Strathmore University
SU+ @ Strathmore
University Library

Electronic Theses and Dissertations

2018

Distributed fuzzing for software vulnerability discovery

Farhiya O. Maalim
Faculty of Information Technology (FIT)
Strathmore University

Follow this and additional works at https://su-plus.strathmore.edu/handle/11071/5988

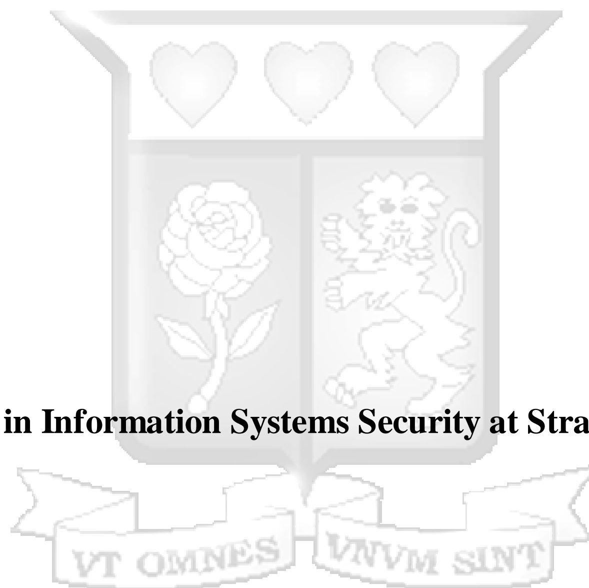
Recommended Citation

Maalim, F. O. (2018). *Distributed fuzzing for software vulnerability discovery* (Thesis). Strathmore University. Retrieved from https://su-plus.strathmore.edu/handle/11071/5988

This Thesis - Open Access is brought to you for free and open access by DSpace @Strathmore University. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of DSpace @Strathmore University. For more information, please contact librarian@strathmore.edu

Distributed Fuzzing for Software Vulnerability Discovery

Maalim, Farhiya Osman



Master of Science in Information Systems Security at Strathmore University

2018

Distributed Fuzzing for Software Vulnerability Discovery

Maalim, Farhiya Osman

067111

A Dissertation Proposal Submitted in partial fulfilment of the requirements for the Degree of Master of Science in Information Systems Security at Strathmore University

Faculty of Information Technology

Strathmore University

Nairobi, Kenya

VT OMNES UNUM SINT

April 18

Declaration

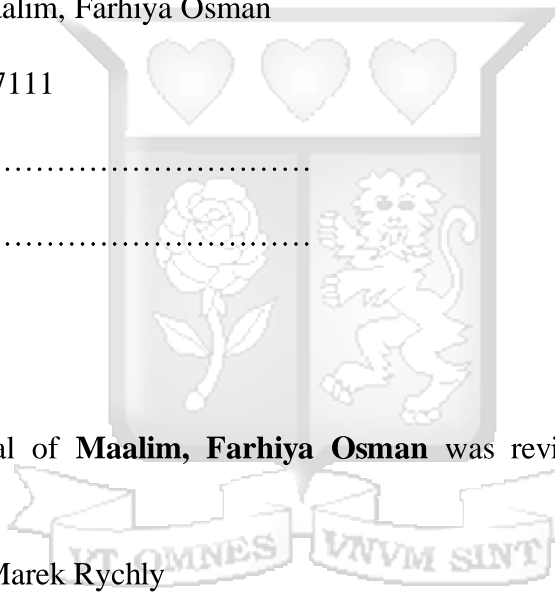
I declare that this work has never been previously submitted and approved for the award of a degree by Strathmore University or any other university. To the best of my knowledge and belief, this dissertation proposal contains no material previously published or written by another person except where due reference is made in the proposal itself.

Student Name Maalim, Farhiya Osman

Student Number 067111

Signature ....

Date ....



Approval

This dissertation proposal of **Maalim, Farhiya Osman** was reviewed and approved by the following:

Supervisor Name Dr Marek Rychly

Faculty Affiliation Faculty of Information Technology

Institution Brno University of Technology

Signature ....

Date ....

VT OMNES
VNVM SINT

VT OMNES VNVM SINT

VT OMNES
VNVM SINT

Abstract

Information Security is concerned with effectively protecting the confidentiality, integrity and availability of data. Software bugs/defects threaten these three elements of information security. By failing to identify and focus upon the root causes of risks such as software vulnerabilities, there is a danger that the response to Information Security compromises become solely reactive.

Fuzzing is a software testing technique that is used to discover software vulnerabilities. The project undertaken is a Distributed Fuzzer that runs on multiple computing environments in the cloud. The advantage of distributed fuzzing compared to regular fuzzing is the ability to run multiple test cases concurrently thus increasing the efficiency of fuzzing.

The aim of this project is to improve fuzzing in order to increase the efficiency of discovering vulnerabilities and software defects. This will ultimately increase the security of a software/application.

The research study was accomplished by using Ansible as a system orchestration tool to run AFL Fuzzers on multiple computing environments in the cloud. The results were collected and presented in this study.

Keywords: fuzzer, distributed fuzzer, fuzzing, scalability

VT OMNES VNVM SINT

Table of Contents

Abstract iii

Table of Contents iv

List of Figures vii

List of Tables viii

List of Abbreviations ix

Chapter 1: Introduction 1

1.1 Background Information 1

1.2 Problem Statement 2

1.3 Research Objectives 2

1.4 Research Questions 3

1.5 Scope and Limitations 3

1.6 Research Relevance... 3

Chapter 2: Literature Review 4

2.1 What is Fuzzing? 4

2.2 The Importance of Fuzzing 4

2.3 The Importance of Distributed Fuzzing 5

2.4 Benefits of Fuzzing 6

2.5 Fuzzing categories 6

2.6 Features of a Fuzzer 6

2.7 Fuzzing Stages... 6

2.8 Ansible Tool for System Orchestration 6

2.9 American Fuzzy Lop (AFL) fuzzer 7

2.10 Previous research done 7

2.12 Summary 7

Chapter 3: Research Methodology... 8

3.1 Introduction 8

3.2 Requirements Analysis 8

1.3 System Design and Architecture 8

1.4 Experimental/ System Implementation 9

1.5 Testing and validation methodology 10

3.5.1 Functional Testing 10

3.5.2 Non-functional requirements 10

Chapter 4: System Design and Architecture 11

4.1 Introduction 11

4.2 Proposed System Modules 11

4.3 Logical System design 11

4.3.1 Use case diagram 11

4.3.2 Sequence Diagram 12

4.3.3 Flowchart 14

4.4 Security design 15

Chapter 5: System implementation and Testing 16

5.1 Overview 16

5.2 Configuration of AWS Instances 16

5.3 Configuration of Ansible 18

5.4 Setup and running of AFL Fuzzers on AWS Instances using Ansible 18

5.5 Results 23

5.6 Testing 26

5.6.1 Non-functional Testing 26

5.6.2 Functional Testing 27

Chapter 6: Discussion of results 28

Chapter 7: Conclusions 29

7.1 Future Work 29

Chapter 8: References 30

List of Figures

Figure 1-1: History of Fuzzing *1*

Figure 4-1: Use Case Diagram *12*

Figure 4-2: Sequence Diagram *13*

Figure 4-3: Flowchart *14*

Figure 5-1: Creating Instances *16*

Figure 5-2: Select Instance OS *17*

Figure 5-3: Selecting number of instances *17*

Figure 5-4: List of VMs *18*

Figure 5-5: Identify Hosts IP Addresses *18*

Figure 5-6: Install AFL Playbook *19*

Figure 5-7: Results of Install AFL Playbook *19*

Figure 5-8: Run AFL Playbook *20*

Figure 5-9: Transfer of demo file to all VMs *21*

Figure 5-10: Target Input File *21*

Figure 5-11: First Testcase *22*

Figure 5-12: Second Testcase *22*

Figure 5-13: Third Testcase *22*

Figure 5-14: Fourth Testcase *22*

Figure 5-15: Fifth Testcase *22*

Figure 5-16: Results of AFL on Local Machine *23*

Figure 5-17: Fuzzer stats from first VM *24*

Figure 5-18: Fuzzer stats from Second VM *24*

Figure 5-19: Fuzzer stats from Third VM *25*

Figure 5-20: Fuzzer stats from Fourth VM *25*

Figure 5-21: Fuzzer stats from Fifth VM *26*

Figure 5-22: Logging into VM with SSH *27*

List of Tables

Table 2-1: The exponential rise cost in correcting defects *4*

VT OMNES VNVM SINT

List of Abbreviations

AFL - American Fuzzy Lop

VPS - Virtual Private Server

SSH **-** Secure Shell

AWS - Amazon Web Services

VM -Virtual Machine

OS - Operating System

VT OMNES
VNVM SINT

Chapter 1: Introduction

1.1 Background Information

Fuzz testing, or Fuzzing, is an automated or semi-automated process that involves repeatedly manipulating and supplying data to target software until a vulnerability is discovered. It is a software security testing method that discovers vulnerabilities by providing unexpected input and monitoring for exceptions (Sutton, Green, & Amini, 2007). The definition of distributed fuzzing is the application of distributed computing for the use of fuzzing (Doyle, Fly, Maynor, Miller, & Naveh, 2011).

In 1989, a professor from the University of Wisconsin introduced the term fuzzing for the very first time whilst testing the robustness of a UNIX System (Sutton, Green, & Amini, 2007). Fuzzing went on to be used by multiple security researchers and software testers over the years. The image below depicts the history of fuzzing from its infancy in 1989 up to 2007.

