Deceptive Environments for Cybersecurity Defense on Low-power Devices

Alexander L. Kedrowitsch

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Computer Science and Applications

Danfeng Yao, Chair
Gang Wang
David R. Raymond

April 12, 2017
Blacksburg, Virginia

Keywords: Security, Honeypots, Linux Containers
Copyright 2017, Alexander L. Kedrowitsch

Deceptive Environments for Cybersecurity Defense on Low-power Devices

Alexander L. Kedrowitsch

ACADEMIC ABSTRACT

The ever-evolving nature of botnets have made constant malware collection an absolute necessity for security researchers in order to analyze and investigate the latest, nefarious means by which bots exploit their targets and operate in concert with each other and their bot master.

In that effort of on-going data collection, honeypots have established themselves as a curious and useful tool for deception-based security. Low-powered devices, such as the Raspberry Pi, have found a natural home with some categories of honeypots and are being embraced by the honeypot community. Due to the low cost of these devices, new techniques are being explored to employ multiple honeypots within a network to act as sensors, collecting activity reports and captured malicious binaries to back-end servers for later analysis and network threat assessments. While these techniques are just beginning to gain their stride within the security community, they are held back due to the minimal amount of deception a traditional honeypot on a low-powered device is capable of delivering.

This thesis seeks to make a preliminary investigation into the viability of using Linux containers to greatly expand the deception possible on low-powered devices by providing isolation and containment of full system images with minimal resource overhead. It is argued that employing Linux containers on low-powered device honeypots enables an entire category of honeypots previously unavailable on such hardware platforms. In addition to granting previously unavailable interaction with honeypots on Raspberry Pis, the use of Linux containers grants unique advantages that have not previously been explored by security researchers, such as the ability to defeat many types of virtual environment and monitoring tool detection methods.

Deceptive Environments for Cybersecurity Defense on Low-power Devices

Alexander L. Kedrowitsch

GENERAL AUDIENCE ABSTRACT

The term 'honeypot', as used in computer security, refers to computer systems that are intended to be targeted by malicious third parties, but contain little value. While these systems are being attacked, the honeypot collects as much data as it can on the actions being performed by the attacker; information that is extremely useful for security researchers in understanding the latest techniques and methods that are employed by cyber-criminals. Unfortunately, not all honeypot architectures are equal and often trade-offs have to be made between ease of setup, cost of hardware, and how realistic the honeypot is capable of behaving.

This thesis proposes that by using a new and useful software package available to Linux computer systems called 'Linux Containers', it is possible to implement honeypots that significantly reduce the amount of trade-offs required. Specifically, honeypots that are capable of highly realistic behavior can be run on highly affordable, low-power devices, such as the Raspberry Pi.

In addition to granting realistic honeypots the ability to operate on low-cost devices, Linux containers also provide the benefit of defeating several, difficult to overcome methods that malicious software authors implement in order to prevent their malware from being monitored and analyzed by security experts. Defeating the investigated forms of environment detection has remained a difficult challenge for security experts and remains an open-ended problem in the field.

Dedication

I dedicate this work to my loving wife and two beautiful daughters. Without my wife's constant encouragement during the journey, I would be far less of a person than I am today. I am also indebted to the understanding of my daughters for spending far less time with them each day then they deserve. I love you all.

Contents

1 Introduction **1**

2 Threat Model **2**

3 Background **3**

3.0.1 Botnets 3

3.0.2 Botnet History and Evolution of Architecture 3

3.0.3 Motivation for Botnet Development and the Threat to the Internet Community 5

3.0.4 Bot and Botnet Detection 6

3.0.5 Challenges in Botnet Infiltration and Takedown 6

3.1 Honeypots 6

3.1.1 Case Study on *Glastopf* honeypot 9

3.1.2 Low/Medium-Interaction vs High-Interaction Honeypots 18

3.1.3 Aggregate Honeypot Servers 19

3.2 Rise of the Raspberry Pi and the Single-Board Computer (SBC) 21

4 The Problem **23**

4.1 Vital Role of Virtual Machines 23

4.2 VM Aware Malware 24

4.3 Related Work in Deception and VM Aware Malware 28

5 Our Contribution **32**

5.1 Linux Containers 33
5.1.1 Linux Container Security Assumptions 34
5.2 Linux containers as a Viable Alternative to Virtual Environments on Low Performance Machines 35
5.3 Experiment Proposal 36

6 The Experiment **38**
6.1 Design 38
6.1.1 CPU-based Environment Detection Tests 39
6.1.2 File System-based Environment Detection Tests 40
6.1.3 Hardware Attribute-based Environment Detection Tests 41
6.1.4 System Memory Performance-based Environment Detection Tests 41
6.1.5 Network-based Environment Detection Tests 41
6.1.6 Linux Container Expected Performance 42
6.2 Experiment Lab 42
6.2.1 Hardware 43
6.2.2 Operating System and Software 46
6.2.3 Ubuntu Core 47
6.2.4 Virtual Environment Software 47
6.3 Experiment Tests 49
6.3.1 CPU-based Environment Detection Test Details 49
6.3.2 File System 51
6.3.3 Hardware Attributes 51
6.3.4 System Memory Performance 51
6.3.5 Network 52

