Evaluating Software Security Aspects through Fuzzing and Genetic Algorithms

Version of December 7, 2008

Kevin Yu Zhang

Evaluating Software Security Aspects through Fuzzing and Genetic Algorithms

THESIS

submitted in partial fulfilment of the
requirements for the degree of

in

MASTER OF SCIENCE

by

Kevin Yu Zhang
born in Beijing, China

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Netherlands National Communication Security Agency
Ministry of the Interior and Kingdom Relations
P.O. Box 20010 2500 EA
The Hague, The Netherlands

Evaluating Software Security Aspects through Fuzzing and Genetic Algorithms

Author: Kevin Yu Zhang
Student id: 1015915
Email: KevinYuZhang@nl.ibm.com

Abstract

This document contains the Master of Science thesis research findings of Kevin Yu Zhang on evaluating software security aspects through fuzzing and genetic algorithms.

As software grows in size and complexity, evaluating the security aspects of it becomes very complicated. Static software security inspection is one of the most effective ways of evaluating software security. However, it is still mostly being done manually. Therefore, finding a way to automate part of it would be very helpful.

The combination of fuzzing and genetic algorithms creates a new approach which generates test cases to help evaluate software on their software security aspects.

Thesis Committee:

Chair:	Prof. Arie van Deursen
University supervisor:	Dr. Phil. Hans-Gerhard Gross
University supervisor:	Dr. Leon Moonen
Company supervisor:	Mr. Kevin van der Raad
Company supervisor:	Mr. Bart Botma
Committee member:	Ir. Bernard R. Sodoyer

Preface

I would like to thank Leon Moonen for giving me the possibility to do this thesis project and Gerd Gross for giving me guidance and support during me thesis research. I would also like to express my sincere gratitude to my two company supervisors Kevin van der Raand and Bart Botma. Without their advice, patience and supervision this thesis would not have existed.

I would like to thank my parents for offering me the opportunity to study and for their understanding and love through thick and thin. Furthermore, a big thanks to all family and friends for their support and interest. Last but not least, I would like to thank my girlfriend, Jade, for encouraging me and for her never ending support. I would never have been able to finish this without her motivation.

Kevin Yu Zhang
Delft, the Netherlands
December 7, 2008

Contents

List of Figures

Chapter 1

1 Introduction

As computer hardware becomes progressively revolutionized, computer software also becomes increasingly complex. Today, it is no longer uncommon to see computer programs contain more than millions of lines of code. The increase in size and complexity of computer software has led to a rise in software security problems. New software vulnerabilities are frequently discovered. It is hence important for all software systems to be thoroughly tested before they can be deployed. This holds especially true for software systems that are critical or are used to transfer secure information. Consequently, software audit review has become a new rising aspect of software development.

Software audit review is defined in IEEE Standard for Software Reviews as a type of software review in which one or more auditors, who do not belong to the software development organization, conduct "an independent examination of a software product, software process, or set of software processes, to assess compliance with specifications, standards, contractual agreements, or other criteria". NLNCSA, the company this master project is directed at, performs such software review to ensure the safety of the software used by the Dutch government. As an initial effort, this very first chapter briefly looks at the current software security inspection situation at NLNCSA and how software audit review is performed within the company. It is also intended to give a general introduction to the topics such as software vulnerability and input validation. The chapter is further ended with a definition of the research questions of the project as well as a short outline of this master thesis.

1.1 The Company - NLNCSA

NLNCSA stands for Dutch National Communication Security Agency, located in Zoetermeer, the Netherlands. The company is responsible for the development, evaluation and/or procurement of security properties of (collections of) ICT products that will be used by the Dutch government for processing classified information. One of the instruments it uses is the Common Criteria (CC) for Information Technology Security Evaluation, which is adopted as ISO/IEC 15408. The company also contributed to the development of version "CC 3.X" of the CC [1].

The CC addresses protection of information from unauthorized disclosure, modification, or loss of use. A CC evaluation process establishes a level of confidence that the security functionality of an ICT product and the assurance measures applied to the product meet the defined requirements. The philosophy of the CC is to provide assurance based upon the evaluation (active investigation) of the ICT product that is to be trusted. The CC further proposes having expert evaluators measure the validity of the documentation and the resulting ICT product with an increasing emphasis on the scope, depth, and rigor. The CC, however, do not state how vulnerability assessment should be conducted. Currently, there is little automated support for doing such assessment, making the process labor-intensive and error-prone. One of the assessment techniques used by NLNCSA is software security inspection. It should be obvious enough that such inspection has to be conducted as thoroughly as possible.

1.1.1 Software Security Inspection

Software security inspection can be performed in two ways - dynamic software security inspection and static software security inspection, both of which are part of the software security inspection process and are an important aspect of software audit review. For a complete and thorough inspection both approaches should be used.

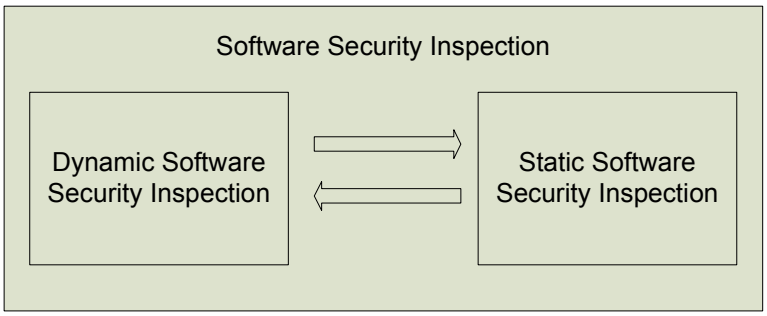

Figure 1: Software Security Inspection

Dynamic software security inspection is focused on testing software security by running the software with a set of possible inputs in a controlled environment in order to detect software vulnerability. In this case, the strength of inspection relies on how well the inputs are chosen. Reliable inputs should contain as many variations as possible, yet remain at a reasonable size.

Static software security inspection, on the other hand, is focused on inspecting source code of a target of evaluation. Due to the lack of proven working tools, a large amount of static software inspections are carried out manually. Detailed inspection requires the auditors to discover potential software vulnerabilities by looking closely at the source code and the design documents. This process generally consumes a great deal of time and human resources and thus is very costly. It also relies on the auditors' personal ability to perform the inspection correctly. Even experienced auditors can make mistakes inspecting software. Therefore, a more automated way to perform such inspection is desired. This thesis focuses on improving static software security inspection.

1.2 Problem Context

Assessing software security can also be divided into two phases. The first phase is to evaluate the process of software development. The second is to evaluate the source code of the software itself. In the case of NLNCSA, the target of evaluation is already developed; therefore, it can be classified as dynamic and product driven.

In general, the software to be tested comes in various shapes and sizes. The software this project is directed at is network traffic software written in C with the size of approximately 100,000 lines of code. This, however, does not mean that there is no software in Java or web applications in XML that need to be audited, but considering that they are less common, they are not the focus of the project. Before defining the research goal of this project, it is useful to have more background information on the subject.

1.2.1 Software Security

There are different opinions about what is meant by secure software. In [2] Butler W. Lampson explains that it is very costly and mostly inefficient to build absolutely secure computer software.

In real life, the users are often satisfied with reasonably secure software that can report and track down attacks so that attackers can be identified and punished for their actions. This is comparable to a security system in real world where a thief steals from a store. The store manager generally does not expect to avoid theft at all, as long as he is able to record and find out immediately when there are items getting stolen. The same applies to the computer software world.

However, in some cases where the target of evaluation is developed for a security-sensitive department of the government, it is vital that the information stays secure. In this case the common track and trace techniques may not be sufficient. Punishing the attacker after an attack, as described in [3], may prove to be already too late. More thorough software vulnerability assessment is hence needed even before the questionable software is deployed, in order to ensure software security.

1.2.2 Software Vulnerability and History of Software Vulnerability Assessment

It is considered necessary to pay attention to several terminologies used in this report before moving onto details. Software vulnerability is seen as the most important term in software auditing. The Common Criteria [1] defines software vulnerability as a weakness in the target of evaluation that can violate the security functional requirement in certain environments. This means that software vulnerability is part of the code that makes the system behave differently from what is defined in its security functional requirement. A system's security functional requirement is used as a reference point of comparison to decide whether the program is behaving accordingly or not. It is therefore important to have a high-quality security functional requirement document. However in cases where such documents are not offered, it is imperative to be able to find software vulnerabilities in a systematic way.

As shown in [4], software vulnerability assessment has a long history: from the periods of formal approaches applied to system certification and validation, to the periods when 'simplistic' tools are introduced to perform the task of vulnerability assessment, then to macroscopic approaches for studying the fundamental output of the complex nonlinear system known as software development, and finally to the present, when state-of-the-art tools and methodologies are beginning to apply principles of formal methods to the evaluation of software.

1.2.3 Exploitable Software Vulnerability

Aside from software vulnerability, exploit is also frequently mentioned in software security testing. In the software security field the word 'exploit' is a short-form expression of 'exploitable software vulnerability'. Anil Bazaz points out in his work [5] the difference between vulnerability and exploit: An exploit consists of a vulnerability present in software application and the method used to take advantage of that vulnerability. In the book he also gives an example of a buffer overflow exploit, which contains a vulnerability by means of an unbounded buffer and the way to exercise that vulnerability by storing data larger than the size of the buffer. The combination of the two makes the overflowed buffer an exploit.

In general, a software program can be viewed as a block, which takes certain inputs from either the users or other computer programs, performs calculations, and outputs a result at the end. Combining this concept with the definitions of software vulnerability and exploit, exploitable software vulnerability can thus be defined as a weakness in the target of evaluation that can be exercised by user inputs.

1.2.4 Static Analysis

In static analysis, the target program is analyzed without running it. As pointed out by Brian Chess and Gary McGraw in [6], static analysis is one of the best ways to protect software security and should always be part of any software audit review process. Static analysis makes use of the tools that attempt to analyze the source code statically without executing the code. There are a number of static analytical tools available, as shown below. Although helpful, these tools are all only useful in a specific programming language or situation and are not overall applicable.

Tool Name	Description
BOON	Determine whether a C program can index an array outside its bounds by applying integer range analysis
CQual	Detect format string vulnerabilities in C programs by using qualifiers to perform a taint analysis
xg++	Attempt to find kernel vulnerabilities in the Linux and OpenBSD by using a template driven compiler extension
Eau Claire	Create a general specification-checking framework for C programs by using a theorem prover
MOPS	Look for violations of temporal safety properties using a model-checking approach
Splint	Extend the lint concept; Try to find abstraction violations, unannounced modifications to global variables, and possible use-before-initialization errors by adding annotations

Table 1: Examples of Static Analysis Tools

In [7] the author shows that even when a static analytical tool has been chosen, using it correctly is not as straightforward as it seems. There are three basic questions to be answered before a static analytical tool can be correctly applied:

- Who runs the tool?
- When is the tool run?
- What happens after the tool is run?

In the case of NLNCSA, the tool is run by the company after the complete program has been developed by a third party. This definition provides a baseline for the scope of further investigations.

1.2.5 Dynamic Analysis - Software Testing

Dynamic analysis is basically software testing. As shown in [8], software testing can be performed at different levels - unit testing and system testing. Unit testing is focused on a small part of a system, while system testing is focused on the system as a whole. There are several well-defined testing criteria for unit testing such as mentioned in [9], [10], and [11]. In the case of NLNCSA, however, a system level solution is required. Jane Huffman Hayes proposes in [8] a preventive method that targets at system level. The method is intended to achieve software security through evaluating system development process, which is hence not applicable for NLNCSA because the company aims at auditing programs that have already been developed. Further, detecting software vulnerabilities at system level can be very costly because most unit level testing techniques and tools cannot be applied to millions of lines of code and system level tools tend to be very complicated. Therefore, what the company looks for is an approach that can aid or automate the current white-box software security inspection.

1.2.6 Input Validation

In [8] input validation refers to those functions in software that attempt to validate the syntax of user-provided commands or information. Through input validation user inputs are checked for completeness and correctness. Software security issues are often related to unreliable inputs from external parties such as the users. For any external parties to exploit software security vulnerability in a program, they must go through input validation. Input validation, if done correctly, does not allow external parties to exploit software security vulnerability. This is the reason why evaluating a system's input validation is an important aspect of software security inspection. This master project is focused on the input validation evaluation part of software inspection. The goal is to develop techniques and tools that can aid software security inspection, by providing support for finding, extracting and evaluating input validation in a software system.

1.3 Research Questions

The initial research goal of the project is defined as:

> **How to use and extend current input validation evaluation techniques and tools to aid white-box software security inspection?**

The following sub-questions are intended to extend the initial research goal towards detailed research directions:

1. *What are the currently available input validation evaluation techniques?*
   It is believed that there are several existing techniques that are interesting to be investigated. During the literature research a list will be compiled of all existing input validation evaluation techniques that are considered to be helpful.

2. *How to use and extend existing static analysis techniques to help software security inspection?*
   To make use of the fact that the source code of a program is generally available, attention will be paid to static analysis techniques first. The source code of the target of evaluation is generally available under a non-disclosure agreement. For each existing static analysis technique gathered in the previous step its application and extension into use will be evaluated.

3. *How to use and extend existing dynamic analysis techniques to help software security inspection?*
   In addition, dynamic analysis to the investigation will be included. The goal is to explore how input validation extracted from static analysis techniques can be used to help generate input validation test cases and hence aid software security inspection.

4. *How to combine static and dynamic techniques to perform automated input validation evaluation?*
   The last step is to combine the static and dynamic techniques that are evaluated to be applicable or extended into a new approach to help carry out input validation evaluation automatically.

The initial research question is defined very broadly which allows literature research into all directions and approaches that might be helpful for white-box software security inspection. Going

through the abovementioned sub-questions stepwise, a more focused research goal is defined as result from the literature research.

How to use and adapt software security inspection approach fuzzing to aid input validation evaluation?

The literature research showed that static approaches are generally too complex when automated. Hence these automated approaches usually have problems with large target programs. Because one of the most important goals of the company was to create a tool that can handle large programs, while the initial focus direction of the research was on static software security inspection it was decided to use fuzzing, which is a dynamic approach as a base approach for next phase research.

1.4 Outline

To answer the research question, first a literature research was conducted. The findings from this research are presented in Chapter 2. It explains and evaluates currently available approaches of input validation evaluation. Both static and dynamic approaches are presented to create a more complete overview. At the end of Chapter 2, three directions are presented for further research. Only one of these approaches was chosen because of the limited time available. Chapter 3 presents fuzzing which is the approach that was chosen at the end of Chapter 2. Both the advantages and short comings of traditional fuzzing are explained as well as a more concrete implementation of one example of traditional fuzzing. Chapter 4 goes a step further and explains how fuzzing can be further enhanced with genetic algorithm to be more effective. Chapter 5 shows how the proposed approach is implemented as a proof-of-concept. Testing and test results of the proof-of-concept implementation are presented in Chapter 6. The results from testing form the basis for Chapter 7 conclusion and future work.

Chapter 2

2 Current Approaches

After defining the research question and clarifying the goal of the project, background information needs to be gathered to formulate an appropriate approach. To do so, a literature research is initiated with the goal of searching for possible software-security-inspection-related software analysis approaches. This chapter aims to summarize the findings from the literature research that are considered valuable for the determination of the research direction of the next phase of the project. The chapter starts with explaining the approaches that are most related to the research goal or are in one way or another helpful to form a new approach. Based on the findings, three possible research directions are proposed. Due to the limitation of time and available resources, only one of the research directions will be chosen to be carried out in the next phase of the project. Therefore, a comparison analysis is conducted among the proposed research directions.

2.1 Current Approaches

The existing approaches can be divided into two categories - static input validation evaluation that mainly looks at the source code of the target of evaluation and tries to find out existing software vulnerability and dynamic input validation evaluation that runs the target of evaluation under controlled conditions with wisely chosen pre-calculated input values containing all boundary conditions. Although the literature research is supposed to cover a complete software analysis area, a selection must be made. Through the literature research only a few approaches are chosen to be further evaluated, as presented in Table 2. The selected approaches are either closely related to the research goal or considered helpful to complete the overview of software security.

Static Input Validation Evaluation	Dynamic Input Validation Evaluation
Tool for input validation understanding and maintenance (TIVUM) [12]; Extracting code fragments that implement functionality from source programs [13]; Framework for deriving verification and validation strategies to assess software security [5]; Program slicing [14]; Software vulnerability assessment version extraction and verification [15]; Matching attack patterns to security vulnerabilities in software-intensive system designs [16]	Fuzzing [17]; Input validation analysis and test method (IVAT) [18]; Using parse tree validation to prevent SQL injection attacks [19]; Input validation analyzer and test case generator (IV-ATCG) [20]; Testing for software vulnerability using environmental perturbation [21]; Input selection and partition validation for fuzzy modeling using neural network [22]; Optimal software testing and adaptive software testing in the context of software cybernetics [23]

Table 2: List of Current Approaches

Static Input Validation Evaluation

While being helpful, the mentioned approaches still lack a number of characteristics in one way or another. For example, a few static input validation evaluation techniques make use of the technique called program slicing. Although program slicing is useful for input validation evaluation, it also has clear disadvantage when target program is large - due to its complexity program slicing requires increasingly more time when target program increases in size. This is the

most common weakness of static input validation evaluation techniques. While working correctly when the target program is small, the required time and effort increase exponentially when the target program grows in size.

Dynamic Input Validation Evaluation
In dynamic input validation evaluation techniques the target program is run with a set of input data. These input data are often called test cases. The difference between different techniques lies in the way these test cases are generated. Because of the nature of testing, these techniques often do not use information from source code and thus cannot show how thorough the investigation has been. Therefore, dynamic input validation evaluation techniques are mostly used in cases where the source code is not available. Because NLNCSA does have source code available, it was looking for an automated static analysis approach as first choice. However, certain aspects of dynamic approaches can still be useful especially when combined with other techniques to take advantage of the fact that source code is available.

2.2 Proposed Directions

As presented in Chapter 2.1, although there are several approaches available, most of them do not directly or effectively deal with input validation evaluation. This chapter proposes three new directions which are either based on or the combinations of the approaches described in Chapter 2.1:

- **Direction 1** Extracting Validation Flow Graph (VFG) from source code to create test cases;
- **Direction 2** Using Fortify SCA to implement the approach proposed in [13];
- **Direction 3** Fuzz testing using information extracted from source code.

Among the three proposed directions only one was selected for further research and implementation based on the requirements of NLNCSA.

2.2.1 Direction 1: Extracting VFG from Source Code to Create Test Cases

This approach consists of two phases. The first phase is to extract VFG of the target program. Thereafter, test cases that aim at input validation evaluation are generated from the VFG. This approach is based on the two approaches proposed in [12] and [20]. Combining the two approaches allow them to compliment each other.

In [12] the author proposes an approach for generating VFG from source code using a smart modification of the old-fashioned program slicing. The main problem with this approach is the fact that, even after the VFG has been extracted, it still requires the software auditors to read the generated VFG manually. This problem becomes more apparent when target programs become increasingly large. The amount of code to be manually inspected is still too much.

In [20] the author proposes an approach that generates test cases using VFG. This approach complements the approach mentioned above. By generating test cases automatically from VFG, auditors no longer need to look at the VFG generated from source code. Instead, they can simply use the generated test cases to test the target program. The weakness of this approach is that it needs a VFG to operate. However, combined with the previous approach which generates VFG of the target program as output, this approach is able to have the desired input.

Analysis
Although both abovementioned approaches provide a tool, respectively, it is difficult to acquire the tools. This means that the project will have to redevelop the tools correctly as described by the authors, which can be tricky. Furthermore, the two tools will need to be integrated. It is not certain whether and how this can be done. Therefore, the risk of choosing this direction is that the result might not be affirmative, resulting in the project to be fruitless.

2.2.2 Direction 2: Using Fortify SCA to Implement the Approach Proposed in [13]

This direction tries to adapt the approach described in [13] to input validation evaluation, using an existing commercial tool. One of the issues that have become clear during the literature research is that there are very few tools available for input validation evaluation. Most of the approaches proposed are either for specific programming languages or merely general descriptions. However, there are commercially available software security inspection tools on the market. These tools often aim at software security in general and are made to be easy to apply and learn. Fortify Source Code Analyzer (SCA) is one of these commercial software security inspection tools. This direction takes Fortify SCA as base and looks at how it can be adjusted to implement the proposed approach.

Fortify SCA and Audit Workbench
Fortify SCA carries out static analysis. After installation, Fortify SCA provides the auditors a full analysis of the target program. The auditors can modify the focus of the analysis by changing the parameters used by Fortify SCA. To do so, they can make use of Audit Workbench which is another tool that works together with Fortify SCA. Audit Workbench provides the auditors the ability to quickly filter out issues that are not desired to be audited, resulting in an analysis with only specified focus.

