Augmenting System Testing

with Symbolic Execution

by

Guodong Zhu

(Under the Direction of Kang Li)

Abstract

Over the past decades, our society has become increasingly dependent on software. They are used in all sorts of different areas such as business, hospitals, crime detection and even to fly planes. One minor software vulnerability might be enough to lead to damage that is unrecoverable. Even though manual software testing, is the most prevalent technique used for software security analysis, is laborious and prone to human error. With the increase of our reliance on computer software, the desire of developing systematic techniques that check the system software we use for critical vulnerabilities increases.

In this dissertation, we investigate the techniques of system software testing for identifying security vulnerabilities and then expand the system software testing techniques by addressing two problems: increasing code coverage and triggering vulnerability. We first explore the feasibility of identifying security vulnerabilities in the implementation of virtualization hypervisor by extracting the implementation of the virtual devices and

modeling them with symbolic execution. Exercising virtual devices independently with symbolic execution engine without the full virtualization running enables the possibility of discovering vulnerabilities with the novel technique.

In the second part of the dissertation, we show how different system software testing techniques can be integrated together to improve the effectiveness by investigating Sezzer, a framework that incorporates fuzzing and symbolic execution yet overcomes the major disadvantages from both of them. Our experiments on different benchmark binaries as well as real-world applications, shows that Sezzer not only outperforms modern system testing techniques in most cases, but also can find security-critical bugs in realworld applications. With the preliminary realworld testing, we have found 6 unique vulnerabilities in GNU-Binutils which resulted in 5 patches and 3 CVEs.

INDEX WORDS: Fuzzing, Symbolic Execution, Systems Testing, Vulnerability Discovery

Augmenting System Testing

with Symbolic Execution

by

Guodong Zhu

A Dissertation Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2018

AUGMENTING SYSTEM TESTING

WITH SYMBOLIC EXECUTION

by

GUODONG ZHU

Approved:

Major Professor: Kang Li

Committee: Kyu Hyung Lee
                     Roberto Perdisci

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
Auguest 2018

Acknowledgments

I couldn't have finished this without the support from all the people I met throughout my academic career.

My supervisor, Kang Li, who has been a constant source of support and inspiration. His non-stop optimism that we could get things to work, even when they seemed impossible, was crucial throughout my time here. He was always there when I needed him and taught me how to be a good researcher. I feel very fortunate to having worked with him and I wouldn't have gotten this far without his mentorship.

I truly acknowledge the valuable time, patience, support of my outstanding thesis committee: Kyu Hyung Lee and Roberto Perdisci. Their insightful comments and advice helped crystallize the vision of the thesis and steers my research towards the correct direction.

I'd like to present my sincere thankfulness to my dear father and my deceased mother, who passed away last May, for their great role in my life and their numerous sacrifices for me and the rest of the family, thank you and I love you forever. Thanks to my sisters Yanli and Yingli, my brother Guoxu for their support throughout my journey and everything they have done for me.

Last but not least, I'd like to express my deepest gratitude to my wife, Song Qi, for her patience and tolerance over the last several years. I could not be able to finish this work without your support. Thank you for being with me and for your appreciated sacrifices. Thank you my cute little son Alex, for being good kids with your mother when I was studying and doing research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software bugs are expensive. A single software flaw is enough to take down spacecrafts [14], make nuclear centrifuges spin out of control [56], or recall 100000s of faulty cars resulting in billions of dollars in damages[36].Worse, security-critical bugs tend to be hard to detect, harder to protect against, and up to 100 times more expensive after the software is deployed [49].The need for finding and fixing bugs in software before they become critical has led to the rapid development of automatic software testing tools.

While software testing, is getting more attentions from researchers, the discovery of the the majority of security vulnerabilities are still relying on manual effort [19]. Automatic testing on the other hand allows developers and users to analyze their software for bugs and potential vulnerabilities automatically. From blackbox random fuzzing [63, 64] to whitebox path-sensitive analyses [43, 44], the goal of automatic testing is the same: identify as many real flaws as possible with minimal user augmentation. Every discovered flaw is usually accompanied by a test case, an input that forces the program to exhibit

the unexpected behavior. Test cases eliminate false positives, ensure reproducibility, and provide the developer with concrete and actionable information about the underlying problem. Even when no flaws are identified, the set of exercised test cases serves as a regression test suite, a standard software engineering practice [69].

An increasingly popular software testing technique is symbolic execution [52, 48, 25]. Unlike manual or random testing, symbolic execution systematically explores the program by analyzing one execution path at a time. For every feasible path, symbolic execution generates an input that exercises the path and then checks for potential vulnerabilities. Over the past decade, numerous symbolic execution tools have appearedfhboth in academia and industryfhshowing the effectiveness of the technique in a vast number of areas, including finding crashing inputs[30, 42], generating test cases with high coverage [29], creating input filters [37], exposing software vulnerabilities [24], and analyzing malware [28].

Another technique that is currently widely used in system software testing is grey-box fuzzing. The term grey-box fuzzing is mentioned for the first time in 2007 [38]. However, the idea of grey box fuzzing originates way before that. It can briefly be explained as a fuzzing technique located between black- and white-box fuzzingcitepmiller1990empirical, miller1995fuzz, godefroid2008automated, godefroid2012sage, where black-box means testing the target system with zero knowledge and only look at the input/output, and white-box testing assumes source code is available. Grey-box fuzzing does not require source code to be available, but is can collect the feedback regarding the internal state of the target application being tested, this is usually achieved by instrumenting the target application with static analysis techniques before-hand or with dynamic instrumentation on

the fly during the execution of the target application [79, 57, 32, 71].

This thesis investigate the feasibility of identifying security vulnerabilities in the implementation of virtualization hypervisor by extracting the implementation of the virtual devices and modeling them with symbolic execution (Chapter 2 and Chapter 3). One of the problems of applying symbolic execution technique in real-world system software testing is that when the target application gets complex, the path constraint becomes harder to solver as the execution goes on and eventually overwhelms the constraint solver, even with all different countermeasures such as cache citepcadar2008klee and simplification [29, 43, 73]. Exercising virtual devices independently with symbolic execution engine without the full virtualization running enables the possibility of discovering vulnerabilities with the well-researched and promising technique.

Even though significant amount of research efforts have been put on improving symbolic execution [15], there are still two fundamental problems that prevent it from being well adapted: path explosion and constraint satisfactory reasoning. As the other most prevalent automatic system software testing technique, fuzzing suffers from the limitation of random mutating and falls short when the target application requires specific input to be able to continue explore the execution paths and increase coverage. We investigate Sezzer Chapter 4, a binary only software testing framework that take advantages from both symbolic execution and grey box fuzzing for better code coverage and target specific vulnerability discovery. We utilize symbolic execution to mitigate fuzzing's limitation on specific condition bypassing and using fuzzing's coverage information to steer the execution of symbolic execution in return to reduce the number of state forks and mitigate path explosion problem.

Fuzzing is used to explore the general conditions of the target program, and with the basic coverage information from fuzzing, concolic execution is used to help getting over the complex constraint checks on the execution path of the application as well as check for specific types of vulnerabilities during the execution which can otherwise be hard to find by fuzzing. The alternation of the two techniques eliminates several major disadvantages from both fuzzing and concolic execution and give SEZZER the ability to find more execution paths and get higher code coverage.

1.1   Contributions

This dissertation makes the following high-level contributions:

1.1.1   Virtual Device States (Chapter 2)

- We differentiated the requirement of device state migration from the application/OS memory migration, and exposed the potential problems of missing key state during live migration.

- We developed an environment that can mimic live migration of virtual machines in a program controlled way. With our environment, a tester can choose at which line of code a live migration event occurs.

- We proposed a solution that extracts the key states of a device by applying program analysis to virtual device implementations, which are essentially software programs. A software tool that does the automatic key device state extraction is current under development .

1.1.2   Detecting Virtualization Specific Vulnerabilities (Chapter 3)

- We identified several types of virtualization specific flaws based on the recently discovered vulnerabilities in virtualization platforms.

- We proposed approaches to detect three types of virtualization specific vulnerabilities.

- We designed a framework to implement these methods to detect virtualization specific flaws .

1.1.3   Augmenting Fuzzing with Concolic Execution (Chapter 4)

- Propose the approach to improve the coverage of software testing by alternating between fuzzing and selective concolic execution. We use selective concolic execution to generate testcases that help fuzzing getting over some complex constraint checks that are otherwise hard to solve. Also mitigating the path explosion problem by using the testcases and coverage information generated by fuzzing and only fork at the places where the newly forked branch can lead to new code coverage Chapter 4.

- Designed and implemented the SEZZER framework to carry out the idea and evaluated the strength of the approach by comparing the testing results with several other state of the art software testing techniques and by discovering several real world vulnerabilities .

Chapter 2

Deducing Key Device States for the Live Migration of Virtual Machines

2.1 Introduction

Cloud computing revolutionizes the IT field by allowing users to access computing resource without physically own the hardware equipments. Virtualization is the underneath enabling technique, which provides the ability to launch, host, and migrate VMs in cloud environments on demand. Among all these cloud computing features, live migration of virtual machines provides significant benefit in resource management and fault tolerance.

Live migration of virtual machines consists of application/OS memory state migration and device state migration [35]. To make the live migration transparent to cloud users, the switching of VMs across physical hosts needs to be as fast as possible. To

6

achieve fast migration, previous work [46, 60, 65] have proposed to use on demand memory copies. Once the virtual machine device states and a small OS footprint have been copied over, a VM can start to execute on the new host with additional application and OS memory copied over when needed.

Because migrating virtual device states is the very first step [46], it is critical to minimize the size of memory being copied in this round. Therefore, for all the memory related to those virtual hardware, only the key states (such as device I/O registers) that governs the upcoming behaviors of the devices need to be migrated. Using QEMU [21], a popular VM platform, as an example, its current practice is to have the virtual device developers specify what are the states need to be saved for the migration. These states are usually selected manually by inspecting the hardware specification and software driver implementation. However, this manual effort is error prone. If some important states are not included, a virtual device may not behave properly after the migration. Moreover, these problems are sometimes hard to detect. Missing device states does not always lead to an immediate crash of the VMs or the virtual devices, and errors could occur at a much later stage of VM execution.

We proposed a project to study the implementations of various VM platforms, especially their behaviors during live migration. The proposed efforts include developing environments to evaluate the virtual device implementations at the time of live migration, developing tools to analyze VM platforms and extracting critical device states, and scanning current VM platforms to detect implementation bugs.

This paper presents our initial effort of studying the virtual device behavior at the time of live migration. In this paper, we present the design of **VDSChecker**, a tool that

can automatically check the completeness of the transferred device state during the live migration of VMs. By exploring the the difference in the behavior of the same virtual device between whether a live migration is involved during the execution, we can detect whether the migrated device state is sufficient for the virtual device to continue running as expected, and which device state is missing if not.

This paper makes the following contributions:

- We differentiated the requirement of device state migration from the application/OS memory migration, and exposed the potential problems of missing key state during live migration.

- We developed an environment that can mimic live migration of virtual machines in a program controlled way. With our environment, a tester can choose at which line of code a live migration event occurs.

- We proposed a solution that extracts the key states of a device by applying program analysis to virtual device implementations, which are essentially software programs. A software tool that does the automatic key device state extraction is current under development.

2.2   Virtual Devices and Key Device State

Although live migration can be applied on many full system virtualization platforms, our efforts focus on QEMU [21] virtual devices. We choose QEMU for two reasons. First QEMU is open source, meaning all the virtual device implementations are publicly

available and easily accessible. Second, QEMU serves as the basis for many other widely used virtualization platforms such as KVM[53], XEN HVM[18], and VirtualBox[78].

2.2.1   QEMU and its Virtual Devices

QEMU is a generic CPU emulator and virtualizer that is able to emulate many different processor architectures including x86, SPARC, and ARM. As a full system virtualization platform, QEMU is able to emulate an elaborate set of peripheral devices including keyboards, mice, disk controllers, graphics cards, and network interface cards. The programs that emulate these devices are called *virtual devices*. These virtual devices are usually modeled after existing pieces of hardware. For example, the e1000 virtual device is modeled after Intel's E1000 gigabit network card.

The layout of a QEMU virtual device implementation shares many similarities with a device driver code. Most devices define a state structure, and register a set of interface functions to enable communication with the guest operating system or the environment through QEMU, these interface function then invoke transaction functions to handle the requests or the environment variables.

2.2.2   Live Migration of QEMU

Live migration includes device state migration and memory state migration. QEMU implemented post-copy live migration [47], as shown in Figure figure 2.1, with which the CPU and device state will be transferred before the memory state. During the migration, the virtual machine will be brought down on origin host when migrating the device state and continue running on destination host as soon as the CPU and device state be
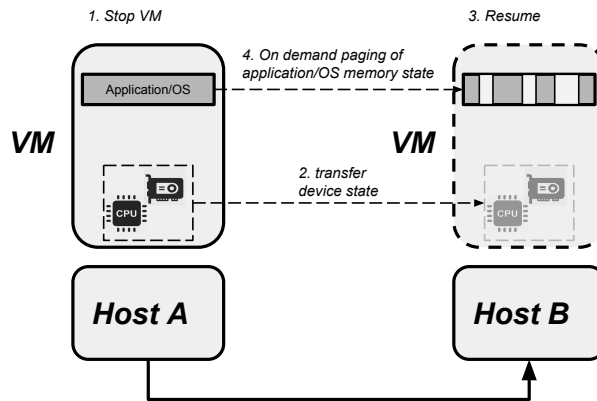
Figure 2.1: QEMU post copy live migration

transferred, the memory state will be transferred to the destination host along the normal execution of the virtual machine.