7 The Results **55**
7.1 Experiment Results 55
7.1.1 CPU-based Environment Detection Tests 55

7.1.2 CPU Information . . . 62
7.1.3 Execution time of privileged instructions . . . 63
7.1.4 Artifacts in the File System . . . 65
7.1.5 Artifacts in System Hardware Attributes . . . 66
7.1.6 System Memory Performance . . . 66
7.1.7 Variability in TCP Timestamp Clock Skew . . . 67
7.1.8 Experiment Conclusion . . . 72

8 Benchmarking Variability **73**
8.1 Disk I/O Throughput . . . 73
8.2 CPU Cache Latency . . . 74

9 Container Detection **85**
9.1 Linux Namespaces . . . 85
9.2 Permissions . . . 86
9.3 Raspberry Pi Detection . . . 88
9.4 Analysis Tool Detection . . . 91

10 Deception Enabled by Containers **93**

11 Future Work **96**

12 Additional Discussion **97**
12.1 Refactoring Honeypots for Aggregate Server Interfaces . . . 97
12.1.1 Refactoring . . . 97
12.1.2 Refactoring Workflow Template . . . 98
12.1.3 Developing a Refactoring Template . . . 99
12.1.4 Case Study . . . 100
12.1.5 Case Study Results . . . 105
12.1.6 Refined Refactoring Workflow . . . 105
12.1.7 Code Review Results . . . 106

12.1.8 Considered Solutions . . . 106
12.1.9 Refactoring Conclusions . . . 107
12.1.10 Future Work of Honeypot Refactoring . . . 107

13 Conclusion **109**

Bibliography **111**

Appendix **119**

A Implemented Code **120**
A.1 CPU Clock Variability Test . . . 120
A.1.1 ClockTest.sh . . . 120
A.1.2 clocks.c . . . 121
A.1.3 ClockBench.cpp . . . 123
A.1.4 ClockBench.cpp . . . 127
A.2 CPU Information Test . . . 131
A.2.1 CPUInfo.sh . . . 131
A.2.2 CPUInfo.py . . . 132
A.2.3 CPUID.c . . . 132
A.3 CPU Execution Time Test . . . 133
A.3.1 timer_test.c . . . 133
A.4 Artifacts in the File System Test . . . 141
A.4.1 file_search.sh . . . 141
A.5 Artifacts in System Hardware Attributes Test . . . 142
A.5.1 hw_attribute_check.sh . . . 142
A.6 System Memory Performance Test . . . 145
A.6.1 CPUID_TLB_flush.c . . . 145
A.7 Network Test . . . 149
A.7.1 collect_timestamps.sh . . . 149

A.7.2 timestamp_capture.c 152
A.7.3 tcpServerLinux.c 170
A.7.4 tcpClientLinux.c 174
A.8 Analysis Tool Detection Test 178
A.8.1 self_status.c 178
A.9 Container Detection Tests 179
A.9.1 detect_lxc.c 179
A.9.2 perm_test.sh 180
A.10 Raspberry Pi Detection Tests 181
A.10.1 io_perf.c 181
A.10.2 io_perf2.c 183

List of Figures

3.1 Illustrating the number of attacks from the top 10 attack origin countries recorded during the case study. 11

3.2 World map illustrating the countries of origin for attacks on *Glastopf* web-server honeypot. 13

3.3 World map illustrating the Internet connectivity density [22]. 13

3.4 Illustrating the frequency of attacks on the *Glastopf* web-server honeypot. The day value is the number of days the honeypot was online for. 14

3.5 Top URL attack strings recorded by the *Glastopf* web-server honeypot and the number of times recorded during the case study 16

4.1 Hypervisor placement in Type I and Type II virtual machines compared with bare metal 26

6.1 Raspberry Pi 3 Model B 44

6.2 Minnowboard Turbot 45

7.1 Standard deviation of CPUID+RDTSC timing source on Cindy. X-axis is environment of execution. Y-axis is standard deviation. 56

7.2 Standard deviation of CPUID+RDTSC timing source on Dani. X-axis is environment of execution. Y-axis is standard deviation. 57

7.3 Standard deviation of CPUID+RDTSC timing source on Emma. X-axis is environment of execution. Y-axis is standard deviation. 58

7.4 Average time between CPUID+RDTSC timing polls on Cindy. X-axis is environment of execution. Y-axis is average number of clock ticks. 59

7.5 Average time between CPUID+RDTSC timing polls on Dani. X-axis is environment of execution. Y-axis is average number of clock ticks. 60

7.6 Average time between CPUID+RDTSC timing polls on Emma. X-axis is environment of execution. Y-axis is average number of clock ticks. 61