Rulepack
Rulepack is a functionality of Fortify SCA which allows the auditors to write their own custom rules. This opens up a lot of possibilities. For NLNCSA it is interesting to see whether and how this custom rules can be written in the way that only input validation is analyzed. The direction focuses on building a tool that can be integrated into Fortify SCA, using Rulepack to analyze input validation.

Analysis
Fortify SCA is a commercial tool. If the project is successful and able to generate Rulepack that implements the proposed approach, the approach is still only available in one tool, Fortify SCA. Since all work is done by Fortify SCA, there may be license issues if using the tool commercially. Further, if the project fails at generating Rulepack for the proposed approach, it does not mean that it is impossible. The tool chosen may just not be suitable for the case. In general, the main risk of this direction is that it may be too narrow.

2.2.3 Direction 3: Fuzz Testing Using Information Extracted from Source Code

Fuzzing is a special form of black-box testing. It involves feeding the target program with randomly generated inputs. As explained in Chapter 2.1, fuzzing is extremely suitable for the need of NLNCSA because of its ability to handle large programs, in contrast to code extraction approaches which are usually fine when tested on small programs but become increasingly complex when applied to large programs.

In spite of its strengths, fuzzing has two major weaknesses which make it less suitable for the project. First, it requires a huge amount of time just to run and test the target program because generated inputs need to go through the same path, which is inefficient. Second, passing a fuzzing test does not mean that the program is secure. The test set is always only part of the possible inputs.

Direction 3 aims at overcoming these two weaknesses so that fuzzing can be used to check input validation vulnerabilities. This is achieved by using source code information to generate test cases, which makes the generated test cases less repetitive and thus the testing process more efficient, compared to the old-fashioned fuzzing. Test cases generated using source code information, if done correctly, should also be more complete than fuzzing alone.

Test Case Generation and Fuzzing
If test cases are generated only based on source code, the result will be limited in the sense that only the vulnerabilities that testers have thought of are included. By adding fuzzing to the process, the vulnerabilities that testers do not foresee will also be exposed. In general, fuzzing and test case generation based on source code are contradictory. The former generates random test cases, while the latter generates test cases following a pre-defined set of rules. The right balance of these two approaches may, however, prove effective.

Analysis
As explained above, the proposed direction does not have a solid theoretical background. Whether or not it will work remains to be proven. Also, the direction is dynamic, which means that target programs will need to be run under controlled environment. This is, however, not the initial goal of the project.

One of the most important benefits of this direction is that unlike other directions it is suitable for large systems. Furthermore, the literature research shows that fuzzing has not been modified to suit the need of input validation testing before. This direction is thus new in this aspect. If proven successful, it may pave a new way to input validation evaluation. As NLNSCA has shown full-size interest in theoretical research, this direction will also be beneficial from this perspective.

2.2.4 Selected Direction

As explained earlier, this project, due to its time limitation, is only able to work out one of the proposed directions. The selection is made based on the criteria given by NLNCSA. The two most important criteria are as follows:

- **Scalability**
  Scalability demands that the proposed solution must be able to take on small programs as well as large programs consisting of around 100,000 lines of code. This is the most important criterion in the case when the target program is large such that it is too much work to perform the evaluation manually. Direction 1 becomes increasingly complex when the target program gets larger; therefore, it is not selected.
- **Automation**
  This is because that when the program becomes larger, it is very likely to test the target program by hand. Therefore ideally all actions needed to perform the test should be automated.
- **Prior knowledge**
  Because the random nature of fuzzing it is clear that it is impossible to create an automated fuzzing tool that can find all input validation vulnerabilities. However, the

proposed approach can still be useful if it can provide a good first round check. This means that the approach should be able to be run without too much prior knowledge of the target program.

Based on the criteria and after discussing with NLNCSA, Direction 3 - Fuzz testing using information extracted from source code - is selected as it matches the best with the need of the company. While no further researches were conducted on the other proposed directions in this thesis project, they do form a good foundation for future projects.

After further research into fuzzing it becomes clear that fuzzing alone will not be sufficient to solve the input validation evaluation problem. Therefore, a new approach smart fuzzing was developed to provide more intelligence to fuzzing by enhancing fuzzing with genetic algorithm. Chapter 3 provides an overview of traditional fuzzing and describes how it can be applied to input validation problem by showing an example implementation of a proof-of-concept tool. Chapter 4 focuses on smart fuzzing.

Chapter 3

3 Traditional Fuzzing

Although the proposed approach is based on fuzzing, it is different than traditional fuzzing. This chapter explains how traditional fuzzing works in theory. The chapter starts with a look at traditional fuzzing, its advantages and shortcomings, followed by a typical implementation of traditional fuzzing algorithm. This chapter also explains why traditional fuzzing is incapable of solving the problem at hand. The next chapter introduces the SF approach, which is "smarter" than traditional fuzzing.

3.1 The Process of Fuzzing

Fuzzing is the approach that was found most suitable to solve the input validation evaluation problem after literature research. It is a black box testing approach which as explained in [24] can be divided into the following phases:

1. *Identify Target*
   When applying fuzzing, the actual tool and technique applied is determined by the fuzz target. It is therefore important, first to choose the target then apply the corresponding technique and tool. A different target will need totally different approaches to fuzz. For example, fuzzing network protocols and web browsers using completely different technologies and tools, only the framework of fuzzing remains the same.

2. *Identify Inputs*
   Most exploitable vulnerabilities are caused by incorrect input validations. This is why locating all possible source of input is the base of any successful fuzzing. In the end, anything that goes into the system should be considered user input and should be fuzzed accordingly. This includes things such as headers, filenames, environment variables, registry keys which are usually considered system files instead of user input.

3. *Generate Fuzzed Data*
   When all user inputs are located, each of them should be tested using fuzzed data. The generation of fuzzed data can be done using several different techniques. Each of them has their own strengths and weaknesses. The approach used here uses a fuzzed data generation method which specifically targets input validations as that is the main aim of this research.

4. *Execute Fuzzed Data*
   The key point in this phase is automation. This is where fuzzing differs from most other techniques. Through automation, a huge amount of test cases can be tested in a relatively short time.

5. *Monitor for Exceptions*
   Exception or fault monitoring is often overlooked, yet it is imperative for a successful fuzzing. Throwing 1000 test cases at target system and seeing it crash gives no other knowledge than the fact that there is vulnerability in the target system.

6. *Determine Exploitability*

Once an exception occurs, to find the actual vulnerability that causes the exception it is necessary to trace the exception back to the vulnerability. This is usually done manually as it requires specialized security knowledge.

As can be seen in the figure above, the key quality of fuzzing is that it is a fully automated process. After the fuzzer is implemented anyone with minimal programming knowledge should be able to run the fuzzer and find vulnerabilities. Although, as shown later in this chapter, fuzzing alone is not sufficient to find all input validation vulnerabilities, it is a good first round check to get rid of the easy-to-spot vulnerabilities. Experts' knowledge will still be needed to find the real hard-to-find vulnerabilities, but the fuzzer will free them from doing a lot of work by finding the easy ones for them, which is what this approach aims to achieve. Therefore, the proposed approach is aimed at ease of use and automation with verifiable completeness and thoroughness.

3.2 Implementation of the Traditional Fuzzing Approach

This section shows a detailed description of one approach of traditional fuzzing which provides more details on how fuzzing works. Although it is slightly modified to suit the target program, the idea behind remains the same. Assuming that the target program has been chosen and the inputs identified, the first step of traditional fuzzing is the generation of test cases. In fuzzing test cases can be generated using different randomized methods. These methods are mostly either based on mutating a set of correct inputs or generating a whole set of inputs from program specification and/or communication protocols. Either way, neither complete path coverage nor the consistency which are needed can be guarantied.

For comparison purpose the test cases are generated using a two-step algorithm. The first step generates seed test cases that are then randomized in the second step to create fuzzed test cases. This method is a slight modification of traditional fuzzing, but is essentially the same. The target system is run with the fuzzed test cases, and the exceptions are monitored. In general, the approach can be divided into three steps: fuzzed data generation, automated testing, and exception handling. Each of these steps will be explained in more details in this section.

3.2.1 Test Case Generation

In this step fuzzed data are generated using a combination of generation rules and randomized mutations. Fuzzed data are in fact test cases that are going to be fed to the fuzzer. The generation of fuzzed data is split in two steps. The first step generates seed fuzzed data followed by the second step which generates the actual fuzzed data. Seed fuzzed data are generated using a predefined rule. Each variable in the input stream is given a few edge values that usually cause vulnerabilities. This seed fuzzed data is then mutated a few times depending on the settings to generate random fuzzed data. The two steps are carried out sequentially. A complete view of fuzz generation is shown below.

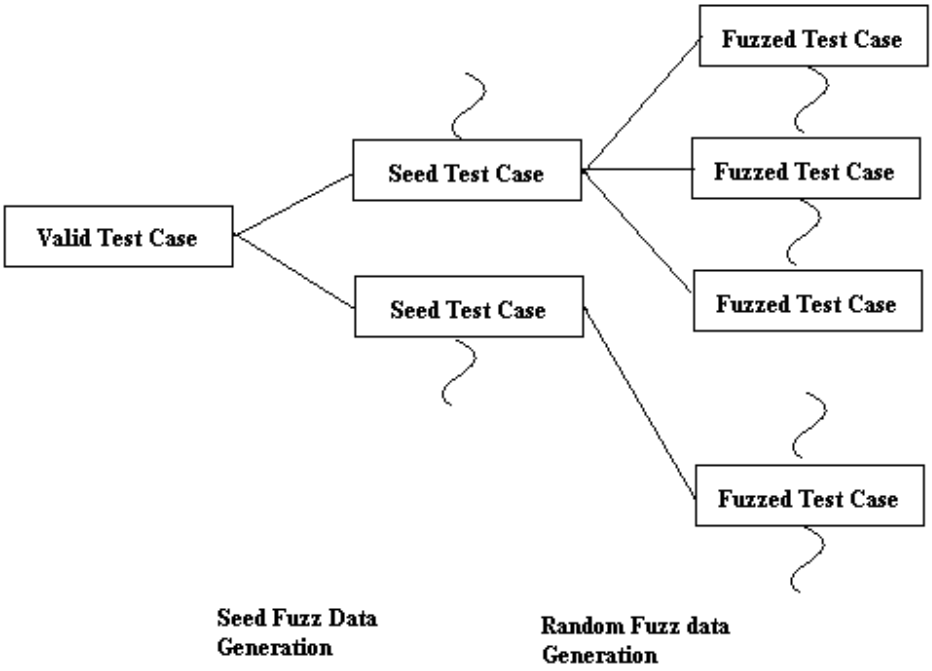

Figure 2: Two Steps of Fuzzed Test Case Generation

The fuzzed data generation is split in two steps because generation using preset rules and random mutations should be in balance. If only the preset rules were used to generate fuzzed data, the core of fuzzing, which is its ability to find overlooked vulnerabilities, would be missed and only those validations that are implemented would be tested. If only randomly mutated fuzzed data were used, either there would be too many test cases to test or only a very small part of the target program could be covered. Next, the predefined seed fuzzed data generation rules and mutations of random fuzzed data will be explained in more detail.

Seed Fuzzed Data Generation

The seed fuzzed data is generated from a set of valid test cases using a predefined rule. The valid test cases are chosen to cover as much functionality of the target program as possible. Ideally all functionality should have its own valid test cases in the set. Also functionality such as ignoring a certain input should be included. In the end, the more complete the set of valid test cases the better.

One seed fuzzed data is generated for each variable in every valid test case. The corresponding variable is overwritten with values that are likely to cause vulnerabilities. These values are chosen beforehand and are based on past experience. The position of each variable is marked and stored in a table together with its length, both of which are used in random fuzzed data generation.

The seed fuzzed data are intended to set the focus of fuzzing on the variables in the input stream one by one. It also allows the auditors to only fuzz one specific variable when needed. This greatly reduces the total number of test cases. However, it also reduces the chance of finding vulnerabilities because by using seed fuzzed data the freedom of fuzz data generation is being severely limited.

Random Fuzzed Data Generation
After all seed fuzzed data have been generated, they are used to create the random fuzzed data. This is done using a few control variables. The most important control variable here is the one called “random factor”. The “random factor” can be set to any positive integer number by the auditors. It determines how many random fuzzed data should be generated per seed fuzzed data. This is strongly related to the available time and the targeted thoroughness. While more random fuzzed data per seed fuzzed data does result in better thoroughness, it also increases the time needed to run the fuzzer.

The random fuzzed data is generated based on the following rules. Each random fuzzed data is derived from one corresponding seed fuzzed data. The total number of random fuzzed data per seed fuzzed data is determined by the random factor. When creating the random fuzzed data of a certain seed fuzzed data, the part that is actually being validated by the implemented validation will be fuzzed while the rest of the input will stay intact. This restricts the random part so that the input is only fuzzed at the targeted part. The position and length of the variable corresponding to the seed fuzzed data are stored in a table when seed fuzzed data is generated.

3.2.2 Automated Testing
After all fuzzed data are generated, the next step, testing, can be started. As mentioned before, the key of fuzzing is that the testing should be automated. In this step the fuzzer will start up the target program and feed the generated test cases to it automatically. The automated testing can be done using seed fuzzed data only or using seed fuzzed data and random fuzzed data both. This gives the users more control over which test cases they want to test and the possibility to balance the generated fuzzed data with random fuzzed data. The right balance may be different for different target programs.

In order to be efficient the testing is done entirely automatically. If the target program fails and terminates as result of the input, the fuzzer will record the exception and keep going with the next input. If the target program does not fail as result of the given input, it will be terminated by the fuzzer after a preset time limit. Although this does not mean that the given input will not cause any exception as it might not have had enough time to generate an exception, considering the overall time and efficiency a time limit is necessary. This is again one of the variables that need to be fine tuned.

3.2.3 Exception Handling
When exceptions do happen, the fuzzer will record it during the testing phase. But in order to find out if the vulnerability is actually exploitable or not, the exception needs to be able to be recreated. This is done with the help of the exception records from the testing phase. Using these records, the input that causes the exception in the first place can be found and fed to the target system again with a debugger. The debugger will help track the execution of the target system step by step, which should reveal more details about the vulnerabilities. This is however outside the scope of this project.

3.3 Advantages of Fuzzing
The main reason that fuzzing was chosen is because of the unique advantages that it provides when compared to other similar approaches. This section shows these advantages and explains why they are important and are suitable to fit the specific needs.

Automation

The key factor that makes fuzzing unique is the fact that once implemented it is a fully automated approach. This allows the auditors to fuzz the target system without too much effort. The idea is to use this approach on the target system before passing it to the specialists. However, because of the automation which simplifies a large part of the input validation evaluation process, the approach will most likely only be able to find those vulnerabilities that are easy to spot. More complex vulnerabilities that only occur after the target system has been put into certain state are hard to find for a fuzzer. However, as explained before, the aim here is to have a fully automated tool that can help the specialists find the input validation vulnerabilities without too much time and effort as their time are expensive and should be spent on more complex vulnerabilities. This makes automation one of the most important factors to consider when choosing an appropriate approach. For this reason alone is fuzzing the best suitable approach.

Scalability
One of the most common problems that approaches based on program slicing have is the huge increase in complexity while the target program grows larger. As target programs are mostly large it is imperative that the chosen approach can handle large programs. Target programs at NLNCSA usually contain more than 100,000 lines of code. Because of the nature of fuzzing, a fuzzer can handle large programs without too much increase in complexity of the tasks to be performed in the approach. This is the second reason why fuzzing is chosen.

Although, fuzzing fits the two most essential needs and is therefore the most suitable approach, there are still some shortcomings of fuzzing that need to be overcome before it can be applied successfully.

3.4 Disadvantages of Fuzzing

As explained in the previous chapters this project is aimed at finding input validation vulnerabilities. After an input has entered the target system, usually there will be several input validations before it can be used to execute the target system. The aim is to find out how thorough these validations are. Thus, it is in fact a search problem. It is a search to find the vulnerabilities in the input validation of a program. The chance of finding something using randomly generated test cases as in traditional fuzzing is very small.

3.4.1 Coverage

When using fuzzing, test cases are created to test input validations. However, when doing so, one of the problems is that sometimes certain input validations could not be reached. When applied to input validation evaluation, fuzzing employs testing techniques with the specific goal to find vulnerabilities in the input validation part of the program. Because the test case generation is based on the input validation implemented in the source code, if the test cases cannot reach certain input validation, the unreached validation is never tested and the created fuzzed data set is thus incomplete. Even when the validation is reached, if it is not reached with the exact input variable combination that can cause the program to crash, the vulnerability is still not visible. There is usually one specific set of variable values that will trigger the vulnerability and crash the target program. Therefore, to test the target program thoroughly, complete path coverage is needed. Path coverage tests if every possible route through a given part of the code has been executed.

3.4.2 Consistency

Although it is not necessary to find all input validation vulnerabilities, it is important that it is consistent in finding the easier-to-find vulnerabilities. This means that traditional fuzzing must be

able to show that it has covered a certain type of vulnerabilities consistently. This is however not possible with randomly generated test cases in traditional fuzzing.

Both complete path coverage and consistency are the requirements that traditional fuzzing does not meet. Therefore, traditional fuzzing alone is not sufficient to solve the input validation evaluation problem here. The next chapter presents an intelligent fuzzing algorithm which takes care of these weaknesses of traditional fuzzing.

Chapter 4

4 Smart Fuzzing

As mentioned in the previous chapter, traditional fuzzing alone is not sufficient to solve the input validation evaluation problem. This chapter introduces the approach SF (Smart Fuzzing) which uses GA (Genetic Algorithm) to enhance traditional fuzzing. This chapter starts with looking at the exact reason why traditional fuzzing is not sufficient, followed by explaining what GA is and why and how it can be utilized to enhance traditional fuzzing. The chapter ends with a description of the Smart Fuzzing Approach.

4.1 Why Is Traditional Fuzzing Not Sufficient?

When inspecting input validation evaluation, the goal is to find those input streams that cause the target program to behave differently than what it is specified to do. Unlike in general testing where the focus is set at the functionalities of the target program, input validation evaluation inspection focuses on outcomes of possible hazardous input stream. In most cases this will result in the target program terminating abnormally. This is because some statements in the program are run with the values they are not designed to be run with. In order to test the target program thoroughly it is thus very important that all statements are executed with possible vulnerable variable values. Most exploitable vulnerabilities are associated with the types of variables that are known to trigger them. Since the values that a statement uses are the results from the calculations before it, it is often not possible to calculate the input variables in order to create the required values for the statement.

In the example below, the validation at "if" statements requires a specific combination of a, b, c and d to execute the vulnerable code strcpy(y, x). The function strcpy is vulnerable to buffer overflow. In this case, char array y is initialized with size 8. When the size of x is larger than 8, inputs will be written out side of the bound of y because strcpy function does not do boundary checking.

```
char y;
if(a+1==b && a-1==0)
{
      if(c+1==d){
            if(d-2=b){
                  strcpy(buffer,x); // Vulnerable code
            }
      }
}
```

A random algorithm such as used by traditional fuzzing will have great difficulty finding this combination. If the chance of having one variable right is one out of thousand, the chance of having all four variables right is one out of a trillion. This is not acceptable. One may argue that these values can also be calculated manually. However, when a, b, c and d are results of previous complex calculations, the right input values are much harder to calculate manually. Complex calculations such as while loops and run time values can be very hard to predict and calculate.

In order to be able to find any type of vulnerabilities consistently, it is necessary to be able to reach all statements. However, traditional fuzzing is not capable of finding the correct

combinations of inputs that lead the execution to vulnerable code. Therefore, traditional fuzzing alone is not sufficient.

4.2 Fuzzing as a Search Problem

As shown before, traditional fuzzing is not sufficient in finding exploitable vulnerabilities because it is very unlikely to randomly select the correct combination of inputs that leads the execution path to the vulnerable code. In the end the problem is to find the correct combination of inputs. Therefore, the input validation evaluation problem is essentially a search problem. The goal is to find the right combination of input variables that leads the execution path to the vulnerable code segment if there is one.

While input validation evaluation is a unique problem in software security, search problems is not uncommon in software engineering. In the next section a number of known heuristic search techniques are presented and one of them selected to aid traditional fuzzing.

There are a number of heuristic search techniques that can be applied here to enhance the search result. Four of the most representative search techniques are shown below:

- Random Testing
- Hill Climbing
- Simulated Annealing
- Evolutionary Algorithm

Fuzzing is basically random testing, also called brute force testing. It is one of the least effective search methods. Hill climbing is a well know local search algorithm. Simulated annealing is similar to hill climbing but with better search result. Evolutionary algorithm is the heuristic search technique probably with the best result in this case.