2.2.3 Key Device State

When a virtual device gets migrated from one host to another, not all its variables need to be copied during the migration to ensure the correct execution. Only a subset of all the virtual device implementation's variables are needed and we call them **key device states**, which govern a virtual device's upcoming behavior after migration.

Each QEMU virtual device implementation has a specific piece of code that defines what variables to save/load at the time of migration. These device state variables are defined in the *VMStateDescription* structure, which is manually specified by the developer. Figure figure 2.3 shows code excerpts for e1000 implementation of *VMStateDescription*

structure.

This *VMStateDescription* data structure is supposed to capture all the key device state variables and only those variables. Containing variables that are not critical to the virtual device will affect the performance of the live migration, because the downtime of the live migration of QEMU is heavily dominated by the time spent on copying the device state[46]. Missing key device state variables from the *VMStateDescription* data structure is even worse, which case lead to misbehaviors and possible crashes of VM after migration.

Unfortunately, the current practice of QEMU virtual device implementation is that – this *VMStateDescription* structure is manually defined and filled in by the developer. For example, in the e1000 virtual device implementation in QEMU 1.7.1, there are more than 80 lines of code for defining what device state to transfer during the live migration. Figure figure 2.3 shows code excerpts for the e1000 implementation of this piece of code. Hypothetically, if the *rxbuf_size* state is not transferred during the live migration, the virtual device will not respond to any network packets or I/O requests before a reset request is sent to the device.

2.3 Harnishing Virtual Device Implementation with Symbolic Execution

We propose to design a software tool to automatically extract the key device states that must be copied at the beginning of a live migration. During a VM live migration, all the device states on the destination host will be set to initial value except the ones that are
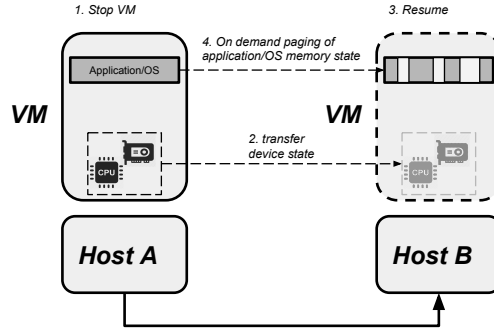
Figure 2.2: Architecture of VDSChecker

included in the *VMStateDescription* structure. Ideally, this *VMStateDescription* structure should contain all the key variables of a given virtual device implementation.

Our approach is to use program analysis techniques to study the behavior of these programs. The key observation is that, although mimicking complex hardware, virtual devices are actually software implementations. Therefore, the idea of automatically deducing the key device states is to analyze the virtual device implementation and detect which variables can not be reset to initial value during the execution without changing the program's behavior.

Although conceptually promising, we need to solve a couple of challenges in order to analyze virtual device implementations and extract key device states. The major challenges are the followings:

1. *Not Standalone Programs* – virtual device implementation is not a standalone program, but existing program analysis techniques [29] prefer to have complete pro-

grams as inputs.

2. *Many Non-deterministic Inputs* – A virtual device program executes as a part of the virtual machine platform. The input to the program can vary, and any program analysis effort that tries to derive the key variables of the program needs to consider all possible inputs.

3. *Large Test Space for Migration* – A live migration can occur at many different execution points of a virtual machine. In terms of software implementation, a live migration could occur in between any two event handlers. A key state variable might only be needed under a special event sequence, and thus finding all key state variables needs to consider a large test space.

For each of the above challenges, we provide the following solutions (and some of them are still under development).

2.3.1 Virtual Device Extraction

The virtual devices by themselves are not executable, in order to extract a virtual device from QEMU implementation for the analysis purpose, a wrapper is needed to initialize the virtual device and steer its execution to explore the paths. With this wrapper we need to implement two main features:

Virtual Device Initialization

In order for the extracted virtual device to function properly, the device state need to be initialized, just like the device initialization process when starting a virtual machine in

```
static const VMStateDescription vmstate_e1000 = {
    .name = "e1000",
    .version_id = 2,
    .minimum_version_id = 1,
    .pre_save = e1000_pre_save,
    .post_load = e1000_post_load,
    .fields = (VMStateField[]) {
        ...
        VMSTATE_UINT32(rxbuf_size, E1000State),
        ...
        VMSTATE_BUFFER(tx.data, E1000State),
        ...
        VMSTATE_UINT32(mac_reg[CTRL], E1000State),
        ...
    },
}
```

Figure 2.3: VMStateDescription structure defines the key device states need to be transferred during live migration

QEMU.

Because of the hierarchical model of QEMU hardware implementation, there are lots of references to the parent device objects in the device implementations, thus the initialization of one particular device requires its parent initialized first. So we need to recursively include the initialization of the parent devices before the device of interest can be properly initialized.

Steering the Execution of the Virtual Device

The virtual devices communicate with QEMU through its interface functions. When the device driver issues a request or there are I/O events related to the device, QEMU
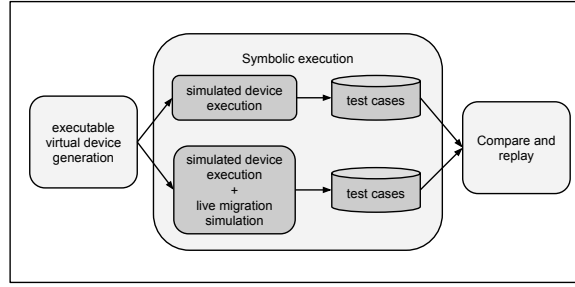
Figure 2.4: Architecture of VDSChecker

will invoke the corresponding interface functions to handle the task. We can steer the execution of the virtual device by invoking the interface functions in the wrapper.

2.3.2 Virtual Device Behavior Exploration

VDSChecker takes advantage of a software testing technique called symbolic execution [52] to solve the non-deterministic input problem, in particular, we use KLEE symbolic execution engine, which is capable of automatically generating test cases to enumerate the execution paths of the virtual devices [29]. With KLEE, we are able to achieve very high code coverage when testing a virtual device implementation and generate inputs that will lead to each of the paths. Also, by generating the path constraints that the inputs and variables have to satisfy for the virtual device to take a particular path, we are able to keep track of which device state under what condition may lead to errors of live migration.

2.3.3 Live Migration Simulation

In our approach, live migration is simulated by mimicking the the operations performed on the virtual devices during the migration, it consists of two parts:

- *Device State Save and Restore* – Save all the device states defined in *VMStateDescription* structure and load them to a newly initialized virtual device.

- *Massaging Functions* – QEMU introduced massaging functions, i.e. pre-save, pre-load and post-load functions, to solve the problems that there are situations when the key device state need to be copied from outside the virtual device data structure, our simulated migration also need to execute these functions to keep the device state correct.

We can address the problem of large test space for migration by enumerating all the combinations between two of the interface functions of a virtual device non-deterministically with simulated migration involved in between of the two interface function invocations.

Once the test cases are generated, we can compare the two sets of test cases, one with live migration involved during the execution of the virtual device and one without, and determine the completeness of the saved device states. The comparison consists of two steps:

1. Compare the two sets and find out all the test cases that exists in only one of the sets.

2. For each test case we get in the first step that belonged to the group of symbolic execution with live migration, match the most similar test case in the other group

according to their path constraints. Here similar means they share the longest path during the execution. In this way we are able to locate the exact statement that lead to the difference and thus find out the virtual device state involved in the statement.

2.4   Preliminary Prototype and Known Challenges

2.4.1   Preliminary Result

With the solutions for the major challenges introduced in the previous section, we implemented the prototype of VDSChecker and the live migration simulation component. We tested it with an artificial test program in which we didn't save one of the key device states that need to be saved during the migration. The result showed that we are able to detect the differences in the behavior of the program and locate the device state that lead to the differences.

There are still ongoing work related to extracting virtual devices implementations that are complicated from QEMU, and we expect to finish the tool and test it with different classes of devices on QEMU.

2.4.2   Known Challenges and Future Work

In our approach, we simplified the exploration of the execution of virtual devices by extracting the their implementations from QEMU and treating them as standalone programs. However, this kind of approximation may change the behavior of the virtual device to some extent and thus lead to inaccuracy when observing the behavior of the virtual devices because of the interactions across different components in the VM or the

function calls that are outside the scope of the virtual devices implementation. To solve this, one of our ongoing effort is trying to run the whole virtual machine with symbolic execution technique while only keep tracking of only the states of the virtual device we are interested in.

Another challenge in our work, which is brought in by the symbolic execution technique, is that starting from one interface function of a virtual device implementation, there may exist a huge number of execution paths, and many of the times, the number of paths increase exponentially. This makes it hard to enumerate all the possibilities and figure out the key device states by naive exploration. We can alleviate the problem by adding a loop bound and force exiting the loops after several number of iterations, however, with this approach we will definitely miss some of the cases that requires us to dig deep into the loops.

2.5 Conclusion

In this work, we proposed a study on live VM migration and the need of derive key device states from virtual device implementations. We developed an environment that mimics live migration of virtual machines in a program controlled way and proposed a solution that extracts the key states of a device by applying program analysis techniques to virtual device implementations. This is still an ongoing work, and we expect to finish the tool and test it on QEMU implementations.

Chapter 3

Detecting Virtualization Specific Vulnerabilities in Cloud Computing Environment

3.1 Introduction

Cloud computing is a fast growing computing technology and has been adopted as one of the key elements in modern IT infrastructure. With cloud computing, users have the ability to request computing resources on demand and on the fly without physically possessing the hardware [62]. Being the core technology in cloud computing, virtualization enables the dynamic allocation and modification of multiple VMs with one physical host machine underneath or the migration of one VM between different hosts.

The implementation of the virtualization is called hypervisor or virtual machine mon-

itor(VMM). As a software layer that lies in between the VMs and the physical hardware, VMM manages the resource allocation and deceives the guest operating system by providing virtual hardware devices. A VMM usually consists of hundreds of thousands of lines of code [33]. In recent years, more security vulnerabilities in cloud platforms have been discovered and documented. These vulnerabilities in the virtualization layer often lead to performance degradation, service interruption, information leaks and even control flow hijacking at the VMM level.

While many such vulnerabilities can be detected by existing software analysis techniques [59] , some flaws as witnessed in the past years are related to particular characteristics of the virtualization layer such as hardware logic and VM state migration requirements. Existing code verification techniques cannot easily be applied to capture these flaws specific to virtualization. Because of this, some of them may have been living in the code for years before they can be found and fixed. In this paper, we argue that the detection of these virtualization specific flaws often requires specific domain knowledge integrated in the detection process.

The goal of our work is to analyze the characteristics of security vulnerabilities and develop a systematic approach to detect them accurately. We studied virtualization security vulnerabilities in different VMMs that are documented by Xen Security Advisory(XSA) , Common Vulnerabilities and Exposures(CVE) , and National Vulnerability Database(NVD). Then we distinguish the unique characteristics of vulnerabilities in VMMs from traditional vulnerabilities. Based on the observation of these documented vulnerabilities, we proposed the idea of extending existing dynamic software analysis techniques, i.e. symbolic execution [52] to detect virtualization security vulnerabilities.

We also implemented the prototype based on QEMU, a popular open source VMM, to apply the new detection methods.

Correspondingly, our contributions in this paper are:

- We identified several types of virtualization specific flaws based on the recently discovered vulnerabilities in virtualization platforms.

- We proposed approaches to detect three types of virtualization specific vulnerabilities.

- We designed a framework to implement these methods to detect virtualization specific flaws.

3.2 Virtualization Vulnerabilities

In a virtualized environment, each of the VMs is isolated from the rest of the system by the VMM. A successful exploit can break this isolation and thus lead to various issues regarding the confidentiality, integrity, or availability of the VMs. The number of virtualization security vulnerabilities disclosed is increasing year by year and more researchers are focusing on this field. In Pwn2Own 2016, an additional bounty of $75K will be rewarded to contestants who are able to escape a Microsoft Windows guest operating system(OS) running in a VM created by VMware Workstation [11].

Out of the 4 dominant VMMs in the market (Microsoft Hyper-V [10], VMware [12], Xen [18] and Kernel-based Virtual Machine(KVM) [53]) [39] , our study focused primarily on Xen and KVM because Microsoft Hyper-V and VMW are closed-source com-

mercial software, which makes it hard to interpret the internal logic of the VMMs and analyze their vulnerabilities.

The vulnerabilities we studied are collected from NVD's CVE database and Xen XSA database. Our initial search criteria covers all the vulnerabilities related to KVM and Xen from the very beginning through December 2015, which gives us 96 matches for KVM and 215 matches for Xen. In addition to that, Xen XSA gives us 6 vulnerabilities that are not documented in the CVE database. Including three CVEs that are shared by both VMMs, there are 314 vulnerability reports in total.

In order to better understand the characteristics of the vulnerabilities and build the detection framework, we categorized the vulnerabilities into three groups: *Virtual Hardware Logic Errors*, *Device State Management Errors* and *Resource Availability Errors*. The definition of these three categories will be discussed in detail in Section III.

Table 3.1: Vulnerabilities Found In 2015

	KVM	Xen
Virtual Hardware Logic Errors	5	20
Device State Management Errors	1	11
Resource Availability Errors	4	15

We analyzed all of the vulnerabilities disclosed in 2015 and the statistical results are shown in Table table 3.1. During the study, we noticed that while some of the vulnerabilities exist in traditional computing environments and can be located by existing techniques, many others have specific properties related to virtualized systems, such as software emulated hardware logic and an adversary's ability to control the execution flow of some virtual hardware that cannot easily be addressed.