7.7 Execution time of 10,000 memory map operations. X-axis is environment of execution. Y-axis is execution time in seconds. 64

7.8 Clock skew variance for Emma, Bare Metal with least linear squares fit line and equation and low MSE. X-axis is time since start of packet capture (in seconds). Y-axis is clock skew offset (in seconds). 68

7.9 Clock skew variance for Xen VM running on Emma with least linear squares fit line and equation and high MSE. X-axis is time since start of packet capture (in seconds). Y-axis is clock skew offset (in seconds). 69

7.10 Clock skew variance for Linux container running on Dani with least linear squares fit line and equation and low MSE. X-axis is time since start of packet capture (in seconds). Y-axis is clock skew offset (in seconds). 70

7.11 Clock skew variance for VMWare VM running on Dani with least linear squares fit line and equation and high MSE. X-axis is time since start of packet capture (in seconds). Y-axis is clock skew offset (in seconds). 71

8.1 Box Plot for iozone FRead throughput benchmark by file size on Raspberry Pi 3 running with 2 threads at CPU frequency of 1.2GHz. System is using Deadline I/O scheduler. X-axis is file size in kB. Y-axis is throughput in kB/s. 75

8.2 Box Plot for iozone FRead throughput benchmark by record size on Raspberry Pi 3 running with 2 threads at CPU frequency of 1.2GHz. System is using Deadline I/O scheduler. X-axis is record size in kB. Y-axis is throughput in kB/s. 76

8.3 Box Plot for iozone FRead throughput benchmark by record size on server running with 2 threads at CPU frequency of 1.2GHz. Both Hypervisor and VM are using Deadline I/O Scheduler. X-axis is file size in kB. Y-axis is throughput in kB/s. 77

8.4 Box Plot for iozone FRead throughput benchmark by record size on server running with 2 threads at CPU frequency of 1.2GHz. Both Hypervisor and VM are using Deadline I/O Scheduler. X-axis is record size in kB. Y-axis is throughput in kB/s. 78

8.5 Box Plot for iozone FRead throughput benchmark by record size on server running with 2 threads at CPU frequency of 1.2GHz with filesize of 1,024kB. Both Hypervisor and VM are using Deadline I/O Scheduler. X-axis is record size in kB. Y-axis is throughput in kB. 79

8.6 Box Plot for iozone FRead throughput benchmark by record size on Raspberry Pi 3 running with 2 threads at CPU frequency of 1.2GHz with filesize of 1,024kB. Both Hypervisor and VM are using Deadline I/O Scheduler. X-axis is record size in kB. Y-axis is throughput in kB. 80

8.7 Box Plot illustrating variance in CPU cache latency by machine. X-axis is the system/VM. Y-axis is latency in nanoseconds. 81

8.8 Line graph illustrating cache latency over range of array sizes with read stride length of 512kB on Dani. X-axis is the array size in MB. Y-axis is latency in nanoseconds. 82

8.9 Plot comparing cache latency between bare metal Dani and VMWare virtual machine running on Dani with read stride length of 512kB. X-axis is the machine. Y-axis is latency in nanoseconds. 83

8.10 Plot comparing cache latency between bare metal Emma and Xen virtual machine (PV) running on Emma with read stride length of 512kB. X-axis is the machine. Y-axis is latency in nanoseconds. 84

9.1 Box Plot for iozone FRead throughput benchmark by record size on Raspberry Pi with 1 thread at CPU frequency of 1.2GHz with filesize of 1,024kB. Kernel is using Deadline I/O Scheduler. X-axis is record size in kB. Y-axis is throughput in kB. 89

9.2 Box Plot for iozone FRead throughput benchmark by record size on server with 1 thread at CPU frequency of 1.2GHz with filesize of 1,024kB. Both Hypervisor and VM are using Deadline I/O Scheduler. X-axis is record size in kB. Y-axis is throughput in kB. 90

10.1 Line graph illustrating cache latency over range of array sizes with read stride length of 512kB on Betty. X-axis is the array size in MB. Y-axis is latency in nanoseconds. 94

12.1 An overview of the file structure and module interaction of Artillery. 101

List of Tables

3.1 Listing several popular open-source honeypots and their interaction category. 7
3.2 System Resource Expense vs Honeypot Interaction Level 19