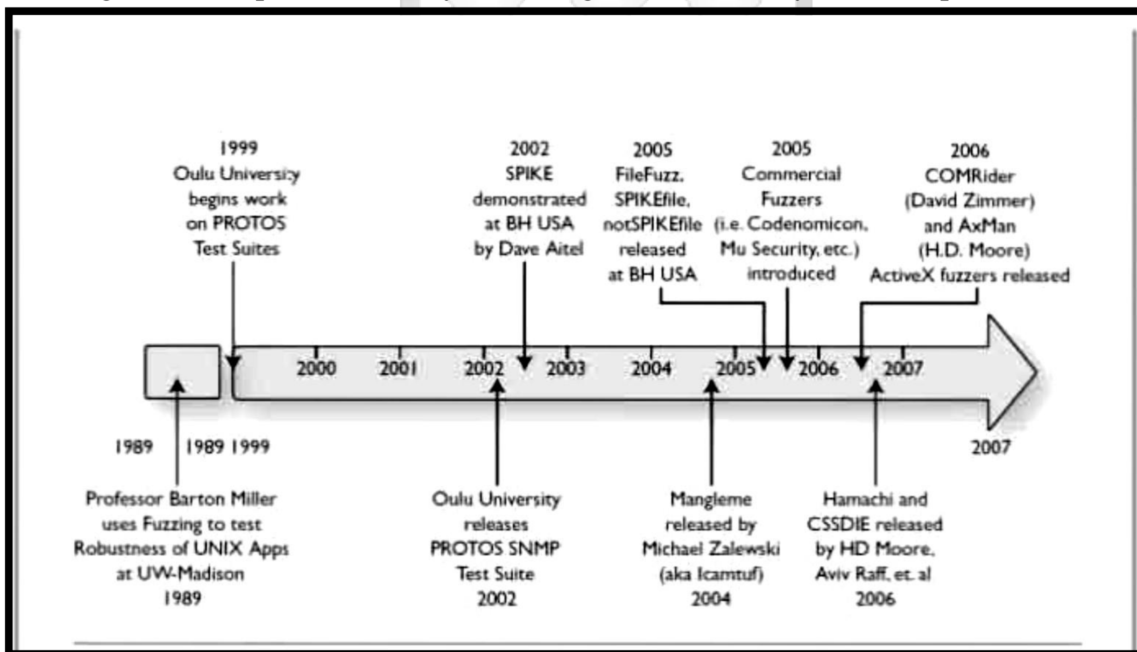


Figure 1-1: History of Fuzzing

(Sutton, Green, & Amini, 2007)

Various testing methodologies exist that discover vulnerabilities in software/ web applications. The three main methodologies are White box, Black box and Grey box methodologies. Whitebox methodology is a software testing method in which the internal structure, design and implementation of the item being tested is known to the tester (White Box Testing Fundamentals, 2018).

Black box testing a software testing method in which functionality of the software under test is tested without looking at the internal code structure, implementation details and knowledge of internal paths of the software (What is BLACK Box Testing? Techniques, Example & Types, 2018).

Grey box testing, on the other hand, is a combination of both white box and black box methodologies. The tester only partially knows the internal structure and design of the software. Fuzzing is a software testing tool that is categorised under the Black box testing methodology. Despite the advances made thus far, fuzzing is a relatively new technology that will see many further innovations.

Information Security is concerned with effectively protecting the confidentiality, integrity and availability of data. Software bugs/defects threaten these three elements of information security. Software testing is a highly complex but important element that should be integrated into the software development life cycle. Software testing for security is usually perceived as an afterthought by software developers, which leads to software defects and vulnerabilities that can be exploited by attackers.

The purpose of the proposed project is to apply distributed computing to carry out fuzzing in order to increase efficiency. The advantage of distributed fuzzing compared to regular fuzzing is the ability to run multiple test cases concurrently thus increasing the efficiency of fuzzing. This will increase the effectiveness of fuzzing to discover vulnerabilities, leading to less runtime and increased discovery of vulnerabilities.

VT OMNES VNVM SINT

1.2 Problem Statement

The purpose of fuzzing is to test the integrity of a software application by adding/removing random files, data, or other information to/from the software application. Unfortunately, utilising a single computing resource to carry out fuzzing can take many hours or days. This leads to a delay in results of the fuzzing. This is a problem that plagues software developers that would like a quick and efficient way of testing for defects and the security of their software.

1.3 Research Objectives

i. To understand previous research undertaken in Distributed Fuzzing.

ii. To identify the gap in the current approaches. iii. To develop and test a distributed fuzzing solution. iv. To validate the solution.

1.4 Research Questions

i. What are the gaps and solutions related to the problem?

ii. What have other researchers done in relation to the problem? iii. How do I develop and design a distributed fuzzing solution to solve the problem? iv. Does the system developed solve the problem?

1.5 Scope and Limitations

The distributed fuzzing project will be limited to fuzzing executable applications running on the Ubuntu Operating System.

1.6 Research Relevance

The researched work will be useful to software testers and developers that would like an efficient way of testing for vulnerabilities and defects.

Chapter 2: Literature Review

2.1 What is Fuzzing?

The discovery of fuzzing as a means to test software reliability is captured in a paper written in 1989 by authors Miller, So and Fredriksen (Clarke, 2009).

Fuzz testing, as described by Michael Sutton is an automated or semi-automated process that involves repeatedly manipulating and supplying data to target software until a vulnerability is discovered. It is a software security testing method that discovers vulnerabilities by providing unexpected input and monitoring for exceptions (Sutton, Green, & Amini, 2007).

Fuzzing is the process of sending intentionally malformed inputs to a piece of software to see if it fails. Each malformed input is a test case. Failure indicates a found bug, which can then be fixed to improve the robustness and security of the target software (What is Fuzzing: The Poet, the Courier and the Oracle, 2017).

2.2 The Importance of Fuzzing

The software development process does not produce secure software applications by default. Historically, according to (Clarke, 2009), this has been due to a number of factors, including:

the increasing level of software complexity; the use of unmanaged programming languages such as C and C++, which offer flexibility and performance over security, a lack of secure coding expertise, due to a lack of training and development and users having no awareness of application security.

Clarke (Clarke, 2009) points out that the cost of correcting software defects rises exponentially as the development stages are completed as is shown in Table 2-1.

Table 2-1: The exponential rise cost in correcting defects

Phase	Relative Cost to Correct
Definition	$1
High-Level Design	$2
Low-Level Design	$5
Code	$10
Unit Test	$15
Integration Test	$22
System Test	$50

Post-Delivery	$100

By failing to identify and focus upon the root causes of risks such as software vulnerabilities there is a danger that the response to Information Security becomes solely reactive (Clarke, 2009).

Fuzz testing is a method that is relatively cheap, requires minimal expertise, can be largely automated, and can be performed without access to the source code, or knowledge of the system under test (Clarke, 2009).

2.3 The Importance of Distributed Fuzzing

Typical fuzz testing is scalable, automatable and does not require access to the source code. It simply feeds malformed inputs to a software application and monitors its failures. Yet it also suffers from several problems: a single unsigned int value can vary from 0 to 65535, indicating the immensity of the input space, which can hardly be covered with limited time and cost (Dai, Murphy, & Kaiser, 2010).

The importance of distributed fuzzing is to address the limitation described above. Distributed fuzzing will have the ability to run multiple test cases concurrently thus decreasing the time taken to run a test case and the cost.

Doyle terms distributed fuzzing as the application of distributed computing for the use of fuzzing (Doyle, Fly, Maynor, Miller, & Naveh, 2011). The process of distributed computing the fuzzing jobs is described as dividing fuzzing jobs between different attacking clients and servers, covering more of the combination space in a significantly shorter time frame (Rathaus & Evron, 2007).

Fuzzing faces many issues, as described by Conger. Fuzzing tasks may be aborted so that an associated resource can be freed-up. Adding to the issue, separate fuzzing outputs are usually reviewed manually, which can occupy many hours of a reviewer's time (U.S. Patent No. US 7743281 B2, 2010). A useful fuzzer must keep records, produce actionable reports, and provide a smooth remediation process to reproduce failures so that they can be fixed (What is Fuzzing: The Poet, the Courier and the Oracle, 2017).

Proactive security testing approaches include fuzzing, protocol mutation, robustness testing, and the like. Fuzzing is a very effective way of discovering software vulnerabilities, as it requires no intelligence of the internal operations of the device or system under test (Takanen, 2010).

2.4 Benefits of Fuzzing

Some of the major benefits of fuzzing includes ensuring high throughput with less manual efforts and pre-knowledge of the target software (Xu, Kashyap, Min, & Kim, 2017) , bugs found by fuzzers could be exploitable by hackers and can often fall into areas overlooked by testers or areas that are omitted due to time and resource constraints (Pierce, 2012).

Fuzzing techniques allows detection of almost all types of security vulnerabilities including, but not limited to, buffer overflows, integer overflows, format string vulnerabilities, Race condition vulnerabilities, SQL injection amongst many others (Juranić, 2006).

2.5 Fuzzing categories

Fuzzers fall into 5 main categories, which are *Mutation-based fuzzers*: they apply mutations on existing data samples to create test cases, *Generation-based fuzzers*: they create test cases from scratch by modelling the target protocol or file format (Sutton, Green, & Amini, 2007), *Replay fuzzer*: they take saved sample inputs and simply replay them after mutating them, *Man-in-theMiddle or Proxy*: they set your fuzzer up as a proxy and mutate requests or responses depending on whether you are fuzzing the server or the client and *Evolutionary fuzzing*: an advanced technique that allows the fuzzer to use feedback from each test case to learn over time the format of the input (Hillman, 2013)

2.6 Features of a Fuzzer

All fuzzers share a similar set of features, namely: Data generation (creating data to be passed to the target); Data transmission (getting the data to the target); Target monitoring and logging (observing and recording the reaction of the target), and Automation (reducing, as much as possible, the amount of direct user-interaction required to carry out the testing regime) (Clarke, 2009).

2.7 Fuzzing Stages

Sutton has listed the stages of fuzzing as being: Identify target, Identify inputs, Generate fuzzed data, Execute fuzzed data, Monitor for exceptions and Determine exploitability (Sutton, Green, & Amini, 2007).

2.8 Ansible Tool for System Orchestration

Ansible is a consistent, reliable and secure way to manage a computing environment. Ansible configurations are simple data descriptions of your infrastructure (both human-readable and machine-parsable) (Configuration Management, 2017).

Ansible relies on the most secure remote configuration management system available as its default transport layer: OpenSSH. OpenSSH is available for a wide variety of platforms, is very lightweight and when security issues in OpenSSH are discovered, they are patched quickly (Configuration Management, 2017).

2.9 American Fuzzy Lop (AFL) fuzzer

American fuzzy lop, a tool developed by lcamtuf, is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. This substantially improves the functional coverage for the fuzzed code (American Fuzzy Lop (2.52b), n.d.).

It uses binary instrumentation (a method that adds instrumentation code by modifying or rewriting compiled code) to create edge-cases for a given software. AFL is the tool that will be utilised to run the test cases on the distributed computing environments.

2.10 Previous research done

Conger, Srinivasamurthy and Cooper filed a patent in 2007 on the invention of embodiments to provide distributed file fuzzing functionality. In an embodiment, a number of computing devices can be used as part of a distributed fuzzing environment. Fuzzing work can be distributed to the number of computing devices and processed accordingly. A group or team can be defined to process particular fuzzing operations that may be best suited to the group. A client can be associated with each computing device and used in conjunction with fuzzing operations (U.S. Patent No. US 7743281 B2, 2010).

Cloud fuzzing, a similar project done by Kirsch (Kirsch, 2017), used a tool called softScheck Cloud Fuzzer Framework to run the AFL Fuzzer on the cloud.

