國立交通大學

資訊科學與工程研究所

碩士論文

基於 QEMU 所模擬系統與軟體之回歸測試框架

A Regression Testing Framework for QEMU-based System and Software Development

研　究　生：林伯謙

指導教授：黃世昆　教授

中華民國一百零二年六月

基於 QEMU 所模擬系統與軟體之回歸測試框架

A Regression Testing Framework for QEMU-based System and Software Development

研　究　生：林伯謙　　Student : Po-Chien Lin

指導教授：黃世昆　　Advisor : Shin-Kun Huang

國　立　交　通　大　學

資　訊　科　學　與　工　程　研　究　所

碩　士　論　文

A Thesis

Submitted to Department of Computer and Engineering

College of Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

June 2013

Hsinchu, Taiwan, Republic of China

中華民國一百零二年六月

基於 QEMU 所模擬系統與軟體之回歸測試框架

學生: 林伯謙　　　　　　　　指導教授: 黃世昆教授

國立交通大學資訊科學與工程研究所碩士班

摘要

藉由軟體測試技術，我們可以檢測程式執行路徑，確認是否含有可能之缺陷與問題，以減少程式錯誤的發生率。然而現有的技術，在處理龐大程式上，軟體測試仍有其困難性與軟硬體資源不足的阻礙。其中，一個限制因素是目前的測試操作都還需要人工介入。雖然大部分測試工具的執行過程已經可以自動化，但在其測試前後的系統環境建置、餵送測試資料、判讀實驗結果以進行下一步驟，都尚未做到完全自動化，使得難以進行大規模、完整的測試。回歸測試的概念，是讓相同的測試工作能被重複執行，以幫助程式設計人員能在修改、調整程式細節等相關開發過程後，再次進行測試，以檢驗功能是否正常、或效能是否有所改進。在雲端計算的概念下，我們利用虛擬化資源的管理概念，將測試平台抽離出硬體主機的限制，藉此只要有足夠的硬體資源，我們的平台就能在不更改架構的情形下，進行大量測資的測試。在此論文中，我們藉由回歸測試框架的實作來管理軟體測試的過程。先試驗測試標的為 Linux 與 Windows 的環境，進一步將建立 Android-x86 系統的環境。如此，我們就能夠進行多適應性大規模的回歸測試。

關鍵字: QEMU 模擬、軟體測試、回歸測試、測試自動化、測試框架

A Regression Testing Framework for QEMU-based System and Software Development

Student : Po-Chien Lin Advisor : Dr. Shih-Kun Huang

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

With the help of software testing techniques, we are able to find potential defects along the execution path of programs, and reduce the software quality in the software system. However, to handle test cases for a large scale program using current software testing technique, it still requires much human efforts and manual intervention to operate, manage software and hardware resources. We currently perform testing and experiments by QEMU-based system tools in an automatic way. However, other processes are still in manual way, including the system environment setting up, input data feeding, result analysis and verification. This hinders us from performing large scale and comprehensive testing. The concept of "regression testing" is to perform the same testing repeatedly, which will help programmers test the integrity of functionalities or efficiency improvements after modifying or tuning the programs in the development stage. By the concept of "cloud computing", we can abstract away the testing platform from the hardware resources restriction by virtualization technique. Our platform is therefore able to manage large scale testing without reorganizing the architecture if we have sufficient and available hardware resources. We propose to manage the software regression testing process by the cloud implementation of a testing framework. We have first deployed the testing framework in the environments on Linux and Windows, and then Android-x86. The process of regression testing has been automated by performing several testing benchmarks which originally take several days to complete, needing much human efforts. The results reveal that our framework implementation can carry out an applicable and large scale testing in a cloud environment.

Keywords: QEMU, QEMU-based system, software testing, regression testing, testing automation, testing framework

Contents

摘要 i

Abstract ii

Contents iii

List of Codes v

List of Figures vi

List of Tables vii

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Problem Description . . . 3
- 1.3 Objective . . . 5
- 1.4 Background . . . 6
  - 1.4.1 Automatic Exploit Generation System . . . 6
  - 1.4.2 Cloud Management . . . 8
  - 1.4.3 Software Testing . . . 9
- 1.5 Overview . . . 10

2 Related Work 11
- 2.1 Cloud management tools . . . 11
  - 2.1.1 libvirt . . . 11
  - 2.1.2 OpenStack . . . 12
  - 2.1.3 Eucalyptus . . . 13
- 2.2 Windows GUI Operation Tools . . . 14
  - 2.2.1 Sikuli . . . 14
  - 2.2.2 AutoHotkey . . . 15
  - 2.2.3 AutoIt . . . 15
  - 2.2.4 Comparison . . . 15
- 2.3 Software Testing . . . 15
  - 2.3.1 Test Automation . . . 16
  - 2.3.2 Unit Testing / xUnit . . . 17
  - 2.3.3 Regression Testing . . . 17
- 2.4 Related Implementations . . . 18
  - 2.4.1 D-Cloud . . . 18

2.4.2 ETICS . . . 19
2.4.3 Inadequacies . . . 19

3 Method 20
3.1 Scenarios . . . 20
3.1.1 Windows and Linux . . . 20
3.1.2 Web Application . . . 21
3.1.3 Android . . . 22
3.1.4 Fuzzer . . . 23
3.1.5 Malware . . . 23
3.1.6 Embedded System . . . 24
3.2 Requirements . . . 24
3.3 Our Method . . . 26
3.3.1 Host Resource Management . . . 26
3.3.2 Guest Machine Management . . . 26
3.3.3 Communication System between Host and Guest . . . 27
3.4 Windows GUI Object Manipulation . . . 28
3.4.1 Symfile . . . 28
3.4.2 Software Operation . . . 28
3.4.3 Exploit Verification . . . 28

4 Implementation Details 30
4.1 Implementation Language . . . 30
4.2 Regression Testing Framework Architecture . . . 31
4.2.1 Cloud-based Management . . . 32
4.2.2 Disk Images and Snapshots Management . . . 33
4.2.3 Connection Channel and Shared File System . . . 33
4.3 Regression Testing Framework . . . 35
4.3.1 A Workflow of a Regression Testing . . . 35
4.3.2 A Sample Test Case . . . 37
4.3.3 More Functionalities . . . 38

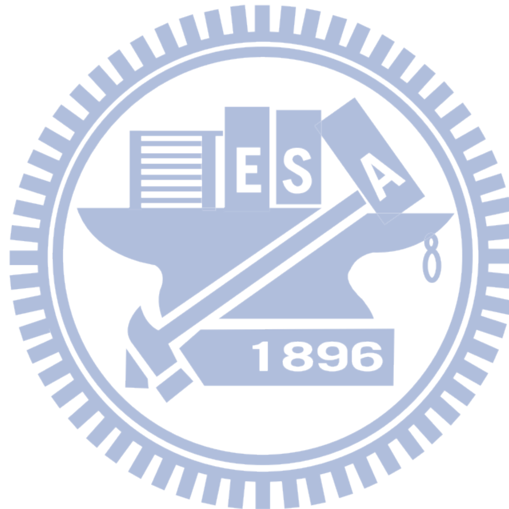
5 Results 41
5.1 Regression Testing Framework for QEMU-based System . . . 41
5.2 Testing Branch Database . . . 42
5.3 Improvement . . . 42
5.3.1 Time and Efficiency Comparison . . . 43
5.3.2 Feature Comparison . . . 43

6 Conclusion and Further Work 45
6.1 Conclusion . . . 45
6.2 Further Work . . . 46

Reference 47

List of Codes

1 A sample test case: test1.py 39
2 Showing test cases example 39
3 Selecting test cases example 40
4 Showing test results example 40
5 An example of testing wrapper 40

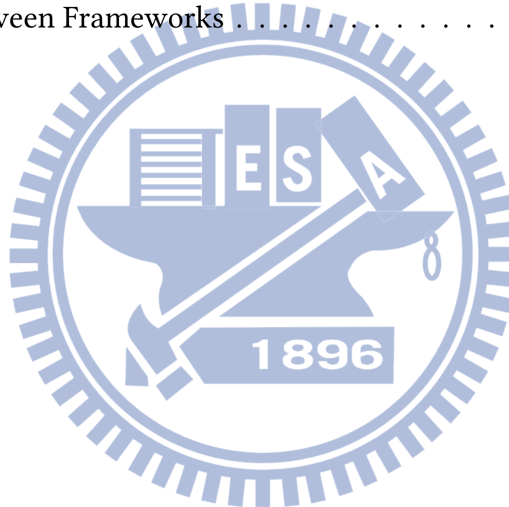


List of Figures

1 An overview to our idea . . . 2
2 Current method flow . . . 3
3 OpenStack overview (*source: OpenStack*) . . . 12
4 Eucalyptus platform architecture . . . 13
5 A sample of Sikuli script (*source: Sikuli.org*) . . . 14
6 Flow of D-Cloud *(source: D-Cloud paper)* . . . 18
7 The ETICS System Architecture *(source: ETICS paper)* . . . 19
8 Overall architecture for web automatic testing . . . 21
9 The architecture of x86-Android on top of S^2E . . . 22
10 New testing model modified from past method . . . 24
11 Framework over cloud computing . . . 27
12 Communication channel outline . . . 27
13 CRAXUnit Architecture . . . 31
14 Hypervisor support by libvirt . . . 32
15 Regression testing framework on libvirt . . . 33
16 A workflow of single testing . . . 35
17 An Example of Result Output . . . 37

List of Tables

1 Comparison between Automation Tools 16
2 Communication channel protocol commands 34
3 Configurations in a Test Case 38
4 Basic type in CRAXUnit 39
5 APIs in CRAXUnit 40
6 Exploitable Test Case in Database 43
7 Comparison of running one testing 43
8 Comparison of running a testing benchmark 43
9 Comparison between Frameworks 44



Chapter 1

Introduction

Software testing is an important stage in the field of software engineering, whereas it is often overlooked. Nowadays, we rely more and more on computer programs to make our life going well; however, even a small unexpected error can cause enormous damage. Therefore, we need software testing to check the codes and look into those vulnerabilities left in programs due to negligence by programmers. Otherwise, these vulnerabilities can become the exploitable entrance utilized by malicious users.

With the soaring importance and popularity of computers in our life, the information and software industry are getting more and more popular. However, we should also notice that more and more security holes and software vulnerabilities are leaked at the same time. The upstream and maintainers of not only operating systems like BSD, Linux, and Windows, but also the implementations of popular programming languages such as JAVA, Python, Ruby, and many other software that millions of people use every day, are releasing their patched versions of vulnerable ones continuously.

The truth is, we are not only at the era of information life, but also at the era of information security. Society and economic order may jump into chaos without information technology; however, the world can not running on track without information security even if we have excellent information technologies.

1.1 Motivation

The quality of software not only depends on what functionalities it provides, but also depends how secure it is. The vulnerabilities and potential defects hidden in software (or programs) may led to unexpected results. By utilizing these software vulnerabilities, malicious users can execute any commands they want through a clever and designed input.

The development of software is a long period process, and it involves more than one programmer most of the time. As the scale of program becomes larger and larger, the existence of the vulnerabilities caused by negligence become an inevitable result.

With the help of software testing techniques, we are able to find potential defects along the execution paths of a program, and reduce the software quality in the software system. However, to handle test cases for a large scale program using current software testing technique, it still requires much human efforts and manual intervention to operate, manage software and hardware resources.