5.1 System Resource Expense vs Honeypot Interaction Level with Containers . . 36

6.1 Table of lab hosts and installed operating systems. 47
6.2 Table of systems and installed hypervisors. 48

7.1 Results of CPU Information Test 63
7.2 Summary of Virtual Environment Detection Test Results. 65

9.1 Results of Container Detection Tests 87
9.2 Results of Raspberry Pi Detection Tests 91

Chapter 1

Introduction

Attacks to online systems can come in a variety of methods and from a variety of attackers ranging from humans conducting perimeter defense penetration and lateral movement to sophisticated, distributed, automated attacks made by botnets. The methods, methodologies, and technologies utilized by attackers changes daily, increasing the need for in-depth research and study of new attacks to a desperate level. Security researchers have developed a number of tools, referred to as honeypots, that are intended to deceive an attacker into believing they are attacking a valuable system and provide defenders with the opportunity to monitor the behavior of attackers and study the tools utilized. However, honeypots have historically suffered from several drawbacks. Initially, defenders must decide whether to implement difficult-to-manage and potentially dangerous-to-operate clones of production systems (referred to as high-interaction honeypots) or easier-to-deploy-and-manage, but limited in deceptive capabilities, emulations of production systems (referred to as low/medium-interaction honeypots). This decision affects the interaction an attacker may have with the honeypot as they attempt to penetrate the target network, which reduces the opportunities to observe new attack techniques and methodologies. Second, once an attacker has "penetrated" the honeypot and begins interacting with the system, attackers oftentimes upload tools in the form of malicious binaries in order to further exploit certain vulnerabilities on the victim. In an effort to avoid study and analyzation by security researchers, an increasing number of malicious binaries are able to test for and determine whether they are executing in a virtual environment (where they are potentially monitored) and decide to alter the tools behavior in order to prevent such monitoring.

Chapter 2

Threat Model

This study addresses threats to Linux systems from automated attackers, such as those used by botnets for bot proliferation.

The threat model this thesis proposes to address consists of the following components:

- Attackers penetrating network-facing service providing systems using software exploits, service vulnerabilities, and the like. In an effort to maintain a feasible scope, this thesis will discuss system penetrations in a general sense rather than delve into specific technical characteristics focusing on only a sub-set of possible system penetrations.
- The execution of malicious binaries on a system once it has been penetrated by an attacker.

Internet-facing systems can expose a number of different attack vectors based on the size of it's attack surface. Restricting available network ports through the use of a hardware or software firewall, restricting the number of services running on a host, and maintaining the latest security updates are all methods in which a defender may be able to minimize the attack surface of a system.

Unfortunately, oftentimes not all available defense measures are taken or possibly attacks are performed that these measures do not defend against. Unknown service vulnerabilities can be exploited (known as 0-day exploits), vulnerabilities that have not yet had a security patch applied, or simply a vulnerability that is due to the nature of the system are all potential avenues in which an attacker may gain unintended access to a system.

Chapter 3

Background

3.0.1 Botnets

Bots and the botnets they create are a curious blend of technologies that pose a serious and legitimate concern to the Internet community at large. Bots are exploited computer systems that have been infiltrated by malicious software, in many respects similar to normal computer virus/worm infections. However, rather than working independently once a host system has been compromised, a bot establishes a connection to a command and control network that allows it to communicate (usually surreptitiously) with it's human controller, referred to a bot herder or bot master [4]. The bot master can then issue commands to the bots in it's botnet, directing all or a portion of the bots to perform various activities (a classic example being the execution of a Distributed Denial of Service Attack (DDoS)).

3.0.2 Botnet History and Evolution of Architecture

The IRC bot Eggdrop is attributed by [49] as the very first botnet. Like many original technological advancements, Eggdrop was benign in nature, using the Internet Relay Chat protocol (IRC) to communicate with other Eggdrop bots for useful purposes (so useful, in fact, that it is still continuing active development with Eggdrop v1.8.0 Stable being released on December 4th, 2016 [1]). However, leveraging the utility of independent, distributed IRC bots on a shared communication channel for nefarious purposes followed shortly after. GT-Bot, the first malicious botnet, was created in 1998 [49] which allowed attackers to infect systems and direct the bots to either perform actions on the infected host or instruct the infected host to perform actions against a remote system.

As the Internet has matured, the strength of early botnets, characterized by the use of an IRC server that provides Command and Communication (C2) between the bots and the bot master, became it's greatest weakness: Being dependent on a single, centralized

server allowed early botnets to be rendered either compromised or mute if the single C2 server was infiltrated or taken down. This threat motivated the evolution of a botnet's C2 network into other technologies, specifically decentralized communication protocols, such as Peer-to-Peer, in order to maximize the botnet's resiliency against take-down or infiltration. Not all network topologies are equalally balanced in terms of resiliency versus reaction time and often require specific tradeoffs depending on the botnet author's specific goals. The C2 methodologies employed by botnets can be roughly divided into the following categories [83]:

- Centralized
- Decentralized / Hybrid
- Unstructured

Centralized botnet C2 architectures encompass the "classic" view of a botnet, where each bot connects to a single server (although not technologically restricted to only one), and awaits commands. The most common protocols used for this architecture are IRC and HTTP [49]. Commands issued to bots are received immediately by all systems logged in to the botnet (some researchers have identified that modern botnets don't have all bots logged in simultaneously [13]). This architecture is the least resilient as the entirety of the botnet relies on a single, centralized communication channel in order to receive instructions from the botmaster. Detection of the C2 protocol is assumed to be relatively difficult due to the ubiquity of the leveraged protocols and the challenge it would be for network sniffers to isolate botnet traffic from legitimate protocol usage. However, with the shift away from IRC usage, this will not always remain the case for IRC-based botnets.