2.12 Summary

The project used Ansible as a means of system orchestration (automation) of the distributed computing resources. It was used as a configuration trigger of the distributed fuzzer on many hosts. AFL was the fuzzer utilised to run the test cases on the distributed computing environments. The works of Sutton and Clarke was taken into consideration in my research. The works of Conger, Srinivasamurthy and Cooper was also largely taken into consideration.

The gaps identified in the research of fuzzing is applying distributed computing to increase the efficiency of fuzzing as opposed to regular fuzzing.

Chapter 3: Research Methodology

3.1 Introduction

The below chapter describes the methodology that was utilised in the completed project. The methodology used to implement the project was Experimental Simulation.

3.2 Requirements Analysis

The objective of this phase is to define in more detail the system inputs, processes, outputs and interfaces (University of Connecticut, 2017).

Below were the steps taken to define and analyse the requirements of the distributed fuzzer:

1. Identify target

The user will identify the target input file. For the purpose of this research study, the target identified was executable applications (Executable and Linkable Format) in the Ubuntu Operating System.

The Executable and Linkable Format, also known as ELF, is the generic file format for executables in Ubuntu systems. Generally speaking, ELF files are composed of three major components: **ELF Header**, **Sections** and **Segments** (Sanmillan, 2018).

2. Identify inputs

The second stage is to the user identifying the input vectors to be utilised for the fuzzers. The identified input vectors are C binary files. The following steps will be taken by the user when carrying out fuzzing of the files:

- Identify areas of code to receive fuzzing
- Expand use cases to improve code coverage

1.3 System Design and Architecture

System design is the process of defining the elements of a system such as the architecture, modules and components, the different interfaces of those components and the data that goes through that system (System Design, 2018).

The development of the distributed fuzzer was guided by the following features:

i. Data generation (creating data to be passed to the target);

ii. Data transmission (getting the data to the target); iii. Target monitoring and logging (observing and recording the reaction of the target), and; iv. Automation (reducing, as much as possible, the amount of direct user-interaction required to carry out the testing regime)

The physical architecture of the system was guided by the following architectural diagrams:

i. Use Case Diagram: It describes a set of actions that some systems should perform in collaboration with one or more external users of the system (UML Use Case Diagrams, 2017).
ii. Sequence Diagram: It describes an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding occurrence specifications on the lifelines (UML Sequence Diagrams, 2017).
iii. Flowchart: A diagram that represents a process undertaken by a user of a system

The diagrams mentioned above were created using Microsoft Visio (Microsoft Visio, 2018).

1.4 Experimental/ System Implementation

The fuzzer was run on one main host/computer that had Ansible *(an automation engine that automates cloud provisioning, configuration management and application deployment)* installed on it. Multiple fuzzers were replicated on a VPS forming a distributed fuzzing cluster. Ansible was used to automate the fuzzers which ran multiple test cases concurrently, which increased the efficiency of fuzzing.

The fuzzer has the following functionalities:

1. Generate fuzzed data

Once input vectors have been identified, fuzz data must be generated. Mutations will be applied in the identified input described in the phase above to create test cases.

2. Execute fuzzed data

This phase is where fuzzing becomes a verb. Execution will involve the act of sending a data packet to the target (identified in the phase above).

Described below are the tools and software that was utilised to run the distributed fuzzer:

a. Ubuntu Operating System 16.04.4 (Ubuntu OS, 2018)
b. Ansible (Ansible, 2017)
c. Digital Ocean VPS to run and scale multiple computing environments with multiple test cases (Digital Ocean, 2018)

VT OMNES VNVM SINT

   d. AFL fuzzer (American Fuzzy Lop (2.52b), n.d.)

3. Determine exploitability

Once a fault is identified, it will be necessary to determine if the uncovered bug can be further exploited (Sutton, Green, & Amini, 2007).

1.5 Testing and validation methodology

3.5.1 Functional Testing

Functional requirements capture the intended behavior of the system. This behavior may be expressed as services, tasks or functions the system is required to perform. (Malan & Bredemeyer, 2001)

Described below are the tests that were undertaken in functionality testing, in order to ensure that it conforms to functional requirements:

1. Installation and setup of multiple computing environments on the VPS to run the fuzzers
2. Check the monitoring and logging functions of the fuzzer:
   This is a vital step where the applications will be monitored for exception or fault monitoring. By categorising crashes, I can identify when one test case is triggering the same bug as another and only keep the cases relating to unique crashes (Hillman, 2013).

3.5.2 Non-functional requirements

Non-functional requirements or system qualities, capture required properties of the system, such as performance, security, maintainability (Malan & Bredemeyer, 2001).

Described below are the tests that were undertaken in non-functional testing, in order to ensure that it conforms to non-functional requirements:

1. Stress Testing of the file fuzzers by increasing the number of test cases and monitoring the amount of time taken and number of cycles
2. Security testing of the fuzzers to ensure that the distributed fuzzing clusters are secure

After the distributed fuzzers are installed and running, validation must take place to ensure that the performance, security and reliability of the distributed fuzzing cluster. The results generated by the distributed fuzzing cluster must be unique in order to avoid one test case triggering the same bug as another.

Chapter 4: System Design and Architecture

4.1 Introduction

The purpose of the design phase is to decide how to build it. System design is the determination of the overall system architecture—consisting of a set of physical processing components, hardware, software, people, and the communication among them—that will satisfy the system's essential requirements.

The purpose of this chapter is to describe the system design of the distributed fuzzer. Multiple fuzzers were replicated on a VPS forming a distributed fuzzing cluster, which will be automated by a software named Ansible.

4.2 Proposed System Modules

The system modules are:

i. Ansible module: determine the number of hosts for system orchestration

ii. Test cases: The testcases are supplied by the user and contain sample input data. This is required by the fuzzer

iii. AFL fuzzer input: in this module, the user will input the test case. The fuzzer will instrument the file and proceed to repeatedly mutate the file

iv. AFL Fuzzer output: this module will collect 3 outputs:

   a. Unique crashes: unique test cases that cause the tested program to receive a fatal signal (e.g., SIGSEGV, SIGILL, SIGABRT). The entries are grouped by the received signal.

   b. Hangs: unique test cases that cause the tested program to time out.

   c. Queue: test cases for every distinctive execution path, plus all the starting files given by the user.

4.3 Logical System design

4.3.1 Use case diagram

Use case diagrams are used to describe a set of actions that some systems should or can perform in collaboration with one or more external users of the system (actors). Each use case should provide some observable and valuable result to the actors or other stakeholders of the system (UML Use Case Diagrams, 2017).

UT OMNES VNVM SINT

Figure 4-2 below shows the processes (use cases) actors of the system should perform. There are 3 main actors of the system, which are:

1. User (this could be a software developer or security tester)
2. Fuzzer (the AFL Fuzzer in this case)
3. Ansible (the system orchestration tool)

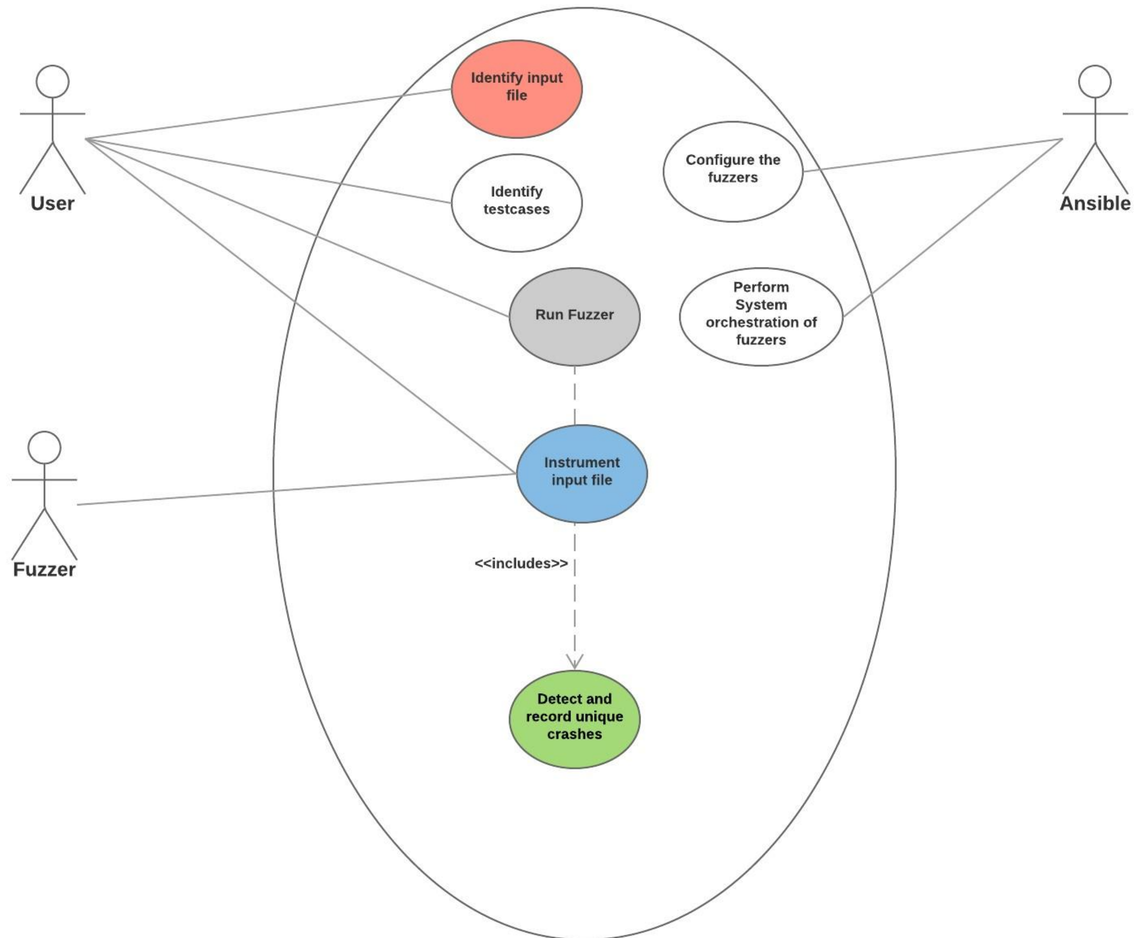


Figure 4-1: Use Case Diagram

4.3.2 Sequence Diagram

Sequence diagrams specifically focus on *lifelines*, or the processes and objects that live simultaneously, and the messages exchanged between them to perform a function before the lifeline ends (UML Sequence Diagram Tutorial, 2018). Figure 4-3 below depicts the processes and users of the system.