So we continue to look forward to finding some more efficient methods to solve these problems.

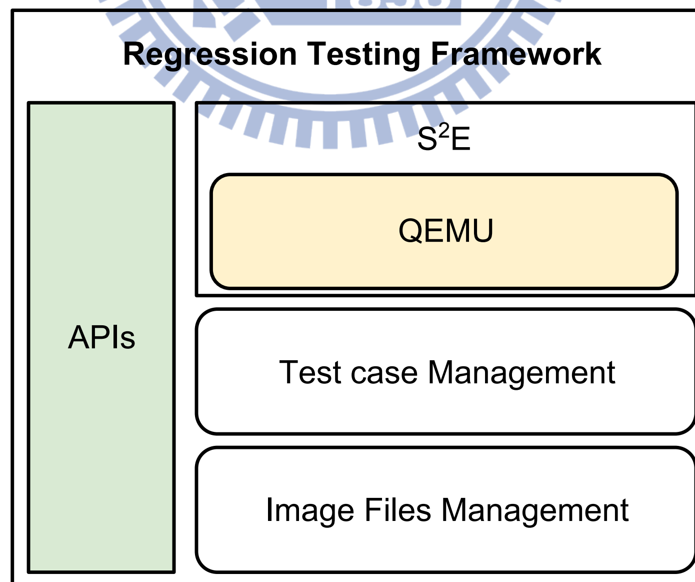


Figure 1: An overview to our idea

We currently perform testing and experiments by QEMU-based system tools in an automatic way. However, other processes are still in manual way, including the system environment

setting, data feeding, result analysis and verification. This hinders us from performing large scale and comprehensive testing.

The concept of "regression testing" is to perform the same testing repeatedly, which will help programmers test the integrity of functionalities or efficiency improvements after modifying or tuning the programs in the development stage. By the concept of "cloud computing", we can abstract away the testing platform from the hardware resources restriction by virtualization technique. Then we can plan a series of procedure and run it without human intervention.

As shown in *Figure 1*, We want to implement a host-based QEMU testing system by utilizing current techniques like cloud management tools and *libvirt* API to make the software testing method fully automated coming from this concept. We will first deploy the testing framework in the environments on Linux and Windows, and then Android-x86. The goal of our implementation is to be able to carry out an applicable and large scale testing in a cloud environment.

1.2 Problem Description

We can divide the current method of exploit generation process using QEMU-based testing system into several stages (See *Figure 2*).

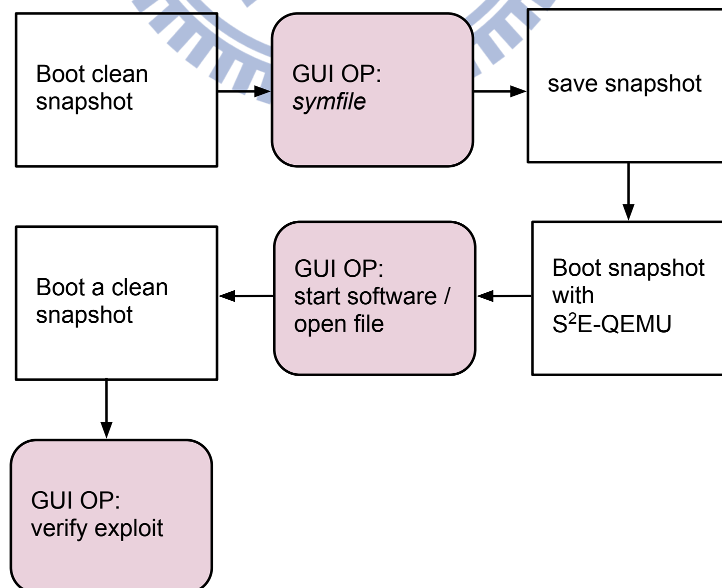


Figure 2: Current method flow

1. Set up the QEMU image (*image-1*) of target vulnerability environment, both operation

system and target software or program installed inside may be bind to specific versions which have vulnerability.

2. Block 1~3: Boot up the virtual machine *image-1* with normal version QEMU, and save the snapshot (*snapshot-1*) after doing *symfile* to crash input.

3. Block 4, 5: Boot up the snapshot *snapshot-1* with S^2E version QEMU. In guest OS, testers should operate the target software to open or process the crash input.

4. Then host system will start to run symbolic execution and generate exploit if possible.

5. Block 6, 7: If the exploit generated successfully, we take the generated exploit to test on the clean environment *image-1* to verify if the exploit is usable.

Besides *stage 4*, each other stages should be operated by human manually now. This is not practical while we want to do a large scale of software testing.

Human intervention can be found in each stages, and moreover, we need to take a look at each step to see how the process going and what time to go forward to next step. Thus, we hope that we can integrate the testing environment and enhance the testing processes to a more usable and less manual work.

In the testing processes described above, we can see several cons:

- Testing on host machine requires human intervention.
- GUI operations are not programmable. Testers need to interactive with GUI environments.
- Testing procedure are not the same through different testing target OS platform.
- No reliable and efficient management method to the test cases and tested results.

And because we are trying to build a testing platform to integrate the different and inconvenient testing processes, here we sort out the problems needed to be solved:

1. The image files format used by official QEMU and S^2E-QEMU are not compatible [*]. We may need to find a method to convert between these two formats or make them compatible to each other.

[*] See: https://s2e.epfl.ch/embedded/s2e/ImageInstallation.html

2. We need a file sharing mechanism between guest OS and host OS.

3. We need a communication method between guest OS and host OS. Thus host OS can control guest OS and get information for guest OS.

4. We need a methods on host OS to take snapshots of the running guest OS, instead of issuing commands in guest OS.

5. We need GUI automation method to take a series of operations under Windows and Linux.

6. We need to provide management ability to our testing.

Before implementing, we also need to survey some methods that we may need. These include:

1. Collect information about software that can be tested based on an QEMU emulator. On this basis, we can design our system to let those software can be tested on our platform.

2. Survey cloud-based image files management method and analysis those techniques that we can use on managing a large number of hardware resources and VMs.

3. Survey the possibility of combining S^2E and cloud-based VM management.

4. Survey the cloud-init mechanism to initialize required testing environments on demand.

5. Survey the testing procedure in embedded system development, and see if it can be integrated with our system.

1.3 Objective

Software testing really takes an important role in software development life cycle. It provides the quality assurance to upcoming products, and make an early discovery of vulnerabilities in software that may cause huge losses.

Our goal is to build a testing framework that will provide comprehensive functionalities to let us create, manage, and run test cases easily. With the implementation of such framework, we can apply it to many environments to test various of software designed for different OSes.

As a testing framework basis, we will require to rebuild a set of test cases that we tested manually before. Besides, we are also planning to build test cases for CVE [†] environments that can be used for later experiments. Also, Constructing the test case database is important for us to review and improve our framework by running real data in it.

Moreover, as the software testing and exploit generation technology improved, we may want to test those test cases we have tested before again. Being a well-designed testing framework, it will also be better for the system to provide a flexible interface that allows testers to customize testing factors. Thus we can setup and run a bunch of wanted testing with changeable parameters quickly through this framework.

1.4 Background

In this section, we explain the terminologies that are used in the thesis. Three parts are introduced: automatic exploit generation system, cloud management, and software testing.

1.4.1 Automatic Exploit Generation System

In the thesis, we first develop based on the former research of automatic exploit generation system to improve the experimental process. Here we simply explain some key points in this field.

- Vulnerability

  In computer security, **vulnerability** is a weakness which allows an attacker to reduce a system's information assurance [‡].

- Exploit

  An exploit is a piece of instructions, commands or data that can take advantage of bugs or vulnerabilities to cause an unintended behavior on computer software. Such behavior frequently includes such things as gaining control of a computer system or allowing privilege escalation or a denial-of-service attack.

[†] Common Vulnerabilities and Exposures, see more information at: http://cve.mitre.org. CVE® International in scope and free for public use, CVE is a dictionary of publicly known information security vulnerabilities and exposures.

[‡] Wikipedia: Vulnerability (computing)

- Symbolic execution

  Symbolic execution [1, 2, 3, 4] is a popular technique of software testing. In contrast with concrete execution that treats the tested program as a black box and find next new path without any information, symbolic execution attempts to explore all paths in the program more systematically by transforming the path feasibility problem into Boolean satisfiability problem [§]. The main idea of symbolic execution is to replace variables controlled by external environments with symbolic values rather than actual data. The value range of those variables represented by symbolic expressions is unlimited, i.e. any value, when the program runs initially. With program execution, those symbolic variables will taint other non-symbolic variables, and its value will be gradually restricted.

- symfile

  When doing symbolic execution, we need first make the input data symbolical to let our program can access those input in symbolic form. **Symfile** is what we call the process to make a file symbolical and map the symbolic file into memory. The action usually should be done on the testing target environment, i.e., in the guest OS if we are using QEMU-based system to do testing works.

- S²E

  S²E [5] is a software analysis platform which allows running the whole operating system in a testing environment. It uses dynamic binary translation which provided by QEMU, selective symbolic execution and relaxed execution consistency models to find the execution paths. So we can analyze not only the user-mode but also the kernel-mode binary.

- CRAX

  CRAX [6] is a framework for automatically generating exploit. It is developed based on S²E and concolic execution. An User can feed the framework with crash inputs of a particular software, and then the framework starts to address the exploitable point of the software by concolic execution. Then shellcode combinations are inserted to test if any of them can be a valid and exploitable input. If success, an exploit is generated as an output to the user.

[§]For more information, see Wikipedia: Boolean satisfiability problem

- Input / crash input

  Input is the kind of data provided by user environment to the software or program to let it analysis, process, or generate some different output according to different input. According to different programs, the type of input may be a series of numbers or strings, files, sockets, etc. When we try to generate an usable exploit in CRAX system, we need first provide an input to the target software (or program) we interested in that will cause the program run into crash finally. This kind of input are called **crash input**.

1.4.2 Cloud Management

We want our implementation to be isolated from physical hardware resources. That is, the scale of testings should only be restricted by computation capability but not the framework architecture scalability. So we surveyed the cloud solution to abstract the hardware restriction.

- Host-guest VM architecture

  A virtual machine (VM), typically has two components: the host and the guest. **The host** is the virtual machine host server; the underlying hardware provides computing resources, such as processing power, memory, disk and network I/O, and so on. **The guest** is a completely separate and independent instance of an operating system and application software. Guests are the virtual workloads that reside on a host virtual machine and share in that server's computing resources.

- QEMU

  QEMU [7, 8] is a free and open-source software product that performs hardware virtualization. QEMU is a hosted virtual machine monitor: It emulates central processing units through dynamic binary translation and provides a set of device models, enabling it to run a variety of unmodified guest operating systems.

- Image / Snapshot

  An image file is used to boot a virtual machine by QEMU. It likes the hard disk saving the operating system files and user data in a physical machine. **A snapshot** is a special format in a QEMU image file. It saves a runtime status of the virtual machine (i.e., via `savevm` command) and could be reloading at any time (via `loadvm` command) with user's

intention. The snapshot information is saved within the same image file of the VM image that used to boot the machine when doing `savevm` action.