3.3   Detecting Virtualization Specific Vulnerabilities

Our analysis in Section III showed that other than traditional software vulnerabilities, there are virtualization specific flaws that can not be addressed by conventional software verification techniques. Many of the times these flaws are caused by logical errors of device emulations. These logical errors are not the conventional software vulnerabilities that usually lead to hijacking of execution flow. Also, cloud users usually by default have full control over the guest OS and can interact with the underlying virtual hardware directly, which is another special characteristic that makes VM implementations more vulnerable than regular systems. By considering the unique aspects of virtualization, we categorize three types of VM implementation flaws based on VM specific properties.

3.3.1   Virtual Hardware Logical Errors

Type of Problem

Virtual hardware implementation is designed to closely mimic physical hardware devices. When virtual hardware implementation behaves exactly the same as its physical counterpart, the OS on top of the VMM executes as if it runs on a physical machine. However, virtual hardware implementation tends to behave differently than physical hardware. Sometimes the differences are introduced intentionally as workarounds for technical difficulties of the implementations and sometimes they are just flaws in the design or development.

While the differences do no harm in most cases, some create vulnerabilities that can be exploited by adversaries to attack the VMM. For example, CVE-2014-9718 indicates

that due to the inconsistency in interpreting a function's return value, a guest OS user is able to cause a host OS Denial of Service(DoS) via crafted code.

Flaws in the implementation of the virtual hardware and differences in behavior are not necessarily software bugs that can be verified by conventional software verification techniques. For example, in CVE-2015-8567, the VMWARE VMXNET3 paravirtual NIC emulator failed to check if the device is active before activating it. This results in the possibility of launching a DoS attack by calling the device activation repeatedly and thus exhaust the host's resources.

Proposed Solution

The solution to detecting these types of errors is to find the differences in behavior between physical hardware and its virtual implementation. For a specific virtual hardware implementation, the detection of thees differences requires a reference that defines the correct behavior of the corresponding physical hardware. This reference can be either the formal specification/datasheet of the physical hardware or the physical hardware itself. In some cases, the reference can be a hardware model provided by the vendor or a third party.

Even with a reference, the detection of these differences in behavior is non-trivial. The challenge is similar to those in equivalence testing between different versions of software, except that one side of the comparison may not be software.

To overcome this challenge, we propose to approximate equivalence testing by enumerating the execution of virtual hardware implementation, followed by comparing the execution outcome between virtual and physical hardware. Even with this approach,

there are difficulties such as how to control the environment so that the virtual and physical hardware are being compared under the same situation.

Our initial effort relies on the ability of observing the behavior of the virtual hardware and tracing the execution path. In virtual hardware implementation, the enumeration of software execution can be addressed by the dynamic symbolic execution technique. We have extended QEMU's unit test framework to achieve the capability of exercising individual virtual devices with a relatively small footprint of a running virtual machine.

3.3.2   Device State Management Errors

Type of Problem

The second type of virtualization specific problem is related to VMM device state management. The value of the device registers captures the status of the virtual hardware, and thus these device states are important for managing the proper execution of the virtual hardware. Also, the VMM needs to manage and monitor the device state for each virtual hardware device for specific VM functions such as snapshot and VM live migration.

Incorrect handling of the device states, such as failing to save or restore some register values during VMM pause and resume actions, usually lead to misbehaviors in the OS level and possibly crashes of VMs. For example, according to CVE-2012-5634, while using VT-d hardware, an error occurred when registering a interrupt handling register of a device that is behind a legacy PCI bridge could result in denial of service attack that affecting VMM.

Device state management is vital during live migration of virtual machines in the cloud environment. Missing device states that governs the virtual hardware's behavior

after live migration will potentially result in unexpected behaviors of the virtual hardware. This type of problem can be illustrated by considering the registers that represents the transmission queue (*rxbuf_size*) status of a network device. If this device state is not transferred during the live migration, the device will not respond to any network packets or I/O requests before a reset request is sent to the device.

Proposed Solution

In order to detect device state management problems, such as the failure of initializing or restoring device register values, we need to ensure that the VMM implementation captures all of the variables that define the device states.

Although hardware behaviors are defined by design specifications, manual efforts to locate these variables from virtual hardware implementation are tedious and error-prone.

We propose to apply software analysis techniques to automatically detect all variables in a virtual hardware implementation at each critical execution points during the running time of virtual hardware. This effort can help detect VMM device state management errors. This approach still faces the challenge of how to get a complete set of variables that capture the behavior of virtual hardware implementation. Although we cannot ensure a full capture of all device states, these techniques have been shown to be useful in our previous work of detecting some VMM device management bugs [80].

3.3.3   Errors Related to Resource Availability

Type of Problems

The third type of VMM specific error is related to resource availability. Physical hardware (such as a sensor) by nature typically executes in an infinite loop once it is initialized. The hardware silicon continuously polls the environment and signal the software (e.g. through interrupts). If the corresponding virtual hardware implementation faithfully implements this hardware specification, the virtual implementation would run in an infinite loop, feeding data to software through registers or interrupts. However, the VMM is not based on silicon and a software device implementation has to consider the resource usage and avoid busy looping when necessary.

Unfortunately, the guest OS running on top of the VMM makes no effort to differentiate between virtual and physical hardware. Mistakes in VMM device implementation combined with guest OS behaviors can lead to high resource usage. Although resource usage problems occur in almost all types of software, VMMs are especially prone to this type of errors due to the lack of physical isolation.

For example, CVE-2015-5279 showed that be exploiting a buffer overflow vulnerability in the ne2000 NIC, an attacker is able to cause a denial of service (by creating an infinite loop) or possibly execute arbitrary code via vectors related to receiving packets.

Proposed solution

To detect errors related to resource usage in VMM implementation, we would need to include resource usage measures in program analysis techniques. We are in the process of applying a previous software analysis solution [27] with a focus on resource usage to

Figure 3.1: Architecture of the detection framework
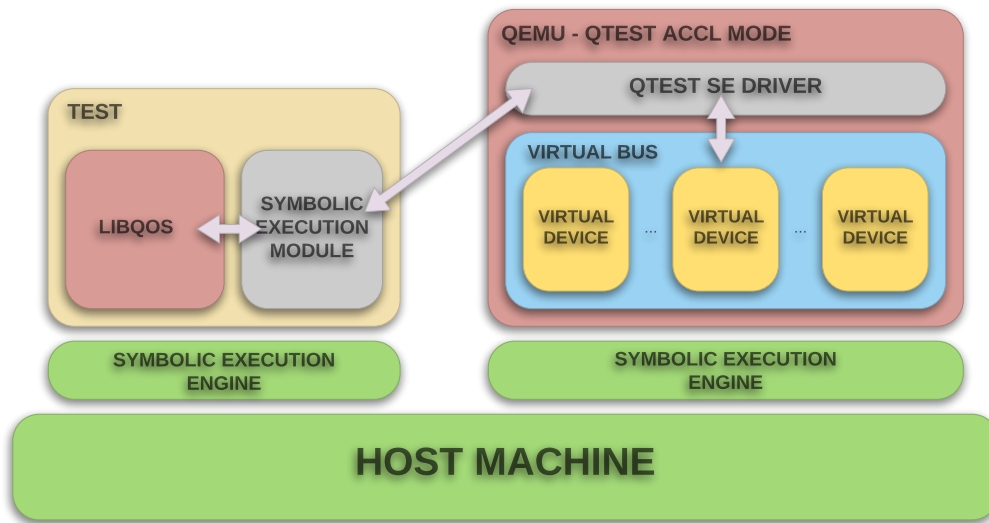
VMM implementation. Our goals are to be able to 1). accurately detect situations that might lead to resource availability vulnerabilities such as infinite loops in virtual hardware or repetition of memory allocation during the normal execution of virtual hardware and 2). craft arbitrary input values for virtual hardware that can lead us to the location of execution of our interest.

3.4 Combining QTest framework and Symbolic Exection

With all of the key challenges to detecting virtualization vulnerabilities as discussed previously in mind, we designed a framework that is able to detect virtualization specific vulnerabilities by extending symbolic execution techniques and QEMU's unit test tool, namely QTest.

3.4.1 Architecture

As shown in Fig. figure 3.1, the framework consists of two major components: a modified version of QEMU that launched in QTest Accl Mode and *LibQOS* that is responsible for handling the test cases of the virtual hardware. These two parts operate in a client-server based fashion and communicate with each other via sockets. QEMU, when running in QTest Accl Mode, is able to initialize only the device to be test and several other crucial components in order for the device to be able to execute properly. LibQOS provides API for steering the execution of QEMU under test by adding basic operations such as clock cycle and IRQ.

The execution of the test is guided by the symbolic execution engine, which runs inside the host OS. Symbolic variables can be defined via *symbolic execution module* inside a unit test, passed to QEMU by *LibQOS*, then interpreted and assigned to corresponding device states by *QTest SE Driver*. Symbolic execution engine will log the execution trace and generate test output representing the execution paths of the virtual hardware during the test.

3.4.2 Modified QTest Framework

We extended QTest's capability of testing the functions of individual virtual hardware to the detection of virtualization specific vulnerabilities in a particular virtual hardware implementation by adding symbolic execution module. Because QTest is maintained in QEMU's codebase, it would be convenient for virtual device developers to conduct not only functional testing but also security testing that helps eliminate implementation flaws that can lead to virtualization vulnerabilities.

With the framework, it is possible to:

1. Conduct testing of virtual hardware for vulnerabilities without having to modify the source code to mark a particular device state of interest, which is required by traditional symbolic software testing techniques. The symbolic value of the device state is passed into the running instance of the VM by LibQOS through IPC.

2. Test only the hardware of interest without having the whole VM up and running. With QTest Accl Mode enabled, QEMU will only initialize the virtual hardware to be tested together with all the components that are required for the virtual hardware to be running properly.

One other advantage of using QTest framework when testing individual QEMU device is that as a result of the hierarchical model of QEMU hardware implementation, there are lots of references to the parent device objects regarding the device implementations, thus the initialization and proper execution of one particular device requires a recursive initialization of all the ancestors and depending components. This will become overwhelmingly difficult to deal with manually when the device implementation gets

complicated.

As a prototype implementation, we integrated our previous work of extracting key device states for the live migration of virtual machines in cloud computing[80] and are able to check for device state vulnerabilities in virtual hardware implementations.

3.5 Evaluation

We evaluated the effectiveness of our approach with regard to detecting logic errors and device state management errors by reproducing with our system several of the CVEs that are discovered and reported manually. We measured the code coverage and the efficiency of the execution of the virtual devices during the experiments and compared them to the testing results of the real hardware.

Since the symbolic execution is a time-consuming process, sometimes the execution can get stuck in one code chunk for hours, we defined a heuristic that if there is no new coverage generated within a given time, the engine will force terminate the current input iteration, mark it as suspicious and switch to the next one. The reason of marking the input as suspicious is that the input might have caused an infinite loop and thus needs further investigation.

3.5.1 Analyzing the behavior of the virtual devices

Even though most of the emulated devices in a virtual machine are based on the specifications of real hardware, they tend to behave differently than their physical counterpart. While some of the differences are there as workarounds for technical difficulties of the

implementations, sometimes they are just mistakes made by the developer in the process of the design or development. When such difference creates vulnerabilities that can be exploited by adversaries to attack the VMM, we call it a *logical error* in the virtual hardware implementation.

In order to be able to compare the execution difference between virtual devices and real hardware as well as the behavior difference between different runs of tests, we implemented *SE-Diff*. *SE-Diff* is a set of script wrappers that can be used to automatically conduct symbolic execution testing on a given piece of annotated source code and generate the results of the behavior differences and other execution information such as code coverage and execution time. With the scripts, we are able to achieve:

1. Given two annotated c files as input, output the differences between the generated symbolic execution test cases. This is useful when comparing between two different runs of the same test with different seeds given or comparing between two sets of tests in which slight differences are introduced.

2. When a specific input and device state is given, analyze the differences between the virtual device and the real hardware by comparing the output of the devices and the state change.

3.5.2   Detecting Virtual Hardware Logical Errors

In Section III we discussed three types if virtual machine implementation flaws based on VM specific properties, and most of the vulnerabilities fell into these categories are triggered by improper I/O or memory manipulation. Since we are able to control clock,

memory, I/O, IRQ with QTest API, theoretically we are able to get full control of the virtual device under test and trigger the vulnerabilities with our test platform. To prove the effectiveness of our proposed device vulnerability detection technique, we conducted experiments that automatically reproduce CVE-2015-7504 and CVE-2015-5278.

The vulnerability of CVE-2015-7504 lies in QEMU's AMD PC-Net II Ethernet Controller emulation implementation. There is a location vulnerable to heap-based buffer overflow in the *pcnet_transmit* function in *hw/net/pcnet.c*. Adversaries can exploit this vulnerability to cause a denial of service attack to crash the QEMU instance or take control of the execution of the QEMU host and achieve arbitrary code execution with privileges of the QEMU process.

CVE-2015-5279 is a flaw in QEMU's NE2000 NIC, when a packet received from the network satisfies certain condition, the *ne2000_receive* function in *hw/net/ne2000.c* will enter an infinite loop and thus resulting in a denial of service.

In both experiments, the test cases were implemented in less than 100 extra lines of code based on the QTest template. The tests consist of device initialization and I/O manipulation. The device initialization stage is used to locate and parse the memory address of the virtual device data structure in the running QEMU memory, which is part of the QTest template. During the I/O manipulation stage, based on the knowledge we get from the initialization stage, the value of device registers of the virtual hardware are marked as symbolic and the test case starts to generate IRQs, with which the symbolic execution engine starts to explore different execution paths of the device.