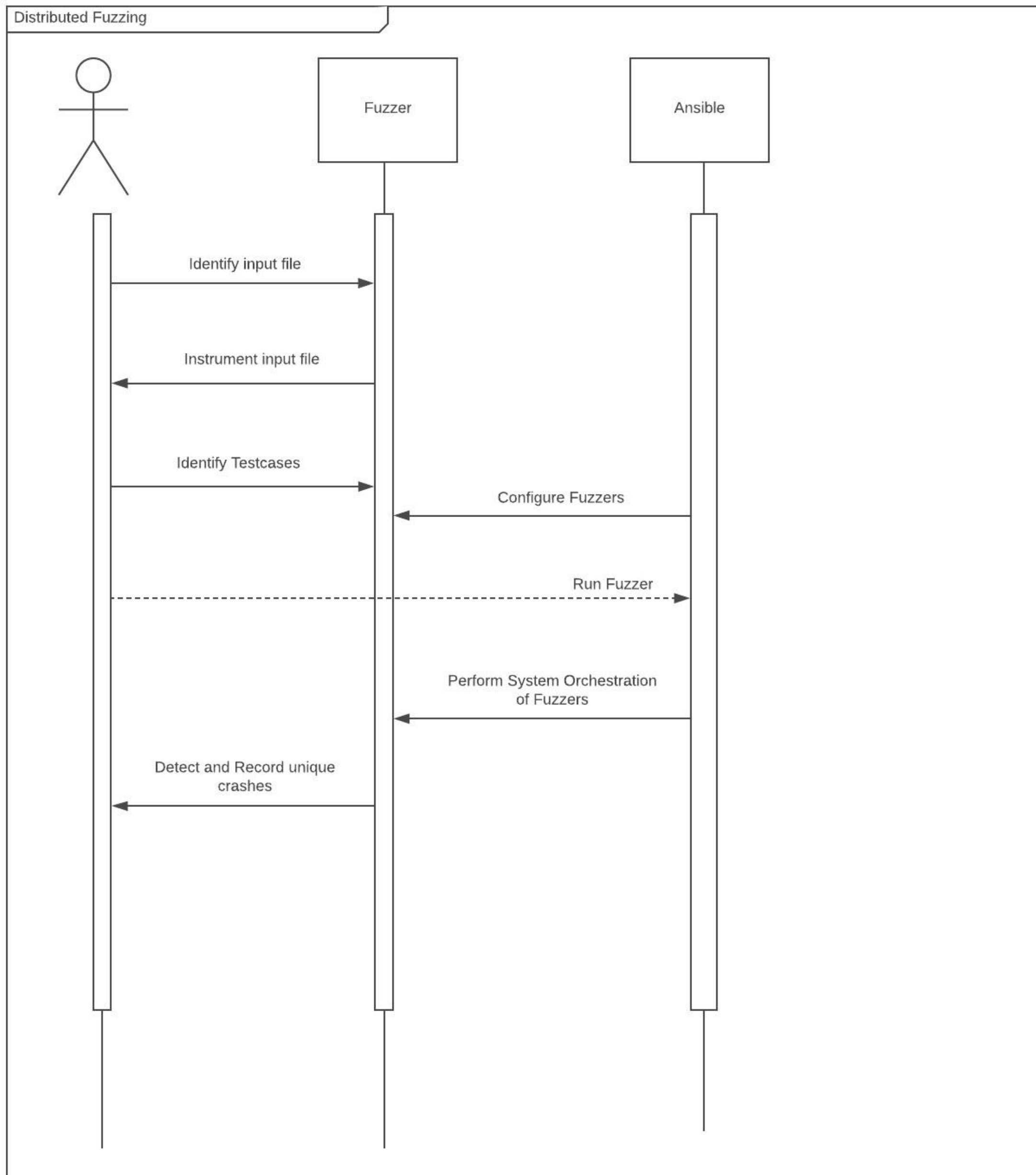


Figure 4-2: Sequence Diagram

The sequence diagram in Figure 4-3 explains the processes implemented by the user. The user will first will select an input file. The user will then select testcases to fuzz the input file. Once this is done, the target file will be instrumented by AFL. After Ansible has configured the fuzzers, the user then responds by running the fuzzers. Ansible will perform system orchestration of the fuzzers on the instances. The fuzzer will then detect and record unique crashes.

4.3.3 Flowchart

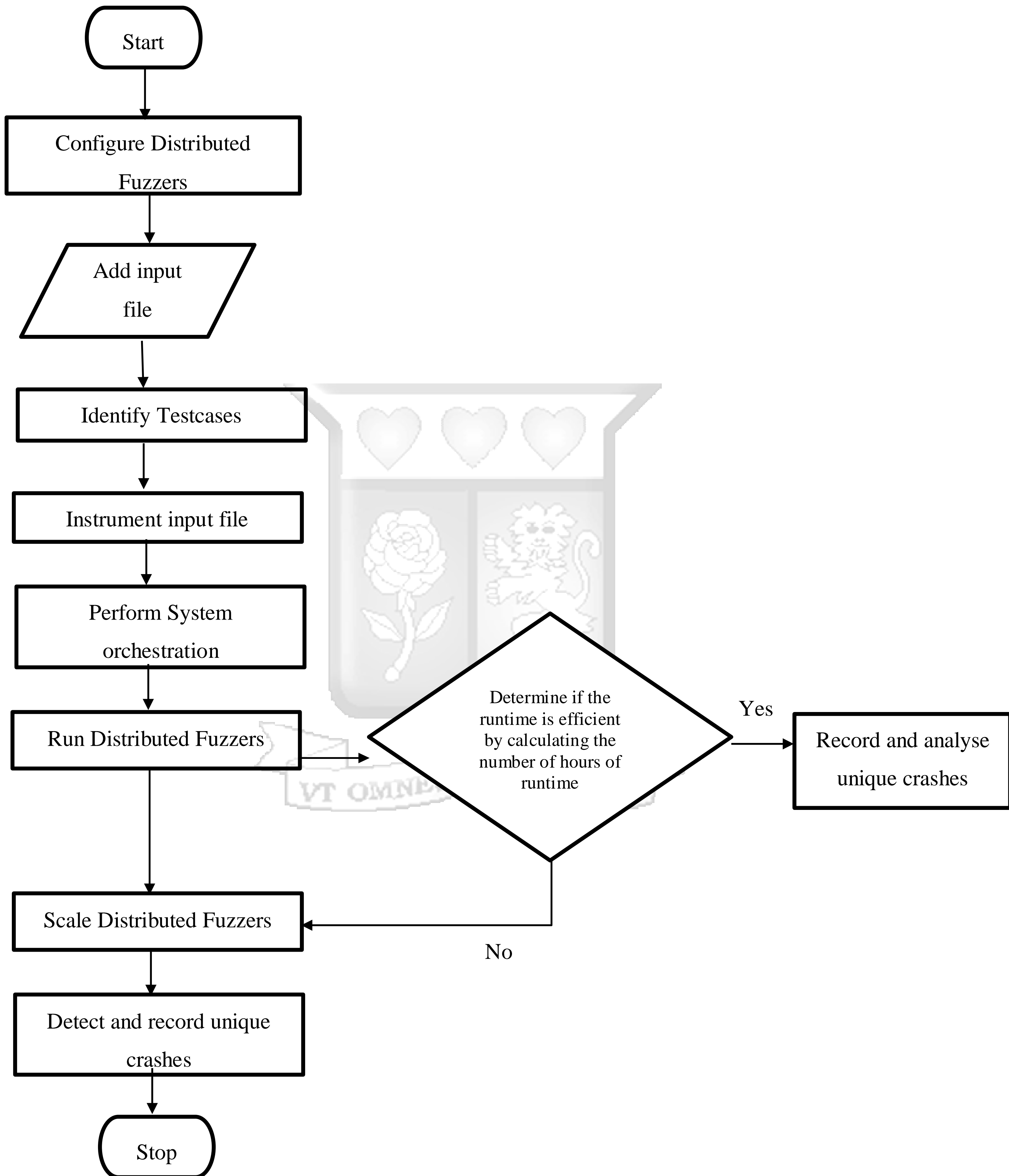
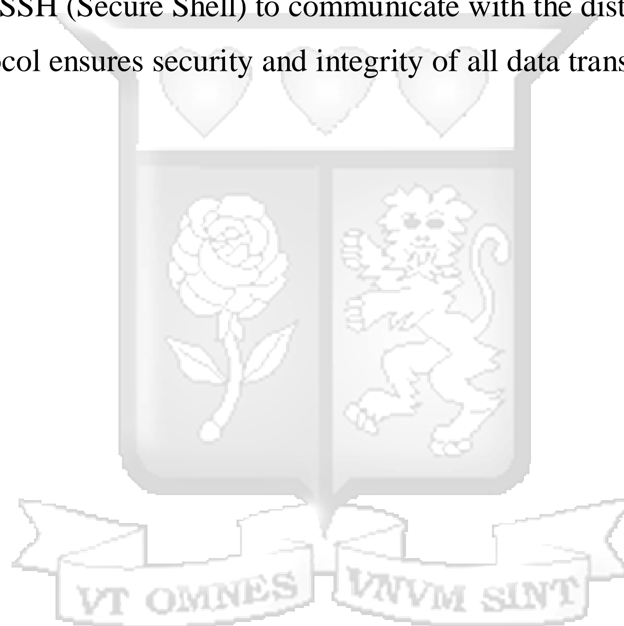


Figure 4-3: Flowchart

The flowchart above describes the flow of processes taken by the user of the system. After configuration of the instances, Ansible and the Fuzzers, the user will select an input file. This is the target file that will be fuzzed. The user will the select testcases to fuzz the input file. Once this is done, the target file will be instrumented by AFL. The fuzzers will then be run by Ansible. The amount of time taken will be calculated and made to be as efficient as possible. If it is, the user will analyse the recorded unique crashes of the system. If the system is still not efficient enough, the user will run scale the number of instances to increase efficiency of the runtime of the fuzzers.

4.4 Security design

Security principle adopted to the Distributed Fuzzing System is Data Protection. Ansible tool uses a protocol called SSH (Secure Shell) to communicate with the distributed fuzzing clusters on the VPS. The protocol ensures security and integrity of all data transmitted using strong encryption.



Chapter 5: System implementation and Testing

5.1 Overview

This chapter describes and depicts the results derived from the distributed fuzzing clusters. The distributed fuzzing clusters were created with the following tools:

i. Ansible (a tool that provides system orchestration)

ii. AFL Fuzzer (the fuzzer utilized for this project)

iii. Ubuntu 16.0.4 LTS (the target operating system)

iv. C++ File (the target input file)

v. Amazon Web Services

5.2 Configuration of AWS Instances

First, the VM Instances need to be setup. As depicted below in Figure 5-1.

I created an account on the Amazon Web Services, a cloud services platform and launched instances (virtual computing environments).