- libvirt

  libvirt [9] is an open source API, daemon and management tool for managing platform virtualization. It can be used to manage Linux KVM, Xen, VMware ESX, QEMU and other virtualization technologies. These APIs are widely used in the orchestration layer of hypervisors in the development of a cloud based solution.

1.4.3 Software Testing

Software testing techniques can assure the quality of a software. By looking into the various testing methods, we can make our implementation to achieve the goal of software testing more closely.

- Unit Testing

  Unit testing is a method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine if they are fit for use. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended. Its implementation can vary from being very manual to being formalized as part of build automation.

- Unit Testing Framework

  Unit testing frameworks help simplify the process of unit testing. It is generally possible to perform unit testing without the support of a specific framework, whereas once a framework is in place, adding unit tests becomes relatively easy.

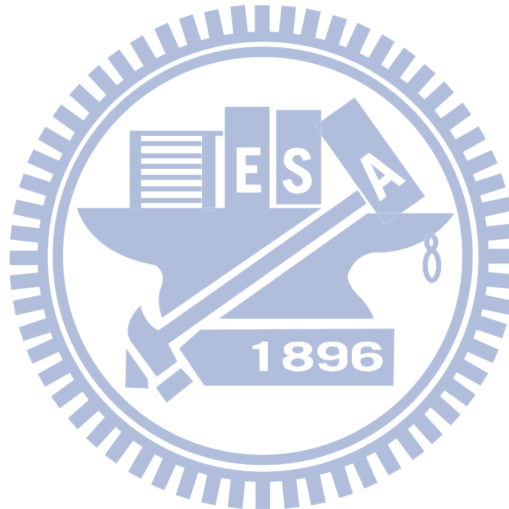
- Regression Testing

  Regression testing is any type of software testing that seeks to uncover new software bugs, or regressions, in existing functional and non-functional areas of a system after changes, such as enhancements, patches or configuration changes, have been made to them. The intent of regression testing is to ensure that a change such as those mentioned

above has not introduced new faults. One of the main reasons for regression testing is to determine whether a change in one part of the software affects other parts of the software.

1.5 Overview

The structure of this thesis is shown as follows. *Chapter 2* describes the related work about cloud managements and software testing. *Chapter 3* and *Chapter 4* explain our method and implementation. *Chapter 5* shows the experimental results. Finally, *Chapter 6* concludes our achievements with further work that may enhance our results.



Chapter 2

Related Work

Before implementation, we first collect and survey the materials that are related to the testing framework. Cloud managements tools are the techniques that can make our framework scalable; Windows GUI operation tools are used to solve the human intervention problem in the experiments, and software testing surveys make out implementation more complete. In the end of this chapter, we also introduce two software testing frameworks whose goals are similar to ours, and compare them to our design.

2.1 Cloud management tools

With the concept of ”Cloud”, the hardware and software resources on host are virtualized to the guest machines running on host. We will benefit from it while we are going to design a testing framework targeted for large scale testing.

We survey some cloud management tools to help us manage the resources on our testing platform.

2.1.1 libvirt

Libvirt provides a wide range of API for managing platform virtualization. As the development team announced [*], the goal of *libvirt* is to provide a common and stable layer sufficient to

[*]Project goals: *http://libvirt.org/goals.html*

securely manage domains † on a node ‡.

Therefore, *libvirt* is designed to provide all APIs needed to do the management, such as provision, create, modify, monitor, control, migrate, and stop the domains. So *libvirt* is intended to be a building block for higher level management tools and for applications focusing on virtualization of a single node. We users don't need to worry about which underlying hypervisor is used, and almost all of the managements will be handled by *libvirt* through its APIs.

The internals of *libvirt* is a C library, but it also supports a variety of bindings in common languages, such as Python, Perl, Ruby, Java, and PHP. Also, it provides command line interface `virsh` to allow administrators can easily do management affairs.

2.1.2 OpenStack

OpenStack [10] is an Infrastructure as a Service (IaaS) cloud computing project. The project aims to deliver solutions for all types of clouds by being simple to implement, massively scalable, and feature rich. OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a data center, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface (*Figure 3*).

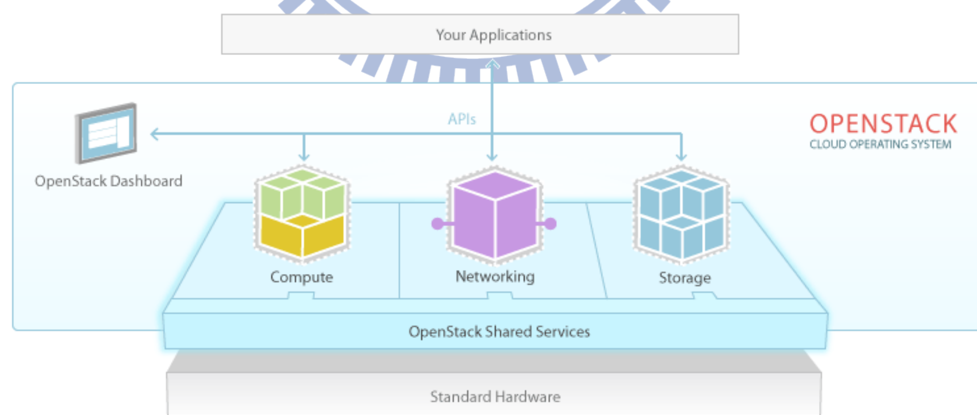


Figure 3: OpenStack overview (*source: OpenStack*)

The OpenStack Dashboard provides administrators and users a graphical interface to access, provision and automate cloud-based resources. OpenStack APIs are compatible with Amazon

†domain: An instance of an operating system (or subsystem in the case of container virtualization) running on a virtualized machine provided by the hypervisor

‡node: A single physical machine

EC2 [§] and Amazon S3 [¶] and thus client applications written for Amazon Web Services can be used with OpenStack with minimal porting effort. Developers can automate access or build tools to manage their resources using the native OpenStack API or the EC2 compatibility API.

2.1.3 Eucalyptus

Eucalyptus [11, 12] is an open source software for building Amazon Web Services (AWS)-compatible private and hybrid clouds.

As shown in *Figure 4*, Eucalyptus is designed to be compatible with AWS APIs so that users can leverage Eucalyptus commands to manage either Amazon or Eucalyptus instances. Cloud users can also move instances between a Eucalyptus private cloud and the Amazon public cloud to create a hybrid cloud. Eucalyptus leverages operating system virtualization to achieve isolation between applications and stacks. Operating system virtualization dedicates CPU, RAM, disk, and network resources to systems and applications so that they don't interfere with each other.

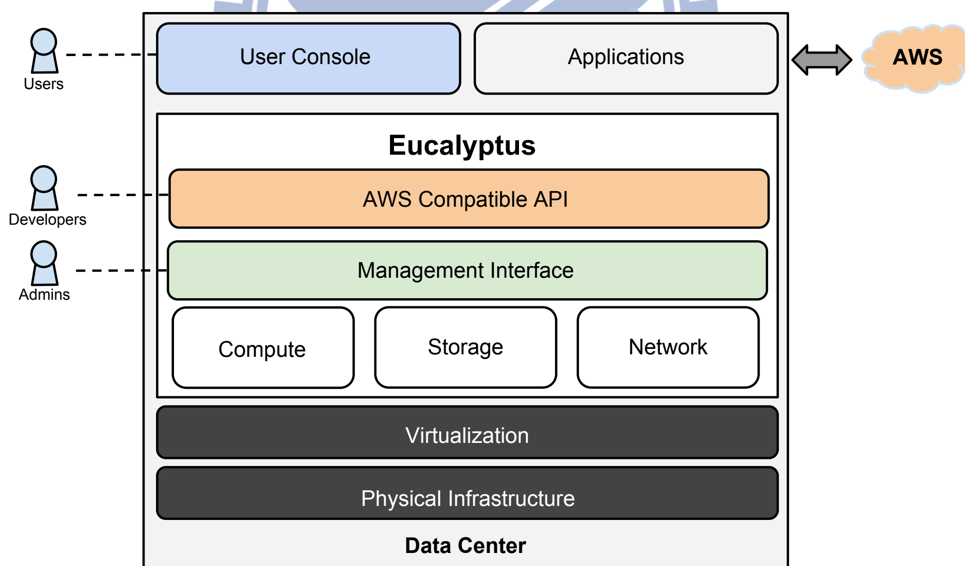


Figure 4: Eucalyptus platform architecture

Both OpenStack and Eucalyptus are Cloud Management Platforms (CMPs), and they meet

§Amazon Elastic Compute Cloud (Amazon EC2): A web service that provides resizable compute capacity in the cloud, see *http://aws.amazon.com/ec2/*.

¶Amazon Simple Storage Service (Amazon S3): An online file storage web service offered by Amazon Web Services, see *http://aws.amazon.com/s3/*.

the same need in cloud management field[‖].

2.2 Windows GUI Operation Tools

One of the biggest inconvenient things in the past testing methods is that testers need to involve in the manual operations in the guest OS to interactive with the target software or OS itself either using CLI (command line interface) or GUI (graphical user interface).

To achieve the goal of automatic testing, we are not willing to step into the testing process humanly. Hence we need some automation and programmable methods to handle the series of operations in guest OS.

2.2.1 Sikuli

Sikuli[**] [13] is a visual technology to automate and test GUI environment. Moreover, Sikuli is cross-platform, written in Jython, so users can use Python syntax and modules to extend when writing Sikuli scripts to identify and control GUI components either in Windows, OS X, or Linux.

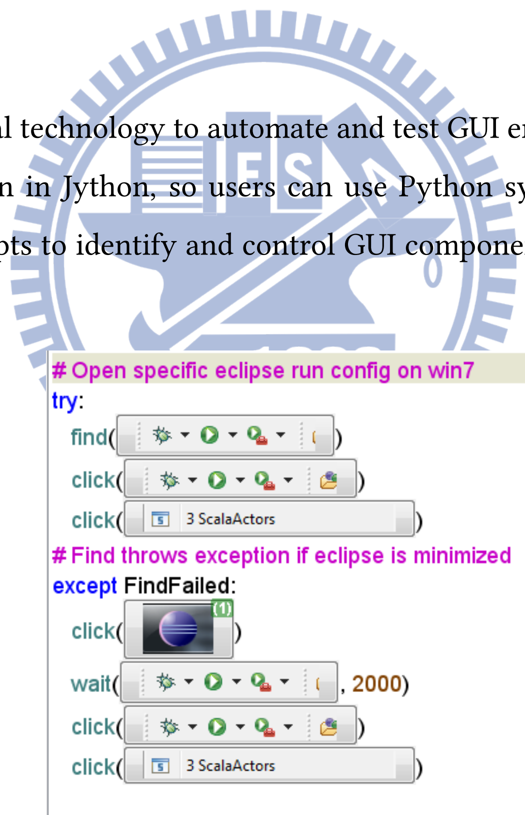


Figure 5: A sample of Sikuli script (*source: Sikuli.org*)

The most impressive feature of Sikuli is that it is triggered by image recognition (See *Figure 5*). So even if the resolution changes or GUI elements moves, Sikuli will work fine as well. On the other hand, Sikuli is said that it is slower than other similar tools which are triggered by elements

‖The comparison between CMSs could be seen at: *http://blog.opennebula.org/?p=4042*

**Sikuli Script: *http://www.sikuli.org/*

position or element ID cause it need to recognize image patterns on screen. Scripts will also not work if the theme style of OS or software changes.