Decentralized botnet C2 architectures represent the next evolution in botnets as they do not rely on a single C2 server in order to receive commands, instead leveraging the decentralized nature of peer-to-peer (P2P) protocols. In 2002, the Linux Slapper Worm was discovered to be communicating via P2P protocol [20]. Many of these earlier evolved botnets, however, were not purely P2P, but instead were Hybrid architectures; rather, the P2P network would send an alert to the botnet, signaling the bots to connect to a centralized server in order to receive specific commands [21]. More recent versions of P2P botnets are now developed to be strictly P2P with no centralized C2 utilized at all. The P2P botnets, Nugache and Storm, both received significant press interest in the mid-2000, raising general awareness to the threat the new generation of botnets posed [28].

Botnets that utilize Fast Flux DNS are categorized as decentralized. Fast Flux botnets use DNS address record lists with very short TTL (time to live) to rotate through IPs of compromised systems. Double Flux also includes the authoritative name server being registered/de-registered to a number of different IPs as well [56]. Fast Flux is a technique used to prevent IP blocking as a means of preventing malicious traffic. Being decentralized, the possibility of compromising a single C2 server is removed, making the botnet highly

resistant to single take-down efforts. However, as a compromise, this resiliency does decrease the speed in which a botnet can react to issued commands as there is significant delay in the propagation of transmitted messages.

Unstructured botnet C2 architectures are the most fluid in their make-up and are simply defined as a network whose bots remain idle and wait for incoming connections. Commands are issued by the botmaster scanning the Internet for specific connection characteristics and passing the commands when a bot has been identified [49]. This does not appear to be an architecture thoroughly studied within the security community and it's application hints at being self-limiting due to the amount of effort involved for a botmaster to scan for and identify individual bots.

As with most technologies, the more advanced a method is, the more challenging and time consuming it is to implement; as such, some researchers maintain that centralized, IRC-based botnets remain the most prevalent [83][92].

3.0.3 Motivation for Botnet Development and the Threat to the Internet Community

While the media more readily identifies the link between botnets and DDoS attackers, security researchers and professionals frequently quote one of the prime motivations for establishing a botnet, like most other criminal activity, is for profit [46] [92]. Methods in which criminals generate funds from the use of botnets include:

- Generating income through spam email distribution
- Extortion campaigns through the use of Distributed Denial of Service Attacks (DDoS)
- Setting the stage for follow-on attacks [4]
- Providing botnets as a service to other criminals [13]

It has been argued that the botnet threat has yet to be fully understand [49] [92], so it's difficult to place a monetary figure on the threat botnets pose to both companies and the Internet in general. However, a single botnet campaign thwarted by the FBI in 2010 prevented the theft of $70 million dollars from global bank accounts [46], illustrating the non-trivial monetary threat posed by botnets.

Additionally, considering the distributed nature of botnets and the sizes they are able to achieve across the globe, some botnets have been quoted as having the combined processing power of modern supercomputers and having enough Internet bandwidth available to conduct DDoS attacks powerful enough to knock entire countries offline [32].

3.0.4 Bot and Botnet Detection

As bots are simply malicious binaries that attempt to conduct automated proliferation across the Internet, they are susceptible to a number of detection methods, such as signature or anomaly-based detection [32]. However, due to their unique feature of establishing communication with it's authoring source, bots are able to perform period code updates in-place in order to take advantage of the newest exploits or otherwise modify their shape and behavior to defeat traditional signature and anomaly-based detection methods [78].

Due to regular updates made to bot binaries, the security community has a persistent need to monitor network activities and capture and collect bot binaries for analysis on how the bot and botnet operates and the vulnerabilities they exploit [2] and it has been suggested that the use of honeypots are ideal for understanding botnet technology and characteristics [49].

3.0.5 Challenges in Botnet Infiltration and Takedown

Botnets have become a legitimate threat to Internet security in the last decade [2] and continue to evade many of the best security practices. They make a potent blend of previous threats that attempts to leverage their individual strengths [4]. Botnets are notoriously resistant to take-downs attempts as demonstrated by the repeated appearance of the Kelihos botnet after several take-down efforts [56] and the rise of the Gameover Zeus botnet following a large multi-national effort to eliminate it [43]. Botnets are also incredibly pervasive, previous studies have shown that as many as 40% of the computers connected to the Internet are infected by a bot and controlled by a botnet [25].