For CVE-2015-7503, once the execution of QEMU is crashed because of the heap overflow, a sequence of input that will trigger the vulnerability will be generated by the

framework automatically. We verified the result by manually fed the generated test input into the virtual devices and observed the crash at the vulnerable location. For CVE-2015-5279, an assertion failure will be triggered once the execution enters the infinite loop and the corresponding test input will be generated. The test input is also verified by manual effort.

With the experiment results, we believe that the proposed virtual device vulnerability detection technique is able to detect different types virtual machine vulnerabilities introduced by device implementation flaws. However, in order to test a virtual hardware with the framework, it requires the developer to have a certain level of understanding of the specification of the virtual device to be tested as well as the emulation implementation. Diminishing the prerequisite knowledge of the real hardware logic while testing the virtual device for vulnerabilities will be one of the fureture works.

3.6   discussion

As more researches are focusing on security in cloud computing, there are a considerable number of works exists that emphasizing the importance of security of cloud computing,

In the work of [68],Perez-Botero et al. did a thorough survey in great details of possible attacks in hardware virtualization and proposed some countermeasures to mitigate the influence when there is an attack. Pk, Gbor etal. studied CVEs related to KVM and Xen vulnerabilities and mapped them into different categories based on trigger source, attack vector and target[67]. There has also been work on classification of threats based on the different service delivery models of cloud computing like [77]. Different from all

these works, our work also focuses on the detection of virtualization vulnerabilities and we propose a framework to find flaws in the implementation of virtual hardware that may lead to vulnerabilities.

Although there may appear to have some overlapping between the three categories we defined, for example, CVE-2012-5634 shows that a flaw in the logic of virtual hardware can lead to virtual device state vulnerability, that's OK because it won't impact the detection of the vulnerability. In fact, flaws of this kind are a lot easier to catch because the vulnerable characteristics are exposed from different perspective.

This work is an on going research, as of now, we are able to test virtual hardware implementation and check for device state vulnerabilities, future work on the project includes the conformity inspection of the virtual hardware and resource availability vulnerability detection based on the framework.

3.7   Conclusion and Future Work

In this work, we conducted analyses of the known vulnerabilities disclosed in recent years in KVM and Xen, studied their characteristics as well as the differences between vulnerabilities in virtualization and traditional software vulnerabilities. Our study showed that some of the unique features of cloud computing and virtualization makes many of the vulnerabilities hard to address using existing software verification and validation techniques.

Based on our findings, we presented three categories of vulnerabilities that are unique to virtualization, identified the challenges and proposed ideas to detect these vulnerabili-

ties, designed a framework to implement these ideas to catch different kinds of flaws that are buried deep within the implementation of the virtualization by combining QEMU function test framework and KLEE LLVM Execution Engine. We are thus able to test virtual hardware implementation and check for device state vulnerabilities. Current on-going work includes conformity inspection of the virtual hardware and resource availability vulnerability detection based on the framework.

Chapter 4

Augmenting Fuzzing with Concolic Execution

In this chapter, we present SEZZER, a binary only software testing framework targets linux binaries and analyze for memory bugs. SEZZER take advantages from both dynamic concolic execution and grey box fuzzing for better code coverage and target specific vulnerability discovery. Fuzzing is used to explore the general conditions of the target program, and with the basic coverage information from fuzzing, concolic execution is used to help getting over the complex constraint checks on the execution path of the application as well as check for specific types of vulnerabilities during the execution which can otherwise be hard to find by fuzzing. The alternation of the two techniques eliminates several major disadvantages from both fuzzing and concolic execution and give SEZZER the ability to find more execution paths and get higher code coverage.

4.1   Introduction

Even though software developers are getting more aware of software security flaws and vulnerabilities and measures are applied to prevent vulnerability when implementing the software. Applications with multiple vulnerabilities are still very common.

While many of the vulnerabilities are discovered by manual effort, many researchers are working on implementing software testing systems that can analyze applications and discover vulnerabilities automatically. Out of all those efforts, there are three techniques that are most prevalent: fuzzing, static analysis and symbolic execution. While each of the techniques has its own merit, they have their own disadvantages as well.

- Fuzzing can be black box or grey box, the basic concept is of the technique very simple and it can be easily applied to different applications without preconception s about system behavior. Since it usually employs random approaches, fuzzing is able find bugs and vulnerabilities that would have often been missed by human. Because the random approach, fuzzing tend to find only simple bugs, the more complex constraint checks involved in the target application, the less likely it will be able to successfully reach the piece of code hid behind the constraints.

- Static code analysis can be helpful with identifying software security issues during the development stage and is usually very fast. However, it can only detect bugs that has strong pattern and has a lot of false positive. And the requirement of source code of the target application prevents the technique to be applied to close sourced, binary only applications.

- Symbolic execution can be both sound and complete in the case of exploring the

execution paths of the target applications in theory. But the constraint solving is relatively slow and by nature it suffers from "path explosion" problem.

In this chapter we introduce SEZZER, a system for automatically testing system software and find security vulnerabilities. One of the assumptions of SEZZER is that the target application can be Commercial off-the-shelf(COTS) applications, which means there is no source code available. With that in mind, we build SEZZER upon the state of art grey box fuzzing technique, with which the source code of the target application to be analyzed is not required. Combined with fuzzing is a selective symbolic execution engine that can also operates linux binaries directly without any instrumentation. With the SEZZER framework, fuzzing is used to explore the general input and populate the easy-to-find testcases, when the fuzzing process gets stuck and can't generate testcases efficiently, the concolic execution kicks in, leveraging the coverage information and the testcases generated by fuzzing, the concolic execution helps with generating testcases that lead to uncovered code blocks. At the same time, the fuzzing process takes advantage of the new testcases generated by symbolic execution and continue explore the newly discovered code blocks of the target application.

The framework operates on linux binaries and does not require any source code information. The experiment is done on different benchmark binaries as well as real-world applications. We evaluate compare the performance in three categories: node coverage, path coverage and number of bugs found. The result showed that with bench mark binaries, SEZZER is able to achieve an average of more than 50% more node coverage comparing to several other software testing tools. With Lava-M test suite [40], SEZZER is able to find all the inserted vulnerabilities in 3 out of 4 binaries in a short amount of

time, and exploited 2076 out of 2163 vulnerabilities of the 4th binary in a 24 hour run. We also found several bugs in master branch of GNU-binutils that resulted in 5 patches and 3 CVEs.

4.1.1 Background

Automated software testing and vulnerability discovery is getting more popular. Traditional black/grey box testing (fuzzing) did a good job of covering general conditions of programs and are the most prevalent techniques for system testing. However, they are not good at finding/triggering bugs that depends on specific inputs.

Symbolic execution as another dynamic testing approach, is good at understanding the program's control flow and solve the requirements of the user inputs for the program to take particular execution paths. But symbolic execution has it's own drawbacks. The exploration of the target program is slow comparing to fuzzing and it suffers from path explosion and resource constraint problems.

Greybox Fuzzing

The term grey-box fuzzing is mentioned for the first time in 2007 [38]. However, the idea of grey box fuzzing originates way before that. It can briefly be explained as a fuzzing technique located between black- and white-box fuzzing, where black-box means testing the target system with zero knowledge and only look at the input/output, and white-box testing assumes source code is available. Grey-box fuzzing does not require source code to be available, but is can collect the feedback regarding the internal state of the target application being tested, this is usually achieved by instrumenting the target application

with static analysis techniques before-hand or with dynamic instrumentation on the fly during the execution of the target application[71].

The main strength of grey-box fuzzing is that it does not require the source code to perform the fuzzing. This is beneficial for testing applications that do not have source code readily available[38]. Since grey-box fuzzing is usually based on the lightweight black-box fuzzing technique, but may still glean information about the target application, code coverage can still be leveraged without sacrificing time on program analysis [23].

Two of the most significant changes were the use of dictionaries while mutating the testcases and the comparison unrolling. The former one allows the fuzzing engine to take a set of predefined byte sequences as dictionary when testing a specific target application, be it obtained from general knowledge or through static analyze techniques, that allows the test case mutation to try those combinations and getting over some specific magic number checks to achieve better code coverage. The later transforms complex constant comparisons to into multiple smaller ones which can guide the fuzzing engine towards the correct value [55]

Out of those grey-box fuzzing techniques, American Fuzzy Lop (AFL) is a state-of-the-art coverage based technique that has been proved to be successful [79] for discovering significant security vulnerabilities in several different major software projects, including X.Org Server [8], PHP [3], OpenSSL [5, 6], pngcrush, bash [1], Firefox [2], BIND [4] and Qt [7].

During our implementation of SEZZER, we leveraged AFL as the fuzzing backend. Since the only addition to fuzzing backend is to have it cooperate with symbolic execution and share context, and no changes were made to the mutation strategy, it is possible to

switch the fuzzing backend with other fuzzing techniques.

Concolic Execution

While concolic execution, as one of the most prevalent automated software techniques for finding security vulnerabilities, is getting more and more attention from both academia and industry during the past decade.

The concept of concolic execution was first brought to sight with EXE [30] and with KLEE [29], and later the technique was applied to applications without source code with Mayhem [31] and S2E [34],

Concolic execution engines model marked memory region of a running application using concolic variables, accumulates constraints introduced by conditional jumps, and solves those constraints with constraint solvers to generate inputs that can lead to the application to execute following a specific path.

Symbolic execution, while is both sound and complete in theory [52]. In practice there are many real-world constraints that limiting the technique to be applied to real-world software testing, out of which the path explosion problem is one of the fundamental problem. Efforts have been made by researchers to alleviate the problem in the past few years, [54] merges state opportunistically so that the resulting path constraints do not stress the underlying constraint solver. Veritesting [16] employs static symbolic execution to amplify the effect of dynamic symbolic execution and merging redundant paths. MultiSeE [74] and the work of [50] reduces the number of paths by pruning out redundant or unnecessary executions.

All those attempts either postpones the path explosion problem to a later stage or

loses information and potential code coverage by discarding states.

4.1.2 Related work

SEZZER is not the first one trying to improve the efficiency and effectiveness of system software testing by combining different types of software analysis techniques and have them work together.

Fairfuzz [57], Angora [32] and Vuzzer [71] all based on AFL and trying to improve the mutation strategy by indtroduing taint analysis and control flow analysis.

Dowser [45] use static analysis to guide symbolic execution to check for buffer overflow at certain locations, but in order for it to work, a valid testcase must be present to lead the symbolic execution to the desired location.

Veritesting [16] combines static symbolic execution and dynamic symbolic execution to reduce the number of paths being executed, but the proposed path merging technique only delays the path explosion problem to an extend.

Driller [76] also leverages fuzzing and concolic execution, but the concolic execution engine still blindly generates test cases with the seed from fuzzing, hoping one of the newly generated testcase will help with fuzzing to discover new code blocks.

4.1.3 Contributions

By implementing SEZZER framework, we make the following contributions:

- Propose the approach to improve the coverage of software testing by alternating between fuzzing and selective concolic execution. We use selective concolic execution to generate testcases that help fuzzing getting over some complex constraint

43

checks that are otherwise hard to solve. Also mitigating the path explosion problem by using the testcases and coverage information generated by fuzzing and only fork at the places where the newly forked branch can lead to new code coverage.

- Designed and implemented the SEZZER framework to carry out the idea and evaluated the strength of the approach by comparing the testing results with several other state of the art software testing techniques and by discovering several real world vulnerabilities.

4.2 Overview

The key idea is from the observation that out of all the conditional jumps inside a program that related to user input, some conditional checks are easy to satisfy and others require specific values to be given. Listing 4.1 demonstrates those two types of constraints, While the check on line 4 can be satisfied with a wide rage of values, line 8 asks the input for a specific value in order for the code inside the bracket to be executed.

The work of SEZZER incorporates the ability to populate the search space that does not require specific values from fuzzing and the power of constraint solving from concolic execution to satisfy complex constraint checks. Also, with the coverage information from fuzzing, we are able to mitigate the path explosion problem by disable state forking at locations that are already covered fuzzing and only fork and generate testcases which can lead to new code coverage.

The high level design of SEZZER consists of several elements.

Fuzzing . Fuzzing is responsible for the initial corpus generation as well as exploring

Listing 4.1: Easy to satisfy constraints Vs. specific constraints

```
1  int main(int argc, char **argv){
2      unsigned int input = 0;
3      read(0 , &input , sizeof(input));
4      If (input > 0xff)
5      {
6          do_something();
7      }
8      If (input * input == 0x9156cb1)
9      {
10         do_something_else();
11     }
12     return 0;
13 }
```

the search spaces of the target application that does not have complex constraint checks. When there is no new code coverage in a given amount of time, the scheduler will select one of the testcases created by the fuzzing based on the basic block coverage information together with the seed selection heuristic and use it as the seed to launch concolic execution to generate testcases that can lead to uncovered code of the target application.