2.2.2 AutoHotkey

AutoHotkey[††] is an automation software utility that allows users to automate repetitive tasks in Microsoft Windows. AutoHotkey scripts can be used to send keystrokes and send mouse clicks and movements. It is well designed for Windows environments and has very nice APIs to control Windows specific programs.

The cons of AutoHotkey are that its mouse moving is implemented by coordinate positioning. This sometimes causes the script fail since the windows size and appearance model are not always the same.

2.2.3 AutoIt

AutoIt[‡‡] [14] is an automation language for Microsoft Windows. It supports COM (Component Object Model) objects and Win32 DLLs functional calls, so users can precisely locate Windows objects without concerning about coordinates position and the change of theme styles.

2.2.4 Comparison

Sikuli is convenient with its image recognition system, and users can simply focus on the GUI operations and make a quick scripting on what they want to do. AutoHotkey has very nice support to keystrokes related operations, but AutoIt is more focus on GUI interactions than AutoHotkey.

We summarize the comparison between these tools in *Table 1*.

2.3 Software Testing

The past testing method is manual, and isn't repeatable efficiently. If an user wants to run the same test again, the user needs to do all the repetitive processes ones more. This is not

[††] AutoHotkey: *http://www.autohotkey.com/*
[‡‡] AutoIt, AutoItScript: *http://www.autoitscript.com/site/autoit/*

Table 1: Comparison between Automation Tools

Tool	Sikuli	AutoHotkey	AutoIt
Windows element recognition	image	-	COM object
Mouse movement	image	coordinate	coordinate
Resolution portable	yes	-	yes*
Theme style portable	-	-	yes*
Standalone executable support	-	yes	yes
Suitable environment to apply	Android emulator / Linux	Windows	Windows

* Note: Support of "resolution" and "theme style" portable for AutoIt are only restricted in those scripts using COM object positioning.

reliable since we human may make mistakes and the environment between two runs may be different.

In software engineering, we need to test our product in a reliable way to decide if the result is the same as what we expect. Through a well-designed routine of the testing process, we can focus more on our core jobs, says, software quality and functionalities. To make the testing process efficient and repeatable, we need to construct a testing framework to wrapper the details of testing processes, including those works done by human in the past.

We survey the test automation techniques, build a system that can control the execution of tests and compare actual outcomes to predicted outcomes [15].

2.3.1 Test Automation

Some software testing tasks, such as extensive low-level interface operations, can be laborious and time consuming to do manually. In addition, a manual approach might not always be effective in repetitive tasks. Once tests have been automated, they can be run quickly and repeatedly if needed.

In our cases, we use QEMU-based environments to test our targets. In most situations, we need to operate the guest VM to let it run some commands or take GUI actions. After that, we then need to take a look on host OS and to do something corresponding to the guest VM. Actually, the testing processes are always switching between the guest VM and host OS to and fro **manually**.

Therefore, we are truly required to make the process automated to reduce time and labor consuming. Then we note that we must make the test cases **runnable** first. Not running by human operation, not determined what to do next by human, but keep the test cases in a runnable format either through scripts or other ways.

Through test automation, we can automate previous repetitive but necessary testings in a formalized process, or add additional testing that would be difficult to perform manually.

2.3.2 Unit Testing / xUnit

Unit testing is one type of testing methods. It involves testing the fundamental units of the software, and usually carried out by writing code that tries out the target unit, checking inputs and outputs, one detailed factor at a time [16, 17].

By keeping such automated testing code, programmers can verify the correctness of their code. Software to manage these tests are often called code-driven testing frameworks, and various such frameworks have come to be known collectively as **xUnit**. The main advantage of xUnit frameworks is that they provide an automated solution with no need to write the same tests many times, and no need to remember what should be the result of each test.

These frameworks are based on a design by Kent Beck [18]. The design is originally implemented for Smalltalk as SUnit [19], and later, Erich Gamma and Kent Beck ported SUnit to Java, creating JUnit [20, 21]. From there, the framework was also ported to other programming languages.

2.3.3 Regression Testing

To shorten the software development cycle, making the test cases reusable is necessary. In a software development cycle, we may change testing parameters, improve algorithms, tune the environment settings, etc. To make sure the changes do not break the software functionalities, we are required to run a test on those testing cases we have used.

It is the target of regression testing. Regression testing is an integral part of the extreme programming software development method. The intents are to ensure that a change such as those mentioned above has not introduced new faults and to determine whether a change in one part of the software affects other parts of the software. Furthermore, we are not only concern

about the correctness, but we also interested in the improvement of the software after some changes have made.

Through a runnable testing case and well-formed testing framework, we can easily achieve these requirements by simply replace what we want to make changes of.

2.4 Related Implementations

Here we introduce two implementations of software testing framework: D-Cloud and ETICS. They are both designed for automatic testing, and want to give a solution for developers to figure out if there are some defects in software systems. We also summarize the cons of these two implementations, and try to solve these inadequacies in our design.

2.4.1 D-Cloud

D-Cloud [22] is a software testing environment using cloud computing technology and virtual machines, proposed in 2010. As shown in *Figure 6*, D-Cloud takes a pre-defined scenario as input. The testing environments such as hardware, software, or network are set through structured configurations. D-Cloud sets up a test environment on the cloud resources and executes tests automatically according to a given scenario. It provides fault injection facility to test the system working status and performance.

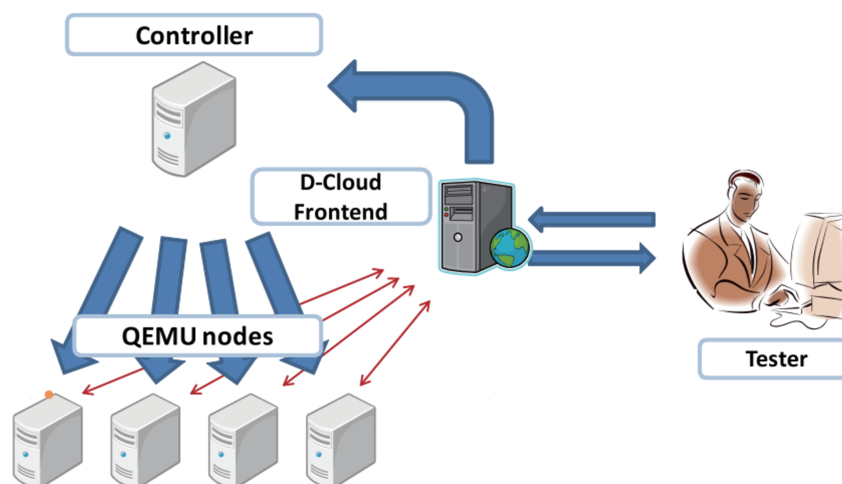


Figure 6: Flow of D-Cloud *(source: D-Cloud paper)*

2.4.2 ETICS

ETICS [23, 24] is an integrated infrastructure for the automated configuration, build and testing of Grid and distributed software, proposed in 2007. The architecture of ETICS is shown in *Figure 7*. It provides a service for software projects by integrating well-established procedures, tools and resources in a coherent framework and adapting them to the special needs of distributed projects.

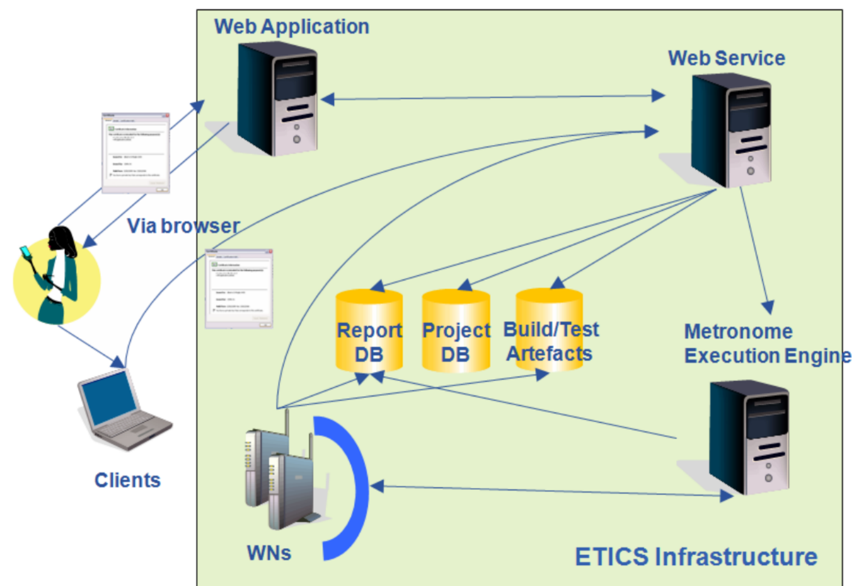


Figure 7: The ETICS System Architecture *(source: ETICS paper)*

1896

2.4.3 Inadequacies

Both the designs of D-Cloud and ETICS have following cons.

- Applicable testing platforms are restricted to Windows and Linux systems.
- Testing targets are restricted to non-interactive software that don't need human manipulations, such as web or database servers.
- Do not support GUI software testing since it usually needs interaction.
- Cannot accept parameterizable inputs, so users need to make another test case when they just want to change some factors.
- Testing functionalities are non-extensible or hard to extend.

Chapter 3

Method

In order to build a regression testing framework meeting our needs, we first illustrate the scenarios. By conceived the scenarios, we can know what requirements are needed to be provided in our system, such as what factors should be parameterized for flexibility and extensibility, what procedure should be designed differently for different testing targets, and what material should be provided to build a new test case.

With comparison to the current method, we also know what processes are needed as well as what processes could be pruned to save costs. And then we can start to build our system, design the architecture of the regression testing framework, and choose the techniques to be used.

3.1 Scenarios

For future extensibility, the applicability of the regression testing framework not only focus on current testing targets like Windows and Linux, but we hope it can also apply to more targets that use QEMU as a testing environment. We enumerate several testing targets here to demonstrate the using context but not limited to these scenarios.

3.1.1 Windows and Linux

We test vulnerable software on Windows and Linux, and trying to find usable exploit by feeding crash inputs to them.

On the whole, the testing processes taken in Windows and Linux are describe in *section 1.2*. At *stage 3*, we need to take operations in guest OS, through Linux command line via SSH login session or via VNC [25] (Virtual Network Computing) session if we are required to operate GUI environment.

In the software development cycle, we may change our symbolic or exploit generation tools to make it works efficiently, or we want to change the crash input file, GUI operation order, software startup parameters, etc. We will make these factors to be parameterized in our framework design.

3.1.2 Web Application

Automatic web testing method is also mainly based on symbolic execution to automate the web exploit generation process. Symbolic socket [26] is used to propagate symbolic execution through socket between applications. In QEMU, we set up web and database (DB) server with target web application running on it. Symbolic socket is implemented by replacing each crawled HTTP request with symbolic data which will be sent to web and DB server. In the other hand, there are also corresponding handlers for HTTP and DB query response to transferring received data to exploit generator at host OS. The overall architecture is shown in *Figure 8*.