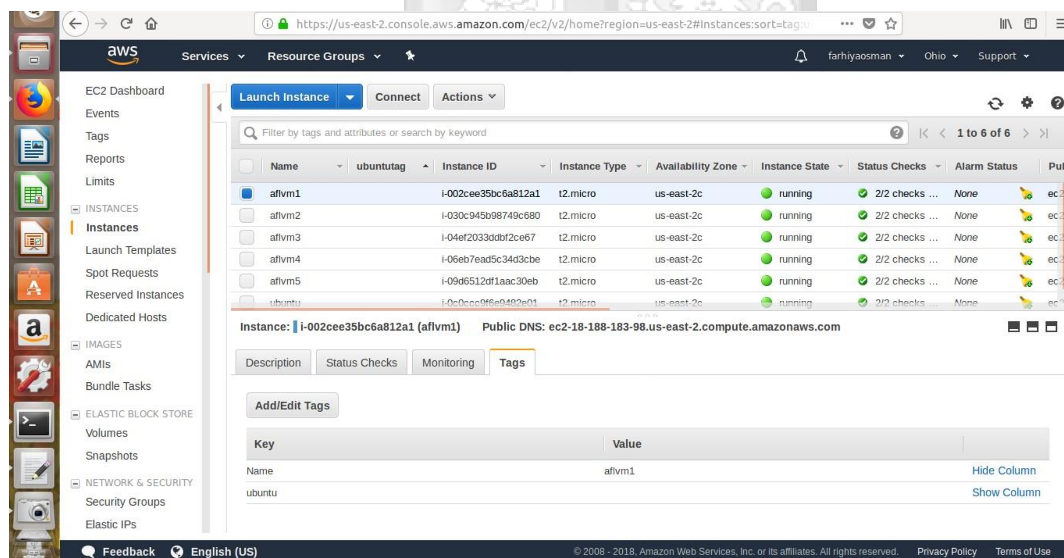


Figure 5-1: Creating Instances

Once on the Instances Webpage, I clicked Launch Instance to create an instance. I selected the Ubuntu Server 16.04 LTS, as this Operating System was selected for the scope of this project.

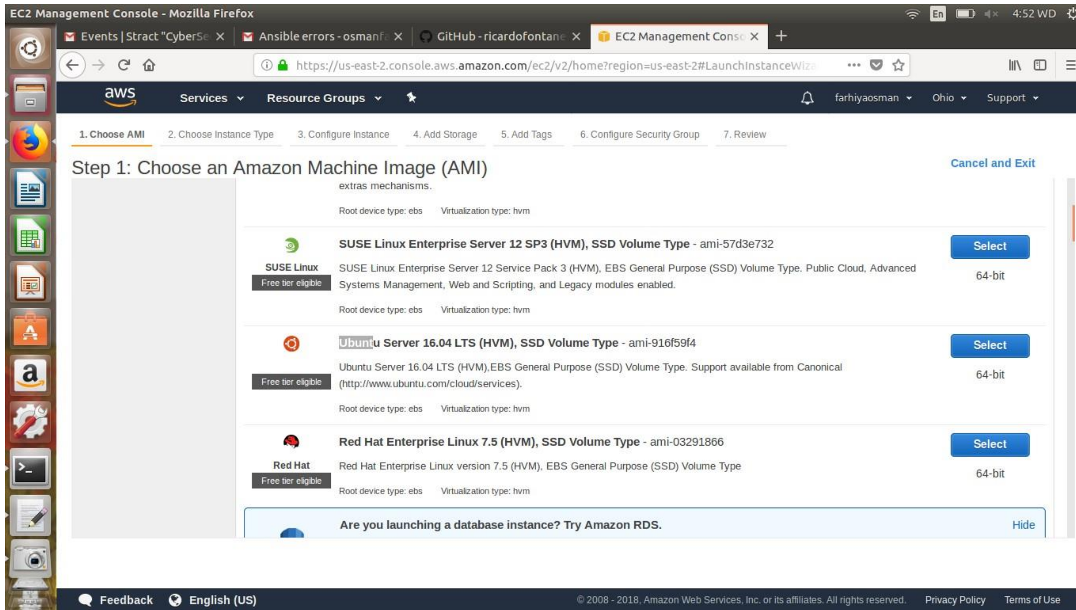


Figure 5-2: Select Instance OS

After selecting the Ubuntu Server, I configured the following components, as depicted below:

i. Number of instances: 5 with the option of ***Auto-scaling*** to a maximum of 10 instances. This was configured as depicted in Figure 5-6 below.

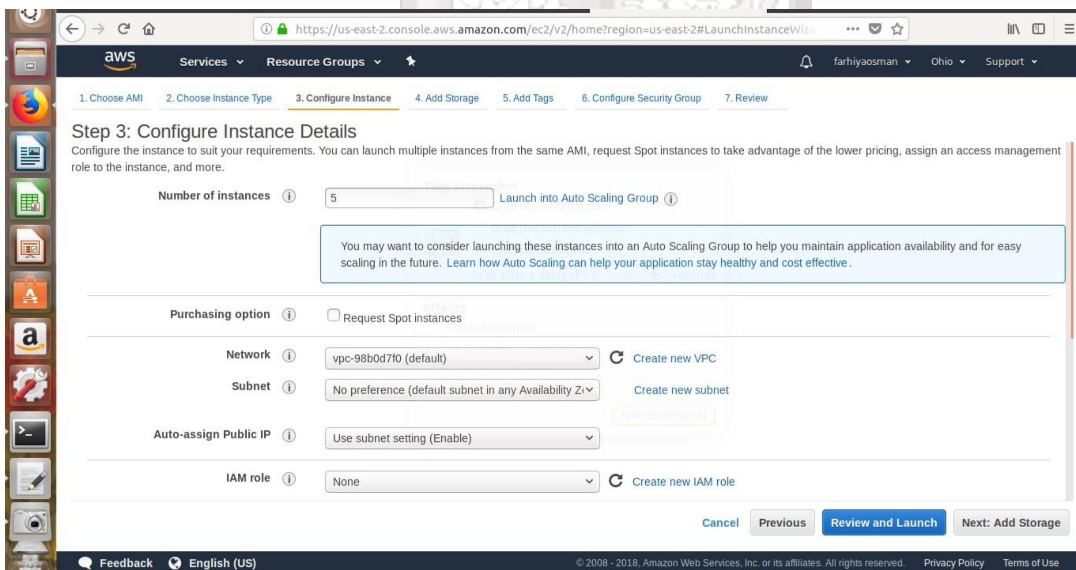


Figure 5-3: Selecting number of instances

After the 5 instances have been successfully launched, I renamed each instance to AFLVM1 (AFL Virtual Machine 1) up to AFLVM5. The purpose of renaming them is to easily identify them when configuring Ansible to connect with the VM's.

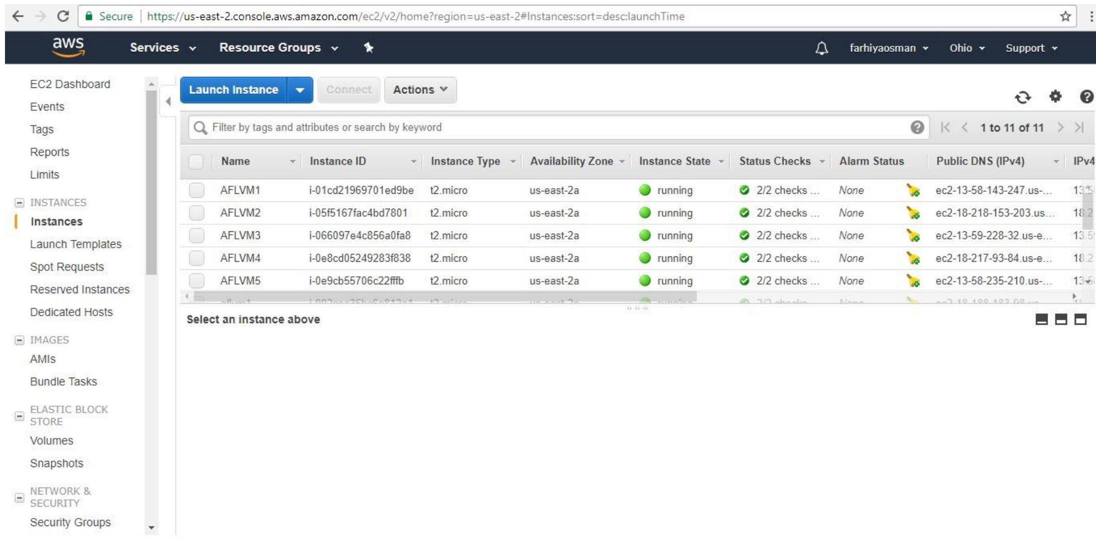


Figure 5-4: List of VMs

After successfully configuring the instances to run the AFL Fuzzer, configuration of Ansible on the local machine comes next.

5.3 Configuration of Ansible

Ansible is the tool that will be utilized to automate the fuzzers on the Virtual Computing Instances created in the step above.

I first installed Ansible on my local machine.

I identified the hosts IP Addresses from the instances created above as shown in Figure 5-5 below. Ansible will connect with the instances from the hosts file.