Basic block coverage analysis . Two major tasks are performed by the basic block coverage analyzer: The first one is to generate the full control flow graph of the target application with reverse engineer techniques before the fuzzing and concolic execution starts, this control flow graph(CFG). The second one is for each of the testcases generated by fuzzing, record the execution path of the target application and all the basic blocks/edges covered. The control flow graph together with basic
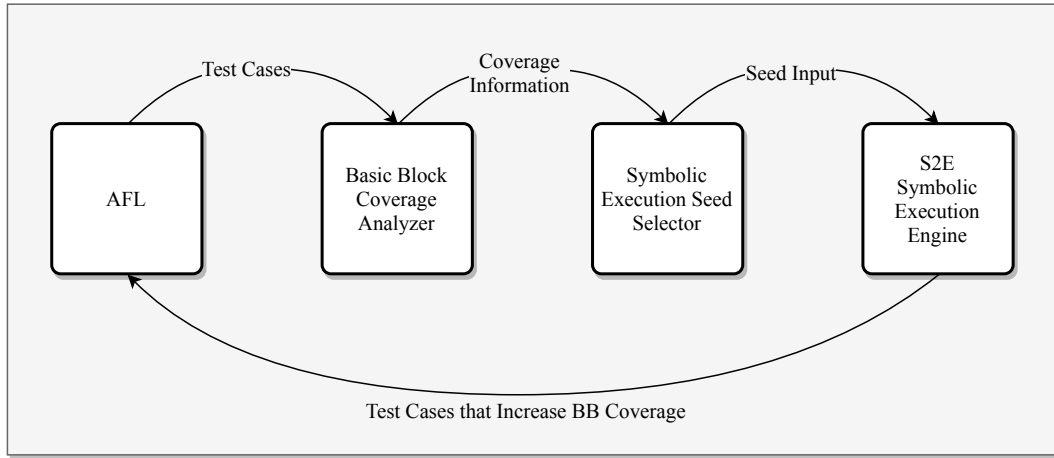
Figure 4.1: high level cizzer architecture

block coverage information of each independent testcase will be used later by the seed selector to determine which testcase from fuzzing has the best potential to discover new code coverage.

Seed selection . The seed selector employs the CFG and coverage information and ranks all the testcases generated by AFL with a weighted score. For each invocation, a testcase with highest score will be selected as the seed of concolic execution that steers the execution of the target application and helps exploring new code coverage.

Concolic execution . Concolic execution engine execute the target application with the seed selected from fuzzing and tries to generate testcases that lead to new code coverage. With the coverage information obtained from the coverage analyzer, state fork is disable at locations that have already been covered by fuzzing, only ac-

cumulates path constraints along the execution of the target application. Besides generate testcases that can lead to new code coverage, the concolic execution engine is also used to check for specific properties during the execution and trigger security vulnerabilities that are otherwise hard to find by fuzzing.

Figure 4.1 demonstrates the high level concept of SEZZER, all the components are coordinated by a scheduler, which is used to launch the basic block coverage analyzer when new testcase is found by fuzzing and initiate concolic execution by selecting optimal seed and define which address the concolic execution engine should be forking at.

Demonstrate with a sample program.

4.3   Implementation

In this section we will present the implementation of SEZZER in detail by walking through each of the components in the order of invocation when SEZZER is launched. And demonstrate the scheduling process that enables the sharing of the context between fuzzing and concolic execution and coordinates different components to work together.

4.3.1   Fuzzing

During our implementation of SEZZER, we leveraged AFL as the fuzzing backend. The reason we are using AFL is because: 1. It is readily available at the time of the development of the system and works on binary only target applications as a grey box fuzzing technique. 2. It is proven to be one of the most successful fuzzing backend and has a lot of potential improvement.

The only changes that made to AFL is to let it prioritize the testcases generated by concolic execution, other than that, no other changes were made.

Strength and Weaknesses

To improve the effectiveness of test input generation, AFL applies the concept of genetic algorithm [20, 41] when mutating the test input and ranks the fitness of the mutated population with the *trace_bits*, an internal representation of the control flow transition (edge) coverage of the target application with a particular input. A newly mutated testcase is considered favored passed to the next generation if the *trace_bits* contains transitions that haven't been covered by other testcases before.

For handling recurrence and loops, the *trace_bits* of all the testcases are stored in a single data structure, *bitmap*, in which each byte represents one single edge that is calculated by hashing the virtual address of both the predecessor and successor basic block. Those said bytes are also called a *bitbuckets*, whereas each bit in the byte represents a particular number of times the transition has been executed by a single testcases. The number of times that represented by each bit are increased in a exponential fashion, if the first bit in a bitbucket is set by a test input, it means with the input, the target application executed the edge once, second bit means more than twice but less than 4 times, third bit 4 7 times and so on. In this way, not only inputs that discover new edge coverage can be considered favored, but also those resulted in different number of times an edge being executed.

All these features together with other optimizations allowed AFL to be able to mutate and execute a program hundreds or thousands of times a second as well as guide the mu-

tation towards finding inputs that can lead to new code coverage. However, AFL together with its *bitmap* abstraction the still suffers from the limitation of random mutating and falls short in certain scenarios.

Consider the code snippet in Listing 4.1. The code reads from user as input and performs different tasks according to comparison result against different values. Unlike the easy comparison in 4, which has a chance of $(2^{32} - 256)/2^{32}$ to be satisfied, the comparison in 8 can be satisfied with only one specific value and, because the dependence of the input bytes, is unlikely to be discovered.

4.3.2   Basic Block Coverage Analysis

In SEZZER, the concolic execution is driven by the coverage information from fuzzing. The benefit of the approach is that with the tracing knowledge of the target application with the testcases from fuzzing, we are able to avoid unnecessary state branching in concolic execution by disabling forking at locations that have been covered already and thus mitigate the path explosion problem.

The coverage information we are utilizing is the control flow graph of the target application and basic block coverage information of each testcase. Even though our chosen fuzzing backend, AFL, as a grey box fuzzer, has its own representation of the internal state of the target application, we decided not to use it and choose CFG coverage instead for 2 reasons.

- AFL's bitmap is using the hashing of the addresses of two consecutive basic blocks to captures branch (edge) coverage, along with coarse branch-taken hit counts. Because it is using binary value to mark whether an edge has been covered or not, it

Branch cnt	Colliding tuples%	Example targetsc
1,000	0.75%	giflib, lzo
2,000	1.5%	zlib, tar, xz
5,000	3.5%	libpng, libwebp
10,000	7%	libxml
20,000	14%	sqlite
50,000	30%	-

Table 4.1: The chance of hash collisions based on branch count.

only records which edge has been covered and impossible to indicate which branch is not and how to get there. On the other hand, with the CFG of the target application, and the basic block coverage information, it is easy to see which testcase has covered basic blocks that can lead to uncovered edges, and thus find new code coverage.

- One of the goal of AFL's bitmap is to achieve high performance while retaining the coverage information. So the default bitmap size is limited to 2^{16} in size, which will very likely suffer from the bitmap hash collision problem - The hash of two different edges falls into the same bitmap index after the normalization and thus one of them is considered "visited" if the byte in the bitmap is tainted by the other map before hand. This can lead to a problem that even though a testcase lead to new code coverage, it can still be discarded by AFL. Table 4.1 showed the odds of hash collision with different number of branch counts. It is certain that when the target
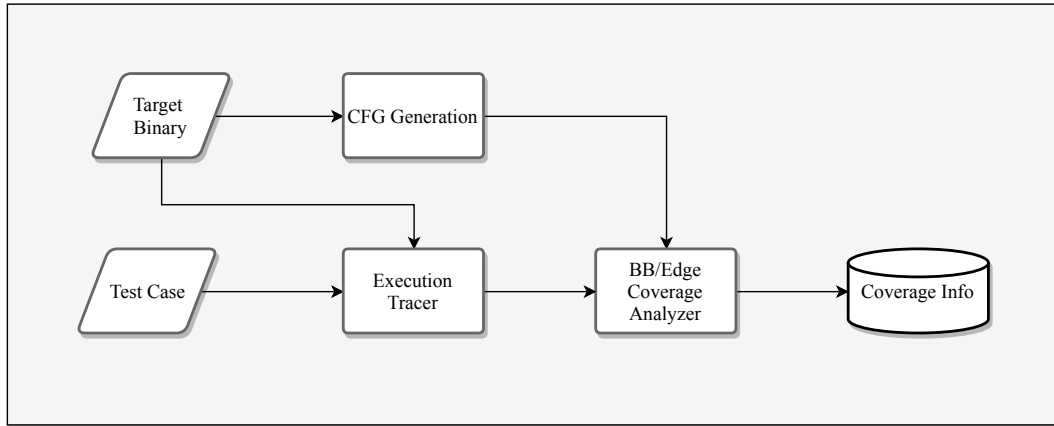
Figure 4.2: Coverage Analyzer

application gets complex, the AFL will get less effective because of hash collision. We will discuss our approach to work around the problem later in Section 4.3.4

As shown in Figure 4.2. The Coverage analyzer consists of two parts. CFG builder and basic block tracker. Given a target application, the CFG builder is first initiated and construct the control flow graph information using reverse engineering techniques. For each of the testcases generated by AFL, it will be executed again in basic block tracker to collect the basic block coverage information.

Control Flow Graph Recovery

A control flow graph is a representation, using graph notation, of all paths that might be traversed through a program during its execution. The control flow graph is due to Frances E. Allen[13], who notes that Reese T. Prosser used boolean connectivity matrices for flow analysis before[70].

The goal of the CFG recovery is to obtain a partial control flow graph of the target application. A recovered CFG is an underapproximation if all edges of the CFG represent feasible paths. A recovered CFG is an overapproximation if all feasible paths in the program are represented by edges in the CFG. Statically recovering a perfect (non-approximate) CFG on binary code is known to be a hard problem and the subject of active research [51, 17]. A recovered CFG might be an underapproximation or an over-approximation, or even both in practice.

There are various different tools and techniques for recovering CFG from binary only, we are applying reverse engineering technique to recover the control flow graph from the target application and leverage two off-the-shelf tools, IDA Pro [72] and Angr [75] to generate the CFG from the target application. With the native API support, it take both of the tools less than 100 LOC to extract CFG from a given program.

Both of the two tools can recover CFG on their own, the reason we are using two different kind of technique is that during the early stage of development, we noticed that, since the CFG is built by reverse engineering the target application, the final control flow graph is not complete. There may be some edges missing here and there, and many of the times, those pieces missed by one tool can be found in the recovered CFG from the other tool. A simple heuristic is implemented to merge the results from both tools and reduce the number of incorrect basic blocks/edges.

Testcase Basic Block Coverage Tracking

The goal of testcase basic block coverage tracking is to be able to trace the execution of the target application with the testcases produced by AFL, collect all the basic blocks and

edges covered for each of the testcases.

We are using a modified version of QEMU to handle the dynamic instrumentation and trace collection. QEMU works with the concept of translation blocks[22]. A translation block is a sequence of instructions that ends with a control flow change. It is different from a basic block, which has the additional constraint of having a single entry point globally. This has two consequences:

- Two basic blocks could be merged into one translation block: QEMU doesn't know anything about the CFG of the program, rather it starts translating code at whatever program counter is given to it. It stops when encountering a ctrl flow change.

- One basic block could be chopped into pieces: if there is an exception in the middle of a basic block, execution aborts, the exception handler is run, then exception resumes at the interrupted program counter. At that point, QEMU will try to fetch an existing translation block starting at that new PC, won't find any, and will re-translate a new translation block. This will end up with two partially overlapping translation blocks starting at two different addresses.

The basic block coverage analyzer is constantly compares the newly traced translation blocks and the existing basic blocks, and will perform basic block merging or splitting as needed.

Once we have the basic block coverage information and the recovered CFG from the target binary, with graph analyzing techniques, it is easy to find out which state transitions haven't been covered by existing testcases, as well as with which testcases the concolic execution engine has the potential to be able to reach the uncovered basic blocks by state forking at the corresponding parent nodes.
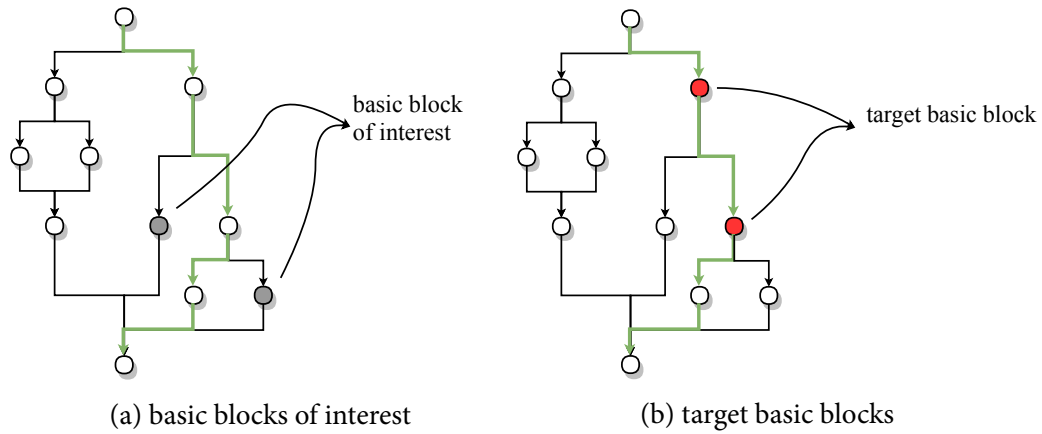
(a) basic blocks of interest   (b) target basic blocks

Figure 4.3: basic blocks of interest and target basic block during seed selection

4.3.3 Seed Selection

The goal of seed selection process is to find a optimal seed from the testcases generated by fuzzing that once executed by concolic execution engine, has the potential to lead to the most new basic block discovery. We define the notion basic block of interest and target basic block.

As shown in Figure 4.3a, assume the dotted graph is the CFG of one target application and the execution path of a testcase is marked green. The basic blocks of interest are the basic blocks that :

- Haven't been executed by any of the testcases from fuzzing.

- Direct successor of the basic blocks that are covered by at least one of the testcases.

This means that for any of the basic blocks of interest, and for any testcases that can reach the predecessor of the said basic block. If the execution of the target application can take

the alternate branch instead of the one the testcase lead to, the basic block of interest can be executed.

With the definition of basic blocks of interest, we define the target basic block during seed selection as the direct predecessor of any of the basic blocks of interest that are on the execution path of at least one of the testcases from fuzzing. The target basic blocks are demostrated in fig:target. These target basic blocks are the locations we need to enable state fork during the concolic execution.