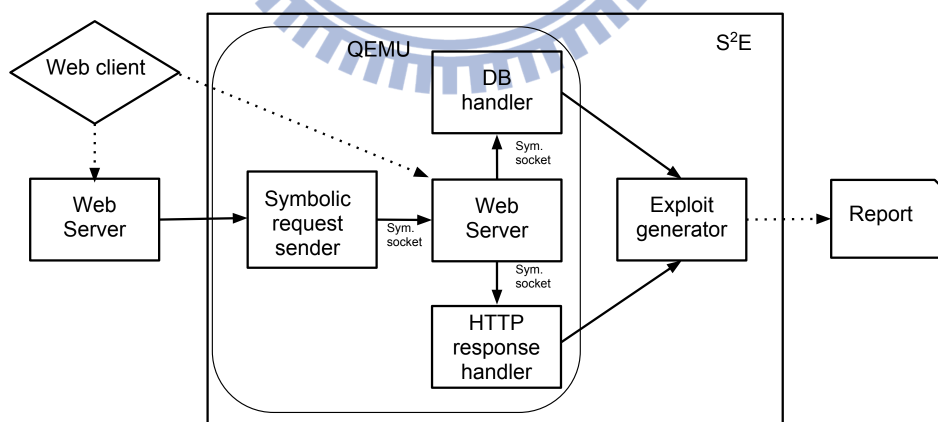


Figure 8: Overall architecture for web automatic testing

In web automatic testing, we will change the input query from web client, and the handler behavior may be changed, too. So make these part tunable in the framework will let the testing process reusable.

3.1.3 Android

As the smartphone market arising, there are more and more mobile applications (mobile apps) appear every day. This booming industry indicates the necessity of software testing.

The app developers need to test their product work normally no matter what version of mobile OS * or apps are. Also, mobile software distribution platforms providers need to know whether the apps uploaded by developers are vulnerable or risky.

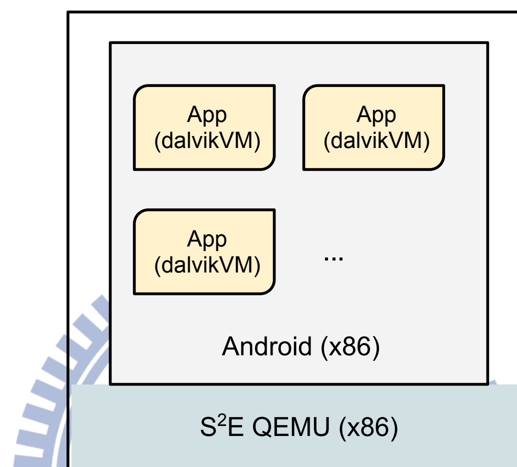


Figure 9: The architecture of x86-Android on top of S^2E

In *Figure 9* we can see, by symbolic execution and fuzz testing, we can satisfy these requirements. To run test on all versions of mobile OS and for each release of apps is nearly impossible in the past. Developers can only choose some popular combination to test if their product may work well. But with a regression testing framework, the problem are solved.

We can build the test cases one time, and run them easily whenever we make changes in a mobile system, either OS version, apps version, or other factors. The regression testing framework will be great beneficial to the mobile apps industry.

Since Android is an open source system and its marketing policy is open too, we will take Android apps as our testing target. By verifying apps not harm to us, users can get more protection.

*Mobile OS: i.e., mobile operating system. The most common mobile OS are Android, BlackBerry 10, iOS, etc.

3.1.4 Fuzzer

Fuzz testing is one of the common techniques of software testing. It often threats the tested applications under test as a black box, so it's especially useful in analyzing closed source software and proprietary systems since in most cases it does not require any access to the source code.

Fuzzer is a kind of fuzz testing tools, which will generate data or events to repeatedly feed the application with random input. The fuzzing result will be a valid input to the application, but will cause the application work improperly or even terminate unexpectedly. If the software crashes, we can use the input, i.e., crash input as material in automatic exploit generation. By feeding appropriate data as input instead of using truly random sequence, we are more likely to find software oversights or vulnerabilities. Improving fuzzing method is what the fuzzing technique are dedicated to.

We would be pleased if there is a testing framework that can help us test the efficacy and usability of new fuzzing method. If we run fuzzer in QEMU environment, we can achieve this.

3.1.5 Malware

Malware is software used by attackers to disrupt computer operation, gather sensitive information, or gain access to private computer systems. They may have bugs because they are also software, but current researches rarely focus on this field.

We are interested in find malware vulnerabilities and usable exploits cause that most of the malware users only focus on "attack" but not "defense". If we successfully find exploit worked on malware, we may have the ability to anti-attack.

On the other hand, one common way of anti-malware strategies is using anti-virus or anti-malware software. The method of detecting malware attacking is often by some specific behavior or functioning pattern. The detecting technique may also need regression testing after applying new method.

3.1.6 Embedded System

Embedded system is the most commonly present forms of computers in the world. The method of factory and acceptance testing in development period is usually black box testing by human or component unit testing on specific functions. If we can run embedded system software on QEMU-based emulator, we can have a reusable and quick testing method through our regression testing framework.

3.2 Requirements

As a summary of previous section, we can modify our past testing method in *Figure 2*, and transfer it to a new testing model as shown in *Figure 10*.

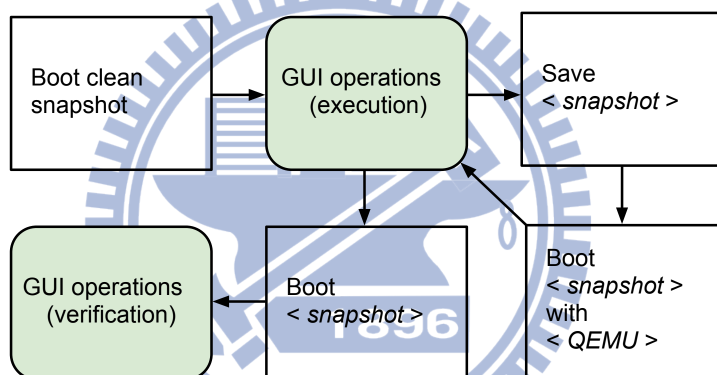


Figure 10: New testing model modified from past method

Then we can digest the requirements of our regression testing framework from this model and the scenarios described in previous section here:

- Setting up testing environment

  Before each testing, the host-guest VM environment should be set up first. The testing target need to be installed in the guest OS, as well as the S^{2}E tools to do *symfile* works. Furthermore, the network environment should also be set up to so that the framework could access to it.

- Operating QEMU via host

  Framework should have the ability to operate guest OS through QEMU, including guest OS booting, *savevm* and *loadvm* of snapshots, guest OS status monitoring, symbolic execution,

etc.

- Building shared file system

  Basically, guest OS need get crash input to run symbolic execution and generated exploit to do verification from host OS. Therefore, a shared file system needs to be provided in the framework to exchange files between host and guest OS.

- Building communication channel

  Guest OS and host need a method to communicate to each other within a test running. Without a communication channel, guest OS will not know when to do next step cause it doesn't know when host OS has done preparing those required works and vice versa.

- Recording output and errors in each testing

  As a regression testing framework, well-formed results and report are required undoubtedly. It is also hoped that the result of each testing can be easily found and viewed at any time.

- Automating operations in guest OS

  To automate the testing processes, human operations in guest OS should be avoided. So the operations in guest OS need to be transferred into programmable and automated methods through the GUI automation tools and scripts described in *section 2.2*

- Parameterizing tunable factors

  To keep flexibility, the regression testing framework is required to abstract as more as factors from core, and use the parameters got from test cases configurations.

- Providing API for creating test cases

  Creating a test case easily to run testing is one of the most important functionalities of a regression testing framework. In the design, appropriate APIs should be provided to let users can create their test cases through giving required material. The generated test case will be runnable and reusable once it is created.

- Managing running tests

  The regression testing framework is required to have ability to manage the tests running on it. Users may add new testing or deleting unwanted testing in queue through the

management interface. Also, viewing the progress of current running testing will be a nice feature in the framework.

- Verifying exploit

  After generating exploit, the regression testing framework needs to verify it to see if it is usable. To achieve this, the verifying procedure is required to be automated.

- Generating readable report

  After testing is done, users may want to get a detail report about the testing. A report that is well formed and easy to analyze will benefit users if they run a large scale of testing.

3.3 Our Method

In this section, we propose our design on the implementation of the testing framework. First, we introduce a hardware resource management layer on the bottom of the framework. Then we provide a management structure to handle various testing environments. Finally a communication method to let host and guest co-work to each other is designed.

3.3.1 Host Resource Management

We want to abstract the host resources from testing environment, thus we can do a large scale of software testing if we increase the computing power by adding hardware.

Therefore, we think of the concept of "cloud computing". Through cloud computing, the hardware resourcees will be virtualized, and the testing nodes running on the framework can share the computing resource. (See *Figure 11*)

3.3.2 Guest Machine Management

On the other side of guest machine management, we want a structured configuration of guest machine for each test case. In this case, we can boot up and run any prepared test case at any time if we want, and no need to worry about what combination of images, shellcode [†], or S^{2}E configurations should be used.

[†] A shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability.

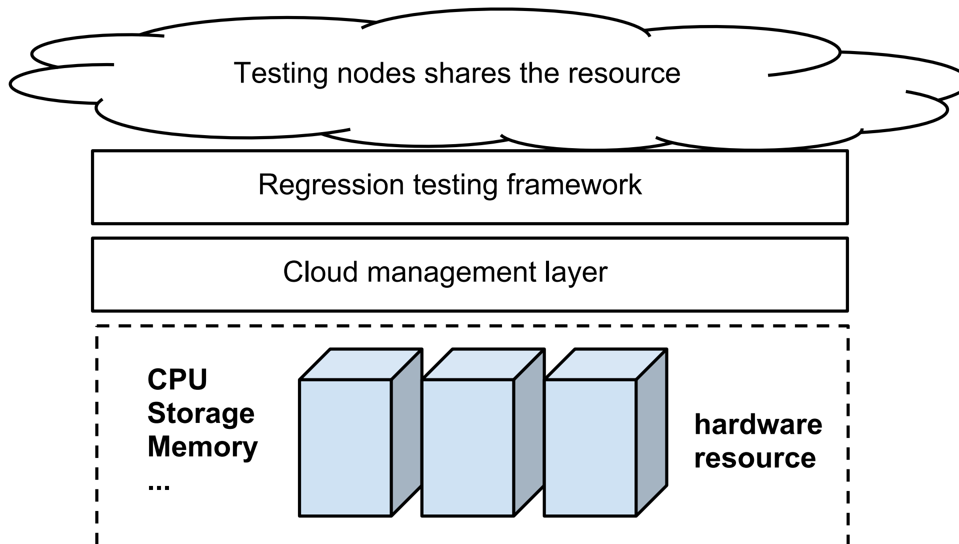


Figure 11: Framework over cloud computing

Furthermore, we want an overview of current running testing to monitor the test status.

3.3.3 Communication System between Host and Guest

To solve the manual operations problem on guest OS, we need a method to take responsibility of communications between host and guest. Therefore we testers don't need to copy the messages of one side to another or decide what the next action should be done according to the responses from either side.

In this situation, we have some choices to solve the problem, such as modifying QEMU to embed custom OP codes, or setting up a central sever to take responsible of delivering messages. Here we choose option of designing a simple protocol to let guest and host OS communicate to each other through network socket (*Figure 12*).