4.2.1 Hill Climbing

Hill climbing is a local search algorithm. It starts with one initial solution and improves it until an optimal is found. An optimal in the search space is a point in the search space with a maximum or minimum search value. First, the initial solution is chosen randomly. Then, the neighborhood of the initial solution is inspected. If a better solution is found, the initial solution will be replaced by it. This process repeats until no improvement can be found from the neighborhood of the best solution known.

This solution is best suitable for search spaces with only one optimal in which case the algorithm "climbs" the hill until it has reached the top. However, most search spaces consist of a number of local optimal besides the global optimal. In this case, the hill climbing algorithm will most likely end up finding a local optimal instead of the global optimal. Furthermore, the hill climbing algorithms also have problems when there is a plateau in the search space. A plateau in the search space is a number of points in the search space which all give the same search values. In that case, the neighbors of the current solution will not have better solution and the algorithm is stuck. To counter these problems, most programs restart the hill climbing algorithm a number of times with different randomly chosen initial solutions. This ensures a larger coverage of the search space. However, it still does not give any certainty of finding the global maximum.

There are different types of hill climbing algorithms. "Steepest ascent" climbing strategy evaluates all neighbors. The neighbor with the best solution is chosen as replacement of the current solution. "Random ascent" climbing strategy evaluates neighbors at a random order. The

first one giving a better solution is used to replace the current solution. However, none of these algorithms can solve the problem of hill climbing ending in a local optimal or a plateau.

4.2.2 Simulated Annealing

Simulated annealing is very similar to hill climbing; however, it is less dependent on the starting solution. It allows, with a certain preset chance, acceptance of poorer solutions. This allows for less restricted movement around search space. The probability of accepting a poorer solution changes as the search proceeds. It is calculated with the formula: $p=e^{-d/t}$. In this formula, p is the chance of accepting a poorer solution, d is the difference in search value between the current solution and the neighboring poorer solution and t is a control parameter known as temperature.

The simulated annealing algorithm originates from the chemical process of annealing. The chemical process annealing is the cooling of a material in a heat bath. When a solid material is heated past its melting point and cooled back into solid state afterwards, the structural properties of the material depend on the rate of cooling. The control variable, temperature, controls the rate at which poorer solutions are accepted in time. The temperature is cooled during the search progress, similar to the annealing in chemical term. This means in the beginning phase the solution is allowed to walk around the search space while later on it will climb the hills just like in hill climbing algorithms. Because of this, the dependency on the starting solution is much less than hill climbing.

The temperature variable is a very important factor here. When the temperature cools down too fast, the solution will not have the chance to leave the starting area and end up with local optimal. However, if the temperature cools down too slow, the approach will take too long to end. The simulated annealing is also usually extended with repeated restarts, just like hill climbing, to give it more coverage.

4.2.3 Evolutionary Algorithm

Evolutionary algorithm originates from evolution in nature. It uses genetics and natural selection as search strategy. There are two breeds of evolutionary algorithm: genetic algorithm and evolution strategies. Both algorithms use generations of solutions to find the optimal in the search space. One generation consists of a number of individuals of solutions. The next generation is generated based on the information from the current generation.

Genetic algorithm primarily uses reproduction to generate new generations. With reproduction, information from two or more solutions is used to create new solutions. Evolution strategies usually use mutation, randomly modifying part of the solutions to create new solutions. These two breeds can also be integrated.

As shown in [25], of all heuristic search algorithms evolutionary algorithm yields the best result. Furthermore, genetic algorithm is more suitable for procedure programming languages such as C, which is the main target programming language in this project. Therefore, genetic algorithm was chosen. The next chapter provides a more detailed view of genetic algorithm and shows how it can be applied to enhance fuzzing.

4.3 Genetic Algorithm

As shown before, genetic algorithm is the most efficient search algorithm at the moment [26] [27]. In genetic algorithm a generation of solutions evolves into the next generation using a reproduction rule. The solutions, also called individuals in genetic algorithm, used to reproduce the next generation are selected based on a fitness function. The fitness function determines how

fit each of the individual is. How fit an individual is, is determined by its search value. Individuals with better fitness are used to reproduce the next generation. Genetic algorithm consists of three important phases: initialization, selection and reproduction.

4.3.1 Initialization

In the initialization phase the first generation is initialized. This can be done using several different methods. The most obvious and easy-to-implement method is random selection where the first generation is selected randomly. However, certain key information can be seeded into the first generation to help the genetic algorithm find the solution faster. Although such information is not always available, it is helpful to seed information related to a certain type of vulnerabilities. How this can be done becomes more apparent in the next section where different types of vulnerabilities are explained.

4.3.2 Selection

After the initialization of the first generation, a fitness function is used to determine which individuals are used to produce the next generation. Individuals are chosen based on their fitness values. These fitness values are determined by the fitness function and show how close the individuals are to the supposed target.

Individuals for reproduction can be chosen based on a few different algorithms. In fitness-proportionate selection, individuals are selected with a chance proportionate to their fitness. In short, the fitter an individual is the better chance it has to be selected. Fitness-proportionate selection has difficulty maintaining a constant selective pressure. Selective pressure is the probability of the best individual being selected. To counter this, linear ranking sorts out individuals by selection based on ranks rather than fitness values. This allows a constant bias being applied throughout the search. Selective pressure is more balanced and controlled in this case. In tournament selection, instead of sorting all individuals, two individuals are selected randomly and the fitter one "wins" with a chance of p, with $0 < p < 1$. P is the chance of the fitter individual being selected. It is usually between 0.5 and 1. When $p = 1$, only fitter individuals will be chosen.

4.3.3 Reproduction

When reproducing next generation, the genes of parent solutions are used. Genes, in this context, are the bit stream of a solution. The reproduction can be achieved using several different ways. The most straightforward way is a one-point recombination in which case one cross-over point is selected at random. If the two individuals below are selected and the cross-over point is chosen at locus 4, the offspring will be the ones shown below.

```
                                ↓Cross over point
Parent one:     <0, 255>    0000000011111111
Parent two:     <255, 0>    1111111100000000
-------------------------------------------------
Offspring one: <15, 0>      0000111100000000
Offspring two: <240, 255>   1111000011111111
```

As can be seen here, the offspring do not always have the same characteristics as its parents. To circumvent this problem, more advanced reproduction rules such as multi-point recombination can be used. In multi-point recombination there are several cross-over points where the genes of the parent solution cross. With more information on the format of the input, the cross-over points can be chosen on those points where inputs separate from each other. This way, the input that

causes one solution to be fit can be preserved. Furthermore, there can be more than two parent solutions per offspring in which even more complex cross-over techniques can be applied.

4.3.4 Stop Condition

There are several ways to decide when to stop a genetic algorithm. The mostly used way is to stop after a maximum number of iterations. This number can be found through experience. Although easy to implement, this way does not ensure that the goal will be reached after the genetic algorithm has finished. Maximum no change approach can be used to solve this problem. In this approach the genetic algorithm stops after a number of iterations in which no change in the best evaluation can be observed. This usually means that the genetic algorithm is not making any progress anymore and additional iterations will not be helpful anyway. Another approach is to look at the differences of the fitness values between individuals in the population. When the similarities become too much, the genetic algorithm will be stopped. A high similarity usually means that the individuals in the population have become too clustered. In this case further reproduction will generate offspring that are also similar; thus, the effect of genetic algorithm is lost.

Each stop condition has its advantages. Maximum iterations approach is easy to implement and can always be applied. Maximum no change approach can truly indicate the end of a genetic algorithm. Maximum similarity approach can be used to generate a population that is similar to the best individual. This can be useful to generate a group of suitable individuals instead of one best individual.

4.3.5 Control Variables

As shown above, there are several control variables in the genetic algorithm. Each of these control variables has its own influence on the outcome of the algorithm. Below, all control variables and their effects are shown.

Control Variable	Effect
Number of solutions per generation	The more solutions there are per generation, the bigger the chance of genetic algorithm finding the optimal. However, more solutions per generation also cause the algorithm to run slower.
Number of parents	The more parents per child, the more genetic information that carries on to the next generation. However, this information will also be more mixed when more parents are involved in reproducing the next child.
Number of children	The more children that are reproduced from the same set of parents, the more genetic information passed on to the next generation. However, too many children per set of parents also means that diversity of the population will be lost pretty quickly.
Number of cross-over point	A larger number of cross-over points are only helpful when these cross-over points are chosen based on input format information. Otherwise, it only means that the genetic information will get more mixed as the algorithm proceeds.
Chance of selection	The bigger chance that fitter solutions have to be selected to be parents, the faster genetic algorithm can find the optimal. However, finding the optimal too fast will mean that the genetic algorithm will have a bigger chance ending in a local optimal.

Table 3: Control Variables of the GA

Eventually, the correct setting of control variables for different search problems is different and can only be found through fine-tuning. After a few rounds of trial-and-error the eventual tests performed to evaluate the approach was conducted with 1500 solutions per generation for 1000 generations with two parents per two children and one cross-over point. This set of control variables has been proven to be effective.

4.4 Vulnerabilities

As explained before, with the help of a genetic algorithm, it is possible to reach the vulnerable code. However, only reaching the vulnerable code is not enough. Different types of vulnerabilities require different specific input variables to trigger. Therefore, it is necessary to add additional requirements to the fitness function. These requirements help to determine how likely a solution is to trigger vulnerabilities after the execution path has reached the vulnerable code. Therefore, they have lower priority than other fitness function requirements.

Furthermore, because the input streams for triggering different types of vulnerabilities are different, it is necessary to look at different types of vulnerabilities separately. The genetic algorithm is then run separately for each type of the vulnerabilities. Also during the years when there were attempts to categorize all software vulnerabilities such as in [28], there were still no general standards available. The most representative types of vulnerabilities are listed below:

- Buffer Overflow
- Format String Vulnerability
- Double Free
- Integer Overflow
- Race Condition

Buffer overflow is the most common vulnerability. It is a vulnerability which can only happen in C and C++ because these languages do not do bound checking. Java, for example, does perform bound checking at run time and is therefore freed from buffer overflow vulnerabilities, though other forms of vulnerabilities can still occur. Because most target programs of the company uses C as programming language and buffer overflow is the most common vulnerability, the proof-of-concept tool will be implemented, targeting buffer overflow in C. Format string vulnerability and double free are also looked at for comparison. To create a more complete view of all vulnerabilities, all the abovementioned vulnerabilities will be explained in more detail.

4.4.1 Buffer Overflow

In buffer overflow, a buffer is initialized with a bound which is not enough to contain the information meant to be stored there. When too much information is being written to the buffer after its initialization, excessive information will be written past the bound of the buffer. When the information written past the boundary is formatted correctly, an attacker may gain control of the system. Buffer overflow can be avoided by doing bound checking properly. However, many programmers make mistakes during bound checking, or they simply forget to do it. This may occur in many ways, which is why buffer overflow is a very common vulnerability.

The example code below shows a typical example of buffer overflow. The function gets() does not do automatic bound check. Therefore, it is the responsibility of the programmers to ensure that the users do not input anything larger than the initialized size of x. When the user input is larger than the size of x, 8 in this case, input after 8 will be written outside of the bound of x and cause a buffer overflow.

```
char x[8];
gets(x);    // Buffer Overflow
```

Functions such as gets(), scanf() and strcpy() are most likely vulnerable to buffer overflow because they are not bound checked. These functions are from C libraries and commonly used by a lot of programmers without being aware of the danger. Using safe libraries can significantly decrease the chance of buffer overflow.

4.4.2 Format String Vulnerability

Format string vulnerability is again a vulnerability strongly associated with C. String functions in C are capable of dynamically changing the contents of the string using conversion specifications. "%s", for example, tells the printf() function to interpret the associated parameter as a string. When a parameter which contains a conversion specification is given to the printf() function, the function will try to interpret the next element on the stack using the given conversion specification. This can cause the program to behave strangely or give away secured information unintentionally.

In the example code below, the first printf() function shows typical format string vulnerability. When x contains conversion specification such as %s, the printf() function will interpret x as a format string. This causes printf() to print the next element on the stack as a string variable. Comparing to the second printf() function which already contains a format string, the result is obvious. Because the second printf() function already contains a format string, it will interpret x as a string even when it contains conversion specification.

```
printf(x);          // Format string
printf("%s", x);
```

When the above two printf() functions are being run with x = %s. The first one will return the string representation of the next element on the stack, while the second printf() function will simply print out %s.

This type of vulnerabilities does not seem to cause too much harm besides the fact that the users may get access to some part of the memory. However, there is one format string %n which can be used to write the number of characters that should have been written so far into the address given. If the attackers could figure out the appropriate address they should use, they could crash the system or even take over the control.

4.4.3 Double Free

Double free occurs when the same free() function is being called twice in a program. In this case a pointer is accidentally being freed twice. This can cause the memory management data structure of the program to be corrupted, which will cause the program to crash.

The example code below shows a typical double free vulnerability. The second free() function corrupts the heap because it will be setting the pointer to a wrong place. A corrupted heap will most likely mean that the program will crash at some point. However, there are ways to exploit it.

```
free(x);
free(x);    // Double Free
```

When a program calls free() twice with the same argument, the program's memory management data structure becomes corrupted. This corruption can cause the program to crash or, in some

circumstances, cause two later calls to malloc() to return the same pointer. If malloc() returns the same value twice and the program later gives the attackers control over the data that is written into this doubly-allocated memory, the program becomes vulnerable to a buffer overflow attack [29].

Double free can be effectively avoided by carefully freeing allocated pointers. Also, setting the pointers to NULL whenever a pointer is freed helps preventing it from being freed again.

4.4.4 Integer Overflow

Integer overflow is caused by the same type of programming mistake as buffer overflow. It is also triggered by a boundary check not being performed properly. It is usually caused by incorrect casts or incrementing an integer value outside of its bound, thus resulting in an integer number being too small or negative for the intended purpose.

The example code below shows typical integer overflow vulnerability. When the input is larger than the maximum value of an integer divided by four, int x will be outside of its bound. Then, the next statement malloc(x) will allocate insufficient space for y, in turn, will cause y to be vulnerable to buffer overflow.

```
int *y;
int x = input * 4        // Integer Overflow
y = malloc(x);
```

As can be seen above, integer overflow alone does not cause too much trouble. However, when the integer is used in initialization of other variables, the unintended small value of the corresponding integer may cause the variable to be too small and thus trigger a buffer overflow later in the program.

4.4.5 Race Condition

A race condition happens when two different threads or processes depend on some shared state. For example, when a thread first checks whether a file exists before opening it, another thread may change the file in between the two operations. When this happens, the first thread will be opening a wrong file or even some files that the users have no access right to. The name race condition comes from metaphor with horse race. Here, a thread is simulated with a horse. While the appropriate horse wins the race, the program will behave correctly; when another horse wins the race, the program behaves strangely or crashes. This means that the program expects the threads to happen in a certain order, and when they do not follow the order, the program crashes.

In the example code below, if the operating system schedules another program in between the access() and open() function and when that program also modifies the required file data, corruption may occur.

```
if(access(filename)){
                                  // Race Condition
      x = open(filename);
}
```

In real-life situation, race condition often occurs with file systems where two or more programs try to access or modify a certain file at the same time with no consideration for each other. This will often result in the target file being corrupted.

4.5 The Approach

As explained before, the input validation evaluation problem is essentially a search problem. Therefore, to solve the problem the smart fuzzing approach utilizes genetic algorithm to aid the generation phase of the fuzzing process. Compared to traditional fuzzing which consists of three main phases test case generation, automated testing and exception handling (the other three phases are not included here because they are out of the scope of this research), the smart fuzzing approach has two extra phases: GA (Genetic Algorithm) test case generation and GA automated testing. In total, smart fuzzing consists of the following five phases:

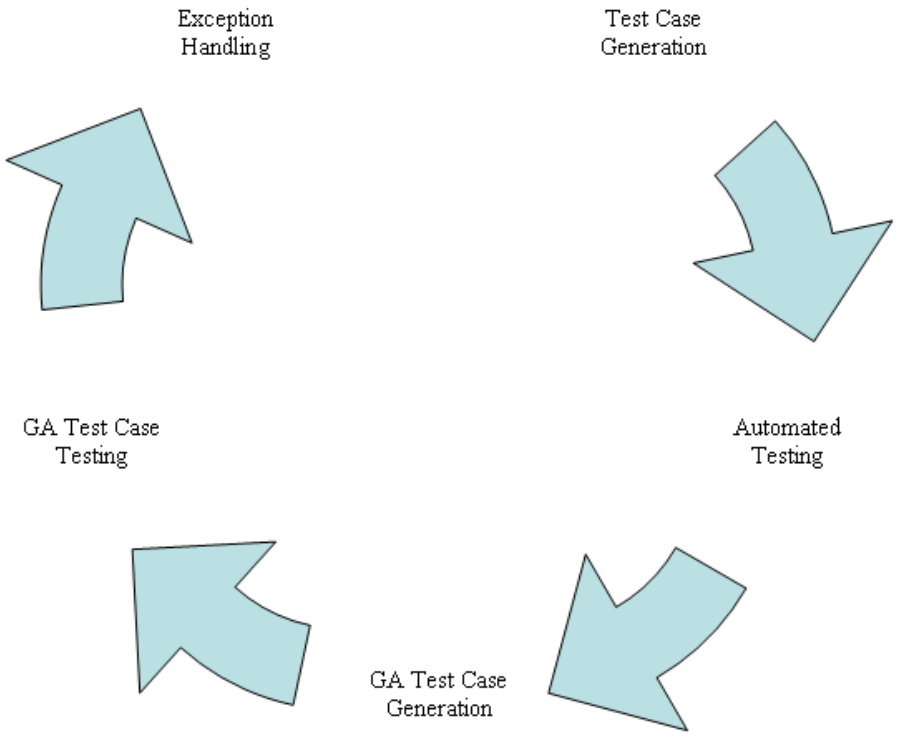

Figure 3: Five Phases of Smart Fuzzing

The first two phases are essentially the same as traditional fuzzing. First, a set of random test cases is generated and run. After the test code has been run, instead of only looking for the exceptions that indicate vulnerabilities, the coverage of the test cases is recorded as well. Although smart fuzzing also starts with random test case generation followed by automated testing, it does not target at finding the vulnerable input here. The random test cases are used to create a general view of which code is being covered.

If any exceptions do happen, the following step will be the same as traditional fuzzing. The test case is run again with debugger to get more detailed information about where the vulnerability is and the cause of it. The test cases that do not cause any exceptions earlier are now used to find the parts of the target program which have not been covered. These parts of the code are usually hard to reach, and if they do contain vulnerabilities, it will be difficult to find using approaches such as fuzzing. This is where genetic algorithm comes in.

In the third phase, GA test case generation, test cases are generated using a genetic algorithm. These test cases are run in the fourth phase, GA test case testing. The exceptions in both automated testing and GA test case testing phases are further analyzed in the last phase, exception handling, which is basically the same as in traditional fuzzing. The major difference between smart fuzzing and traditional fuzzing occurs in the GA test case generation phase and will be further explained in the following sub-sections.

4.5.1 Test Case Generation and Automated Testing

The first two phases of smarting fuzzing is very similar to traditional fuzzing. First, test cases are generated randomly. The difference here is that, the goal is not mainly to directly find vulnerabilities but rather to generate a general coverage view. Therefore, instead of using the usual suspect inputs, it is more beneficial to generate purely random code to ensure that the test cases generated are not only focused on a small part of source code. In fuzzing, experienced auditors usually know a set of inputs that are likely to trigger an exception. Taking an example search space, randomly generated test cases in Figure 5 cover a bigger range than the test cases targeting at certain vulnerabilities in Figure 6. In this case, the test cases target at x = 20 and x = 40. These are inputs that will most likely cause vulnerabilities. These inputs are known to experienced auditors. In traditional fuzzing they are chosen in order for the auditors to focus on the possible weak points instead of spreading the search too broad.