Figure 5-5: Identify Hosts IP Addresses

5.4 Setup and running of AFL Fuzzers on AWS Instances using Ansible

Playbooks are Ansible's configuration, deployment, and orchestration language used to describe a set of steps you want your remote system to enforce.The 3 playbooks created are:

a. Install AFL yml

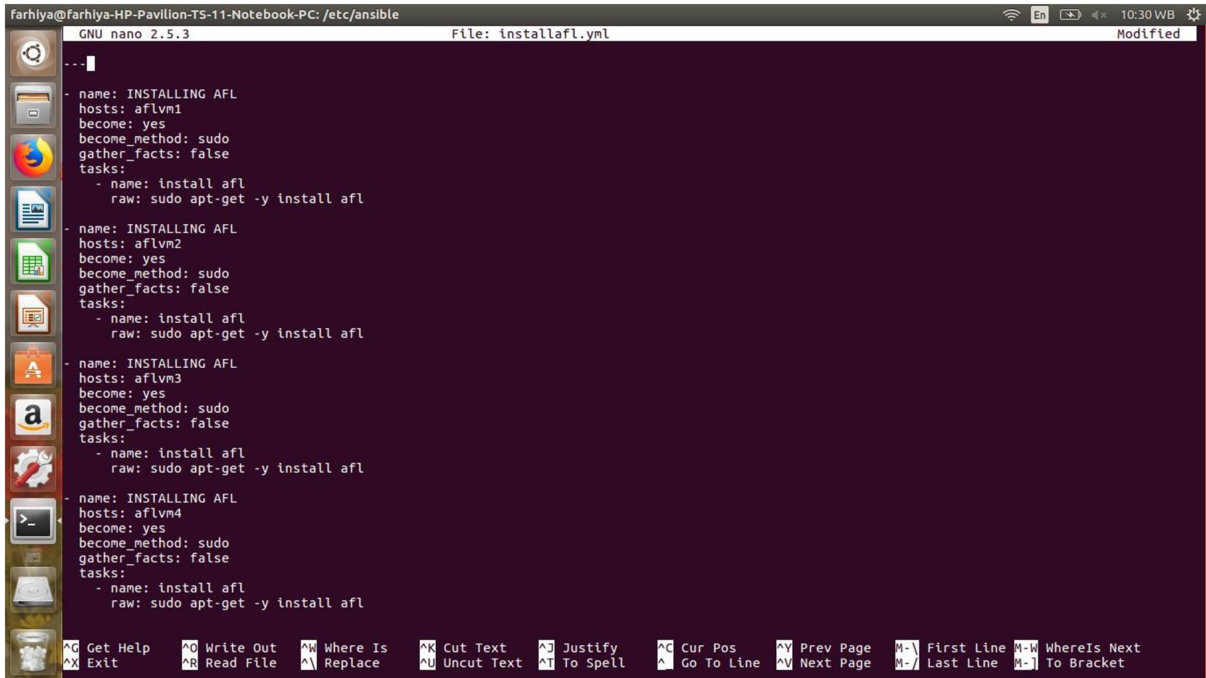


Figure 5-6: Install AFL Playbook

The *installafl.yml* playbook installs AFL Fuzzer on all five VM's.

The screenshot in Figure 5-7 displays the results of the install AFL playbook.

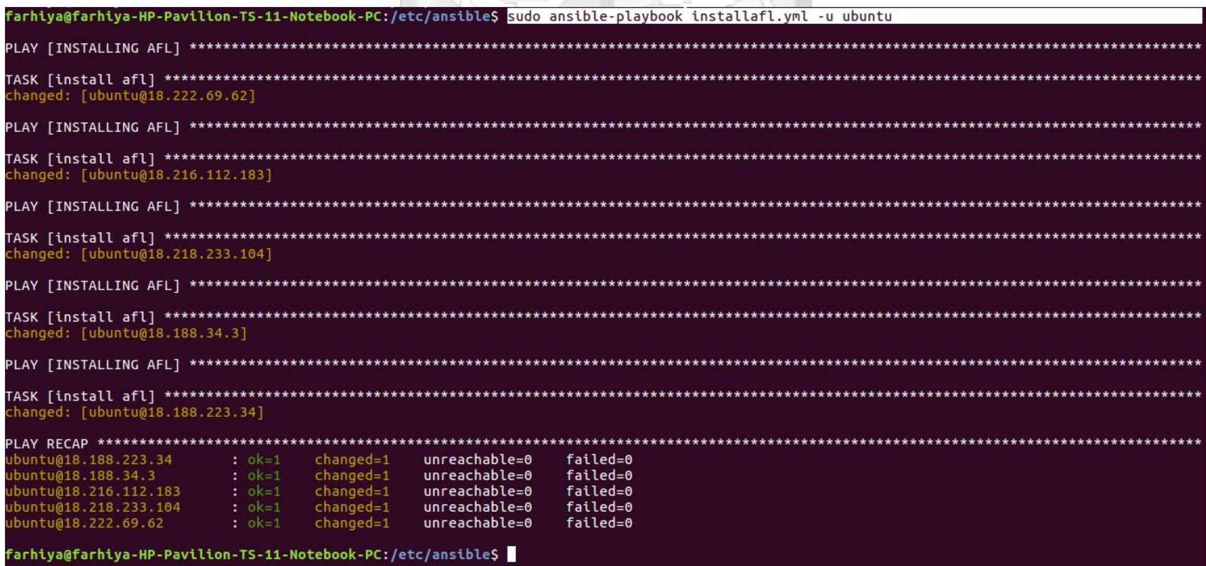


Figure 5-7: Results of Install AFL Playbook

b. Instrument.yml

The *instrument.yml* playbook instruments the demo input file selected for this project. It instruments the demo file on each VM.

```
GNU nano 2.5.3                              File: test.yml                                          Modified

- name: Transfer and execute the first testcase script.
  hosts: aflvm1
  tasks:
     - name: Copy and Execute the script on AFLVM1
       raw: afl-gcc -fno-stack-protector -z execstack demo.c -o demo

 name: Transfer and execute the first testcase script.
  hosts: aflvm2
  tasks:
     - name: Copy and Execute the script on AFLVM2
       raw: afl-gcc -fno-stack-protector -z execstack demo.c -o demo

 name: Transfer and execute the first testcase script.
  hosts: aflvm3
  tasks:
     - name: Copy and Execute the script on AFLVM3
       raw: afl-gcc -fno-stack-protector -z execstack demo.c -o demo

 name: Transfer and execute the first testcase script.
  hosts: aflvm4
  tasks:
     - name: Copy and Execute the script On AFLVM4
       raw: afl-gcc -fno-stack-protector -z execstack demo.c -o demo

 name: Transfer and execute the first testcase script.
  hosts: aflvm5
  tasks:
     - name: Copy and Execute the script on AFLVM5
       raw: afl-gcc -fno-stack-protector -z execstack demo.c -o demo

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos    ^Y Prev Page  M-\ First Line M-W WhereIs Next
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line ^V Next Page  M-/ Last Line  M-] To Bracket
```

c. RunAfl.yml

The playbook depicted in Figure 5-8 runs the fuzzer on each host after successfully instrumenting the target file depicted in the instrument.yml playbook.

```
GNU nano 2.5.3                              File: runafl.yml                                        Modified

- name: Execute 5 different testcases on All  VMs
  hosts: all
  tasks:
     - name: Execute the AFL Fuzzer on each VM running different testcases
       command: afl-fuzz -i testcase -o output ./demo @

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos    ^Y Prev Page  M-\ First Line M-W WhereIs Next
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line ^V Next Page  M-/ Last Line  M-] To Bracket
```

Figure 5-8: Run AFL Playbook

The above playbook runs the testcases (different in each VM) on all hosts (all 5 Virtual Machines). The demo file was copied via SSH to all Virtual Machines as depicted in Figure 5-9.

```
farhiya@farhiya-HP-Pavilion-TS-11-Notebook-PC:~$ cd Downloads/
farhiya@farhiya-HP-Pavilion-TS-11-Notebook-PC:~/Downloads$ sudo scp -i aflproject.pem demo.c ubuntu@18.216.112.183:/home/ubuntu
demo.c                                        100%  322     0.3KB/s   00:00
farhiya@farhiya-HP-Pavilion-TS-11-Notebook-PC:~/Downloads$ sudo scp -i aflproject.pem demo.c ubuntu@18.218.233.104:/home/ubuntu
demo.c                                        100%  322     0.3KB/s   00:00
farhiya@farhiya-HP-Pavilion-TS-11-Notebook-PC:~/Downloads$ sudo scp -i aflproject.pem demo.c ubuntu@18.188.34.3:/home/ubuntu
demo.c                                        100%  322     0.3KB/s   00:00
farhiya@farhiya-HP-Pavilion-TS-11-Notebook-PC:~/Downloads$ sudo scp -i aflproject.pem demo.c ubuntu@18.188.223.34:/home/ubuntu
demo.c                                        100%  322     0.3KB/s   00:00
farhiya@farhiya-HP-Pavilion-TS-11-Notebook-PC:~/Downloads$
```

5-9: Transfer of demo file to all VMs

The input file (target) depicted in Figure 5-9 is a simple C program that contains a buffer overflow vulnerability. A buffer overflow is a common mistake usually made by software developers that results in a program attempting to put more data in a buffer than it can hold. The above input file has 2 inputs: *login* and *passwd*. Both input parameters are passed to a gets () function. This is the mistake made in the program. Gets () has only received the name of the char, it does not know how big the char limit is. This error exposes the program to a buffer overflow attack. The target file explained above will be fuzzed to better depict crashes of the system.

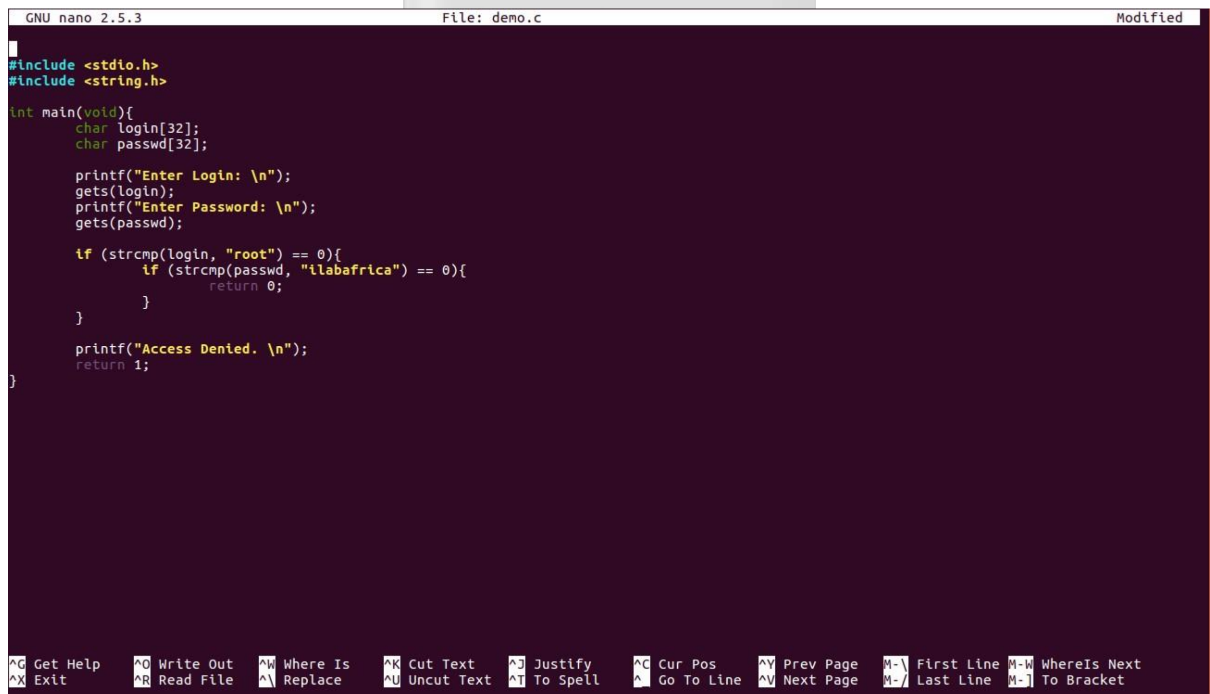


Figure 5-10: Target Input File

The testcases that are executed on each VM in the *aflfuzzer.yml* playbook are shown in Figures 5-10 to 5-14.