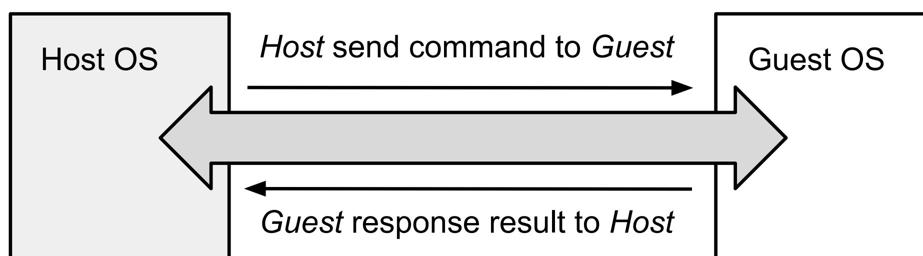


Figure 12: Communication channel outline

3.4 Windows GUI Object Manipulation

There are three parts in the past testing method that need manual operations in guest OS. Each part requires an alternative automation method to make the automatic testing work. In *section 2.2*, we have survey several tools that can be used in GUI operation automation, and here we summarize the method that will be used to solve the problem in each part.

3.4.1 Symfile

Before symbolic execution, we need to make the input symbolic through our own program in guest OS. The programs need to take input as a command line parameter and feed the data in S^2E mode after processing them. This procedure can simply completed by using the `cmd.exe` utility built-in in Windows with command of the full program and input path.

3.4.2 Software Operation

This part is more complicated than previous. There are great differences in types of software operations. The interfaces, the menu and theme styles, and even the methods of opening a file are all different.

Hence we need to use separate procedure for different target software according to actual circumstances. In this stage, we will use automation script corresponding to each target, to operate software acting as what we users will do on them. The script for each software is best able to adapt to variety of environments such as different OS version, different resolution or theme style, etc., thus we don't need to worry the script won't work if the environment changes.

3.4.3 Exploit Verification

At the last stage of regression testing, we need to verify if the generated exploit is usable. The most intuitive way is using the exploit on the target software to see the shellcode will run or not. Mostly, we set up the shellcode to run a `calc.exe` program. So if the exploit is usable, the target software using the exploit as an input will trigger a `calc.exe` process after opening or processing the exploit file as input.

In this step, we need a clean environment installed with the target software, and automatically operate the software to process the generated exploit. Then we also need to check if the `calc.exe` process has been triggered to run on system. If the answer is yes, the exploit pass verification. Otherwise, the testing result may be failed.

Chapter 4

Implementation Details

We separate the implementation of our regression testing framework into four parts. First of all, we handle the cloud-based architecture of managing hardware resources. Then we will implement the guest machine and QEMU images management, as well as the communication protocol between guest and host OS. Finally, we will collect the testing result of each run. Besides, we will show how to build a runnable and reusable test case for the regression testing framework in the end of this chapter.

4.1 Implementation Language

We choose Python as our implementation language. The considerations of choosing it to implement our regression testing framework are:

1. Portable

   If the code base is portable, we can share some common library to use on Windows, Linux, and other OS on any architecture without handling the cross-platform problem.

2. Native bidding library support

   In the regression testing framework design, we are going to use *libvirt* and maybe some cloud management tools. If these tools or library have native bidding for the language we chosen, we can directly use it in our implementation, and don't need to consider about the compatible problem if the low level library changes.

3. Scriptable

A programming language is scriptable means the source code written by the language can be run directly without compiling it. There are two advantages at least if our selecting is scriptable. First, we need to simplify the complicity of deployment. Once the codes are modified, we can directly deploy them without compiling them according to platform differences. Second reason is for creating test cases easily. Make the process of creating a test case as easy as possible, we can easily refine or tune it.

4. Easy maintenance

   The language chosen is better to have good syntax specification. Therefore the maintenance and future extensibility will be less pain.

4.2 Regression Testing Framework Architecture

In this section, we will describe the implementation detail of our regression testing framework from bottom to top.

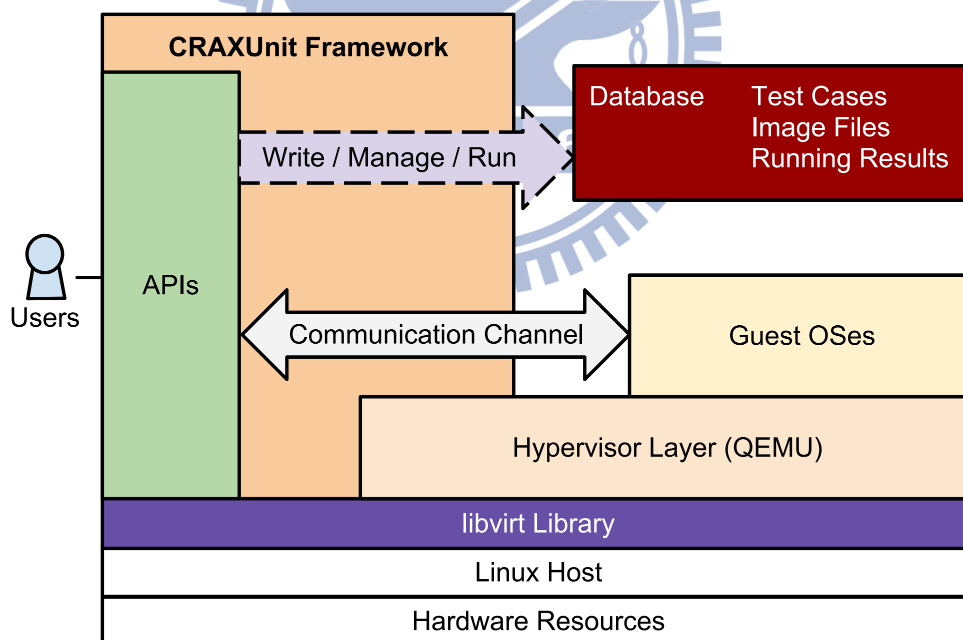


Figure 13: CRAXUnit Architecture

The architecture of our platform is shown in *Figure 13*, we named it *CRAXUnit*. We will introduce how we abstract the hardware resources, how we manage the environment settings, and how we control the guest and host OS to communicate to each other.

4.2.1 Cloud-based Management

We want to abstract the physical infrastructure from our framework by using cloud management concept. *Libvirt* has very comprehensive APIs for manipulating platform virtualization. It can be used to manage Linux KVM, XEN, VMware ESX [*], QEMU, and other virtualization technologies.

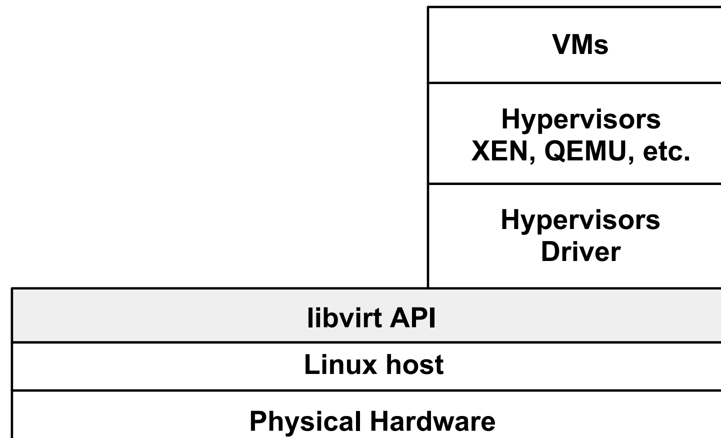


Figure 14: Hypervisor support by libvirt

As shown in *Figure 14*, *libvirt* implements a driver-based architecture, which allows a common API to service a large number of underlying hypervisors in a general way. This means we don't need to worry about what functionality one hypervisor have implemented. We can just use libvirt API to write our own applications, and the hypervisor could be switch from one to another in the future with few costs.

In the beginning, we use *libvirt* to control QEMU and the testing nodes to replace the manual management in our past testing method.

Figure 15 shows how our implementation will work based on *libvirt*. *Libvirt* implements a special daemon called *libvirtd* which runs on host. libvirtd provides the means to access local hosts resources if we need to integrate several hosts via network. The regression testing framework plays the role of management application, and communicates through the local libvirt to the remote libvirtd via a custom protocol. If we only have one host, the framework will directly control the local resource through libvirt API.

[*]VMware ESX is an enterprise-level computer virtualization product offered by VMware, Inc.
See *http://www.vmware.com/products/vi/esx/*

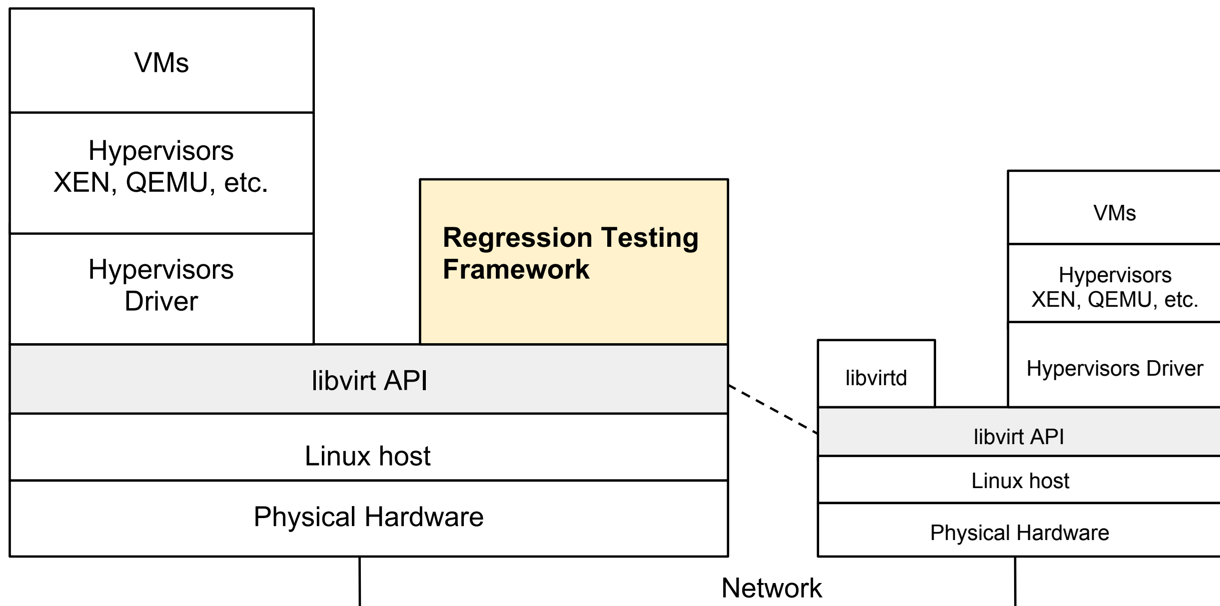


Figure 15: Regression testing framework on libvirt

4.2.2 Disk Images and Snapshots Management

QEMU disk images keep the hardware and software information of specific environments. For each independent test, the required image will be different. And the saved snapshots during the testing will also be saved in the same image file that used to boot the guest OS.

We save the images file collection information in our own configurations.In each testing, we can assign an existing image to be the environment to do the testing. Testing using the same specific image cannot be run simultaneously because of the file lock control policy.

And also, we keep the flexibility in our framework to switch from our own image management system to another one. We implement this by provide an attribute named `get_image_method` to indicate how to get the real image file path to use in one testing. Users can assign this attribute to another function which will use their own image management system to get the image file.