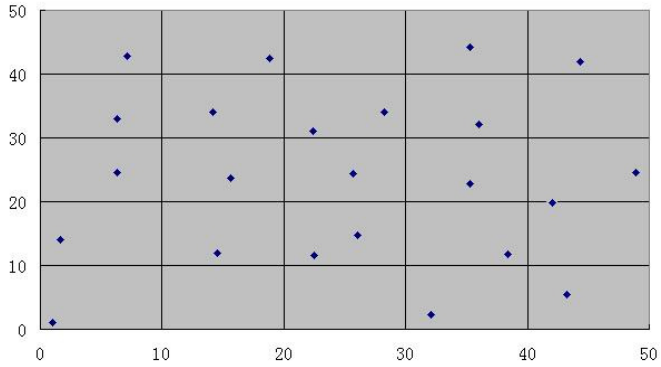

Figure 4: Randomly Generated Test Cases

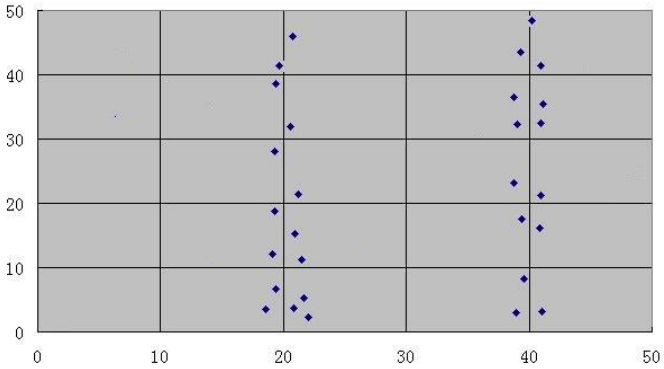

Figure 5: Test Cases Targeting at Vulnerabilities

After the test cases has been generated they are run automatically just like in traditional fuzzing. However, instead of focusing on finding vulnerabilities, the main task in this phase to generate a coverage view of the target program. Although exceptions are still important and are still the first

sign of vulnerable code, the main difference is that the coverage information is gathered. Test cases that did cause a crash or an exception will be stored and examined further in the last phase, exception handling.

Running the test cases under a coverage tool can determine which part of the code is actually being executed. Those codes that have not been reached by random testing are possible vulnerabilities that are hard to reach for traditional fuzzing. Annotations are placed at these places to lead the genetic algorithm to find them in the next phase GA test case generation.

4.5.2 Genetic Algorithm Test Case Generation

The GA-test-case generation phase uses the coverage result from the first two phases. In this phase the test cases are generated using the genetic algorithm led by the annotations left from the second phase. The goal is to find those input data that can lead the execution path to the annotated part of the target program that has not been covered by randomly generated test cases. This ensures a thorough coverage of the source code.

Initialization

In genetic algorithm, the first generation of population is generated either randomly or with preset rules to seed certain information. Both approaches are viable here. While seeding information into the first generation helps the genetic algorithm run much faster, randomly generated first generation is simpler and does not require the auditors to possess any prior knowledge of the source code. This is important because the tool is supposed to be a first check, which means that no types of suspected vulnerabilities or prior knowledge of the source code should be needed to use the tool.

When test cases are generated with preset rules to seed information to aid genetic algorithm, different rules should be applied for different types of vulnerabilities. This means that a group of input data should be generated with a specific type of vulnerabilities as target.

When searching for buffer overflow vulnerabilities, the input data should be larger than the buffer in question to cause the vulnerabilities to emerge after the program has reached the vulnerable code. This means that when generating the first generation, the size of the input data can be set to be larger than the buffer in question. This is important because the genetic algorithm is only capable of leading the execution path to the annotated code; whether the input can actually reveal vulnerabilities depends on other factors of the input. Some of these factors can be added in as a fitness function, but some cannot. In some of the GA's, for example, the size of the input cannot change in because the total size of the input is preset as the length of the chromosome and is used throughout the reproduction phase. It must remain the same as all other individuals from the population in order to reproduce. Therefore, it is necessary to use different GA's for different types of vulnerabilities.

When searching for format string vulnerabilities, the format strings such as %s, %d and %n can be seeded into the first generation, which may speed up the search after the execution path has reached the annotated code. However, the fitness function should also help to find these values specifically to help the genetic algorithm find the format strings even if they are not seeded or are lost during the evolution. Double free vulnerabilities do not require special attention during initialization because they will become apparent as long as they are reached.

Evaluation

After the generation of the first population to begin reproduction, the fitness of each individual must be determined. This is a fundamental part of genetic algorithm. The fitness represents how well an individual suits the goal. In this case, the fitness must show how close a certain input is to reach the annotated code. To achieve this all conditions along the branch are annotated. In front of each condition, a corresponding fitness function is added to help direct the execution path into the condition. The fitness function is created using a preset rule as follows:

Relational Predicate	Fitness Value When True	Fitness Value When False
A > B	F = K	F = (A – B)
A >= B	F = K	F = (A – B)
A < B	F = K	F = (B – A)
A <= B	F = K	F = (B – A)
A = B	F = K	F = – abs (A – B)
A != B	F = K	F = 0
A && B	F = K + K	F = F(A) + F(B)
A || B	F = K	F = max(F(A), F(B))

Table 4: Annotations for Fitness Function

K is a positive integer.

The fitness function is derived from Korel's objective function in [25]. Modifications are made to make the objective function more suitable. The most obvious modification is that the fitness value is at its maximum when the relational predicate is true. This is basically the reverse of Korel's objective function. In this way, the fitness values can be added up every time they pass a fitness function. The genetic algorithm is run with the goal to maximize the fitness value. This means that the larger the fitness value, the closer the input data to the target code.

The fitness function can sometimes also be utilized to help trigger vulnerabilities after the execution path has reached the annotated code. For format string vulnerabilities, the fitness function can be set to find format strings at the annotated code. For buffer overflow, the length of the input buffer must be larger than the receiving buffer. This can only be achieved by having a large enough input to begin with. Therefore, the fitness function is not helpful here. Double free does not need fitness function to help find input either as it will be triggered regardless of input as long as it is reached.

Reproduction

Reproduction consists of two parts: selection and recombination. First, based on the fitness value of the input data a selection is made to choose the parents of the next generation. As explained before in the genetic algorithm section, the selection can be done in several ways. Fitness proportionate selection, linear ranking and tournament selection are all viable selection strategies. Which strategy yields the best result is case-specific and can only be found through trial and error.

Recombination rules are used to reproduce the next generation in genetic algorithm. There are several reproduction rules that can be applied here. One-point recombination and multi-point recombination are both viable ways of reproduction. Again, which one is the best, strongly depends on the target program in question. Although multi-point recombination with cross-over points at the place where input variables separate from each other is obviously more effective because it allows the different input variables to evolve separately, it is too complex to use as the auditors will need to assign those cross-over points themselves. The format of the input is different for each system. Therefore, the recombination process will need to be changed for each system.

Stop Condition
In the stop condition part of the genetic algorithm section a few different stop condition approaches have been explained. In input validation evaluation the genetic algorithm is used to find a set of test cases that can reach the annotated code and hopefully trigger the vulnerabilities there in case it is vulnerable. Therefore, it is useful to have not only one but a group of individuals that all can reach the annotated code; hence, the maximum similarity approach seems to be the best suitable approach.

4.5.3 Testing and Exception Handling

The genetic algorithm test case testing phase is basically the same as automated testing phase in fuzzing. The only difference is, that here the test cases are generated both randomly and in the genetic algorithm test case generation phase. The test cases are run automatically in the same way as in traditional fuzzing.

Chapter 5

5 Implementation

This chapter goes into more details about the implementation in the form of a proof-of-concept. In the first place the plan was to implement a proof-of-concept tool for traditional fuzzing. However, it came apparent during the implementation that traditional fuzzing was not enough to solve the problem of input validation evaluation. The smart fuzzing approach was then introduced to be more effective in finding the vulnerabilities. This chapter starts with a brief description of implementation of traditional fuzzing and some results showing why it is not sufficient. Then, the implementation of the smart fuzzing approach is described in more detail.

5.1 Traditional Fuzzing

In the first phase of this project, traditional fuzzing is implemented as a proof-of-concept tool. The goal is to implement a tool that is fully automated and capable of handling large target programs. Also, to use the limited amount of time more effectively the tool is intended to target at a specific target type instead of a fuzzing framework. This is because the amount of time and effort needed to build a fuzzing framework is huge and it is not the goal of this project to solve the integration problem of fuzzing framework. The tool will generate test cases using the traditional fuzzing approach described before.

5.1.1 Target Selection

The proof-of-concept tool should be able to show that the idea of using fuzzing to help input validation evaluation is effective. To achieve this, file format fuzzing is selected as the type of target programs. This is only the first step. If the result shows that the approach is indeed effective, the next step will be moved on to building input validation enhanced fuzzers for other types of target programs or even a fuzzing framework. However, both of these are for future researches. For now, the focus is at building a proof-of-concept tool that runs file format fuzzing.

In general, fuzzing targets can be divided into the following groups: environment variables and argument fuzzing, web-application and server-fuzzing, file-format-fuzzing, network-protocol-fuzzing, web-browser-fuzzing, and in-memory-fuzzing. In this case, file format fuzzing is chosen. This is because this type of target programs is used the most at the company.

5.1.2 Overall Structure

The proof-of-concept tool is designed in a modular way. This allows different parts of the program to be tested separately. Furthermore, it separates the work of generating and using test cases and thus makes it possible to add and use different test case generation methods to evaluate the effectiveness of fuzzing.

The tool is divided into the following three components: traditional fuzzing test case generator, the fuzzer, and exception handler. The traditional fuzzing test case generator generates test cases using traditional fuzzing. These generated test cases are then fed to the target program by the fuzzer, and the exception handler will catch any exceptions that may happen and provide information to help trace back to the root of the corresponding exception. Figure 7 shows a more complete view of the entire system.

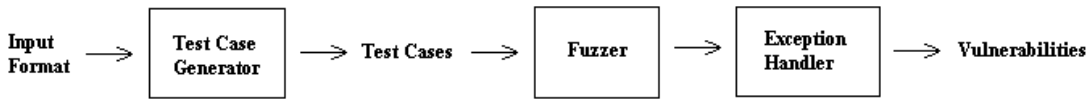

Figure 6: Modular Overview of the Entire System

Because of the modular design, the tool is built modularly as well. Each of the components is built and tested separately and in the end put together. This allows for more progress control and testing during the implementation phase but also leads to a longer implementation phase. Next, the implementation of each component will be explained in more details.

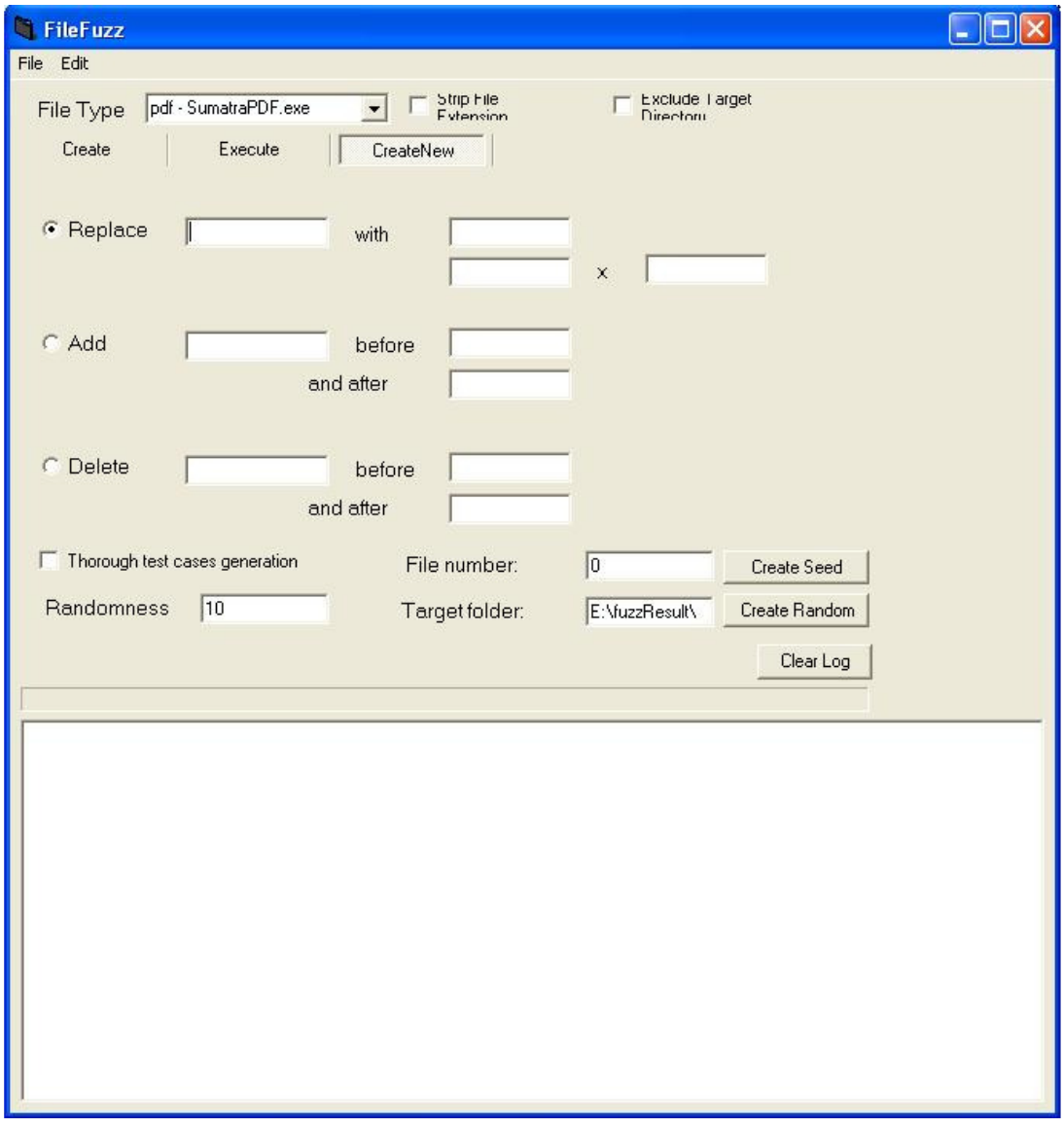

Figure 7: CreateNew Tab of the Proof-of-Concept Tool

5.1.3 Test Case Generator

A well known fuzzer Filefuzz [30] is utilized to generate fuzz data. The reason to use and extend an existing file fuzzer is to avoid reinventing the wheel. Fuzzing is a proven concept; many fuzzers have already been developed. Filefuzz is chosen because it has the basic structure of a file format fuzzer yet is not fully implemented. It can do the basic fuzzing and generate fuzz data at a very simplistic way. While this is certainly not enough, it does give a base upon which the proof-of-concept tool can be built.

Because the fuzz data generator is extended from Filefuzz, the two have similar user interfaces. The user interface of Filefuzz is implemented in main.cs class which is extended to accommodate new functionalities. A new tab, createNew, is added where the users can generate seed test cases from original test cases and random test cases from seed test cases. A screenshot of the tab is given in Figure 8.

As shown in the figure, there are some additional entries to allow the users to specify the place where the fuzzing should be more accurate. These are meant to help limit the generated test cases and improve efficiency.

When the "Create Seed" button is clicked, the function createSeedButton_Click is executed. This function replaces, adds or deletes part of the input template to generate seed inputs which will be fuzzed in the next step. The seed creating step can be repeated several times to create specific seed input data group as the auditors wish. Then, the "Create Random" button is clicked. Based on the randomness number provided by the auditors, the createRandomButton_Click function will be executed. In this function, for each seed input data a few test cases will be generated by changing a few bytes randomly. The number of test cases per seed data is determined by the randomness number. By clicking on the thorough test case generation check box the auditors can choose to generate test cases in a thorough way, which means that every byte will be modified sequentially. This is generally not feasible as it will result in too many test cases; however, for small input set it may be a viable solution.

5.1.4 Fuzzer

Again Filefuzz is utilized to fuzz the generated fuzzed data because the actual fuzzing in the approach is similar to traditional fuzzing. The real difference is in the fuzzed data generation part. With minor modifications it is safe to use Filefuzz fuzzing functionalities for the company's need. Basically, it allows the company to start the application under test with the fuzzed data as input repeatedly.

The application is given with a time limit. If it does not crash or throw any exceptions before this time limit, it is stopped by the fuzzer. In this case the set of input will be considered safe. The time limit here is a variable that can be set by the users, in this case, the auditors. This variable influences the overall performance of the fuzzer to a great extent. When set too short the application may not have enough time to fully execute the entire validation chain, thus leaving part of the application unchecked. However, if the time limit is set longer, each execution of the target system takes longer and eventually the whole execution time is increased dramatically. The fuzzer will go through all generated fuzzed data and execute them one by one.

The user interface shown in Figure 9 is again very similar to Filefuzz. The tab execute of Filefuzz is modified to allow different levels of fuzzing. While the execute button itself still has the same functionality as in Filefuzz, the newly added button "Random" triggers the function executeRandomButton_Click, which executes randomly generated test cases.

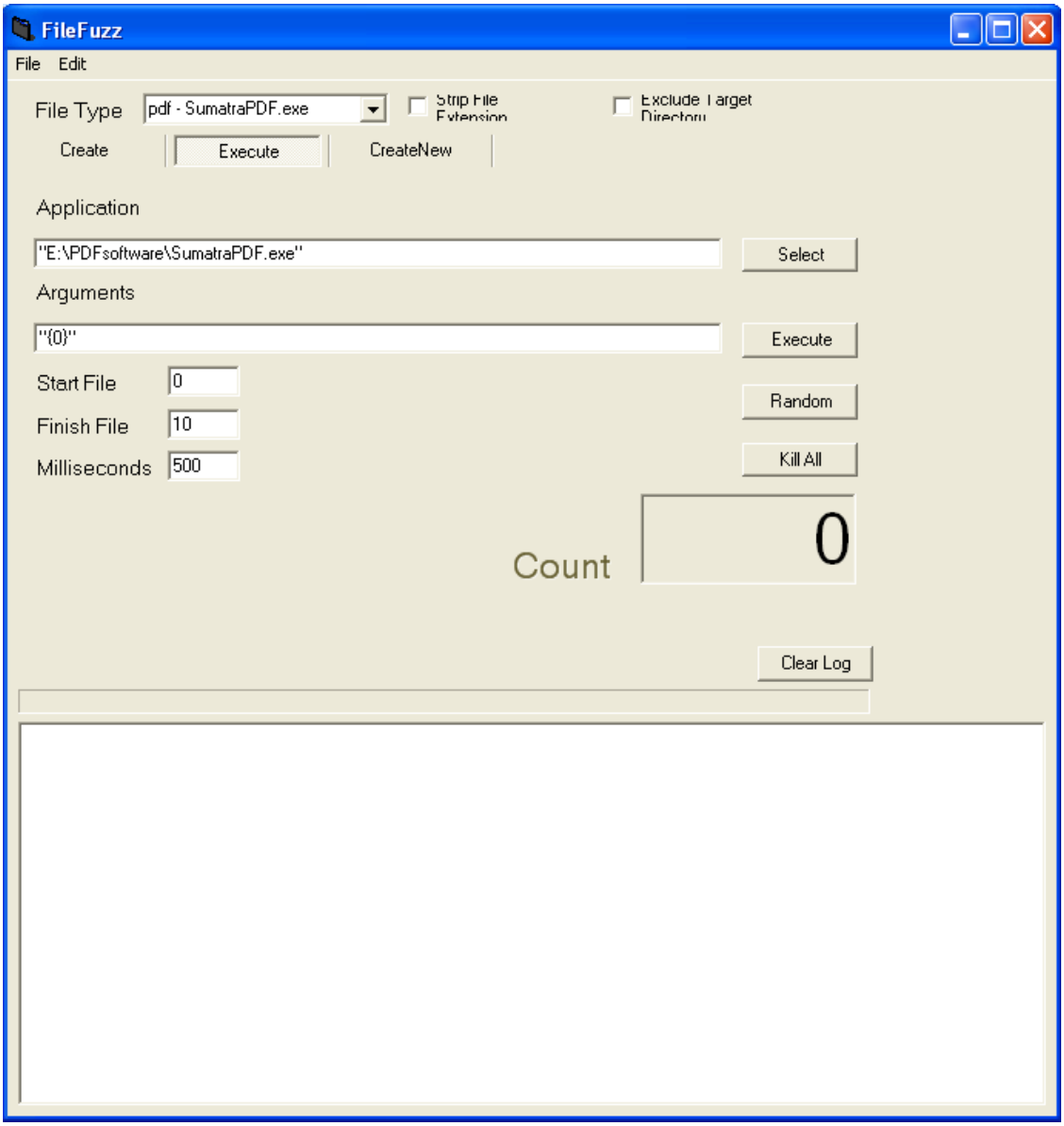

Figure 8: Execute Tab of Proof-of-Concept Tool

The other functions are described in Appendix A.

5.1.5 Exception Handler

The exception handler is responsible for tracing back to the point that causes the exceptions when they do happen. In order to do this the code that is being run at the time must be debugged. Filefuzz provides a simple class crash.c, which records exceptions when they happen. In the proof-of-concept tool the same class is used to track the exceptions. When an exception does occur, the input that causes the exception and the reason of the exception are recorded. This allows the auditors to run the inputs that cause exceptions again under debugger to further analyze them.