3.1 Honeypots

Discussed frequently throughout literature is the importance of the role honeypots play in the ongoing struggle to understand and defeat botnets [83][77][58][41][15][80][68]. A honeypot is typically defined as a system (or group of systems) that is designed to pose as a legitimate server, but has no production value; instead, it monitors and logs all interactions it has with outside entities. As no legitimate traffic is intended to be run on the honeypot and the honeypot is not advertised to normal users, any traffic directed toward the honeypot appears suspect; this allows administrators to monitor attack behavior for logging as well as capturing malicious binaries for later analysis without having to combat the overwhelming quantity of legitimate traffic logs [68].

The functionality of honeypots spans a wide range of capabilities. On one end of the spectrum, high-interaction honeypots are fully functional servers and systems running in virtual environments in order to provide containment and isolation from production systems as well

Low-Interaction	**Medium-Interaction**
Honeyd [67]	*Kippo* [19]
Artillery [76]	Glastopf [71]
Thug [18]	*Dionaea* [70]

Table 3.1: Listing several popular open-source honeypots and their interaction category.

as aid monitoring. From the VM's hypervisor, any system changes performed by malicious code can be monitored and logged. As these are fully operational systems, they most closely emulate the behavior of an actual production system and provide the greatest amount of deception. However, simply because they are fully operational systems, they carry a set of unique challenges. First, the time required to set up and deploy a fully operational system, as well as the time requirements involved with maintaining it, may become overly burdensome [6]. Second, as this is a fully operational system, high interaction honeypots can potentially be commandeered and used to participate in malware distribution or attacks themselves, lending both moral concerns and administrator legal liability [95].

Given this list of valid concerns, a second category of honeypots has been developed known as low and medium-interaction honeypots. Considered a "classic" paper in the honeypot domain, in 2004, Neils Provos developed a new type of honeypot called *honeyd* [68] where portions of a network are *emulated* rather than virtualized. This introduced a host of advantages over high interaction honeypots, such as rapid establishment of entire sub-networks of servers and the reduction in hardware investment. The trade-off, however, was a loss in accuracy in honeypot behavior. Due to the systems and interfaces being emulated, their behavior consisted of rapid approximation of how the interfaces would respond to network traffic rather than actual processing of the network traffic according to the specifics of the advertised server. Within specific security goals, these approximations can be "good enough" to assess network threats by monitoring the attacker's initial behavior. However, oftentimes the deception of low/medium-interaction honeypots doesn't last beyond the initial network traffic exchange.

Following the success of Neils Provos' paper, a budding community of open-source low and medium interaction honeypots arose, with contributions from both the security research community as well as the open-source community; the Honeynet Project being one of the foremost groups promoting opensource honeypots (*https://www.honeynet.org*).

To provide a general assessment of the capabilities of low/medium-interaction honeypots, Table 3.1 lists several popular open-source honeypots that are used by both security professionals and amateurs in order to monitor cybersecurity threats on their networks. The list is divided into low and medium interaction categories based on the response and interaction behavior provided by the honeypot with the potential attacker. As stated earlier, low interaction honeypots limit their responses to basic or templated interactions and tend to not respond further than one or two interactions with their attacker. Medium interaction honeypots are capable of more varied responses than low interaction honeypots and are usually

designed to maintain the deception for longer periods of time. Medium interaction honeypots provide more realistic responses and strive to maintain a greater depth of deception while still retaining an emulation of the advertised service. A summary of the honeypots listed in Table 3.1 follows.

As discussed earlier, *Honeyd* marked a shift in honeypot design that allowed defenders to emulate sections of a network with a topography specific to the defender's configuration [68][67]. Being only an emulation of a network section, the hardware requirement to set up such a honeypot was drastically reduced as well as the time involved with establishing and configuring it. *Honeyd*, however, was designed to provide deception only for an attacker's preliminary reconnaissance and does not present specific hosts or services that an attacker may have in-depth interaction with. However, *Honeyd* did establish a shift in honeypot design philosophy and trigger the development of future low and medium interaction honeypots.

Artillery is another example of a low interaction honeypot and, while being rudimentary in it's design, is still used frequently in the security community. *Artillery* is meant primarily to present open ports on a system and log the IP addresses of anyone that attempts to connect to one of it's open ports [76]. This allows network and system administrators to be aware of any entities that may be attempting to port-scan their network. Additionally, due to the assumption that anyone who interfaces with the honeypot is not a legitimate user, the default behavior of *Artillery* is to blacklist the IP addresses it records in order to prevent any possible follow-on attacks that the port-scan was preceding.

Thug is a low interaction honeypot that attempts to identify malicious behavior using an approach different from standard honeypots. Rather than being established as a system or server for attackers to connect to, a *Thug* honeypot connects to servers and presents itself as a client with vulnerabilities, monitoring any behavior of the server attempting to exploit those vulnerabilities [18]. This style of honeypot is often referred to as a "honeyclient" [78].

Kippo is a medium interaction honeypot that presents itself as a host with SSH available. What defines *Kippo* as medium interaction is that it not only logs brute-force login attempts, but also presents a full shell with an emulated file system to the attacker once it has been penetrated for an attacker to interface with [19]. This allows administrators to monitor the follow-on actions performed by the attacker once they have gained access to a system (to include storing any binaries that are uploaded) and grant a greater understanding of the full attack cycle.