4.2.3 Connection Channel and Shared File System

We design a simple protocol for interchanging information between host and guest OS. Host need to signal guest in the following points:

- When booting, sending testing specific ID to guest that shared between guest and host

- Starting `savevm`
- Finishing `savevm`
- Starting Running in S^2E mode
- Starting verifying

And guest also need to send signal to host:

- After *symfile*
- Guest is in static state, signaling host to start `savevm`
- Verifying result

The implemented protocol commands are shown in *Table 2*. Command started with a `H` means it is used for host sending to guest; command started with a `G` means it is used for guest sending to host.

Table 2: Communication channel protocol commands

Commands	Description
`HUID`	Send testing unique ID
`Hsavevm`	Host start to savevm
`Hsavevmdone`	Host done savevm
`HS2Emode`	Host run as S^2E mode
`Hverify`	Host run as verify mode
`Gstatic`	Guest system is static
`Gsymfile`	Guest start to symfile
`Gsymfiledone`	Guest done symfile
`Gverifying`	Guest return verifying result

On the other side, we provide shared file system between guest and host OS using NFS [27]. It will be used to share files such as crash inputs, exploit, scripts, etc. since their size may be too big to send through a simple network socket protocol.

4.3 Regression Testing Framework

Till now, the components of the regression testing framework are well prepared. We can combine them and go through a scenario to see how the testing framework is working.

4.3.1 A Workflow of a Regression Testing

Here we describe how our regression testing framework (below we just call it "the framework") works.

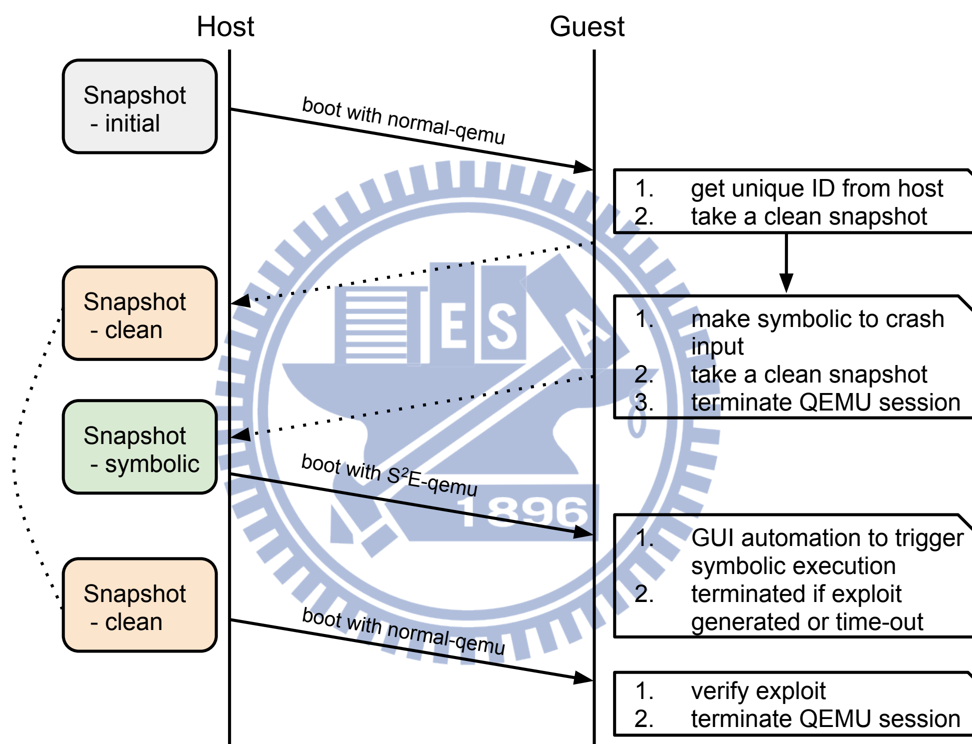


Figure 16: A workflow of single testing

First, when an user starts running a test case, the framework gets the configuration from test case about image file to use, crash input file, testing target, machine hardware settings, and other required attributes. (We will explain the detail of these information later in *section 4.3.2*.) Then the host boots the guest OS.

See *Figure 16*, The guest OS is prepared previously running a daemon that interactives with host, so after it finishing booting, it gets the testing unique ID from host as a key that is used during this testing.

First, the guest signals host to start making a first snapshot by `savevm` action to keep a clean environment that is used to verify generated exploit in the last step. Then the guest signals host again to make a snapshot after doing *symfile* to the crash input received from host. After the two snapshots taken, this booting session is terminated by QEMU.

Next, we go for the symbolic execution stage. The framework boots the snapshot with *symfile* state using S^2E version QEMU, and guest is signaled by host telling that it's S^2E mode, and start to run a GUI automation procedure to trigger the symbolic execution.

The host continues monitoring the state of symbolic execution and recording the output. This booting session is terminated in two situations. First, the exploit successfully generated and host receive a kill-state from S^2E. Second, the time lasting for symbolic execution has exceeded the time-out configuration in the test case. If the latter happens, the testing is terminated and the framework throws a `SymExecTimeout` exception.

If the exploit successfully generated, we can go through to the verification stage. The framework boots the snapshot with clean state using normal version QEMU, and send the exploit into guest. Then the guest starts running GUI automation procedure to verify if the exploit is usable or not.

If the guest system detects that a `calc.exe` process have been triggered after opening the exploit using the target software, it notifies host with successful signal. Otherwise, if the guest doesn't detect such process running within the time limitation, it signals host with an `ExploitRunTimeout` exception. The booting session is terminated in both case.

Finally, we are prepared to yield a general report about this testing. The testing report at least contains the following information:

- The basic configuration of this testing
- The success or failure of this testing
- If failure, the failure type or reason; if success, the time spent in each stage
- The detail output of the symbolic execution

Figure 17 shows an example of result output. It provides the basic information of a particular testing to the user, and can be reviewed in the future at any time.

```
============================
Test Report
============================
Name: office
Type: Exploit Generation
Date: 2013-05-30
----------------------------
OS:     Windows XP
S/W:    Office 2010
----------------------------
Image: Windows_XP_0
Input: CVE-2010-3333_DOC_2011-01-06.doc
symexec_offset: 35650
shellcode: 1
----------------------------
Stage 1: 1:35
Stage 2: 6:24
Stage 3: 0:45
Total: 8:44
----------------------------
Result: Pass
----------------------------
```

Figure 17: An Example of Result Output

4.3.2 A Sample Test Case

The required and optional attributes configurations in a test case are shown in *Table 3*.

First, we need to provide the configuration for the testing environment setting, i.e., image, testing target, and connection channel. Then we need to provide the crash input and shellcode required for the exploit generation process. Besides, we can set optional values to overwrite the default attributes that will be used during testing process, such as the information about offset, size, and timeout.

And a sample of test case to test exploit generation under Windows is shown in *Code 1*.

At *line 5*, we create a test case by using `CRF.test.Test()` method. It needs two arguments. The first one is the naming of this test case; the second is a Python Dictionary type object that indicates the attributes of this test case. Usable attributes are already described in *Table 3*.

We can run a test on the command line. Just use something like this:

```
$ python2 -m CRF -c 'CRF.test.run("test1")'
```

The framework will search the test case named *test1* under current directory, run a testing with it, and generate a readble result.

Table 3: Configurations in a Test Case

Configuration Entry	Description	Type	Required	Default Value
get_image_method	function to get image file	String	Optional	pget_image
image_id	QEMU disk image file ID	String	yes	
crash_id	crash input file ID	String	yes	
target_id	testing target software ID	String	yes	
machine_id	machine hardware configuration ID	String	Optional	default
autoit_id	AutoIt script ID to manipulate GUI	String	yes	
channel_port	connection channel port	Integer	yes	
symexec_timeout	timeout limit for symbolic execution	Integer	Optional	600
verify_timeout	timeout limit for verifying stage	Integer	Optional	60
symbolic_offset	offset for making symbolic on file	Integer	Optional	0
symbolic_size	length for making symbolic on file	Integer	Optional	(full file)
symexec_offset	offset for symbolic execution	Integer	yes	
symexec_size	length for symbolic execution	Integer	Optional	100
symexec_jmp_offset	jump offset for symbolic execution	Integer	Optional	200
symexec_jmp_size	jump length for symbolic execution	Integer	Optional	100
symexec_eip_offset	EIP offset for symbolic execution	Integer	Optional	100
symexec_eip_size	EIP length for symbolic execution	Integer	Optional	200
shellcode_id	shellcode ID to generate exploit	Integer	yes	
exploit_target	target process triggered by exploit	String	Optional	calc.exe

4.3.3 More Functionalities

Being a regression testing framework, our platform tries to provide useful functionalities as many as possible.

- **Showing useable test cases**: We can list the built test cases in our system (*Code 2*).
- **Selecting specific test cases**: When building a test case, we can tag it with attributes. Then we can select a bunch of test cases by these attributes and do some actions on them (*Code 3*).
- **Showing tested results**: We can review the test results of any test case which we have tested before (*Code 4*).
- **Regression testing wrapper**: When some parts in the system are being modified, we may want to take a test to see what will be impacted by this change. Thus we can use a wrapper code to handle this situation (*Code 5*).

Code 1: A sample test case: test1.py

```
1  # Import required module
2  import CRF
3
4  # New a CRF testing and give required attributes for this testing
5  test1 = CRF.test.Test('test1',
6      {
7          'test_type': 'exploitgen',
8          'image_id': 'Windows_XP_0',
9          'crash_id': 'CVE-2010-3333_DOC_2011-01-06.doc',
10         'target_id': 'Office_2003',
11         'autoit_id': 'office_1',
12         'channel_port': 10354,
13         'symexec_offset': 35650,
14         'shellcode_id': 7
15     })
16
17 test1.setTag([
18     'Windows', 'Office', 'Windows_XP', 'Office_2003', 'CVE_2010',
19 ])
```

Code 2: Showing test cases example

```
1  # list all test cases under a directory
2  CRF.admin.listAllCase('/home/data/tests/windows')
```

We list the basic types in *CRAXUnit* in *Table 4*, and the APIs currently provided in *CRAXUnit* are listed in *Table 5*.

Table 4: Basic type in CRAXUnit

Type name	Description
`Test`	A runnable test case including required information
`Set`	A collection of **`Test`**, but not runnable
`TestSet`	A runnable test set composed of a **`Set`** including required information

Code 3: Selecting test cases example

```
1 set0 = CRF.test.selectAll('/home/data/tests/windows')
2 set1 = set0.selectByTagAnd('Windows', 'Office')
3 set2 = set1.selectByTagOr('Windows', 'Office')
```

Code 4: Showing test results example

```
1 date1 = time.mktime((2012, 12, 21, 0, 0, 0, 0, 0, 0))
2 date2 = time.mktime((2013, 2, 1, 0, 0, 0, 0, 0, 0))
3
4 # Show all results for specific test
5 CRF.admin.showTestResults('test1')
6 # Restrict the result between a date range
7 CRF.admin.showTestResults('test1', date1, date2)
```

Code 5: An example of testing wrapper

```
1 set0 = CRF.test.selectAll('/home/data/tests/windows')
2 set1 = set0.selectByTagAnd('Windows', 'Office')
3
4 test2 = CRF.test.TestSet(set1,
5     {
6         'symexec_offset': 21480,
7         'shellcode_id': 7   }
8 ).run()
```

Table 5: APIs in CRAXUnit

Class	Method	Description
test	Test	Create a new test case
test	TestSet	Create a new test set with a set
test	selectAll	Select all test case as a set
test.set	selectByTagAnd	Select a set by tag using AND logic
test.set	selectByTagOr	Select a set by tag using OR logic
test.testCase	run	Run a test case
test.testCase	setTag	Set tags to test case
test.testSet	run	Run each tset case in a test set
admin	listAllCase	List all test cases
admin	showTestResults	Show tested results

Chapter 5

Results

Building a reusable software testing method on QEMU-based system as the goal, we implement the regression testing framework that can easily create, manage, and run tests. We also create an exploitable test case database for the framework using. Thus, we can do software testing and verify our modification to the exploit generation method efficiently.