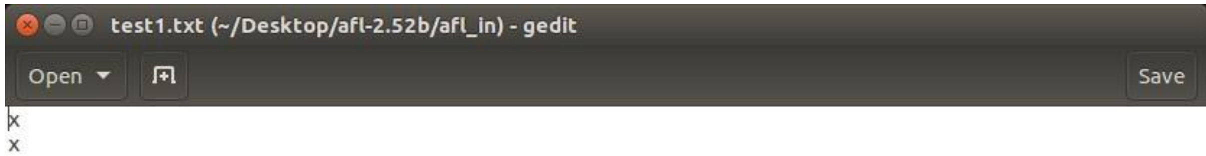


Figure 5-11: First Testcase

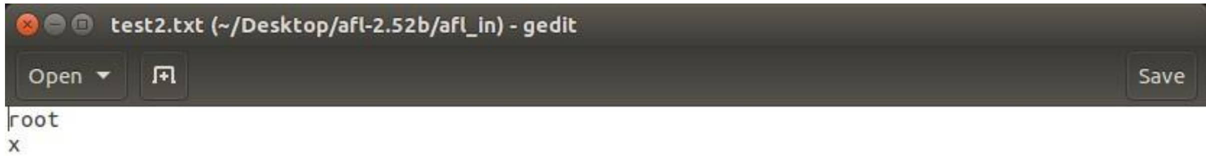


Figure 5-12: Second Testcase

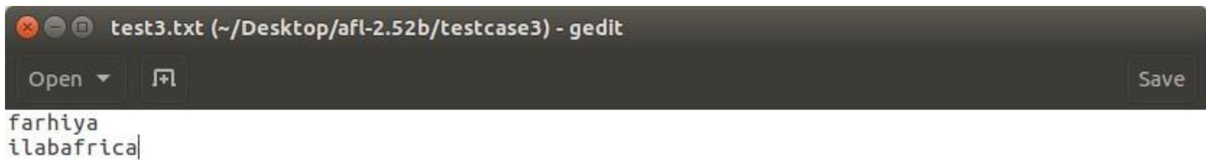


Figure 5-13: Third Testcase

Figure 5-14: Fourth Testcase

Figure 5-15: Fifth Testcase

5.5 Results

The amount of time taken to run the AFL Fuzzer on all five virtual machines will be compared to running the AFL Fuzzer on one machine with different testcases. The Ansible playbooks ran five different testcases on 5 different VMs compared to five testcases on one machine.

Figure 5-15 displays the results of five testcases running on my local machine. The fuzzer ran for a total of 15 minutes, 7 seconds. The unique crashes recorded were 3 in that amount of time.

Figure 5-16: Results of AFL on Local Machine

On the five virtual machines, the results were collected as depicted in Figure 5-16 to Figure 5-21.

5-17: Fuzzer stats from first VM

```
GNU nano 2.5.3                    File: fuzzer_stats

start_time        : 1523962176
last_update       : 1523962310
fuzzer_pid        : 5846
cycles_done       : 75
execs_done        : 752190
execs_per_sec     : 5599.62
paths_total       : 2
paths_favored     : 2
paths_found       : 1
paths_imported    : 0
max_depth         : 2
cur_path          : 0
pending_favs      : 0
pending_total     : 0
variable_paths    : 0
bitmap_cvg        : 0.01%
unique_crashes    : 1
unique_hangs      : 0
last_path         : 1523962176
last_crash        : 1523962177
last_hang         : 0
exec_timeout      : 20
afl_banner        : demo
afl_version       : 1.96b
command_line      : afl-fuzz -i testcase -o output ./demo @

                              [ Read 25 lines ]
^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text   ^J Justify   ^C Cur Pos     ^Y Prev Page  M-\ First Line  M-W WhereIs Next
^X Exit      ^R Read File  ^\ Replace   ^U Uncut Text ^T To Spell  ^_ Go To Line  ^V Next Page  M-/ Last Line   M-] To Bracket
```

5-18: Fuzzer stats from Second VM

```
ubuntu@ip-172-31-44-208: ~/output
GNU nano 2.5.3                    File: fuzzer_stats

start_time        : 1523962176
last_update       : 1523962310
fuzzer_pid        : 1364
cycles_done       : 138
execs_done        : 695346
execs_per_sec     : 5150.22
paths_total       : 1
paths_favored     : 1
paths_found       : 0
paths_imported    : 0
max_depth         : 1
cur_path          : 0
pending_favs      : 0
pending_total     : 0
variable_paths    : 0
bitmap_cvg        : 0.00%
unique_crashes    : 1
unique_hangs      : 0
last_path         : 0
last_crash        : 1523962177
last_hang         : 0
exec_timeout      : 20
afl_banner        : demo
afl_version       : 1.96b
command_line      : afl-fuzz -i testcase -o output ./demo @

^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text   ^J Justify   ^C Cur Pos     ^Y Prev Page  M-\ First Line  M-W WhereIs Next
^X Exit      ^R Read File  ^\ Replace   ^U Uncut Text ^T To Spell  ^_ Go To Line  ^V Next Page  M-/ Last Line   M-] To Bracket
```

5-19: Fuzzer stats from Third VM

```
ubuntu@ip-172-31-42-241: ~/output
  GNU nano 2.5.3                          File: fuzzer_stats

start_time        : 1523962176
last_update       : 1523962310
fuzzer_pid        : 10239
cycles_done       : 137
execs_done        : 689579
execs_per_sec     : 5128.09
paths_total       : 1
paths_favored     : 1
paths_found       : 0
paths_imported    : 0
max_depth         : 1
cur_path          : 0
pending_favs      : 0
pending_total     : 0
variable_paths    : 0
bitmap_cvg        : 0.00%
unique_crashes    : 1
unique_hangs      : 0
last_path         : 0
last_crash        : 1523962177
last_hang         : 0
exec_timeout      : 20
afl_banner        : demo
afl_version       : 1.96b
command_line      : afl-fuzz -i testcase -o output ./demo @

^G Get Help   ^O Write Out   ^W Where Is   ^K Cut Text    ^J Justify    ^C Cur Pos     ^Y Prev Page   M-\ First Line   M-W WhereIs Next
^X Exit       ^R Read File   ^\ Replace    ^U Uncut Text  ^T To Spell   ^_ Go To Line  ^V Next Page   M-/ Last Line    M-] To Bracket
```

5-20: Fuzzer stats from Fourth VM

```
ubuntu@ip-172-31-34-56: ~/output
  GNU nano 2.5.3                          File: fuzzer_stats

start_time        : 1523962174
last_update       : 1523962310
fuzzer_pid        : 8800
cycles_done       : 145
execs_done        : 728075
execs_per_sec     : 5375.36
paths_total       : 1
paths_favored     : 1
paths_found       : 0
paths_imported    : 0
max_depth         : 1
cur_path          : 0
pending_favs      : 0
pending_total     : 0
variable_paths    : 0
bitmap_cvg        : 0.00%
unique_crashes    : 1
unique_hangs      : 0
last_path         : 0
last_crash        : 1523962177
last_hang         : 0
exec_timeout      : 20
afl_banner        : demo
afl_version       : 1.96b
command_line      : afl-fuzz -i testcase -o output ./demo @

^G Get Help   ^O Write Out   ^W Where Is   ^K Cut Text    ^J Justify    ^C Cur Pos     ^Y Prev Page   M-\ First Line   M-W WhereIs Next
^X Exit       ^R Read File   ^\ Replace    ^U Uncut Text  ^T To Spell   ^_ Go To Line  ^V Next Page   M-/ Last Line    M-] To Bracket
```

5-21: Fuzzer stats from Fifth VM

The fuzzers above ran for 5 minutes, and found 1 unique crash each. These results will be compared to the number of unique crashes detected by the fuzzer in the local machine.

5.6 Testing

The purpose of testing is to demonstrate that the developed solution satisfies its specified requirements. This step involves testing of the distributed fuzzing solution.

5.6.1 Non-functional Testing

Described below are the tests that were undertaken in non-functional testing:

i. Stress Testing of the file fuzzers by increasing the number of test cases and monitoring the amount of time taken and number of cycles.

```
farhiya@farhiya-HP-Pavilion-TS-11-Notebook-PC:/etc/ansible$ sudo ansible-playbook runafl.yml -u ubuntu

PLAY [Execute 5 different testcases on All  VMs] ***************************************************************************

TASK [Gathering Facts] *****************************************************************************************************
ok: [ubuntu@18.216.112.183]
ok: [ubuntu@18.188.223.34]
ok: [ubuntu@18.218.233.104]
ok: [ubuntu@18.222.69.62]
ok: [ubuntu@18.188.34.3]

TASK [Execute the AFL Fuzzer on each VM running different testcases] *****************************************************
^C [ERROR]: User interrupted execution

farhiya@farhiya-HP-Pavilion-TS-11-Notebook-PC:/etc/ansible$
```

5-22: Stress testing of afl

The fuzzer above did not stop until the user interrupted the execution.

ii. Security testing of the fuzzers to ensure that the distributed fuzzing clusters are secure

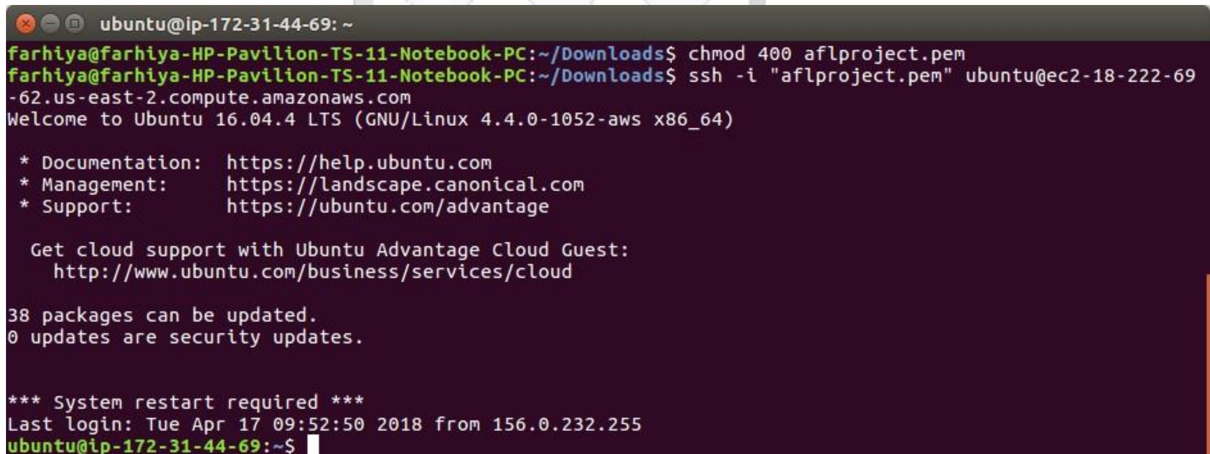


Figure 5-23: Logging into VM with SSH

The user of the solution can only log into the system with SSH. SSH is a secure protocol that ensures secure remote login from one machine to another. It authenticates the login by specifying the private key generated by the user, as you can see in Figure 5-18 above. The private key is called *aflproject.pem.* This authenticates the user logging into the virtual machines.