In order for the concolic execution to be able to discover new basic block coverage more efficiently, we want to prioritize testcases that has the most target basic blocks, while neglect those target basic blocks that have been executed in concolic execution multiple times yet failed to fork to the branch that leads to the basic block of interest.

Our implementation of the seed selection heuristic is very straight forward.

1. Each of the target basic blocks is given a score S_{bb}, the initial weight of an untouched target basic block is set to 1. Every time a target basic block is selected with a testcase for concolic execution. the score is increased by 0.9. The less the score of a target basic block, the less times it is executed by concolic execution engine. The number 0.9 is chosen based on our experiments and yields a good result.

2. For a given testcase, we will first sort the target basic blocks with the score they have and only consider the first 100 target basic blocks with the lowest score. This is because even with the state fork disabled at locations other than target basic blocks during concolic execution, it still is relatively slow to query the feasibility of a particular state fork and solve the constraints to generate testcases. In cases that the execution get stuck at one point for a long time and can't proceed, it will influence
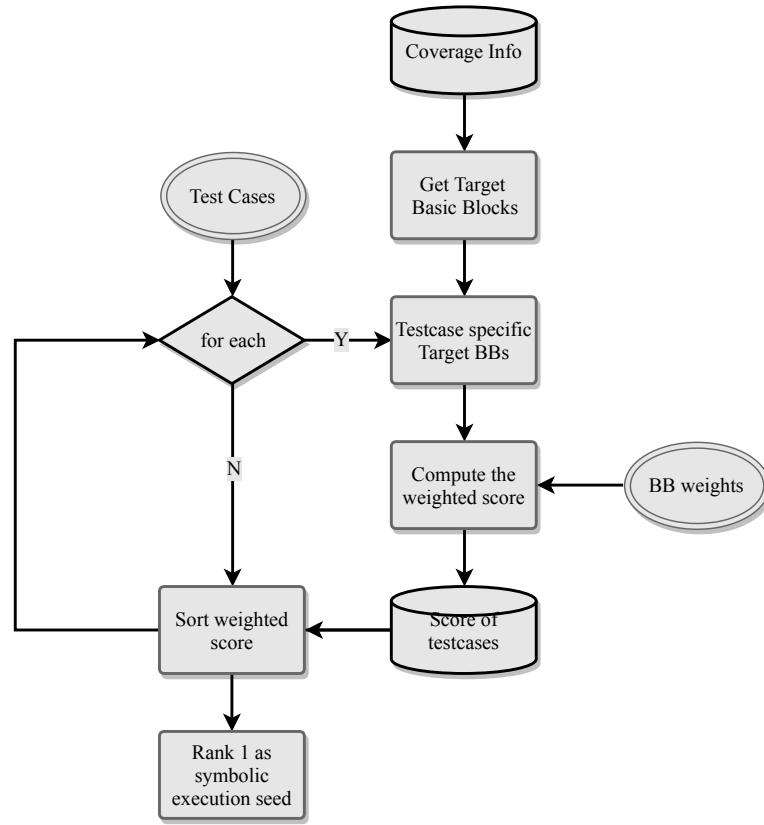
55

Figure 4.4: Concolic Execution Seed Selection

the effectiveness of the framework. We are limiting the number of state forks for each of the concolic execution invocation and leverage the power of multiprocessing.

3. Assume the scores of the all the target basic blocks for one test case are S_1, S_2, ..., S_n. The weighted score of the testcase S_T is calculated with

$$S_T = \sum_{i=1}^{n} \frac{1}{S_i}, (n <= 100)$$

.

weighted of each of the basic block, together calculate a weighted score of each of the testcases

After the weighted score of all the testcases are calculated, the one with the highest score is chosen as the seed of concolic execution. Figure 4.4 shows the work flow of the seed selection process. In the figure, the testcases are generated by fuzzing, coverage info comes from the basic block coverage analyzation and the score the basic blocks (shown as BB weights) are updated by the scheduling process as the target basic blocks are constantly being executed by the concolic execution engine.

4.3.4 Concolic Execution And Property Checking

The concolic execution engine is responsible for two tasks:

- Generate testcases that can steer the execution of the target application to new code coverage. When the scheduling process finds out that the progress of fuzzing gets stuck, the concolic execution kicks in. Leveraging the basic block coverage information from fuzzing, the concolic execution attempts to do state fork at the edges that the fuzzing is unable to get over and satisfies the conditional checks with constraint solver.

- Check for certain types of vulnerabilities along the execution. For certain types of vulnerabilities, even though fuzzing is able to generate testcases that lead the

execution to the location of them. To trigger the vulnerability is another problem. Checking the properties of certain registers along the concolic execution of a testcase can help discover vulnerabilities otherwise inefficient with fuzzing techniques.

Any testcases generated by the concolic execution engine will be placed in a synchronize queue directory waiting for the fuzzing engine to pick up. At the same time, even though no changes were made to the synchronization process of AFL, we modified the code to let AFL swith to the newly synchronized testcase from concolic execution as soon as possible to take advantage of the newly discovered edges and continue explore the code behind them.

We utilize S2E [34] as our concolic execution backend.

S2E introduction and how it works

In order for it to fulfill our requirements. Several improvements are made to the concolic execution engine. Most notably the *ConcolcExploreSearcher* and *PropertyChecker*

Testcase Generation with Concolic Execution

The *ConcolicExploreSearcher* is the responsible for guiding the concolic execution and at every branching condition, make decisions such as whether concolic state fork should be enabled, whether a testcase should be generated for the current execution state, whether a state switch should be performed and so on.

State fork. The concolic execution state fork consists of two parts: the feasibility query and the actual state branching. With *ConcolicExploreSearcher*, whether a state fork feasibility query should be performed at all depends on 3 factors. 1. Which state
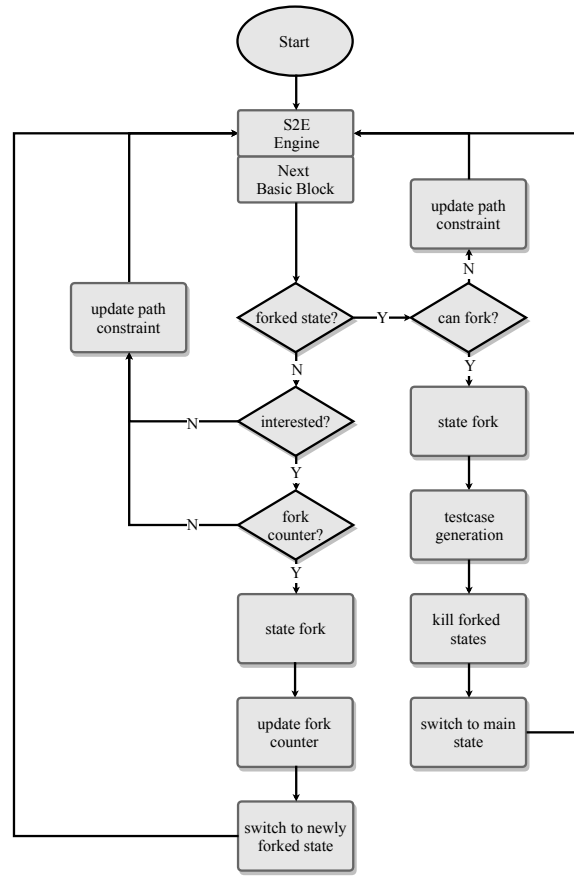
Figure 4.5: The process of concolic exeution given a seed and pre-defined target basic blocks

is the concolic execution engine currently running on. If running on the main state, which is the state that the seed leads to, further analysis should be performed to decide whether the state fork should happen. If running on a forked state, the feasibility query will always be performed and the state branching will take place as soon as the path constraint together with the branching constraint allows the state fork, otherwise the *ConcolicExploreSearcher* will append the branching con-

straint to the path constraints of the forked state and continue evaluate the next translation block. 2. Whether the current basic block being evaluated is a target basic block. This is straight forward, if the current basic block is not in the list of the target basic blocks, no feasibility query will be sent to the constraint solver, the concolic execution engine will continue evaluate the next translation block. The current branching constraint that satisfies the path the concolic execution seed is taken is appended to the path constraints. 3. How many times a target basic block has been hit during the current invocation of the concolic execution. In order to match the *bitbucket* of AFL and the implement the loop awareness, the *ConcolicExploreSearcher* assigns each of the target basic blocks a counter to record how many times the concolic execution engine has encounter a particular target basic block. Fork is only enabled when the hit count meet the predefined creteria.

Testcase generation. With *ConcolicExploreSearcher*, the testcase generation follows the following guidelines: 1. Testcase should be generated if the testcase can trigger a crash is found by the *PropertyChecker*. 2. Testcase generation that increase code coverage always generates a pair of two testcases as soon as a successful state fork is taken place on the forked state, representing both the true branch and the false branch of the state fork. After the testcases are generated, both of the forked states are killed. .

The purpose of generating two testcases each time is to work around AFL's hash collision problem. Because of AFL's fitness evaulation, only when all the edges have hash collision in the bitmap, a testcase that introduces new edges will fail to be synchronized into AFL's testcase queue. This means that if only one new basic

block coverage is introduced by a test case, it will fail to be synchronized by AFL only when both the inbound edge and outbound edge of the new basic block have hash collision. While according to Table 4.1 the chance of this to happen is low, it is not low enough to be neglected. Our approach to mitigate the problem is for each of the basic block of interest, the *ConcolicExploreSearcher* will generate testcases that take different outbound edges, increasing the chance for the newly discovered basic block to be synchronized by AFL. While this approach cannot eliminate the hash collision problem, in practice we didn't notice the scenario where basic blocks of interest failed to be synchronized while there are certain testcases generated by concolic execution rejected by AFL because of hash collision.

State switch. The state switch indicates that the concolic execution engine is about to switch the execution from one concolic state to another. This happens 1. as soon as the a successful state fork on the main state, and 2. the forked state is killed after a successful test case generation.

Property Checking for vulnerabilities

Similar to path constraints, to different types of vulnerabilities require different amount of effort to exploit. Some of the vulnerabilities, such as out of bound memory access with a invalid memory address that can be controlled by user input, can be triggered with a wide rage of invalid input while some other types of vulnerabilities requires particular values to be triggered (such as divide by zero vulnerability). Also, some of the vulnerabilities might not crash the target application while can still do harm (like memory leak). In cases like this is when concolic execution can be useful. By checking the values and

constraints on certain memory locations or cpu registers, concolic execution is able to determine whether the user input controlled behavior can do damage to target application or host system.

Take the code snippet in Listing 4.2 and Listing 4.3 for example. Listing 4.2 is the source code of *tiff2rgba* from libtiff version 4.0.6, with crafted user input, the value of *tif->tif_dir.td_bitspersample* can be set to 0, resulting a divide-by-zero vulnerabilty being exploited. Listing 4.3 is the assembly code of the location of the vulnerability. Since the value of *tif->tif_dir.td_bitspersample* is calculated from different places in the user input instead of directly loaded from one location, the *interest value* strategy of fuzzing does not work in finding this particular vulnerability, and the chance of the crash being triggered by random mutation is slim.

With concolic execution, as soon as an *idiv* instruction is encountered, the concolic execution engine can take the operand that contains the denominator and query the constraint solver whether it can be controlled by user input. And if yes, whether it can be set to 0. When both of the answers are yes, the concolic execution engine can then proceed and generate a testcase that is able to trigger the vulnerability.

We improved S2E by proposing *PropertyChecker*, *PropertyChecker* is implemented in lua scripting language and is expandable on demand. Right now it is able to :

- be invoked at certain stages of the execution, be it on particular instruction, while accessing specific memory location or when certain signal is emitted.

- check for the specific properties of a given register or memory address.

With *PropertyChecker*, we were able to find the divide-by-zero vulnerability mentioned above under 5 seconds and validate the heartbleed vulnerability [9] in less than 1

Listing 4.2: Divide by zero vulnerability from libtiff

```
369  static void
370  fpAcc(TIFF* tif, uint8* cp0, tmsize_t cc)
371  {
372      tmsize_t stride =
                PredictorState(tif)->stride;
373       uint32 bps =
                 tif->tif_dir.td_bitspersample / 8;
374      tmsize_t wc = cc / bps;
375      tmsize_t count = cc;
376      uint8 *cp = (uint8 *) cp0;
377      ...
378  }
```

Listing 4.3: Assenbly code of the vulnerability of libtiff

```
0x4468b7 <fpAcc+61>   mov -0x68(%rbp),%rax
0x4468bb <fpAcc+65>   cqto
0x4468bd <fpAcc+67>   idiv %rbx
0x4468c0 <fpAcc+70>   mov %rax,-0x20(%rbp)
```

minute.

4.4   Experiment

In order to evaluate the effectiveness and efficiency of SEZZER. The experiment is done on different benchmark binaries as well as real-world applications. We evaluate the performance in three categories: node coverage, path coverage and number of bugs found.

The result showed that with bench mark binaries, SEZZER is able to achieve an average of more than 50% more node coverage comparing to several other state-of-the-art software testing tools. With Lava-M test suite [40], SEZZER is able to find all the inserted vulnerabilities in 3 out of 4 binaries in a short amount of time, and exploited 2076 out of 2163 vulnerabilities of the 4th binary in a 24 hour run. The experiment result on GNU Core Utilities showed that Sezzer increased coverage for most of the binaries except for 6 of them. We also found several bugs in master branch of GNU-binutils that resulted in 5 patches and 3 CVEs.