5.2 Extending Fuzzing

After the first phase of this project, it becomes apparent that traditional fuzzing alone is not enough to solve the problem at hand. The results from the initial tests show that traditional fuzzing is not effective and not able to find vulnerabilities as desired. Therefore, it is necessary to find a way to enhance traditional fuzzing. This leads to what I call the smart fuzzing approach, which makes traditional fuzzing smarter with genetic algorithm.

The implementation of smart fuzzing has a different goal from the implementation of traditional fuzzing. While the implementation of traditional fuzzing is intended as a proof-of-concept tool which is fully automated and can take on large real-life systems, the implementation of smart fuzzing is aimed at proving that genetic algorithm can indeed help generate better test cases. This makes target selection of implementation of smart fuzzing completely different. In fact, the whole nature of the implementation is different.

5.2.1 Target Selection

During the first phase of the project, the aim was implement an automatic tool to take on a real-life program because that is the initial requirement. However, after taking a few tests with real-life programs it became clear that it is not practical to perform fundamental researches with real-life programs due to the fact that it is often unknown whether or not there are vulnerabilities in them. This makes finding vulnerabilities similar to shooting blind. For this reason, in the second phase the implementation is focused on the targets that are constructed specifically for testing purpose. In this way the tests are performed with detailed information about where the vulnerabilities are and which types they are.

Because of the way genetic algorithm works, each type of vulnerabilities is targeted separately. Different initialization strategies, fitness function and selection strategies are applied depending on the type of vulnerabilities that is being targeted. Hence, to test the effectiveness of smart fuzzing, a group of c programs that contain different types of vulnerabilities needs to be constructed. However, as explained before, there are many types of vulnerabilities and it is too much work to incorporate them all at this stage. It is hence needed to choose one type of vulnerabilities and show the effectiveness of smart fuzzing on it first.

Buffer overflow is chosen as the main target type of vulnerabilities because it is the most representative type. Furthermore, format string and double free vulnerabilities are also incorporated. This is done for two reasons. First, it shows how additional vulnerabilities can be implemented; secondly, it allows comparison between different types of vulnerabilities.

5.2.2 Overall Structure

The difference between smart fuzzing and traditional fuzzing lies in the test case generation phase. Therefore, the proof-of-concept tool of smart fuzzing is focused on the implementation of test case generation using genetic algorithm. Its other parts are basically the same as traditional fuzzing. It is not meaningful to implement them once again. Therefore, only the test case generation part needs to be implemented.

The test case generation is carried out in a few components. First, the test targets are run with randomly generated test data using a coverage tool. The coverage information is then used to find those parts of the test targets which are not reached by randomly generated test cases. Annotations are then added at these parts. Then, genetic algorithm is run. The focus of the implementation here lies in the genetic algorithm part. The genetic algorithm should be able to

produce a set of input data. These input data are fed to the target program. If successful, they should be able to reveal vulnerabilities which are built in beforehand.

5.2.3 PGAPack

The genetic algorithm is implemented with the help of a GA library PGAPack [31]. PGAPack stands for Parallel Genetic Algorithm Pack. It is a general-purpose, data-structure-neutral, parallel genetic algorithm library intended to provide most capabilities desired in a genetic algorithm library, in an integrated, seamless and portable manner.

The PGAPack library is written and callable from C which is the main target of this project. Furthermore, the library runs on uniprocessors, parallel computers and workstation networks, which makes the proof-of-concept tool easily extensible to target programs involving network. The PGAPack supports binary, integer, real and character valued native data types as well as full extensibility to support custom data types. It also provides multiple cross-over, mutation and selection operators. These properties make the PGAPack helpful for testing the smart fuzzing approach.

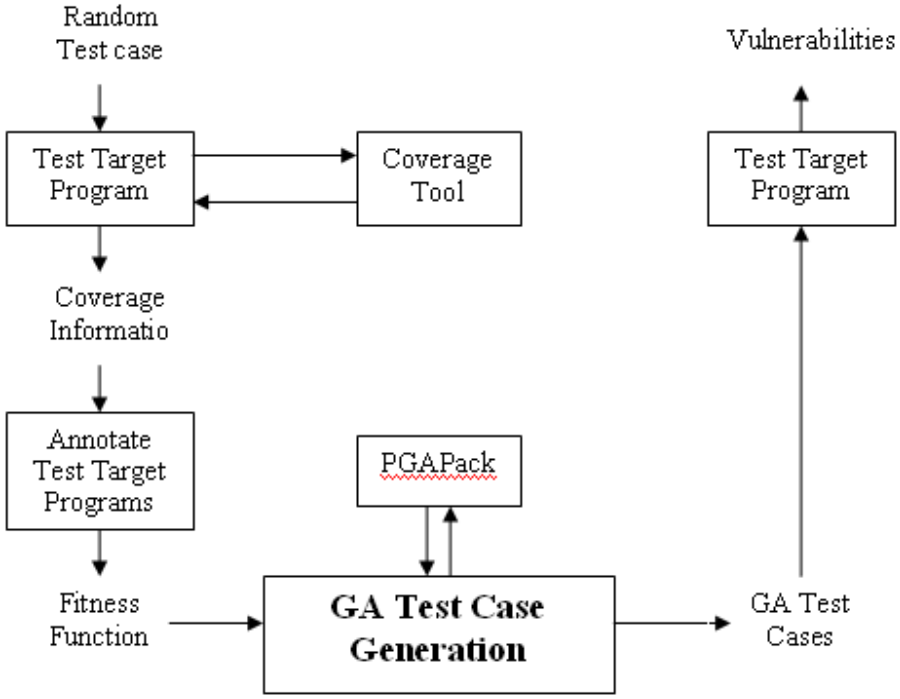

Figure 9: Overview of GA Test Case Generation

5.2.4 GA Test Case Generation

GA test cases are generated using the PGAPack library. The genetic algorithm test case generation uses annotated test target programs as fitness function. Test target programs are programs with specific vulnerabilities built in. The set of test target programs is build by the author for testing purpose alone. More details about these target programs are available in the next chapter. Running these test target programs with randomly generated test cases under coverage tool produces coverage information which tells which part of the target program is covered. Annotations are placed at those places that have not been covered, which in turn produces the fitness function needed by the genetic algorithm. The genetic algorithm is run with

the help of the PGAPack. The result is a group of test cases which should be able to reach the annotated source code. To determine if the test cases can indeed reveal the built-in vulnerabilities, the test cases are fed into the test target programs again. A complete overview of the system can be seen above.

The GA test case generation starts with the initialization of the first generation. As explained before, this is done randomly to cover a wide range of inputs. To do this, the PGAPack function, PGACreate, is used. It initializes the PGAContext variable, which contains the information needed to run the genetic algorithm. In this function the users should define the data type of the individuals, the population size and whether to search for a maximum or a minimum. After the PGAContext has been initialized, the users can choose to use the PGASetUp function to set the default values for the genetic algorithm or use the PGASet functions to set more specific settings such as the number of individuals in a population, which cross-over algorithm to use, and whether to use mutation, *etc*. Then, the PGARun function is called to actually start the genetic algorithm. In this function the users also have to provide a fitness function. The fitness function is derived from the test target programs. Annotations are put in the places where the coverage test has shown that random testing does not reach. Therefore, the fitness function will be different for each target program. In the end, the PGADestroy function is called to destroy the PGAContext variable. The last generation of the genetic algorithm will be stored in a file, which can be used to generate test cases. More details on the functions used can be found in Appendix A.

Chapter 6

6 Evaluation

After the completion of the tool, evaluating the tool is the next phase. Testing is part of the functionalities of fuzzing. Therefore, only running the test cases will not show whether the fuzzer is working correctly or not. In fact, it is more useful to see a test case fail because that will yield more information. In order to evaluate the implemented proof-of-concept tool, the first step is to select a suitable target program. After a target program is selected, a test environment can be set. The tests are then performed as described in the test plan and the results recorded.

This chapter starts with the target selection section, explaining how the target program is selected or constructed. Then, the test plan is presented, which gives more detailed information about how the tests are set. This is followed by the test results which lay the base for the conclusion of the evaluation of the smart fuzzing approach. This chapter ends with an evaluation of the business value of the conducted research.

6.1 Target Selection

As mentioned before, the implementation is done in two steps. First, a proof-of-concept tool of traditional fuzzing is implemented. When it becomes apparent that traditional fuzzing alone is not sufficient, the smart fuzzing approach is introduced which uses genetic algorithm to aid test case generation of fuzzing.

The implementation of traditional fuzzing is tested with a real-life program, an open source PDF program called Sumatra PDF. This choice is made because the company is concerned that the new approach must be able to take on real-life programs. In fact, this is one of the most important criteria because earlier attempts have been proven successful on small scale tests but ineffective against larger target programs. Furthermore, open source programs are chosen as test targets because both their executable and source code are commonly available.

However, this choice also brings trouble with itself. Because the target programs are large open source, it is impossible to know whether or not they contain vulnerabilities. This means that there may not even be vulnerabilities in the target programs; hence, there is nothing to verify the test results with and the results from tests are thus not useful. The only thing that does become apparent is that a large portion of the target programs is not covered during testing, which leads to the initial idea of using genetic algorithm to aid test case generation in smart fuzzing. The execution path needs to be able to reach the vulnerable code before it can actually trigger the vulnerability.

To correct the issue of the implementation of traditional fuzzing, the smart fuzzing approach is tested in a more controlled environment. The test cases are built by the author; hence, both the types and the places of all the vulnerabilities are known. This means that each individual test case is of a smaller scale. However, this should not be a problem because the fuzzing approach has already been proven usable on large target programs by the first series of tests. The smart fuzzing approach essentially only modifies the test case generation part of traditional fuzzing. The fact that test cases are made by the author makes it easier to track them down. When a test is run, whether or not the vulnerabilities are found immediately becomes apparent.

6.2 Test Plan

The test phase is carried out in two separate steps. The traditional fuzzing approach is first tested followed by the testing of the smart fuzzing approach. As explained before, the two approaches are tested with entirely different mindsets. Thus, there are two separated test plans for the two approaches. In this section how these two test plans are set and executed are thus also separately explained.

6.2.1 Traditional Fuzzing

The implementation of traditional fuzzing is mainly based on a well known fuzzer called Filefuzz. Therefore, the testing is not focused on the correctness of the implementation; instead, the focus lies on its ability to test large real-life target programs and the ease at which it can be used. To achieve this, the proof-of-concept tool is implemented in such a way that it is mostly automatic. A software security auditor with little prior knowledge of the test target should be able to run the tool. All he needs is the protocol of the input stream.

Because Sumatra PDF is chosen as the target program, the input stream in this case is the PDF format. A complete PDF format description can be found at [32]. Using the format given in this reference manual, an initial test case is generated, which is then further fuzzed by the tool to create more seed test cases and essentially all other test cases. Using the tool all parts of the input stream are fuzzed separately to create a larger coverage.

At first, the plan is to test more target programs. However, the result from the first test shows that only a very limited portion of the source code has been covered during the performed testing, which leads to further improvement of the approach. Therefore, the traditional fuzzing approach is only tested with one target program. Although this may seem insufficient, it already gives a good view of the strengths and weaknesses of the traditional fuzzing approach. Performing more tests with other similar target programs may cost too much time and most likely not yield any other results partly due to the fact that, as mentioned before, the tests performed are not in a controlled environment.

6.2.2 Smart Fuzzing

The smart fuzzing approach is tested with three types of vulnerabilities: buffer overflow, format string, and double free. The main goal here is to test one type of vulnerabilities as an initial step. This is because different vulnerabilities will require different search requirements, and to cover all types of vulnerabilities is outside the scope of this project. Buffer overflow vulnerability is chosen as the main testing target type because it is the most common vulnerability. Format string and double free vulnerabilities are added to show as an example how other types of vulnerabilities can be added. This also allows for comparison between the effectiveness of smart fuzzing on buffer overflow and its effectiveness on other vulnerabilities.

To test the effectiveness of smart fuzzing on buffer overflow, format string and double free vulnerabilities, three groups of test cases containing each of these vulnerabilities, respectively, are built. In each of these groups, subgroups are made to differentiate between test cases that contain complex execution paths and test cases that contain fairly simple execution paths. Although the built-in vulnerabilities remain the same, the vulnerabilities that are built into the subgroups that contain complex execution paths should be harder to find and trigger than the vulnerabilities that are built into the subgroups that contain simpler execution paths.

The test cases from the simple groups mostly contain only one test condition. This is the most elementary form of any control flow graphs. They are intended to show the effectiveness of

genetic algorithm on the most basic control flows. All conditions are included in the test groups. Also, the else branches are included to illustrate the effectiveness of genetic algorithm on alternative branches. The "and" and "or" relations are also included in their most basic form. A few test cases from the simple groups are shown as the examples in Table 5. The variable "str" here is the input stream. When its size is larger than the size of the receiving buffer, which is 8 in this case, a buffer overflow will occur.

Elementary Test Case	Else Branch	"and or" Relation

char buffer[8];
char a = str[0];
if(a > 'b')
  strcpy(buffer, str);
else
  a = 'b';
	char buffer[8];
	char a = str[0];
	if(a > 'b')
	  a = 'b'
	else
	  strcpy(buffer, str);
		char buffer[8];
		char a = str[0];
		if(a > 'b' && a < 'd')
		  strcpy(buffer, str);
		else
		  a = 'b';

Table 5: Simple Test Case Examples

In the complex test groups, the elementary test cases are expanded into more complex control graphs by adding additional conditions and branches. First, a set of two branches are tested, followed by even more branches. Because the first group of test cases shows that the else branches are essentially the same as the elementary test cases, no else branch test cases are included in complex test groups. Instead, more branches are added. The most complex test cases in the complex groups have nine conditions in them. One example of a complex test case is given below. Again the variable str is the input stream. Because str[6] equals '%' if str[7] is any of the format strings such as 's', the printf() will crash the target program.

Complex Test Cases

a = str[0];
b = str[1];
c = str[2];
d = str[3]);
e = str[4];
f = str[5];
if(f == 'a'){
  if(e >= 'd'){
  if(d < 'd'){
  if(c <= 'a' || c >= 'x'){
  if(b == 'a'){
  if((a > 'd' || a < 'b') && a == 'z'){
  str[6] = '%';
  printf(str);
  }
  }
  }
  }
  }
}

Table 6: Complex Test Case Examples

After the test cases are built, the actual tests are performed in three steps. In the first step the test cases are run with randomly generated test cases under coverage tool. The coverage information is then extracted. Normally, in the second step the genetic algorithm is run with the suspicious

uncovered code sections as targets. However, in this case, the vulnerabilities are built in, so that the positions of the built-in vulnerabilities are known beforehand. Hence, these positions are used as target sections. The third step is the actual fuzzing again. The results from the second step are used as inputs. This means that each individual test case will have a group of inputs which is the result of the genetic algorithm run in the previous step.

6.3 Test Results

As explained earlier, the testing of traditional fuzzing is only done on one target program and is not performed in a controlled environment. This makes it hard to verify the test results. However, the tests still show some informative results. The most important result that can be derived from the performed tests is that only a very small portion of the source code is actually covered by the test cases.

The smart fuzzing approach is tested in a more controlled manner; all test cases are generated by the author. This makes it possible to compare the results with the actual vulnerabilities that are built in. In general, the results show that genetic algorithm is indeed a viable way of generating test cases. Not only are the generated test cases capable of reaching the built-in vulnerabilities, they are also able to trigger the vulnerabilities. More detailed test results are given in the next subsection.

6.3.1 Genetic Algorithm Test Results

The table below shows the test results from the smart fuzzing test cases. The six subgroups are listed in the most left column. The second column shows how many of the twenty test cases in the subgroups have passed random testing, and the third column shows how many of the test cases have passed smart fuzzing. Passing a test in this context means that the build-in vulnerabilities are found correctly.

Subgroups	Random Testing	Smart Fuzzing
Buffer Overflow Simple	10/20	14/20
Buffer Overflow Complex	2/20	14/20
Double Free Simple	10/20	16/20
Double Free Complex	2/20	16/20
Format String Simple	10/20	16/20
Format String Complex	2/20	16/20

Table 7: Genetic Algorithm Test Results

From this table, it becomes apparent how ineffective random testing actually is. Only a few of the built-in vulnerabilities are discovered by random testing. Even when the execution path is fairly simple, the chance of random testing reaching the vulnerable code section is still too small. However, it must be noted that random testing is only performed once in this test. In general, there should be several rounds of random testing as it requires a little to no effort to create and run additional test cases. A successful random testing will probably require many rounds and a huge amount of test cases. Furthermore, it is clear that when test cases become complex, the chance of random testing finding vulnerabilities diminishes drastically.

Furthermore, it also shows that the smart fuzzing approach yields much better results. In all subgroups more than 50% of the built-in vulnerabilities are found. This shows that the smart fuzzing approach is effective in finding vulnerabilities, especially double free and format string vulnerabilities where nearly all the built-in vulnerabilities are found. This is most likely because with double free vulnerabilities, the execution path only needs to reach the vulnerable code to

trigger the vulnerabilities, unlike other vulnerabilities where additional requirements must be fulfilled to actually trigger the vulnerabilities even when the execution path has reached the vulnerable code.

This means that additional search goals must be achieved to reveal the vulnerabilities. It is however not always possible to construct these search goals. Buffer overflow vulnerability requires that the variable being used as input buffer must be larger than the output buffer expecting it. This means that the vulnerability can only be found when the input buffer is large enough, which is not controlled by genetic algorithm. Similar situation occurs for format string vulnerability. Here, in addition to reaching the vulnerable code, the format string identifiers such as %s, %c and %n must also be present to trigger the vulnerability. Although in some cases these can be added as the search goals of genetic algorithm, it is not always possible to do so.

The test cases that fail to reveal the built-in vulnerabilities are often large and have many branches. This makes reaching the last branch where the genetic algorithm fitness function is implemented harder. This may be improved by adding more genetic algorithm search goals along the execution path. Because the fitness value is a positive number, it can be accumulated, and running the genetic algorithm with the goal of finding a maximum fitness value will result in the genetic algorithm going to the vulnerable code, accumulating the fitness values along the intended execution path. However, this requires the extraction of the control flow graph, which will certainly increase the complexity of the approach significantly and can hence be counter-effective. Nevertheless, for larger target programs it will likely be needed as they usually contain very complex and large control flow graphs.

6.4 Business Value

This section discusses the business value of the conducted research. First, the research result will be related back to the research question. Then, the extent to which the research question has been solved will be discussed. Based on this, the logical next step and tasks to be done in the follow-up project will also be suggested. In the end, an estimation of the time needed to implement an integrated proof-of-concept tool is given.

The research question of this project was primarily defined as “how to use and extend current input validation evaluation techniques and tools to aid white-box software security inspection?” After the literature research, the scope of the project has been defined in more detail, directing the focus onto fuzzing, which is a promising software security inspection approach. Another more concrete goal is to focus on input validation evaluation, instead of entire white-box software security inspection. The research question at that point was rephrased as “how to use and adapt software security inspection approach fuzzing to aid input validation evaluation?” However, after the initial research, the implementation of a proof-of-concept of the fuzzing approach showed that fuzzing alone was insufficient, which led to the need of further research on other helpful algorithms. The outcome was genetic algorithm. While the addition of genetic algorithm seems to be outside of the scope of the later defined research question, it is in line with the primary research question. The additional research on genetic algorithm is considered essential for producing a more meaningful and valuable research result for the company.

The research result of genetic algorithm shows that genetic algorithm can be used as an extension to fuzzing. The test cases generated by genetic algorithm have been proven effective in finding certain types of input validation vulnerabilities. Although ideally all types of vulnerabilities should be covered and an integrated proof-of-concept tool should be implemented, it is

considered infeasible to include every aspect given the constraints of this master thesis project. The research has shown that it is possible to use genetic algorithm to produce test cases which can reveal input validation vulnerabilities and that it is possible to generate and run test cases automatically using fuzzing.

One of the most important requirements of the company from a business perspective was that the researched approach must be able to take on large target programs without becoming too complex to use. While it was not possible to show this aspect in an integrated way, the research has shown that the complexities of both fuzzing and genetic algorithm are not directly related to the size of target programs. This means that while the complexities of fuzzing and genetic algorithm increase as a result of the increase in size of the target program, this increase of the complexities is relatively insignificant. Hence, the proposed approach should be applicable to meet the business need of the company in this aspect. Another business goal of the research, which was to find an approach that can serve as a first-round check, has also been fulfilled because through this approach auditors only need the knowledge of fuzzing and genetic algorithm and a little introduction of the target program to carry out a first-round check of the target program. As explained in the Implementation chapter, the use of fuzzing and genetic algorithm ensures that very little prior knowledge of the target program is needed.