Glastopf is a medium interaction honeypot that acts as a web server while advertising vulnerabilities in an effort to lure attackers. The method that vulnerabilities are advertised is through maintaining a file of specific text strings that is indexed by Google's web-crawler service. The text file contains strings that are queried by attackers using Google dorks to search for servers that may contain targeted vulnerabilities. Such text strings can contain version numbers for specific services or applets that attackers may be querying for. Once an attacker finds a *Glastopf* honeypot, they can than attempt to attack it using specifically

crafted attack URLs. *Glastopf* attempts to provide accurate responses based on a template engine in order to prompt additional responses from the attacker. All exchanges are logged and any binaries that are referenced in attack URLs are downloaded and stored for later analysis [71]. Additional details about the *Glastopf* honeypot are discussed in section 3.1.1.

Dionaea is a popularly employed honeypot due to its generic behavior. A *Dionaea* honeypot presents a number of different network protocols that an attacker may interact with and attempt to exploit in order to force an exploitation payload upload. Depending on the payload presented, *Dionaea* will perform different operations in order to receive the final attack binary. For example, payloads containing shell commands will be parsed for download URLs and attempt to download the attack binary for later analysis [70]. The number and popularity of services *Dionaea* emulates has helped promote it's frequent use within the security community.

3.1.1 Case Study on *Glastopf* honeypot

In order to validate the value of honeypots running on low-performance machines for this thesis, the honeypot *Glastopf* was configured and operated on a Raspberry Pi 3 for 43 days, from June 29th, 2016 to August 11th, 2016. As discussed earlier in section 3.1, *Glastopf* is a unique honeypot that not only emulates a webserver for the purpose of recording malicious URL strings that are attempted by attackers, but also attracts malicious behavior by manipulating the index content collected by Google's web crawlers in order to contain text strings that are normally present on systems with targeted vulnerabilities. This is intended to attract attacker that use Google search criteria referred to as 'Google Dorks' in order to discover web servers with specific vulnerabilities. *Glastopf* was chosen for the case study due to its clever use of Google dorks, the capabilities that are offered by the honeypot, as well as it's popularity within the open-source honeypot community.

Glastopf has the ability to automatically classify the following types of HTTP-based attacks made through the 'HTTPRequest' interface:

- Remote file inclusion - HTTPRequests that attempt to include a remote file
- Local file inclusion - HTTPRequests that attempt to include a local file
- phpmyadmin - HTTPRequests that attempt to access phpmyadmin paths
- SQL Injection - HTTPRequests that attempt a generic SQL injection
- Login - HTTPRequests that attempt a generic login string
- phpinfo requests - HTTPRequests that attempt to find a phpinfo debug test page

During the 43 days of data collection, there were over 15,000 individual attacks recorded against the *Glastopf* honeypot; a clear indication that HTTP-based attacks are a non-trivial security threat.

An initial review of the collected data can provide a view of the HTTP-based attack landscape revealed by the *Glastopf* honeypot. During the 43 days of operation, *Glastopf* recorded 15,593 attacks from 55 different countries around the world. While *Glastopf* records any interaction with honeypot as an attack, whether the interaction contained a malicious URL or not, 139 (less than 1% of the total recorded attacks) were basic accesses to the honeypot web page, with the remaining interactions containing non-standard HTTPRequests. The attacks originated from 448 unique IP addresses with 177 of them being repeat offenders (IP addresses that performed more than one attack) who accounted for over 98% of the attacks. Of the repeat offenders, 29 IPs conducted 100 or more attacks against the honeypot, accounting for over 90% of the repeat offender attacks. Attack traffic from repeat offenders originated from 10 different countries with the single highest repeat offender coming from the United States with 4,842 attacks and the second highest coming from Switzerland with 1,215 attacks.

A representation of attack origins can be reviewed in Figure 3.2. Each dot illustrates all of the countries where an attack originated from (with the exception of the United States, which received two dots due to the geographical diversity of it's attacks; the vast majority of attacks originated from the state of California while several other attacks originated from the states of New York and New Jersey). The top 10 countries of origin for attacks against the *Glastopf* honeypot were:

1. United States - 11,302 attacks
2. Switzerland - 1,255 attacks
3. Portugal - 574 attacks
4. Germany - 542 attacks
5. France - 485 attacks
6. United Kingdom - 269 attacks
7. Iraq - 250 attacks
8. Canada - 223 attacks
9. Kazakhstan - 126 attacks
10. Russian Federation - 120 attacks