In this chapter, we will show the experimental result of our implementation and compare it to some similar works.

5.1 Regression Testing Framework for QEMU-based System

The regression testing framework can provide the following functionalities:

- Test cases creation, managing, and running

  We provide a series of API to let testers to manipulate test cases. Through these APIs, one can create tests easily with providing required entries. Later, she or he can classify these tests into different tags according to one's purpose. Test cases maintainers can choose to list these tests by their tags or other attributes, or choose to view previous tests result by constraints. The last, as the most important part of the regression testing framework, one can efficiently run a batch of test cases on demand, and providing different arguments to change the testing method is also acceptable.

- Automation testing procedure

  The past method used on QEMU-based system testing need human intervention in guest

OS. We replace the manual GUI operations with programmable GUI automation scripts and communication channel that can send control message from host to guest. As a result, we can now run through the whole testing procedure without getting into guest OS manually. It's a more efficient and innovation way to do the testing.

- Regression testing ability: parameterizable

  Our framework reserve flexibility for testers to customize their wanted testing. Either testing target, testing environment, S^2E version, shellcode, or symbolic offset are all parameterizable. This feature makes the framework have the ability to do regression testing, both for previous test cases verification and future development testing.

- Flexible extensibility: extending connection channel commands

  We use a connection channel with self-defined communication protocol to handle the actions taken on guest OS and host OS. We provide a series of commands to control some common actions. Developers can extend the protocol by simply adding their own commands and corresponding actions to guest OS by themselves, then the framework will be able to do more extensive testing.

5.2 Testing Branch Database

In our framework, we have rebuilt a test case database from the existed exploitable testing. Those messes such as images files, configurations, and crash inputs we need to handle manually are not needed any more. We can run a test case just using the database to select out what need to be tested.

The built exploitable test cases in our database are shown in *Table 6*.

5.3 Improvement

In order to test our method, we compare our testing framework with others' and the past method to show what functionalities our platform can provide.

Table 6: Exploitable Test Case in Database

#	OS	Software	Crash Input	Image
1	Windows XP	Office 2003	CVE-2010-3333_DOC_2011-01-06.doc	Windows_XP_0
2	Windows XP	Coolplayer	CVE-2008-3408.mp3	Windows_XP_1
3	Windows XP	Distiny	CVE-2009-3429.mp3	Windows_XP_2
4	Windows XP	Dizzy	EDB-ID-15566.mp3	Windows_XP_2
5	Windows XP	GAlan	OSVDB-ID-60897.galan	Windows_XP_2
6	Windows XP	GSPlayer	OSVDB-ID-69006.mp3	Windows_XP_1
7	Windows XP	MPlayer	EDB-ID-17013.mp3	Windows_XP_1
8	Windows XP	Foxit Reader	OSVDB-ID-68648.pdf	Windows_XP_3

5.3.1 Time and Efficiency Comparison

The beginning motivation of our work is to build a framework that provides automatic regression testing without human intervention. Here we list the comparison of testing work using the past method and our CRAXUnit framework in *Table 7* and *Table 8*.

Table 7: Comparison of running one testing

method	past method	using CRAXUnit
time consuming	10 minutes	8 minutes
operation type	human monitoring / operation all the time	just one command
tunable factors	manual	parameterizable input

Table 8: Comparison of running a testing benchmark

method	past method	using CRAXUnit
test case numbers	7	7
time consuming	1 hour 23 minutes	45 minutes
operation type	human monitoring / operation all the time	just one command
involved configurations	7	none (set in test cases)
tunable factors	manual	parameterizable input

5.3.2 Feature Comparison

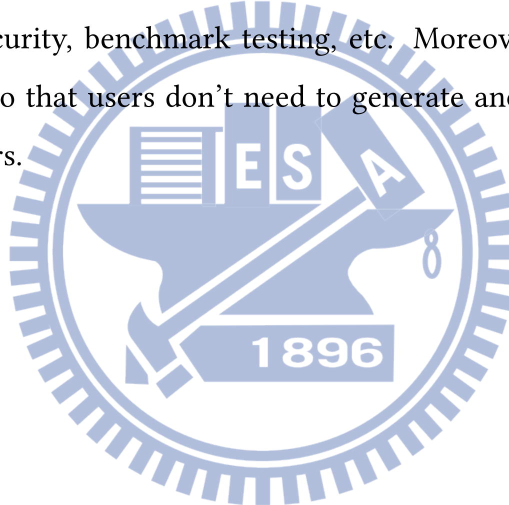
Table 9 shows the comparison between the related implementations mentioned previously and our framework. Our platform, *CRAXUnit* supports almost all QEMU-based system testing, and extending the functionalities of the framework is an easy work just through writing plug-ins

Table 9: Comparison between Frameworks

Framework	D-Cloud	ETICS	CRAXUnit
Image Mgmt.	yes	-	yes
Test case Mgmt.	yes	yes	yes
Parameterizable input	-	-	yes
Target type	Windows/Linux	Windows/Linux	QEMU-based system
Guest operations	-	-	yes
Distributed testing	yes	yes	yes*
Test application type	fault testing	fault testing	fault, security, benchmark testing
Extensibility	modify QEMU	-	Python plug-in

* Note: S^2E only supports one host testing currently.

in Python. Therefore, by extensibility features, we can use *CRAXUnit* to execute not only fault testing but also security, benchmark testing, etc. Moreover, our framework can accept parameterizable inputs so that users don't need to generate another test case when they just want to tune some factors.



Chapter 6

Conclusion and Further Work

In this final chapter, we summarize our work and list some further work to refine our method.

6.1 Conclusion

In this thesis, we implemented a regression testing framework for software testing on QEMU-based system, which is applicable for testing work on Windows, Unix-like system, Android and testing target as fuzzer, malware, embedded system and so on.

Our framework provides an innovative method for software testing. Usually those testing involve GUI operations need human intervention using past methods, and testing based on QEMU also need human to jump into guest OS to manipulate in guest environments. Our method makes the whole testing automatically without any manual operations by establishing a channel to take control between guest and host OS.

As a well-formed framework, we also provide APIs for creation, managing, and running test cases. Tests can easily be operated and repeated through these APIs, and testing results are also managed properly that can be selected and reviewed at any time. Our framework architecture also allows users to do more type of testing by simply extending its functionalities if the built-in ones are not sufficient.

In order to run experiments on our framework, we also rebuild an exploitable test case database transformed from previous manual test cases. Some scenarios that have not been exploited successfully yet have also been created as test cases in our database waiting for method

improving in the future.

6.2 Further Work

Here we propose some further work to implement or to improve that can refine our work:

- Environments setting up and GUI automation method improvements

  The initial disk image files used by our framework require users to customize the environments and install a daemon for the framework. Since these procedures take really a long time in test case preparing stage, it will be better to have an automatic initializing method. On the other hand, the GUI automation scripts we use now are made case by case. To quickly build large amount of test cases, we can take one basic script and modify from it to adapt to dedicated cases, but this is still not efficient. For long-term considerations, we are better to propose a more efficient method to solve the GUI automation problem.

- Running queue management

  Though that we use cloud management techniques to handle underlying hardware resources, the framework still can deploy only one testing instance at a time. This is because our framework doesn't provide parallel APIs. A test will be deployed only when the previous one terminated. If users want to run several tests in parallel, they need to handle the simultaneous problem by using `fork` system call or issue several commands successively. It will be a good enhancement if the framework supports running queue management. Users can add test tasks into queue, and the tasks can be issued depending on hardware resources automatically or by users' decision.

- Testing ability for distributed computing

  The current architecture of S^2E is only support single-machine software testing. Since we are using S^2E as our core element, the range of testable target is restricted as well. To achieve this goal, our framework should have the ability of booting several testing instances at once and communication between co-working guests. The trend of information industry is toward to distributed computing, and more and more software are starting to support distributed co-working. So if our system can support real distributed system testing, the testing coverage will be raise up and comprehensive.

Reference

[1] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[2] Corina S Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer*, 11(4):339–353, 2009.

[3] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331. IEEE, 2010.

[4] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1066–1071. ACM, 2011.

[5] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGARCH Computer Architecture News*, 39(1): 265–278, 2011.

[6] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. CRAX: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 78–87. IEEE, 2012.

[7] QEMU - open source processor emulator. `http://qemu.org/`.

[8] Fabrice Bellard. QEMU, a fast and portable dynamic translator. USENIX, 2005.

[9] libvirt: The virtualization API. `http://libvirt.org/`.

[10] OpenStack open source cloud computing software. `http://www.openstack.org/`.

[11] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, pages 124–131. IEEE, 2009.

[12] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. Eucalyptus: A technical report on an elastic utility computing architecture linking your programs to useful systems. In *UCSB TECHNICAL REPORT*. Citeseer, 2008.

[13] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. Sikuli: using GUI screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 183–192. ACM, 2009.

[14] Jason Brand and Jeff Balvanz. Automation is a breeze with AutoIt. In *Proceedings of the 33rd annual ACM SIGUCCS fall conference*, pages 12–15. ACM, 2005.

[15] Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management*. Wiley-IEEE Computer Society Press, 2007.

[16] Kent Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

[17] Kent Beck and Cynthia Andres. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.

[18] Kent Beck. Simple smalltalk testing: With patterns, 1999.

[19] SUnit - the mother of all unit testing frameworks. `http://sunit.sourceforge.net/`.

[20] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[21] Erich Gamma and Kent Beck. JUnit: A cook's tour. *Java Report*, 4(5):27–38, 1999.

[22] Takayuki Banzai, Hitoshi Koizumi, Ryo Kanbayashi, Takayuki Imada, Toshihiro Hanawa, and Mitsuhisa Sato. D-Cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 631–636. IEEE Computer Society, 2010.

[23] Marc-Elian Bégin, Guillermo Diez-Andino Sancho, Alberto Di Meglio, Enrico Ferro, Elisabetta Ronchieri, Matteo Selmi, and Marian Żurek. Build, configuration, integration and testing tools for large software projects: ETICS. In *Rapid Integration of Software Engineering Techniques*, pages 81–97. Springer, 2007.

[24] Francisco J González-Castaño, Javier Vales-Alonso, Miron Livny, Enrique Costa-Montenegro, and Luis Anido-Rifón. Condor grid computing from mobile handheld devices. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(2): 18–27, 2002.

[25] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual network computing. *Internet Computing, IEEE*, 2(1):33–38, 1998.

[26] David Brumley, Sang Kil Cha, and Thanassis Avgerinos. Automated exploit generation, December 13 2012. US Patent 20,120,317,647.

[27] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985.