The research result of this thesis should give a baseline to conduct further researches in this direction and provide the necessary information for follow-up projects. In chapter 7 Conclusion and Future Work, a few set-ups of follow-up projects have been suggested. To bring the research on "how to use genetic algorithm as an extension of fuzzing to aid input validation evaluation" into the next phase, the researcher will require insightful knowledge of fuzzing [24] and genetic algorithm [31] as well as practical experience of these two approaches. Hands-on knowledge of implementation of one or more types of fuzzing will also be helpful. Moreover, this thesis can be used as a guide to implement fuzzing and genetic algorithm. However, the integration of the two will need to be further worked out during the next phase of the research.

Following the ways described in this thesis, the efforts required in building an integrated proof-of-concept tool should take no more than four-month time. This is based on the time needed to conduct the current research. In this one-year research project three months were spent on the literature research to create a knowledge base from which the focus point was chosen. It was followed by a four-month theoretical research on fuzzing to create a theoretical foundation of the approach and a two-month phase of building a proof-of-concept of fuzzing. In the end, another two months were spent on theoretical research of genetic algorithm, followed by a one-month testing on the viability of using genetic algorithm to enhance fuzzing. A follow-up project with the goal of creating an integrated proof-of-concept will not involve the work such as literature research and theoretical research. However, at least one-month time will be needed to read through this thesis and other literatures to get familiar with the topic. Assuming the implementation of an integrated proof-of-concept in the follow-up project will be done with the same efficiency as in this project, the actual implementation should take around three-month time. Hence, the total project time should be no longer than four months.

Chapter 7

7 Conclusion and Future Work

Based on the test results, this chapter concludes the research performed in this project. As the results clearly indicate, the smart fuzzing approach is effective against at least three types of input validation vulnerabilities. Because this research is done as an initial step of using genetic algorithm to aid test case generation of traditional fuzzing to solve input validation evaluation problems, there are still a great deal of improvements to be made. Therefore, the section Future Work sums up a few interesting research directions that are closely related to this project or can be performed as follow-up projects.

7.1 Conclusion

In general, from the test results it can be concluded that the smart fuzzing approach is effective against at least three types of vulnerabilities. In most cases, genetic algorithm is capable of finding a set of inputs that can reach the vulnerabilities. The only problem is that sometimes reaching vulnerabilities alone is not sufficient in triggering them well. Furthermore, the effectiveness, efficiency and coverage of the smart fuzzing approach are all interesting aspects that should be analyzed. Each of these aspects is further examined in more details in the sections below.

7.1.1 Effectiveness

The test results show that genetic algorithm is indeed a powerful search algorithm. Applied to the test case generation of fuzzing, genetic algorithm is able to generate a group of inputs that are all capable of reaching vulnerable statements. This makes the smart fuzzing approach highly effective. Also, because of the nature of testing, there are no false positives. Each crash indicates that there are certain vulnerabilities in the system. Although they may not always be exploitable, they are certainly not working correctly. This is under the assumption that a correctly working system should not crash under no circumstances.

While the smart fuzzing approach retains the strength of traditional fuzzing, which makes it capable of taking on larger target programs at the same time, it also improves the weakness of traditional fuzzing, the effectiveness, greatly by introducing genetic algorithm as the test case generation part. The test cases generated by genetic algorithm are much more effective because most of them can reach the vulnerable code. Therefore, almost all test cases are right on or at least very close to the spot. Hence, instead of having a huge amount of test cases that are scattered around the search space, the test cases generated by genetic algorithm all focus on the actual vulnerabilities.

7.1.2 Efficiency

Although using genetic algorithm does improve the effectiveness of traditional fuzzing significantly, it also comes with more cost for computing time and power. In the time that genetic algorithm takes to run one round, random testing would have generated much more test cases. However, test cases generated by random testing are less focused and less effective. Therefore, eventually when taking both the time spent and the number of vulnerabilities discovered into account, it can be concluded that the smart fuzzing approach is much more efficient than traditional fuzzing as more vulnerabilities are found with less time spent.

7.1.3 Coverage

One of the research goals of this project is that the new approach must be able to show coverage information. Not only will it be interesting to see which part of the source code is being covered, it will be even more interesting to see which types of vulnerabilities are being covered.

This is partially true for the smart fuzzing approach because different types of vulnerabilities are being targeted separately. When, for example, buffer overflow is being targeted, if no vulnerabilities are found after the completion of smart fuzzing, it is practically safe to say that there are no buffer overflow vulnerabilities in the target program. Even though there is theoretically still a chance that the targeted vulnerability is not found, it is leastwise certain that the vulnerability has been reached by the test inputs and has been tested several times. More thorough testing will eventually reveal the vulnerability. Especially in the second phase when the suspicious code fragment is being used as the search goal of genetic algorithm, when the test cases are generated and run, even if no vulnerabilities are found, it is certain that the suspicious code fragment has been reached and tested.

7.2 Future Work

In this project attempts were made to solve the input validation evaluation problem by using fuzzing. When it became apparent that traditional fuzzing alone was not sufficient to solve the problem and that the test case generation phase was essentially a search problem, genetic algorithm was added to aid fuzzing. Genetic algorithm was chosen because it was by far the most effective heuristic search algorithm. However, other types of heuristic search algorithms can also be effective, or perhaps even more effective. Furthermore, there are many control variables in genetic algorithm that can be set. Sub-section 4.4.5 shows a list of these control variables. The tests performed in this project obviously only used one set of control variable values. How effective other control variable values are still remains unknown. Another research direction can be to add other types of vulnerabilities as targets. Also, only the test case generation part of the smart fuzzing approach was tested in the project. Although the effectiveness of fuzzing in general was tested as well, it was done in a separate test. Hence, it can be meaningful to build a complete working prototype to prove the effectiveness of smart fuzzing. Next, each of these research directions is explained in more details.

7.2.1 Other Types of Heuristic Search Algorithms

The test case generation part of fuzzing is essentially a search problem. The goal is to find those inputs that can reach and trigger vulnerabilities. Although genetic algorithm has been proven to be one of the most effective heuristic search algorithms, there are many other search algorithms available. For example, heuristic algorithms such as hill climbing and local search could be effective in generating desired inputs as well. These algorithms might be even more efficient as they require less computing power in general. In fact, as long as the required inputs can be found, a less powerful but more efficient search algorithm might be more suitable. Therefore, a research should be done to look into how different heuristic search algorithms suit test case generation of input validation evaluation problem.

7.2.2 Further Fine-Tuning Genetic Algorithm

As shown in Section 4.4.5, there are many control variables available. The tests performed in this project only used one possible value of each of these control variables. Although the author did try to fine tune the control variables a bit, other values may still be able to yield better results.

Increasing the number of solutions per generation, for example, can be helpful in finding more test cases; however, this also means that more time is required to run the genetic algorithm, making it less efficient. More cross-over points may lead to faster evolution; however, having too many cross-over points can lose the fitness characteristics too fast. In general, a balance must be found for each of these control variables under different situations. Although finding these balances would require a huge amount of testing, the result could help the auditors that use the smart fuzzing approach to decide which values they should set their control variables to under their specific situations.

7.2.3 Additional Types of Vulnerabilities

Because of the limited time and resources available, this project only included one type of vulnerabilities as the main type of test target and in total three types of vulnerabilities. As shown in Section 4.5 there are many other types of vulnerabilities. How effective smart fuzzing is on other types of vulnerabilities remains to be proven. Vulnerabilities such as integer overflow and race condition are often more complex and cannot be easily traced back to one certain statement. This makes it harder for genetic algorithm to find as multiple statements would have to be reached by the execution path. This could be resolved by adding more search goals. However, how exactly each of these types of vulnerabilities should be handled still requires additional research. Furthermore, additional types of vulnerabilities would make the smart fuzzing approach more complete.

7.2.4 Complete Working Prototype

The smart fuzzing approach at its current state is implemented in two separated parts. The genetic algorithm test case generator is implemented separately. Hence, though the implementation did cover all steps of the smart fuzzing approach, it was not implemented in one whole working prototype. The true power of the approaches such as fuzzing lays in the fact that it can be entirely automatic and requires a little to no prior knowledge of the target program. However, this can only be proven when a complete working prototype is built. Therefore, building a complete working prototype would be an important step for smart fuzzing research.

Abbreviation

CC	Common Criteria
CFG	Control Flow Graph
CMC	Controlled Markov Chains
EAI	Environment-Application Interaction
Fortify SCA	Fortify Source Code Analyzer
GA	Genetic Algorithm
IVAT	Input Validation Analysis and Test
JABA	Java Architecture for Byte Code Analysis
MICASA	Method for Input Cases and Static Analysis
NLNCSA	Dutch National Communication Security Agency
PDF	Portable Document Format
PGAPack	Parallel Genetic Algorithm Library
SF	Smart Fuzzing
TIVUM	Tool for Input Validation Understanding and Maintenance
VFG	Validation Flow Graph
V&V	Verification and Validation

Reference

1. *Common Criteria for Information Technology Security Version 3.1.* 2006.
2. Lampson, B.W., *Computer security in the real world.* Computer, 2004. **Volume: 37**(Issue: 6): p. 37- 46.
3. Lampson, B.W., *Practical Principles for Computer Security.* research.microsoft.com, 2006.
4. Gray, A., *An historical perspective of software vulnerability management.* Information Security Technical Report, 2003.
5. Bazaz, A., *A Framework for Deriving Verification and Validation Strategies to Assess Software Security*, in *the faculty of Virginia Polytechnic*, Institute and State University
6. McGraw, B.C.a.G., *Static Analysis for Security*, ed. E.b.G. McGraw: Published by the IEEE Computer Society.
7. Pravir Chandra, B.C., John Steven, *Putting the Tools to Work: How to Succeed with Source Code Analysis*, ed. E.b.J.S.a.G. Peterson: Published by the IEEE Computer Society.
8. Jane Huffman Hayes, A.J.O., *Input validation testing: A Requirements-Driven, System Level, Early Lifecycle Technique.* 2000.
9. Beizer, B., *Software Testing Technique*. 2nd edition ed. 1990, New York: Van Nostrand Reinhold Inc.
10. Horgan J. R., L.S., *A data flow converage testing tool for C.* Proceeding of the Symposium of Quality Software Development Tools.
11. Jefferson, O., *Investigation of the Software Testing Coupling Effect.* ACM Transaction on Software Engineering and Methodology.
12. Hui Liu, H.B.K.T., *An Approach to Aid the Understanding and Maintenance of Input Validation.* Proceeding of the 22nd IEEE International Conference on Software Maintenance.
13. Hee Beng Tan, J.T.K., *An approach for extracting code fragements that implements functionality from source programs.* Journal of Software Maintenance and Evolution, 2001. **13**: p. 53-75.
14. Weiser, M., *Program slicing.* presented at 5th International Conference on Software Engineering, 1981.
15. Martin Boldt, B.C., Roy Martinsson, *Software Vulnerability Assessment Version Extraction and Verification.* International conference on Software Engineering Advances, 2007.

16. Michael Gegick, L.W., *Matching Attack Patterns to Security Vulnerabilities in Software-Intensive System Designs.* Proceedings of the 2005 workshop on Software engineering, 2005.
17. Arkin, B.S., S. McGraw, G., *Software penetration testing.* IEEE SECURITY & PRIVACY 2005
18. Jane Huffman Hayes, A.J.O., *Input Validation Analysis and Testing.* Empirical Software Engineering, 2006.
19. Gregory T. Buehrer, B.W.W., Paolo A. G.. Sivilotti, *Using Parse Tree Validation to Prevent SQL Injection Attacks.* Proceedings of the 5th international workshop on Software, 2005.
20. Hui Liu, H.B.K.T., *Automated Verification and Test Case Generation for Input Validation.* 2006 international workshop on Automation of software test, 2006.
21. Wen Liang Du, A.P.M., *Testing For Software Vulnerability Using Environmental Perturbation.* Quality and Reliability Engineering International, 2002: p. 261-272
22. D. A. Linkens, M.Y.C., *Input selection and partition validation for fuzzy modeling using neural network.* Fuzzy Sets and Systems 1999: p. 299-308.
23. Cai, K.Y., *Optimal software testing and adaptive software testing in the context of software cybernetics.* Information and Software Technology, 2002: p. 841-855.
24. Michael Sutton, A.G., Pedram Amini, *Fuzzing Brute Force Vulnerability Discovery*. 2007: Pearson Education. 543.
25. McMinn, P., *Search-Based Software Test Data Generation: A Survey.* Journal on software testing, verification and reliability, 2004. **Vol. 14**(No. 2): p. 105-156.
26. Nigel Tracey, J.C., Keith Mander, *Automated program flaw finding using simulated annealing.* International Symposium on Software Testing and Analysis
Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis, 1998: p. 73-81.
27. Nigel Tracey, J.C., Keith Mander, *The Way Forward for Unifying Dynamic Test Case Generation: The Optimisation-Based Approach.* In Dependable Computing and Its Applications- To appear. IFIP, 1998.
28. Michael Howard, D.L., John Viega, *19 Deadly Sins of Software Security*. 2005: McGrawHill Osbourne Media.
29. *Common Weakness Enumeration*. [cited; Available from: http://cwe.mitre.org/data/slices/2000.html.

30. Sutton, M. *iDefense Labs*. 2008 [cited; Available from: http://labs.idefense.com/software/fuzzing.php.
31. Levine, D. *PGAPack Paralle Genetic Algorithm Library*. 2000 [cited; Available from: http://www-fp.mcs.anl.gov/CCST/research/reports_pre1998/comp_bio/stalk/pgapack.html.
32. Incorporated, A.S., *Portable Document Format Reference Manual*. 1999.

Appendix A

7.3 Traditional Fuzzing Functions

7.3.1 createSeedButtonClick

```
private void createSeedButton_Click(object sender, EventArgs e)
        {
                        Read readFile = new Read(tbxSourceFile.Text);
            if (readFile.readAscii() == false)        //End if source file not found
                return;

                        string targetDirectory = tbxTargetDirectory.Text;
                        string fileExtension =
                        tbxSourceFile.Text.Substring(tbxSourceFile.Text.LastIndexOf("."));
                        int fileNumber = Convert.ToInt32(fileNumberText.Text);

            if (replaceSelect.Checked == true)
            {
                if (oldText.Text == "")
                {
                    rtbLog.AppendText("Please fill in what you want to replace.\n");
                    return;
                }
                string ot = oldText.Text;
                string nt = newText1.Text;
                fileNumber = replaceTextFunction(readFile, ot, nt, targetDirectory, fileExtension,
                             fileNumber);
                try
                {
                    int times = Convert.ToInt32(timesText.Text);
                    if (times > 0)
                    {
                        nt = "";
                        for (int count = 0; count < times; count++)
                        {
                            nt += newText2.Text;
                        }
                        fileNumber = replaceTextFunction(readFile, ot, nt, targetDirectory, fileExtension,
                                     fileNumber);
                    }
                }
                catch
                {
                    //rtbLog.AppendText("number of times can not be converted.\n");
                }
            }
            else if (addSelect.Checked == true)
            {
                string toAdd = addText.Text;
                string beforeAdd = beforeAddText.Text;
                string afterAdd = afterAddText.Text;
                if (toAdd == "")
```

```
    {
        rtbLog.AppendText("Please fill in which chars you want to add.\n");
        return;
    }
    else if (beforeAdd == "" && afterAdd == "")
    {
        rtbLog.AppendText("Please specify the location where you want to add the new text.\n");
        return;
    }
    else if (beforeAdd == "")
    {
        string ot = afterAdd;
        string nt = afterAdd + toAdd;
        fileNumber = replaceTextFunction(readFile, ot, nt, targetDirectory, fileExtension,
                     fileNumber);
    }
    else if (afterAdd == "")
    {
        string ot = beforeAdd;
        string nt = toAdd + beforeAdd;
        fileNumber = replaceTextFunction(readFile, ot, nt, targetDirectory, fileExtension,
                     fileNumber);
    }
    else
    {
        string ot = afterAdd + beforeAdd;
        string nt = afterAdd + toAdd + beforeAdd;
        fileNumber = replaceTextFunction(readFile, ot, nt, targetDirectory, fileExtension,
                     fileNumber);
    }
}
else if (deleteSelect.Checked == true)
{
    string toDelete = deleteText.Text;
    string beforeDelete = beforeDeleteText.Text;
    string afterDelete = afterDeleteText.Text;
    if (toDelete == "")
    {
        rtbLog.AppendText("Please fill in which chars you want to delete.\n");
        return;
    }
    else if (beforeDelete == "" && afterDelete == "")
    {
        string ot = toDelete;
        string nt = "";
        fileNumber = replaceTextFunction(readFile, ot, nt, targetDirectory, fileExtension,
                     fileNumber);
    }
    else if (beforeDelete == "")
    {
        string ot = afterDelete + toDelete;
        string nt = afterDelete;
        fileNumber = replaceTextFunction(readFile, ot, nt, targetDirectory, fileExtension,
                     fileNumber);
    }
```

```
            else if (afterDelete == "")
            {
                string ot = toDelete + beforeDelete;
                string nt = beforeDelete;
                fileNumber = replaceTextFunction(readFile, ot, nt, targetDirectory, fileExtension,
                             fileNumber);
            }
            else
            {
                string ot = afterDelete + toDelete + beforeDelete;
                string nt = afterDelete + beforeDelete;
                fileNumber = replaceTextFunction(readFile, ot, nt, targetDirectory, fileExtension,
                             fileNumber);
            }
        }
        fileNumberText.Text = fileNumber + "";
        //rtbLog.Clear();
        rtbLog.AppendText(readFile.sourceString.Length.ToString());
        rtbLog.AppendText(" characters read.\n");
        rtbLog.AppendText((fileNumber).ToString());
        rtbLog.AppendText(" files written to disk.\n\n");
    }
```

7.3.2 *createRandomButton_Click*

```
private void createRandomButton_Click(object sender, EventArgs e)
    {
        Read readFile;
        Write writeFile;
        int fileTotal = Convert.ToInt32(fileNumberText.Text);
        int randomness = Convert.ToInt32(randomnessText.Text);
        string fileExtension = tbxSourceFile.Text.Substring(tbxSourceFile.Text.LastIndexOf("."));
        string sourceDirectory = "E:\\fuzzSource\\pdf\\";
        string targetDirectory = randomTargetText.Text;

        for (int fileNumber = 0; fileNumber < fileTotal; fileNumber++)
        {
            readFile = new Read(sourceDirectory + fileNumber + fileExtension);

            if (readFile.readBinary() == true)    //take next input if source not found
            {
                rtbLog.AppendText("Creating random test cases for seed: " + sourceDirectory +
                                  fileNumber + fileExtension + "\n");
                rtbLog.Update();

                for (int r = 0; r <= randomness; r++)
                {
                    writeFile = new Write(readFile.sourceArray, targetDirectory, fileExtension,
                                          fileNumber, r, fuzzLocations[fileNumber]);

                    if (ThoroughCheckBox.Checked == true)
                    {
                        writeFile.writeRandomThoroughByte();
                    }
```

```
                    else
                    {
                        writeFile.writeRandomByte();
                    }
                }
            }
            else
                rtbLog.AppendText("Could not find seed: " + sourceDirectory + fileNumber +
                                  fileExtension + "\n");
        }
        rtbLog.AppendText("Random test case generation finished. \n");
        rtbLog.Update();
    }
```

7.3.3 executeRandomButton_Click

```
private void executeRandomButton_Click(object sender, EventArgs e)
    {
        int startFile = Convert.ToInt32(tbxStartFile.Text);
        int finishFile = Convert.ToInt32(tbxFinishFile.Text);
        int randomness = Convert.ToInt32(randomnessText.Text);
        string targetDirectory = randomTargetText.Text;
        string fileExtension;
        if (cbxStripFileExt.Checked == true)
            fileExtension = null;
        else
            fileExtension = tbxSourceFile.Text.Substring(tbxSourceFile.Text.LastIndexOf("."));
        int killTimer;
        string applicationName = tbxExecuteApp.Text;
        string applicationArguments = tbxExecuteArgs.Text;

        try
        {
            FileInfo[] targetFiles = null;
            DirectoryInfo targetDirectoryInfo = new DirectoryInfo(tbxTargetDirectory.Text);
            targetFiles = targetDirectoryInfo.GetFiles();
        }
        catch (System.IO.DirectoryNotFoundException ex)
        {
            MessageBox.Show(ex.Message, "Error - Directory not found");
            return;
        }

        try
        {
            killTimer = Convert.ToInt32(tbxMilliseconds.Text);
        }
        catch (System.FormatException ex)
        {
            MessageBox.Show(ex.Message, "Error - Invalid format for milliseconds");
            return;
        }
```

```
            Execute executeApplication = null;

            if (cbxExcludeTargetDir.Checked == true)
                targetDirectory = null;

            executeApplication = new Execute(randomness, startFile, finishFile, targetDirectory,
                                    fileExtension, killTimer, applicationName, applicationArguments);
            executeApplication.pbrStart += new pbrProgressBarStart(pbrHandleStart);
            executeApplication.pbrUpdate += new pbrProgressBarUpdate(pbrHandleUpdate);
            executeApplication.tbxUpdate += new tbxCountUpdate(tbxHandleUpdate);
            executeApplication.rtbLog += new rtbLogOutput(rtbHandleLog);

            //Execute application in new thread
            Thread executeAppThread =
                new Thread(new ThreadStart(executeApplication.executeApp));
            executeAppThread.Start();
        }
```