4.4.1   Testing Metrics

- Node coverage measures the percentage of code covered by generated test cases in terms of basic blocks . Node coverage is a good measurement regarding the performance of a system software tool [81] and has been used constantly in measuring the efficiency of concolic execution systems [54, 29].

- Path coverage analyzes the number of paths covered for the target application by a specific testing technique. Unlike node coverage, which can be analyzed accurately

since it is calculated with the number of basic blocks, many applications have infinite number of execution paths and it is impossible to measure the path coverage percentage.

- The number of unique bugs is measured by counting the number of unique stack hashes [66] among crashes. We report bugs only when a generated test case can produce a core file during concrete execution.

Even though the ultimate goal of a system software testing technique is to find more bugs, the node coverage and path coverage information are also important. Node coverage gives a high level indication of which part of the code have been covered, but complete node coverage does not guarantee vulnerability discovery. Vulnerabilities may hide behind loops and never be triggered. Path coverage indicates the different behaviors of the target application, but path coverage driven techniques can easily get stuck inside infinite loops and fail to explore new code. The two metrics combined can give a picture of the testing progress to some extent.

4.4.2   AFL bitmap coverage

Figure 4.6 shows the results on the benchmark binaries comparing to other state-of-the-art system software testing techniques. The experiment was conducted on 9 different benchmarks. 1. those favored for evaluation by the AFL creator (djpeg from libjpeg-turbo-1.5.1, and readpng from libpng-1.6.29), 2. those used in AFLFastfhs evaluation (tcpdump -nr from tcpdump-4.9.0; and nm, objdump -d, readelf -a, and c++filt from GNU binutils-2.28), 3. and a few benchmarks with more complex input grammars in which
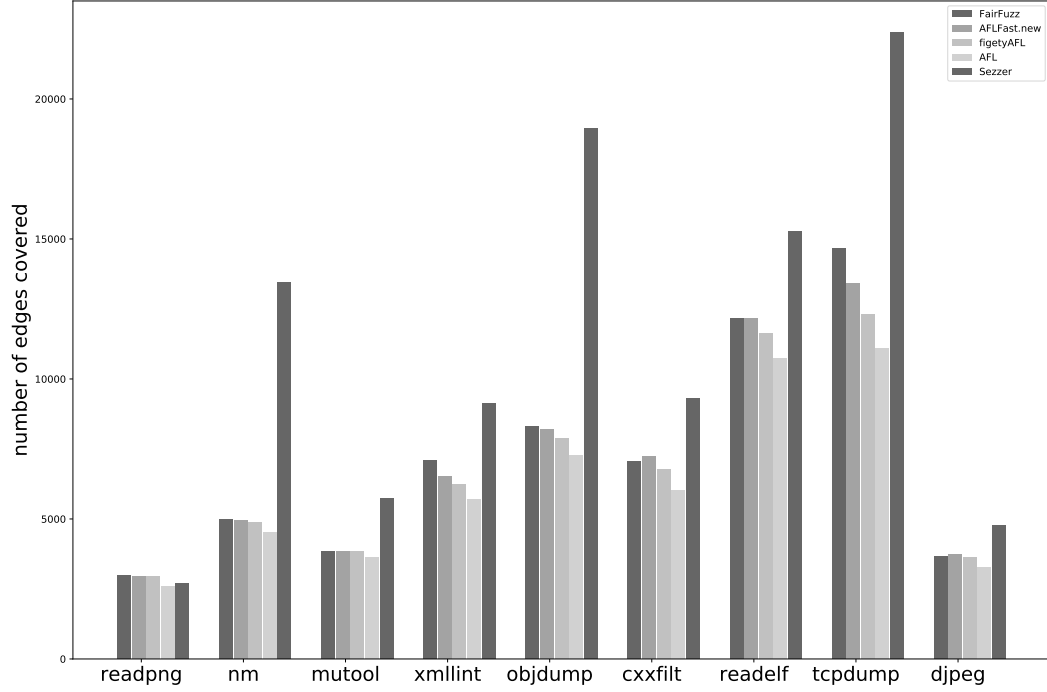
Figure 4.6: experiment results - binaries

AFL has previously found vulnerabilities (mutool draw from mupdf-1.9, and xmllint from libxml2- 2.94).

For Sezzer, we executed each of the target application with 6 AFL instances and 4 concolic execution worker. And for each of the other tools, the target applications were executed for 24 hours with 10 nodes. The result showed that comparing to the best performer other than Sezzer, our framework achieved more than 130% more coverage on 2 of the binaries (nm and objdump), around 30% 50% more coverage on 5 of the binaries (mutool, xmllint, cxxfilt, readelf and tcpdump), 13% more coverage on one binary (djpeg) and 9% less coverage on 1 binary(readpng).

4.4.3   Lave-M bugs

LAVA is a technique dynamic taint analysis-based technique for producing ground-truth corpora by quickly and automatically injecting large numbers of realistic bugs into program source code[40]. The authors created a test suite LAVA-M by injecting multiple bugs into four popular linux applications: *uniq*, *base64*, *md5sum*, and *who*. Each injected bug has an unique ID, and the crash is triggered by calling *printf* with the id of the bug and a crafted memory address. We compared Sezzer with the following state-of-the-art techniques:

FUZZER and SES . The result of FUZZER and SES are retrieved from the work of LAVA [40], the authors only mentioned FUZZER is a coverage-based fuzzer and SES as symbolic execution and SAT solving but did not provide details of the implementation or the symbolic execution search strategy.

VUzzer : a grey box fuzzing technique based on AFL by extending the mutation algorithm with the "magic bytes" strategy [71]. We got the numbers of crashes found from the VUzzer publication.

Steelix : another greybox fuzzing technique that utilizes static analysis and binary instrumentation to provide not only coverage information but also comparison progress information. [58].

Angora : mutation-based fuzzer that utilizes scalable byte-level taint tracking, context-sensitive branch count, search algorithm based on gradient descent, shape and type inference, and input length exploration. We also included the results of vanilla

Listing 4.4: Assembly code of the injected crash of LAVA-M on base64

```
...
 804a75c:       3d 6c 61 75 76          cmp     $0x7675616c,%eax
 804a761:       0f 94 c2                sete    %dl
 804a764:       c1 e3 02                shl     $0x2,%ebx
 804a767:       0f af d3                imul    %ebx,%edx
 804a76a:       8d 44 14 1c             lea     0x1c(%esp,%edx,1),%eax
 804a76e:       83 ec 0c                sub     $0xc,%esp
 804a771:       89 44 24 0c             mov     %eax,0xc(%esp)
 804a775:       68 de 00 00 00          push    $0xde
 804a77a:       e8 a1 f0 ff ff          call    8049820 <lava_get>
 804a77f:       c7 04 24 de 00 00 00    movl    $0xde,(%esp)
 804a786:       89 c3                   mov     %eax,%ebx
 804a788:       e8 93 f0 ff ff          call    8049820 <lava_get>
...
```

AFL-2.51b, which was executed for five hours on each binary by the author of Angora, for comparison.

Along the experiment of LAVA-M test suite, we discovered that the way LAVA injects the bugs leaves hints in the binaries that can help AFL discovering them. Take the code snippet in Listing 4.4 for example. The assembly code is directly extracted from the result of *objdump* of the binary *base64* of LAVA-M test suite. we can see from the two lines marked red that the injected code is comparing the register to a fixed value, which represents a particular injected bug, and calls the crash triggering function *lava_get* with the value. This means that with the approach LAVA injects the bugs, all the magic numbers are hard coded in the binary, we can extract all the magic checks in the binary with static analysis techniques and leverage AFL's dictionary to brute-force through all the magic

Binary	Sezzer	Sezzer+Dict	Angora	AFL	FUZZER	SES	VUzzer	Steelix
uniq (28)	19	33	29	9	7	0	27	7
base64 (44)	34	45	48	0	7	9	17	43
md5sum (57)	47	61	57	0	2	0	-	28
who (2136)	1840	2076	1541	1	0	18	50	194

Table 4.2: Number of crashes found in Lava-M test suite by different tools

checks at each and every mutation.

```bash
#!/bin/bash

objdump -d "$1" | grep cmp | \
                grep %eax | \
                grep -Eo '\$0x[0-9a-f]+' | \
                cut -c 2- | \
                sort -ug > "dict_$1"
```

With the discovery, we conducted two runs with Sezzer, one without dictionary and the other with dictionary extracted for each binary with the following bash command:

We run Sezzer with 6 AFL instances and 4 concolic execution nodes for 5 hours for each of the binary, Figure 4.7 and Table 4.2 shows the result of Lava-M test suite. The number of crashes found are normalized for each of the binary by comparing to the number of bugs established by the author of LAVA. Without dictionary, Sezzer is able to achieve decent results by triggering more than 65
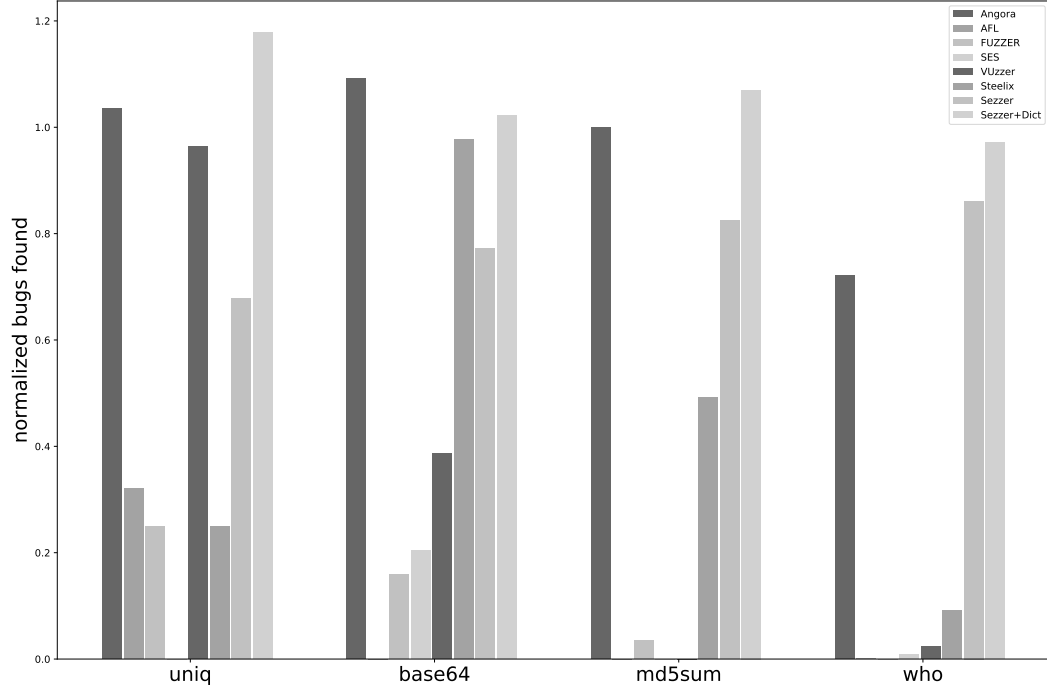
Figure 4.7: experiment results - Lava-M

4.4.4 Coreutils number of testcase

The GNU Core Utilities are the basic file, shell and text manipulation utilities of the GNU operating system. They are the core utilities which are expected to exist on every operating system.

We measured the node coverage of 33 coreutils binaries in terms of total number of valid testcases in AFL's synchronize queue. We conducted two runs for each of the binary application, one with Sezzer and one with vanilla AFL. AFL is executed with 10 nodes in parallel mode and Sezzer runs with 6 fuzzing nodes and 4 concolic execution nodes. And each of the binary is executed for half an hour.

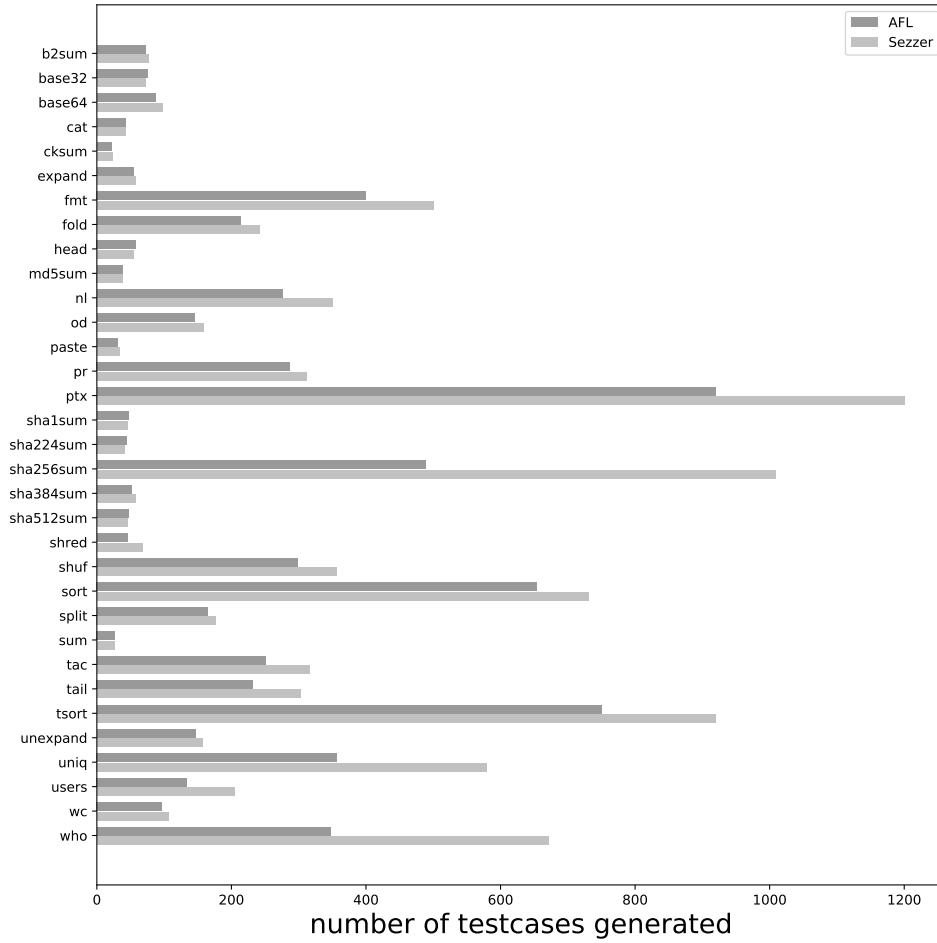Figure 4.8: experiment results - CoreUtils