5.6.2 Functional Testing

i. Installation and setup of multiple computing environments on the VPS to run the fuzzers: The user can install fuzzers and instrument the input file on multiple computing environments.

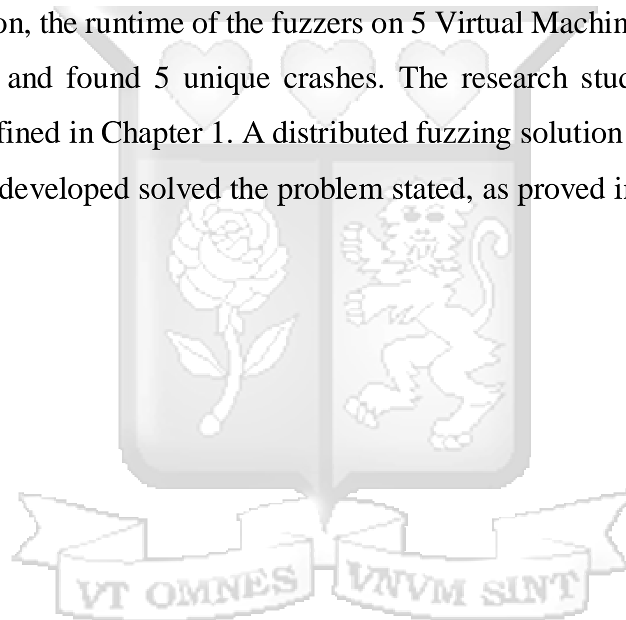
ii. Check the monitoring and logging functions of the fuzzer.

There are 3 sub-directories that are put in the output directory. The sub-directories are queue, crashes and hangs. The directory that collects the results of the unique crashes are in the crashes folder. This is what the user will look at to further analyse the unique crashes.

Chapter 6: Discussion of results

The purpose of the project was to run a distributed fuzzing cluster on multiple Virtual Machines in order to increase the efficiency of fuzzing. The research study undertaken proved that running multiple fuzzers concurrently to fuzz a file takes less runtime compared to running the fuzzer on one machine.

The local machine took 20 minutes, 2 seconds to run 5 testcases and found 3 unique crashes in that time. In comparison, the runtime of the fuzzers on 5 Virtual Machines took 1 minute to run the same 5 testcases, and found 5 unique crashes. The research study carried out met the research objectives defined in Chapter 1. A distributed fuzzing solution was developed and the results of the solution developed solved the problem stated, as proved in Chapter 5.6 Testing.

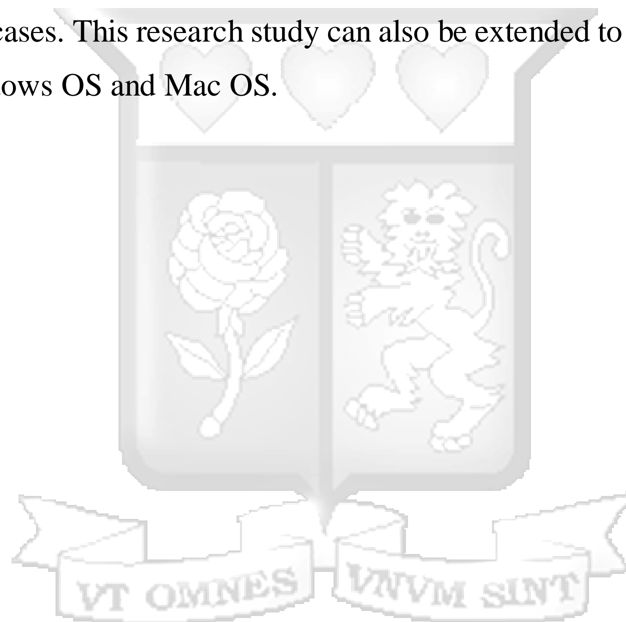


Chapter 7: Conclusions

The results collected from the research study proves that running distributed fuzzing clusters concurrently on virtual instances increases the efficiency and decreases the runtime of the fuzzers to discover software vulnerabilities. The benefits of distributed fuzzing compared to regular fuzzing is the ability to run multiple test cases simultaneously thus increasing the efficiency of fuzzing. This will increase the effectiveness of fuzzing to discover vulnerabilities, leading to less run-time and increased discovery of vulnerabilities.

7.1 Future Work

The purpose of the research study was to fuzz input files on distributed fuzzing clusters. The research study can be extended to fuzz applications and analyse the application's code with multiple complex testcases. This research study can also be extended to run on other Operating Systems such as Windows OS and Mac OS.



Chapter 8: References

American Fuzzy Lop (2.52b). (n.d.). Retrieved March 25, 2018, from lcamtuf: http://lcamtuf.coredump.cx/afl/

Ansible. (2017). Retrieved March 28, 2018, from Ansible: http://docs.ansible.com/

Autoscaling Groups. (2018). Retrieved April 15, 2018, from Amazon Web Services: https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html

Clarke, T. (2009). *Fuzzing for software vulnerability.* Surrey. Retrieved February 26, 2018

Configuration Management. (2017). Retrieved February 27, 2018, from Ansible: https://www.ansible.com/use-cases/configuration-management

Conger, D., Srinivasamurthy, K., & Cooper, R. (2010). *U.S. Patent No. US 7743281 B2.* Retrieved February 26, 2018

Dai, H., Murphy, C., & Kaiser, G. (2010, February 15). *Configuration Fuzzing for Software Vulnerability Detection.* Retrieved February 26, 2018, from NCBI: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3057938/

Digital Ocean. (2018). Retrieved March 28, 2018, from Digital Ocean: https://www.digitalocean.com/

Doyle, F., Fly, R., Maynor, D., Miller, C., & Naveh, Y. (2011). *Open Source Fuzzing Tools.* Burlington: Syngress Publishing. Retrieved February 26, 2018

Hillman, M. (2013, August 8). *15 minute guide to fuzzing*. Retrieved February 26, 2018, from MWR InfoSecurity: https://www.mwrinfosecurity.com/our-thinking/15-minute-guideto-fuzzing/

Juranić, L. (2006, April 25). *Using fuzzing to detect security vulnerabilities.* Zagreb: Infigo IS. Retrieved March 27, 2018, from https://pdfs.semanticscholar.org/c86c/bbb14be4c623b5150e2d0c44a6ee0e9c6292.pdf

Kirsch, W. (2017, March 20). *How we found a tcpdump vulnerability using cloud fuzzing*. Retrieved April 1, 2018, from SoftSCheck: https://www.softscheck.com/en/identifying-security-vulnerabilities-with-cloudfuzzing/

Malan, R., & Bredemeyer, D. (2001). *Functional Requirements and Use Cases.* Bloomington: Bredemeyer Consulting. Retrieved February 25, 2018

Microsoft Visio. (2018). Retrieved March 29, 2018, from Microsoft Products: https://products.office.com/en/visio/flowchart-software

Miller, B., Fredriksen, L., & So, B. (1989). *An Empirical Study of the Reliability of UNIX Utilities.* Madison: University of Wisconsin. Retrieved February 27, 2018

Pierce, J. (2012, November 6). *Security Testing: Web Application Fuzz Testing*. Retrieved March 27, 2018, from Coveros: https://www.coveros.com/our-blogs/

Rathaus, N., & Evron, G. (2007). *Open Source Fuzzing Tools.* Syngress. Retrieved February 26, 2018

Sanmillan, I. (2018, January 2). *Executable and Linkable Format 101*. Retrieved February 26, 2018, from Intezer: http://www.intezer.com/executable-linkable-format-101-part1sections-segments/

Sutton, M., Green, A., & Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery.* Pearson Education. Retrieved February 25, 2018

System Design. (2018). Retrieved March 29, 2018, from Techopedia: https://www.techopedia.com/definition/29998/system-design

Takanen, A. (2010). Proactive Security Testing and Fuzzing. In R. H. Pohlmann N., *ISSE 2009 Securing Electronic Business Processes* (pp. 312-319). Vieweg+Teubner. Retrieved March 27, 2018, from https://pdfs.semanticscholar.org/2efd/8de4341e0757427183f6fb33beaad60bfede.pdf

Ubuntu OS. (2018). Retrieved March 28, 2018, from Ubuntu: https://www.ubuntu.com/desktop

UML Sequence Diagram Tutorial. (2018). Retrieved April 9, 2018, from Lucid Chart: https://www.lucidchart.com/pages/uml-sequence-diagram

UML Sequence Diagrams. (2017). Retrieved March 28, 2018, from UML Diagrams: https://www.uml-diagrams.org/sequence-diagrams.html

UML Use Case Diagrams. (2017). Retrieved March 28, 2018, from UML Diagrams: https://www.uml-diagrams.org/use-case-diagrams.html

VT OMNES VNVM SINT

University of Connecticut. (2017). *Requirements Analysis*. Retrieved March 28, 2018, from UConn: https://sdlc.uconn.edu/activity-3-requirements-analysis/

What is a data flow diagram (DFD). (2012, January 27). Retrieved from Visual Paradigm: https://www.visual-paradigm.com/tutorials/data-flow-diagram-dfd.jsp

What is BLACK Box Testing? Techniques, Example & Types. (2018). Retrieved February 27, 2018, from Guru 99: https://www.guru99.com/black-box-testing.html

What is Fuzzing: The Poet, the Courier and the Oracle. (2017). Retrieved March 27, 2018, from Synopsys: https://www.synopsys.com/content/dam/synopsys/sigassets/whitepapers/what-is-fuzzing.pdf

White Box Testing Fundamentals. (2018). Retrieved February 27, 2018, from Software Testing Fundamentals: http://softwaretestingfundamentals.com/white-box-testing/

Xu, W., Kashyap, S., Min, C., & Kim, T. (2017). *Designing New Operating Primitives to Improve.* Retrieved March 27, 2018, from Georgia Tech: http://iisp.gatech.edu/sites/default/files/images/designing_new_operating_primitives_to_improve_fuzzing_performance_vt.pdf