7.3.4 replaceTextFunction

```
private int replaceTextFunction(Read readFile, string oldTextString, string newTextString, string
targetDirectory, string fileExtension, int fileNumber)
        {
            Write writeFile = null;
            Regex regex = new Regex(oldTextString);
                        int matches = regex.Matches(readFile.sourceString, 0).Count;
            rtbLog.AppendText(matches + " matches found.\n");

            if (matches == 0)
            {
                return fileNumber;
            }

            string replace = newTextString;
                        string source = readFile.sourceString;
                        int location = 0;

            for (int count = 1; count <= matches; count++)
            {
                int newLocation = regex.Match(source, location).Index;
                source = regex.Replace(source, replace, 1, location);
                location = newLocation + newTextString.Length;

                writeFile = new Write(source, targetDirectory, fileExtension, 0, matches, fileNumber);
                writeFile.writeAscii();
                if ( fuzzLocations.Length <= fileNumber )
                {
                    Array.Resize(ref fuzzLocations, fuzzLocations.Length + 10);
                }
                fuzzLocations[fileNumber] = newLocation+2;//??
                fileNumber++;
            }
            return fileNumber;
        }
```

7.3.5 crash.c

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#include "libdasm.h"

int main (int argc, char **argv)
{
    PROCESS_INFORMATION pi;
    INSTRUCTION         inst;
    STARTUPINFO         si;
    DEBUG_EVENT         dbg;
    CONTEXT             context;
    HANDLE              thread;
    HANDLE              process;
    DWORD               wait_time;
    DWORD               start_time;
    BOOL                ret;
    BOOL                exception;
    //BOOL              continueDebug;
    u_char              inst_buf[32];
    char                inst_string[256];
    char                command_line[32768];
    int                 i;

    //
    // variable initialization.
    //

    memset(&pi, 0, sizeof(pi));
    memset(&si, 0, sizeof(si));
    si.cb = sizeof(si);

    memset(command_line, 0, sizeof(command_line));
    memset(inst_buf,     0, sizeof(inst_buf));

    start_time = GetTickCount();
         exception  = FALSE;

    //
    // command line processing.
    //

    // minimum arg check.
    if (argc < 4)
    {
        fprintf(stderr, "[!] Usage: crash <path to app> <milliseconds> <arg1> [arg2 arg3 ... argn]\n\n");
        return -1;
    }

    // convert wait time from string to integer.
    if ((wait_time = atoi(argv[2])) == 0)
    {
        fprintf(stderr, "[!] Milliseconds argument unrecognized: %s\n\n", argv[2]);
```

```
    return -1;
}

// create the command line string for the call to CreateProcess().
strcpy(command_line, argv[1]);

for (i = 3; i < argc; i++)
{
    strcat(command_line, " ");
    strcat(command_line, argv[i]);
}

//
// launch the target process.
//

ret = CreateProcess(NULL,       // target file name.
    command_line,               // command line options.
    NULL,                       // process attributes.
    NULL,                       // thread attributes.
    FALSE,                      // handles are not inherited.
    DEBUG_PROCESS,              // debug the target process and all spawned children.
    NULL,                       // use our current environment.
    NULL,                       // use our current working directory.
    &si,                        // pointer to STARTUPINFO structure.
    &pi);                       // pointer to PROCESS_INFORMATION structure.

printf("[*] %s\n", GetCommandLine());  //Print the command line

if (!ret)
{
    fprintf(stderr, "[!] CreateProcess() failed: %d\n\n", GetLastError());
    return -1;
}

//
// watch for an exception.
//

while (GetTickCount() - start_time < wait_time)
{
    if (WaitForDebugEvent(&dbg, 100))
    {
        // we are only interested in debug events.
        if (dbg.dwDebugEventCode != EXCEPTION_DEBUG_EVENT)
        {
            ContinueDebugEvent(dbg.dwProcessId, dbg.dwThreadId, DBG_CONTINUE);
            continue;
        }

        // get a handle to the offending thread.
        /*if ((thread = OpenThread(THREAD_ALL_ACCESS, FALSE, dbg.dwThreadId)) == NULL)
        {
            fprintf(stderr, "[!] OpenThread() failed: %d\n\n", GetLastError());
            return -1;
```

```
            }

            // get the context of the offending thread.
            context.ContextFlags = CONTEXT_FULL;

            if (GetThreadContext(thread, &context) == 0)
            {
                fprintf(stderr, "[!] GetThreadContext() failed: %d\n\n", GetLastError());
                return -1;
            }*/

           // examine the exception code.
            switch (dbg.u.Exception.ExceptionRecord.ExceptionCode)
            {
                case EXCEPTION_ACCESS_VIOLATION:
                                        exception = TRUE;
                                        printf("[*] Access Violation\n");
                                        break;
                                case EXCEPTION_INT_DIVIDE_BY_ZERO:
                                        exception = TRUE;
                                        printf("[*] Divide by Zero\n");
                                        break;
                case EXCEPTION_STACK_OVERFLOW:
                    exception = TRUE;
                                        printf("[*] Stack Overflow\n");
                                        break;
                default:
                                        //printf("[*] Unknown Exception (%08x):\n",
dbg.u.Exception.ExceptionRecord.ExceptionCode);
                    ContinueDebugEvent(dbg.dwProcessId, dbg.dwThreadId, DBG_CONTINUE);
            }

                        // if an exception occured, print more information.
                        if (exception)
                        {
                                // open a handle to the target process.
                        if ((process = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dbg.dwProcessId)) == NULL)
                                {
                                fprintf(stderr, "[!] OpenProcess() failed: %d\n\n", GetLastError());
                                        return -1;
                                }

                                // grab some memory at EIP for disassembly.
                                ReadProcessMemory(process, (void *)context.Eip, &inst_buf, 32, NULL);

                                // decode the instruction into a string.
                                get_instruction(&inst, inst_buf, MODE_32);
                       get_instruction_string(&inst, FORMAT_INTEL, 0, inst_string, sizeof(inst_string));

                                // print the exception to screen.
                                printf("[*] Exception caught at %08x %s\n", context.Eip, inst_string);
                         printf("[*] EAX:%08x EBX:%08x ECX:%08x EDX:%08x\n", context.Eax, context.Ebx,
 context.Ecx, context.Edx);
                         printf("[*] ESI:%08x EDI:%08x ESP:%08x EBP:%08x\n\n", context.Esi, context.Edi,
 context.Esp, context.Ebp);
```

```
                                return 1;
                        }

        }
         }
    //
    // done.
    //

    printf("[*] Process terminated normally.\n\n");
    return 0;
}
```

7.4 Smart Fuzzing Functions

7.4.1 PGACreate

```
PGAContext *PGACreate ( int *argc, char **argv,
                        int datatype, int len, int maxormin )
{
    int i;
    PGAContext *ctx;

    ctx = (PGAContext *) malloc ( sizeof(PGAContext) );

    /*  We cannot make PGA calls until we sort the FuncNameIndex below,
     *  so we just manually print the (rather severe) error message.
     */
    if( ctx == NULL ) {
        fprintf(stderr, "PGACreate: No room to allocate ctx\n");
        exit(-1);
    }


    /*  We use this (indirectly) in PGAReadCmdLine -- in processing
     *  -pgahelp and -pgahelp debug.
     */
    MPI_Initialized (&ctx->par.MPIAlreadyInit);

    /* Initialize MPI, only if it isn't already running (fortran)  */
    if (!ctx->par.MPIAlreadyInit)
         MPI_Init (argc, &argv);


#if OPTIMIZE==0
    /*  Sort the FuncNameIndex.  This allows us to use a binary search
     *  for finding the function names.
     */
    PGASortFuncNameIndex(ctx);
#endif

    /* Initialize debug flags, then parse command line arguments.  */
    for (i=0; i<PGA_DEBUG_MAXFLAGS; i++)
        ctx->debug.PGADebugFlags[i] = PGA_FALSE;
```

```
PGAReadCmdLine( ctx, argc, argv );


/*  The context variable is now initialized enough to allow this
 *  call to complete successfully.
 */
PGADebugEntered("PGACreate");

/* required parameter 1: abstract data type */
switch (datatype)
{
case PGA_DATATYPE_BINARY:
case PGA_DATATYPE_INTEGER:
case PGA_DATATYPE_REAL:
case PGA_DATATYPE_CHARACTER:
case PGA_DATATYPE_USER:
     ctx->ga.datatype  = datatype;
     break;
default:
     PGAError( ctx, "PGACreate: Invalid value of datatype:",
              PGA_FATAL, PGA_INT, (void *) &datatype );
};

/* required parameter 2: string string length */
if (len <= 1)
    PGAError( ctx,  "PGACreate: Invalid value of len:",
              PGA_FATAL, PGA_INT, (void *) &len );
else
    ctx->ga.StringLen = len;


/* required parameter 3: optimization direction */
switch (maxormin) {
    case PGA_MAXIMIZE:
    case PGA_MINIMIZE:
      ctx->ga.optdir = maxormin;
      break;
    default:
      PGAError( ctx, "PGACreate: Invalid value of optdir:",
                PGA_FATAL, PGA_INT, (void *) &maxormin );
};


/*  For datatype == PGA_DATATYPE_BINARY, set how many full words
 *  are used in the packed representation, and how many extra bits
 *  this leaves us with.  Finally, set how many total words are used;
 *  if there are no extra bits, this is just the number of full words,
 *  else, there is one more word used than the number of full words.
 */
switch (datatype) {
case PGA_DATATYPE_BINARY:
    ctx->ga.fw = ctx->ga.StringLen/WL;
    ctx->ga.eb = ctx->ga.StringLen%WL;
    if ( ctx->ga.eb == 0 )
        ctx->ga.tw = ctx->ga.fw;
```

```
    else
        ctx->ga.tw = ctx->ga.fw+1;
    break;
default:
    ctx->ga.fw = PGA_UNINITIALIZED_INT;
    ctx->ga.eb = PGA_UNINITIALIZED_INT;
    ctx->ga.tw = PGA_UNINITIALIZED_INT;
    break;
}

/*  Clear all the setting.  Later on, PGASetUp() will be called, and then
 *  it will notice which setting are uninitialized, and set them to the
 *  default value.
 */
ctx->ga.PopSize            = PGA_UNINITIALIZED_INT;
ctx->ga.StoppingRule       = PGA_STOP_MAXITER;
ctx->ga.MaxIter            = PGA_UNINITIALIZED_INT;
ctx->ga.MaxNoChange        = PGA_UNINITIALIZED_INT;
ctx->ga.MaxSimilarity      = PGA_UNINITIALIZED_INT;
ctx->ga.NumReplace         = PGA_UNINITIALIZED_INT;
ctx->ga.CrossoverType      = PGA_UNINITIALIZED_INT;
ctx->ga.SelectType         = PGA_UNINITIALIZED_INT;
ctx->ga.FitnessType        = PGA_UNINITIALIZED_INT;
ctx->ga.FitnessMinType     = PGA_UNINITIALIZED_INT;
ctx->ga.MutationType       = PGA_UNINITIALIZED_INT;
ctx->ga.MutateOnlyNoCross  = PGA_UNINITIALIZED_INT;
ctx->ga.MutateRealValue    = PGA_UNINITIALIZED_DOUBLE;
ctx->ga.MutateIntegerValue = PGA_UNINITIALIZED_INT;
ctx->ga.MutateBoundedFlag  = PGA_UNINITIALIZED_INT;
ctx->ga.NoDuplicates       = PGA_UNINITIALIZED_INT;
ctx->ga.MutationProb       = PGA_UNINITIALIZED_DOUBLE;
ctx->ga.CrossoverProb      = PGA_UNINITIALIZED_DOUBLE;
ctx->ga.UniformCrossProb   = PGA_UNINITIALIZED_DOUBLE;
ctx->ga.PTournamentProb    = PGA_UNINITIALIZED_DOUBLE;
ctx->ga.FitnessRankMax     = PGA_UNINITIALIZED_DOUBLE;
ctx->ga.FitnessCmaxValue   = PGA_UNINITIALIZED_DOUBLE;
ctx->ga.PopReplace         = PGA_UNINITIALIZED_INT;
ctx->ga.iter               = 0;
ctx->ga.ItersOfSame        = 0;
ctx->ga.PercentSame        = 0;
ctx->ga.selected           = NULL;
ctx->ga.SelectIndex        = 0;
ctx->ga.restart            = PGA_UNINITIALIZED_INT;
ctx->ga.restartFreq        = PGA_UNINITIALIZED_INT;
ctx->ga.restartAlleleProb  = PGA_UNINITIALIZED_DOUBLE;

/* Operations */
ctx->cops.CreateString      = NULL;
ctx->cops.Mutation          = NULL;
ctx->cops.Crossover         = NULL;
ctx->cops.PrintString       = NULL;
ctx->cops.CopyString        = NULL;
ctx->cops.Duplicate         = NULL;
ctx->cops.InitString        = NULL;
ctx->cops.BuildDatatype     = NULL;
```

```
    ctx->cops.StopCond           = NULL;
    ctx->cops.EndOfGen           = NULL;

    ctx->fops.Mutation           = NULL;
    ctx->fops.Crossover          = NULL;
    ctx->fops.PrintString        = NULL;
    ctx->fops.CopyString         = NULL;
    ctx->fops.Duplicate          = NULL;
    ctx->fops.InitString         = NULL;
    ctx->fops.StopCond           = NULL;
    ctx->fops.EndOfGen           = NULL;

    /* Parallel */
    ctx->par.NumIslands         = PGA_UNINITIALIZED_INT;
    ctx->par.NumDemes           = PGA_UNINITIALIZED_INT;
    ctx->par.DefaultComm        = NULL;
#ifdef FAKE_MPI
    ctx->par.MPIStubLibrary     = PGA_TRUE;
#else
    ctx->par.MPIStubLibrary     = PGA_FALSE;
#endif

    /* Reporting */
    ctx->rep.PrintFreq          = PGA_UNINITIALIZED_INT;
    ctx->rep.PrintOptions       = 0;
    ctx->rep.Online             = 0;
    ctx->rep.Offline            = 0;
    ctx->rep.Best               = PGA_UNINITIALIZED_DOUBLE;
    ctx->rep.starttime          = PGA_UNINITIALIZED_INT;

    /* System
     *
     *  If ctx->sys.UserFortran is not set to PGA_TRUE in pgacreate_ (the
     *  fortran stub to PGACreate), the user program is in C.
     */
    if (ctx->sys.UserFortran != PGA_TRUE)
        ctx->sys.UserFortran  = PGA_FALSE;
    ctx->sys.SetUpCalled        = PGA_FALSE;
    ctx->sys.PGAMaxInt          = INT_MAX;
    ctx->sys.PGAMinInt          = INT_MIN;
    ctx->sys.PGAMaxDouble       = DBL_MAX;
    ctx->sys.PGAMinDouble       = DBL_MIN;

    /* Debug */
    /* Set above before parsing command line arguments */

    /* Initialization */
    ctx->init.RandomInit         = PGA_UNINITIALIZED_INT;
    ctx->init.BinaryProbability = PGA_UNINITIALIZED_DOUBLE;
    ctx->init.RealType           = PGA_UNINITIALIZED_INT;
    ctx->init.IntegerType        = PGA_UNINITIALIZED_INT;
    ctx->init.CharacterType      = PGA_UNINITIALIZED_INT;
    ctx->init.RandomSeed         = PGA_UNINITIALIZED_INT;

    /*  Allocate and clear arrays to define the minimum and maximum values
```

```
     *  allowed by integer and real datatypes.
     */
    switch (datatype)
    {
    case PGA_DATATYPE_INTEGER:
         ctx->init.IntegerMax = (int *) malloc(len * sizeof(PGAInteger));
         if (!ctx->init.IntegerMax)
              PGAError(ctx, "PGACreate: No room to allocate:", PGA_FATAL,
                       PGA_CHAR, (void *) "ctx->init.IntegerMax");
         ctx->init.IntegerMin = (int *) malloc(len * sizeof(PGAInteger));
         if (!ctx->init.IntegerMin)
              PGAError(ctx, "PGACreate: No room to allocate:", PGA_FATAL,
                       PGA_CHAR, (void *) "ctx->init.IntegerMin");
         ctx->init.RealMax = NULL;
         ctx->init.RealMin = NULL;
         for (i = 0; i < len; i++)
         {
              ctx->init.IntegerMin[i] = PGA_UNINITIALIZED_INT;
              ctx->init.IntegerMax[i] = PGA_UNINITIALIZED_INT;
         }
         break;
    case PGA_DATATYPE_REAL:
         ctx->init.RealMax = (PGAReal *) malloc(len * sizeof(PGAReal));
         if (!ctx->init.RealMax)
              PGAError(ctx, "PGACreate: No room to allocate:", PGA_FATAL,
                       PGA_CHAR, (void *) "ctx->init.RealMax");
         ctx->init.RealMin = (PGAReal *) malloc(len * sizeof(PGAReal));
         if (!ctx->init.RealMin)
              PGAError(ctx, "PGACreate: No room to allocate:", PGA_FATAL,
                       PGA_CHAR, (void *) "ctx->init.RealMin");
         ctx->init.IntegerMax = NULL;
         ctx->init.IntegerMin = NULL;
         for (i = 0; i < len; i++)
         {
              ctx->init.RealMin[i] = PGA_UNINITIALIZED_DOUBLE;
              ctx->init.RealMax[i] = PGA_UNINITIALIZED_DOUBLE;
         }
         break;
    default:
         ctx->init.RealMax = NULL;
         ctx->init.RealMin = NULL;
         ctx->init.IntegerMax = NULL;
         ctx->init.IntegerMin = NULL;
         break;
    }

    PGADebugExited("PGACreate");

    return(ctx);
}
```

7.4.2 PGASetUp

```
void PGASetUp ( PGAContext *ctx )
{
```

```
/*  These are for temporary storage of datatype specific functions.
 *  They allow some (understatement of the yesr!!) cleaning of the
 *  code below.
 */
void         (*CreateString)(PGAContext *, int, int, int);
int          (*Mutation)(PGAContext *, int, int, double);
void         (*Crossover)(PGAContext *, int, int, int, int, int, int);
void         (*PrintString)(PGAContext *, FILE *, int, int);
void         (*CopyString)(PGAContext *, int, int, int, int);
int          (*Duplicate)(PGAContext *, int, int, int, int);
void         (*InitString)(PGAContext *, int, int);
MPI_Datatype (*BuildDatatype)(PGAContext *, int, int);
int err=0, i;

PGADebugEntered("PGASetUp");
PGAFailIfSetUp("PGASetUp");

ctx->sys.SetUpCalled = PGA_TRUE;

if ( ctx->ga.datatype           == PGA_DATATYPE_BINARY   &&
     ctx->ga.tw                 == PGA_UNINITIALIZED_INT )
  PGAError( ctx,
           "PGASetUp: Binary: Total Words (ctx->ga.tw) == UNINITIALIZED?",
           PGA_FATAL, PGA_INT, (void *) &ctx->ga.tw );

if ( ctx->ga.datatype           == PGA_DATATYPE_BINARY  &&
     ctx->ga.fw                 == PGA_UNINITIALIZED_INT )
  PGAError( ctx,
           "PGASetUp: Binary: Full Words (ctx->ga.fw) == UNINITIALIZED?",
           PGA_FATAL, PGA_INT,  (void *) &ctx->ga.fw );

if ( ctx->ga.datatype           == PGA_DATATYPE_BINARY  &&
     ctx->ga.eb                 == PGA_UNINITIALIZED_INT )
  PGAError( ctx,
           "PGASetUp: Binary: Empty Bits (ctx->ga.eb) == UNINITIALIZED?",
           PGA_FATAL, PGA_INT, (void *) &ctx->ga.eb );

if ( ctx->ga.PopSize            == PGA_UNINITIALIZED_INT)
  ctx->ga.PopSize                = 100;

if ( ctx->ga.MaxIter            == PGA_UNINITIALIZED_INT)
     ctx->ga.MaxIter            = 1000;

if ( ctx->ga.MaxNoChange        == PGA_UNINITIALIZED_INT)
     ctx->ga.MaxNoChange        = 100;

if ( ctx->ga.MaxSimilarity      == PGA_UNINITIALIZED_INT)
     ctx->ga.MaxSimilarity      = 95;

if ( ctx->ga.NumReplace         == PGA_UNINITIALIZED_INT)
     ctx->ga.NumReplace         = (int) ceil(ctx->ga.PopSize * 0.1);

if ( ctx->ga.NumReplace          > ctx->ga.PopSize)
     PGAError(ctx, "PGASetUp: NumReplace > PopSize",
              PGA_FATAL, PGA_VOID, NULL);
```