Figure 4.9 and Figure 4.8shows the difference for each binary. Sezzer improves the node coverage for 19.3% on average and decreased coverage by a small margin on only 6 of the binaries.
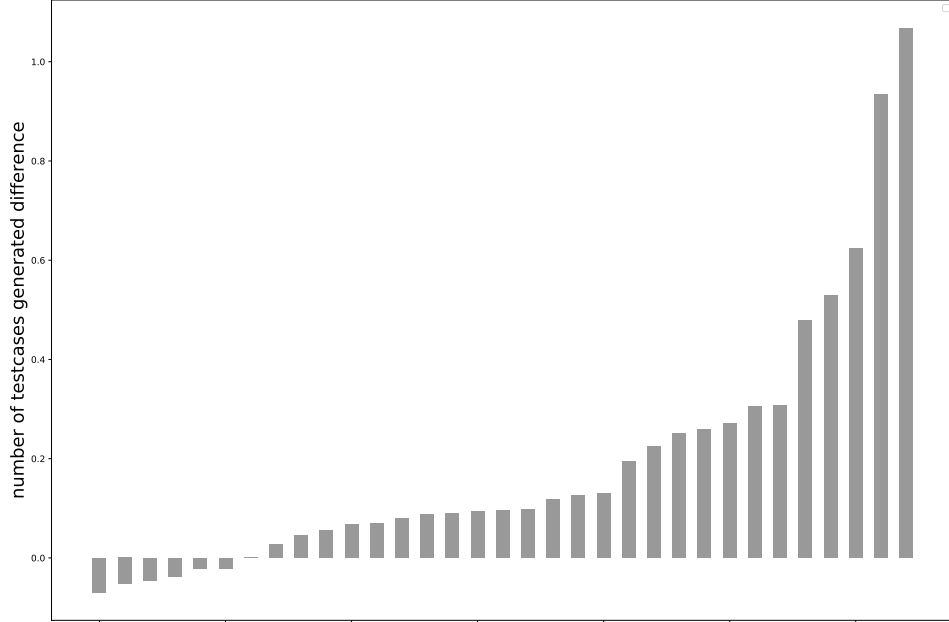
Figure 4.9: experiment results - CoreUtils

4.4.5 Real World Vulnerability Detection

As part of the evaluation, we conducted tests on 11 binaries of the latest version of GNU Binary Utilities(version 2.30.51.20180321 as of this writing), As a result, we found 6 uniq bugs across 4 different applications and resulted in 5 patches upstream and 3 CVEs. Since binutils have been extensively tested by different kinds of tools [29, 54, 31, 16, 26, 61], the vulnerabilities found by Sezzer further indicates the capability of vulnerability discovery of Sezzer.

Further investigation on one of the testcases generated that lead to the vulnerabilities showed that the final testcase is a descendant of testcase generated by concolic execution, which in turn the seed is the descendant of another testcase generated by concolic exe-

cution. This is the evidence that Sezzer extends the reach of both fuzzing and concolic execution.

Chapter 5

Conclusion And Future Work

Software testing is one of the most important phase of software development process. In the dissertation, we investigated the thesis of automatic system testing by:

- Extracting the implementation of virtual device implementation from virtual machine monitor (VMM) and modeling them with symbolic execution. This allows us to be able to utilize the power of symbolic execution to find security vulnerabilities in the implementation of VMM by executing only a fragment of the implementation of the code.

- We show how different system software testing techniques can be integrated together to improve the effectiveness by investigating Sezzer, a framework that incorporates fuzzing and symbolic execution yet overcomes the major disadvantages from both of them. In Sezzer, fuzzing is used to explore the general conditions of the target program, and with the basic coverage information from fuzzing, concolic execution is used to help getting over the complex constraint checks on the

execution path of the application as well as check for specific types of vulnerabilities during the execution which can otherwise be hard to find by fuzzing.

5.1   Problems And Future Work

We conclude the thesis with a list of limitations and challenges that and possibly be solved in future work.

Better Control Flow Graph.   We use control flow graph as our representation of code coverage during the implementation of Sezzer, however, it is hard to get a complete CFG from reverse engineering technique with our selection of CFG building tools. With incomplete information, it is difficult for seed selector to compute basic block of interest and target basic block accurately, and thus lead to poorer than anticipated performance.

Constraint caching and sharing.   Because we are launching the concolic execution repeatedly with very few state forks instead of the traditional "launch once and let it explore". Same path constraints are being sent to constraint solver repeatedly. With the current implementation, the counter example cache is not preserved between different concolic execution initiations, meaning that those recurrences are always treated as new queries thus might waste solving time. A persistent constraint counter example cache that can be shared between different concolic execution instances will boost the performance a lot.

Incomplete Register support.   Even though S2E, the concolic execution engine we were

using during the implementation of Sezzer, utilizes QEMU for full system virtualization on symbolic execution, does not support making certain CPU registers as symbolic such as MMX registers. Those target application that optimizes for such registers are unable to execute symbolically. Extend the symbolic execution engine and support all the CPU registers will yield a wider range of application that can be tested with Sezzer

Appendix A

Binaries Tested in Coreutils and Launching Commands

Binary	Command Line Options
b2sum	@@
base32	@@
base64	@@
cat	-T @@
cksum	@@
expand	@@

fmt	@@
fold	@@
head	@@
md5sum	@@
nl	@@
od	@@
paste	@@
pr	@@
ptx	@@
sha1sum	@@
sha224sum	@@
sha256sum	@@
sha384sum	@@
sha512sum	@@

shred	@@
shuf	@@
sort	@@
split	@@
sum	@@
tac	@@
tail	@@
tsort	@@
unexpand	@@
uniq	@@
users	@@
wc	@@
who	-a @@

Table A.1: The binaries and command line options of coreutils tested

Appendix B

Binaries Tested in Binutils and Launching Commands

Binary	Command Line Options
ar	-xo @@
as-new	@@
cxxfilt	-t
ld-new	@@
nm-new	-aClSs –special-syms –synthetic @@
objcopy	@@ /dev/null

objdump	-a -f -p -P -h -x -d -D -S -s -g -e -G -W -t -T -r -R -l -F -C @@
ranlib	@@
size	-A -t @@
strings	-afw @@
strip-new	@@

Table B.1: The binaries and command line options of binutils tested

Bibliography

[1] Cve-2014-6278, . URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6278.

[2] Cve-2014-8637, . URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-8637.

[3] Cve-2014-9427, . URL https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-9427.

[4] Cve-2015-5477, . URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5477.

[5] Cve-2015-0208, . URL https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0208.

[6] Cve-2015-0288, . URL https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0288.

[7] Qt project security advisory, . URL http://lists.qt-project.org/pipermail/announce/2015-April/000067.html.

[8] Advisory-2015-03-17, . URL `http://www.x.org/wiki/Development/Security/Advisory-2015-03-17/`.

[9] cve-2014-0160. URL `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160`.

[10] Microsoft hyper-v virtualization. URL `http://www.microsoft.com/virtualization`.

[11] Pwn2own 2016 contest rules. URL `http://zerodayinitiative.com/Pwn2Own2016Rules.html`.

[12] Vmware virtualization for desktop and server, application, public and hybrid clouds. URL `http://www.vmware.com/`.

[13] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM. doi: 10.1145/800028.808479. URL `http://doi.acm.org/10.1145/800028.808479`.

[14] Ariane. The ariane catastrophe. URL `https://around.com/ariane.html`.

[15] asergrp. Symbolic execution bibliography. URL `https://sites.google.com/site/asergrp/bibli/symex`.

[16] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1083–1094. ACM, 2014.

[17] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. The bincoa framework for binary code analysis. In *International Conference on Computer Aided Verification*, pages 165–170. Springer, 2011.

[18] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[19] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2015.

[20] Nils Aall Barricelli. Symbiogenetic evolution processes realized by artificial methods. *Methodos*, 9(35-36):143–182, 1957.

[21] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[22] Fabrice Bellard. Tiny code generator, 2009.

[23] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.

[24] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 122–131. IEEE Press, 2013.

[25] Robert S Boyer, Bernard Elspas, and Karl N Levitt. Selectfha formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10 (6):234–245, 1975.

[26] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems*, pages 183–198. ACM, 2011.

[27] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 161–169. IEEE Computer Society, 2009.

[28] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Dawn Song, et al. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 413–425. ACM, 2010.

[29] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[30] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[31] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Un-

leashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[32] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. *arXiv preprint arXiv:1803.01307*, 2018.

[33] Yanpei Chen, Vern Paxson, and Randy H Katz. Whatfhs new about cloud computing security. *University of California, Berkeley Report No. UCB/EECS-2010-5 January*, 20 (2010):2010–5, 2010.

[34] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices*, 46(3):265–278, 2011.

[35] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[36] CNN. Toyota recall costs: $2 billion. URL http://money.cnn.com/2010/02/04/news/companies/toyota_earnings.cnnw/index.htm.

[37] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 117–130. ACM, 2007.

[38] Jared DeMott, Richard Enbody, and William F Punch. Revolutionizing the field

of grey-box attack surface testing with evolutionary fuzzing. *BlackHat and Defcon*, 2007.

[39] Ankita Desai, Rachana Oza, Pratik Sharma, and Bhautik Patel. Hypervisor: A survey on concepts and taxonomy. *International Journal of Innovative Technology and Exploring Engineering*, 2(3):222–225, 2013.

[40] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 110–121. IEEE, 2016.

[41] Alex S Fraser. Simulation of genetic systems by automatic digital computers i. introduction. *Australian Journal of Biological Sciences*, 10(4):484–491, 1957.

[42] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[43] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[44] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[45] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, pages 49–64, 2013.

[46] Michael R Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review*, 43(3):14–26, 2009.

[47] Takahiro Hirofuchi, Hidemoto Nakada, Satoshi Itoh, and Satoshi Sekiguchi. Enabling instantaneous relocation of virtual machines with a lightweight vmm extension. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 73–83. IEEE, 2010.

[48] William E Howden. Methodology for the generation of program test data. *IEEE Transactions on computers*, 100(5):554–560, 1975.

[49] Institute Systems Sciences IBM. It is 100 times more expensive to fix security bug at production than design. URL `https://www.owasp.org/images/f/f2/Education_Module_Embed_within_SDLC.ppt`.

[50] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A Navas. Boosting concolic testing via interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 48–58. ACM, 2013.

[51] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *International Conference on Computer Aided Verification*, pages 423–427. Springer, 2008.

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

[53] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux

virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[54] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Acm Sigplan Notices*, 47(6):193–204, 2012.

[55] laf intel. Circumventing fuzzing roadblocks with compiler transformations. URL `https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/`.

[56] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[57] Caroline Lemieux and Koushik Sen. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. *arXiv preprint arXiv:1709.07101*, 2017.

[58] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637. ACM, 2017.

[59] Qian Liu, Chuliang Weng, Minglu Li, and Yuan Luo. An in-vm measuring framework for increasing virtual machine security in clouds. *Security & Privacy, IEEE*, 8 (6):56–62, 2010.

[60] Yingwei Luo, Binbin Zhang, Xiaolin Wang, Zhenlin Wang, Yifeng Sun, and Haogang Chen. Live and incremental whole-system migration of virtual machines using

block-bitmap. In *Cluster Computing, 2008 IEEE International Conference on*, pages 99–106. IEEE, 2008.

[61] Paul Dan Marinescu and Cristian Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 716–726. IEEE, 2012.

[62] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[63] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[64] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, Technical report, 1995.

[65] F.F. Moghaddam and M. Cheriet. Decreasing live virtual machine migration downtime using a memory page selection based on memory change pdf. In *Networking, Sensing and Control (ICNSC), 2010 International Conference on*, pages 355–359, April 2010. doi: 10.1109/ICNSC.2010.5461517.

[66] David Molnar, Xue Cong Li, and David Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, volume 9, pages 67–82, 2009.

[67] Gábor Pék, Levente Buttyán, and Boldizsár Bencsáth. A survey of security issues in hardware virtualization. *ACM Computing Surveys (CSUR)*, 45(3):40, 2013.

[68] Diego Perez-Botero, Jakub Szefer, and Ruby B Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing*, pages 3–10. ACM, 2013.

[69] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[70] Reese T Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138. ACM, 1959.

[71] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[72] Hex Rays. Ida pro. URL https://www.hex-rays.com/products/ida/index.shtml.

[73] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.

[74] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 842–853. ACM, 2015.

[75] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and

Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[76] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[77] Subashini Subashini and Veeraruna Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of network and computer applications*, 34(1):1–11, 2011.

[78] Jon Watson. Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1, 2008.

[79] Michafk Zalewski. American fuzzy lop. URL `http://lcamtuf.coredump.cx/afl/`.

[80] Guodong Zhu, Kang Li, and Yibin Liao. Toward automatically deducing key device states for the live migration of virtual machines. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 1025–1028. IEEE, 2015.

[81] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.