```
if ( ctx->ga.CrossoverType      == PGA_UNINITIALIZED_INT)
     ctx->ga.CrossoverType       = PGA_CROSSOVER_TWOPT;

if (ctx->ga.CrossoverType       == PGA_CROSSOVER_TWOPT &&
    ctx->ga.StringLen == 2)
     PGAError(ctx, "PGASetUp: Invalid Crossover type for string of length "
              "2", PGA_FATAL, PGA_INT, (void *) &ctx->ga.CrossoverType);

if ( ctx->ga.SelectType         == PGA_UNINITIALIZED_INT)
     ctx->ga.SelectType          = PGA_SELECT_TOURNAMENT;

if ( ctx->ga.FitnessType        == PGA_UNINITIALIZED_INT)
     ctx->ga.FitnessType         = PGA_FITNESS_RAW;

if ( ctx->ga.FitnessMinType     == PGA_UNINITIALIZED_INT)
     ctx->ga.FitnessMinType      = PGA_FITNESSMIN_CMAX;

if ( ctx->ga.MutateOnlyNoCross == PGA_UNINITIALIZED_INT)
     ctx->ga.MutateOnlyNoCross  = PGA_TRUE;

if ( ctx->ga.MutationProb       == PGA_UNINITIALIZED_DOUBLE)
     ctx->ga.MutationProb        = 1. / ctx->ga.StringLen;

if ( ctx->ga.MutationType       == PGA_UNINITIALIZED_INT) {
    switch (ctx->ga.datatype) {
    case PGA_DATATYPE_BINARY:
    case PGA_DATATYPE_CHARACTER:
    case PGA_DATATYPE_USER:
         /* leave PGA_UNINITIALIZED_INT for these data types */
         break;
    case PGA_DATATYPE_REAL:
         ctx->ga.MutationType    = PGA_MUTATION_GAUSSIAN;
         break;
    case PGA_DATATYPE_INTEGER:
         switch (ctx->init.IntegerType) {
             case PGA_UNINITIALIZED_INT:
             case PGA_IINIT_PERMUTE:
                 ctx->ga.MutationType    = PGA_MUTATION_PERMUTE;
                 break;
             case PGA_IINIT_RANGE:
                 ctx->ga.MutationType    = PGA_MUTATION_RANGE;
                 break;
         }
         break;
    default:
         PGAError( ctx, "PGASetup: Invalid value of ctx->ga.datatype:",
                   PGA_FATAL, PGA_INT, (void *) &(ctx->ga.datatype) );
     }
}

if (ctx->ga.MutateRealValue    == PGA_UNINITIALIZED_DOUBLE) {
    switch (ctx->ga.MutationType) {
    case PGA_MUTATION_GAUSSIAN:
        ctx->ga.MutateRealValue    = 0.1;
```

```
            break;
        case PGA_MUTATION_UNIFORM:
            ctx->ga.MutateRealValue   = 0.1;
            break;
        case PGA_MUTATION_CONSTANT:
            ctx->ga.MutateRealValue   = 0.01;
            break;
        case PGA_MUTATION_RANGE:
        default:
            ctx->ga.MutateRealValue   = 0.0;
        }
    }

    if ( ctx->ga.MutateIntegerValue == PGA_UNINITIALIZED_INT)
         ctx->ga.MutateIntegerValue  = 1;

    if ( ctx->ga.MutateBoundedFlag == PGA_UNINITIALIZED_INT)
         ctx->ga.MutateBoundedFlag  = PGA_FALSE;

    if ( ctx->ga.NoDuplicates      == PGA_UNINITIALIZED_INT)
         ctx->ga.NoDuplicates       = PGA_FALSE;

    if ( ctx->ga.NoDuplicates && ((ctx->ga.StoppingRule & PGA_STOP_TOOSIMILAR)
                                  == PGA_STOP_TOOSIMILAR))
         PGAError(ctx, "PGASetUp: No Duplicates inconsistent with Stopping "
                  "Rule:", PGA_FATAL, PGA_INT, (void *) &ctx->ga.StoppingRule);

    if ( ctx->ga.CrossoverProb     == PGA_UNINITIALIZED_DOUBLE)
         ctx->ga.CrossoverProb      = 0.85;

    if ( ctx->ga.UniformCrossProb  == PGA_UNINITIALIZED_DOUBLE)
         ctx->ga.UniformCrossProb   = 0.6;

    if ( ctx->ga.PTournamentProb   == PGA_UNINITIALIZED_DOUBLE)
         ctx->ga.PTournamentProb    = 0.6;

    if ( ctx->ga.FitnessRankMax    == PGA_UNINITIALIZED_DOUBLE)
         ctx->ga.FitnessRankMax     = 1.2;

    if ( ctx->ga.FitnessCmaxValue  == PGA_UNINITIALIZED_DOUBLE)
         ctx->ga.FitnessCmaxValue   = 1.01;

    if ( ctx->ga.PopReplace        == PGA_UNINITIALIZED_INT)
         ctx->ga.PopReplace         = PGA_POPREPL_BEST;

    if ( ctx->ga.restart           == PGA_UNINITIALIZED_INT)
         ctx->ga.restart            = PGA_FALSE;

    if ( ctx->ga.restartFreq       == PGA_UNINITIALIZED_INT)
         ctx->ga.restartFreq        = 50;

    if ( ctx->ga.restartAlleleProb == PGA_UNINITIALIZED_DOUBLE)
         ctx->ga.restartAlleleProb = 0.5;
```

```
/* ops */
    /*  If no user supplied "done" function, use the built in one.
     *  No need to check EndOfGen; they only get called if they
     *  are defined.
     */
    if (((void *)ctx->cops.StopCond == (void *)PGADone) ||
        ((void *)ctx->fops.StopCond == (void *)PGADone))
        PGAError( ctx,
                 "PGASetUp: Using PGADone as the user stopping condition will"
                 " result in an infinite loop!", PGA_FATAL, PGA_VOID, NULL);

    switch (ctx->ga.datatype) {
    case PGA_DATATYPE_BINARY:
        CreateString  = PGABinaryCreateString;
        BuildDatatype = PGABinaryBuildDatatype;
        Mutation      = PGABinaryMutation;

        switch (ctx->ga.CrossoverType) {
          case PGA_CROSSOVER_ONEPT:
            Crossover  = PGABinaryOneptCrossover;
            break;
          case PGA_CROSSOVER_TWOPT:
            Crossover  = PGABinaryTwoptCrossover;
            break;
          case PGA_CROSSOVER_UNIFORM:
            Crossover  = PGABinaryUniformCrossover;
            break;
        }
        PrintString    = PGABinaryPrintString;
        CopyString     = PGABinaryCopyString;
        Duplicate      = PGABinaryDuplicate;
        InitString     = PGABinaryInitString;
        break;
      case PGA_DATATYPE_INTEGER:
        CreateString   = PGAIntegerCreateString;
        BuildDatatype  = PGAIntegerBuildDatatype;
        Mutation       = PGAIntegerMutation;
        switch (ctx->ga.CrossoverType) {
          case PGA_CROSSOVER_ONEPT:
            Crossover  = PGAIntegerOneptCrossover;
            break;
          case PGA_CROSSOVER_TWOPT:
            Crossover  = PGAIntegerTwoptCrossover;
            break;
          case PGA_CROSSOVER_UNIFORM:
            Crossover  = PGAIntegerUniformCrossover;
            break;
        }
        PrintString    = PGAIntegerPrintString;
        CopyString     = PGAIntegerCopyString;
        Duplicate      = PGAIntegerDuplicate;
        InitString     = PGAIntegerInitString;
        break;
      case PGA_DATATYPE_REAL:
        CreateString   = PGARealCreateString;
```

```
  BuildDatatype  = PGARealBuildDatatype;
  Mutation       = PGARealMutation;
  switch (ctx->ga.CrossoverType) {
    case PGA_CROSSOVER_ONEPT:
      Crossover  = PGARealOneptCrossover;
      break;
    case PGA_CROSSOVER_TWOPT:
      Crossover  = PGARealTwoptCrossover;
      break;
    case PGA_CROSSOVER_UNIFORM:
      Crossover  = PGARealUniformCrossover;
      break;
  }
  PrintString    = PGARealPrintString;
  CopyString     = PGARealCopyString;
  Duplicate      = PGARealDuplicate;
  InitString     = PGARealInitString;
  break;
case PGA_DATATYPE_CHARACTER:
  CreateString   = PGACharacterCreateString;
  BuildDatatype  = PGACharacterBuildDatatype;
  Mutation       = PGACharacterMutation;
  switch (ctx->ga.CrossoverType) {
    case PGA_CROSSOVER_ONEPT:
      Crossover  = PGACharacterOneptCrossover;
      break;
    case PGA_CROSSOVER_TWOPT:
      Crossover  = PGACharacterTwoptCrossover;
      break;
    case PGA_CROSSOVER_UNIFORM:
      Crossover  = PGACharacterUniformCrossover;
      break;
  }
  PrintString    = PGACharacterPrintString;
  CopyString     = PGACharacterCopyString;
  Duplicate      = PGACharacterDuplicate;
  InitString     = PGACharacterInitString;
  break;
case PGA_DATATYPE_USER:
  if (ctx->cops.CreateString == NULL)
      PGAError( ctx,
                "PGASetUp: User datatype needs CreateString function:",
                PGA_WARNING, PGA_INT, (void *) &err );
  if (ctx->cops.Mutation     == NULL)
      PGAError( ctx,
                "PGASetUp: User datatype needs Mutation function:",
                PGA_WARNING, PGA_INT, (void *) &err );
  if (ctx->cops.Crossover    == NULL)
      PGAError( ctx,
                "PGASetUp: User datatype needs Crossover function:",
                PGA_WARNING, PGA_INT, (void *) &err );
  if (ctx->cops.PrintString  == NULL)
      PGAError( ctx,
                "PGASetUp: User datatype needs PrintString function:",
                PGA_WARNING, PGA_INT, (void *) &err );
```

```
        if (ctx->cops.Duplicate    == NULL)
            PGAError( ctx,
                     "PGASetUp: User datatype needs Duplicate function:",
                     PGA_WARNING, PGA_INT, (void *) &err );
        if (ctx->cops.CopyString    == NULL)
            PGAError( ctx,
                     "PGASetUp: User datatype needs CopyString function:",
                     PGA_WARNING, PGA_INT, (void *) &err );
        if (ctx->cops.BuildDatatype == NULL)
             PGAError(ctx,
                      "PGASetUp: User datatype needs BuildDatatype "
                      "function:", PGA_FATAL, PGA_INT, (void *) &err );
        break;
    }
    if ((ctx->cops.Mutation     == NULL) && (ctx->fops.Mutation    == NULL))
        ctx->cops.Mutation      = Mutation;
    if ((ctx->cops.Crossover    == NULL) && (ctx->fops.Crossover   == NULL))
        ctx->cops.Crossover     = Crossover;
    if ((ctx->cops.PrintString  == NULL) && (ctx->fops.PrintString == NULL))
        ctx->cops.PrintString   = PrintString;
    if ((ctx->cops.Duplicate    == NULL) && (ctx->fops.Duplicate   == NULL))
        ctx->cops.Duplicate     = Duplicate;
    if ((ctx->cops.InitString   == NULL) && (ctx->fops.InitString  == NULL))
        ctx->cops.InitString    = InitString;
    if (ctx->cops.CreateString  == NULL)
        ctx->cops.CreateString  = CreateString;
    if (ctx->cops.CopyString    == NULL)
        ctx->cops.CopyString    = CopyString;
    if (ctx->cops.BuildDatatype == NULL)
        ctx->cops.BuildDatatype = BuildDatatype;

/* par */
    if ( ctx->par.NumIslands       == PGA_UNINITIALIZED_INT)
         ctx->par.NumIslands        = 1;
    if ( ctx->par.NumDemes         == PGA_UNINITIALIZED_INT)
         ctx->par.NumDemes          = 1;
    if ( ctx->par.DefaultComm      == NULL )
         ctx->par.DefaultComm       = MPI_COMM_WORLD;



/* rep */
    if ( ctx->rep.PrintFreq == PGA_UNINITIALIZED_INT)
         ctx->rep.PrintFreq  = 10;

/* sys */
    /* no more sets necessary here. */

/* debug */

/* init */
    if ( ctx->init.RandomInit == PGA_UNINITIALIZED_INT)
         ctx->init.RandomInit  = PGA_TRUE;

    if ( ctx->init.BinaryProbability == PGA_UNINITIALIZED_DOUBLE)
```

```
     ctx->init.BinaryProbability  = 0.5;

if ( ctx->init.RealType == PGA_UNINITIALIZED_INT)
     ctx->init.RealType  = PGA_RINIT_RANGE;
if ( ctx->init.IntegerType == PGA_UNINITIALIZED_INT)
     ctx->init.IntegerType  = PGA_IINIT_PERMUTE;
if ( ctx->init.CharacterType == PGA_UNINITIALIZED_INT)
     ctx->init.CharacterType = PGA_CINIT_LOWER;

switch (ctx->ga.datatype)
{
case PGA_DATATYPE_INTEGER:
     for (i = 0; i < ctx->ga.StringLen; i++)
     {
          if (ctx->init.IntegerMin[i] == PGA_UNINITIALIZED_INT)
               ctx->init.IntegerMin[i] = 0;
          if (ctx->init.IntegerMax[i] == PGA_UNINITIALIZED_INT)
               ctx->init.IntegerMax[i] = ctx->ga.StringLen - 1;
     }
     break;
case PGA_DATATYPE_REAL:
     for (i = 0; i < ctx->ga.StringLen; i++)
     {
          if (ctx->init.RealMin[i] == PGA_UNINITIALIZED_DOUBLE)
               ctx->init.RealMin[i] = 0.;
          if (ctx->init.RealMax[i] == PGA_UNINITIALIZED_DOUBLE)
               ctx->init.RealMax[i] = 1.;
     }
     break;
}

/* If a seed was not specified, get one from a time of day call */
if ( ctx->init.RandomSeed == PGA_UNINITIALIZED_INT)
     ctx->init.RandomSeed = (int)time(NULL);

/* seed random number generator with this process' unique seed */
ctx->init.RandomSeed += PGAGetRank(ctx, MPI_COMM_WORLD);
PGARandom01( ctx, ctx->init.RandomSeed );

ctx->ga.selected        = (int *)malloc( sizeof(int) * ctx->ga.PopSize );
if (ctx->ga.selected == NULL)
     PGAError(ctx, "PGASetUp: No room to allocate ctx->ga.selected",
              PGA_FATAL, PGA_VOID, NULL);

ctx->ga.sorted          = (int *)malloc( sizeof(int) * ctx->ga.PopSize );
if (ctx->ga.sorted == NULL)
     PGAError(ctx, "PGASetUp: No room to allocate ctx->ga.sorted",
              PGA_FATAL, PGA_VOID, NULL);

ctx->scratch.intscratch = (int *)malloc( sizeof(int) * ctx->ga.PopSize );
if (ctx->scratch.intscratch == NULL)
     PGAError(ctx, "PGASetUp: No room to allocate ctx->scratch.intscratch",
              PGA_FATAL, PGA_VOID, NULL);

ctx->scratch.dblscratch = (double *)malloc(sizeof(double) * ctx->ga.PopSize);
```

```
    if (ctx->scratch.dblscratch == NULL)
         PGAError(ctx, "PGASetUp: No room to allocate ctx->scratch.dblscratch",
                  PGA_FATAL, PGA_VOID, NULL);

    PGACreatePop ( ctx , PGA_OLDPOP );
    PGACreatePop ( ctx , PGA_NEWPOP );

    ctx->rep.starttime = time(NULL);

    PGADebugExited("PGASetUp");
}
```

7.4.3 PGARun

```
void PGARun(PGAContext *ctx, double (*evaluate)(PGAContext *c, int p, int pop))
{
     MPI_Comm comm;                 /* value of default communicator */
     int nprocs;                    /* number of processes in above  */
     int npops;                     /* number of populations         */
     int ndemes;                    /* number of demes               */


     PGADebugEntered("PGARun");
     PGAFailIfNotSetUp("PGARun");

     comm   = PGAGetCommunicator(ctx);
     nprocs = PGAGetNumProcs    (ctx, comm);
     npops  = PGAGetNumIslands  (ctx);
     ndemes = PGAGetNumDemes    (ctx);

     /**********************************************************************/
     /*              Global model, one island, one deme                    */
     /**********************************************************************/
     if     ( (npops == 1) && (ndemes == 1) ) {

         PGARunGM(ctx, evaluate, comm);
     }

     /**********************************************************************/
     /*              Island model, > one island, one deme                  */
     /**********************************************************************/
     else if( (npops > 1) && (ndemes == 1) ) {
         if ( nprocs == 1 )
             PGAError (ctx, "PGARun: island model with one process",
                       PGA_FATAL, PGA_VOID, (void *) &nprocs);
         if ( nprocs != npops) {
             PGAError (ctx, "PGARun: island model no. processes != no. pops",
                       PGA_FATAL, PGA_VOID, (void *) &nprocs);
         }
         PGARunIM(ctx,evaluate,comm);
     }

     /**********************************************************************/
     /*              Neighborhood model, one island, > one deme            */
```

```
    /************************************************************************/
    else if( (npops == 1) && (ndemes > 1) ) {
        if ( nprocs == 1 )
            PGAError (ctx, "PGARun: neighborhood model with one process",
                      PGA_FATAL, PGA_VOID, (void *) &nprocs);
        if ( nprocs != ndemes)
            PGAError (ctx, "PGARun: neighborhood model no. processes "
                      "!= no. demes", PGA_FATAL, PGA_VOID, (void *) &nprocs);
        PGARunNM(ctx,evaluate,comm);
    }

    /************************************************************************/
    /*              Mixed model, > one island, > one deme                   */
    /************************************************************************/
    else if( (npops > 1) && (ndemes > 1) ) {
        PGAError (ctx, "PGARun: Cannot execute mixed models",
                  PGA_FATAL, PGA_VOID, (void *) &nprocs);
    }

    /************************************************************************/
    /*                         E R R O R                                    */
    /************************************************************************/
    else {
        PGAError (ctx, "PGARun: Invalid combination of numislands,"
                  "ndemes, and nprocs.",
                  PGA_FATAL, PGA_VOID, (void *) &nprocs);
    }

    /************************************************************************/
    /*                          E X I T                                     */
    /************************************************************************/
    PGADebugExited("PGARun");
    return;
}
```

7.4.4 PGADestroy

```
void PGADestroy (PGAContext *ctx)
{
    int i;

    PGADebugEntered("PGADestroy");

    /*  These are allocated by PGASetUp.  Free then only if PGASetUp
     *  was called.
     */
    if (ctx->sys.SetUpCalled == PGA_TRUE) {
      /*  Free the population...fly little birdies!  You're FREE!!!  */
      for ( i = 0; i < ctx->ga.PopSize + 2; i++ ) {
        free ( ctx->ga.oldpop[i].chrom );
        free ( ctx->ga.newpop[i].chrom );
      }
      free ( ctx->ga.oldpop );
      free ( ctx->ga.newpop );
```

```
  /*  Free the scratch space.  */
  free ( ctx->scratch.intscratch );
  free ( ctx->scratch.dblscratch );
  free ( ctx->ga.selected );
  free ( ctx->ga.sorted );
}

/*  These are allocated by PGACreate  */
if (ctx->ga.datatype == PGA_DATATYPE_REAL)
  {
    free ( ctx->init.RealMax );
    free ( ctx->init.RealMin );
  }
else if (ctx->ga.datatype == PGA_DATATYPE_INTEGER)
  {
    free ( ctx->init.IntegerMax );
    free ( ctx->init.IntegerMin );
  }

/*  We want to finalize MPI only if it was not started for us (as
 *  fortran would do) AND it is actually running.  It would not be
 *  running if, for example, -pgahelp is specified on the command
 *  line.
 */
MPI_Initialized(&i);
if ((ctx->par.MPIAlreadyInit == PGA_FALSE) && i)
  MPI_Finalize();

/*  We really should perform a PGADebugPrint here, but we can't;
 *  we've already deallocated most of the stuff we need!!
 */
free ( ctx );
}
```