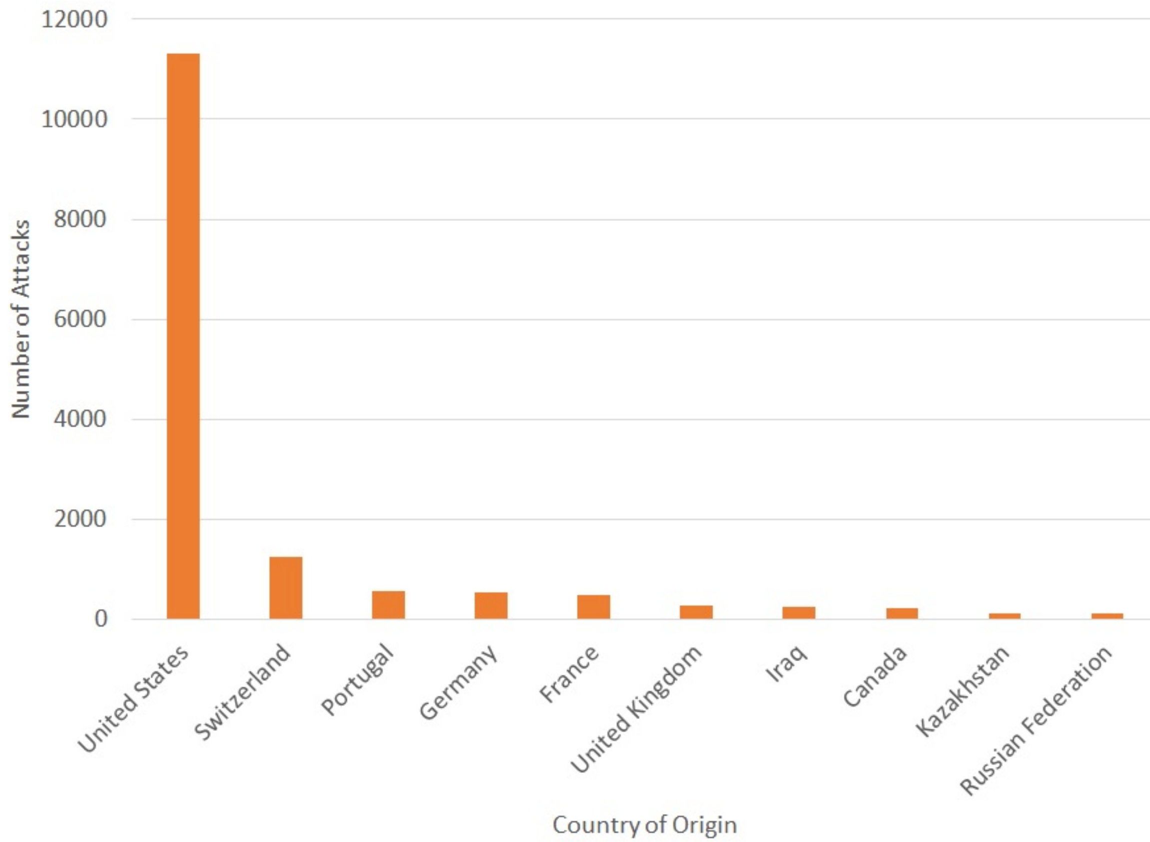

Figure 3.1: Illustrating the number of attacks from the top 10 attack origin countries recorded during the case study.

As shown in Figure 3.1, attacks originating from the United States significantly outnumbered attacks from other countries, with nearly three times the number of attacks than attacks from the remaining 9 countries combined.

While the *Glastopf* logs show where an attack originated from, it must be kept in mind that this does not necessarily indicate where the *attacker* is based. It is very likely, given the high attack count for repeat offenders, that many of the HTTPRequests are automated attacks being performed by malicious software that may have already infected other systems and are simply attempting to proliferate themselves across the Internet. Reviewing the countries indicated in Figure 3.2, a strong correlation can be identified between origins of attacks against the *Glastopf* honeypot and with Internet connectivity in general at a global level, as shown in Figure 3.3.

Comparing the collected statistics with other sources, [79] analyzed attack traffic released by Akamai Technologies for the last quarter of 2012 and identified over 40% of the world's cyber attack traffic originating from within China. This is in contrast to Figure 3.1 identifying the US as responsible for nearly 75% of attack traffic against the *Glastopf* honeypot. While the two datasets clearly are not a direct comparison, Akamai Technologies' review covered all Internet attack traffic and the *Glastopf* honeypot is logging only HTTP-based attacks, some similarities are expected. A reasonable explanation would be the previously mentioned scenario, that the attack traffic logged by *Glastopf* represented the systems that have already been penetrated and are now commandeered into participating follow-on HTTP attacks.

Reviewing the daily activity of the *Glastopf* honeypot, Figure 3.4 shows minimal interaction with the honeypot until day 21 where attack traffic increases significantly and was sustained until the honeypot was taken down on day 43. This jump in attack traffic was due to the honeypot being manually entered into Google's search index the day prior and becoming discoverable through the use of Google dorks. As approximately 99% of the attack traffic arrived after the honeypot was included in Google's search index, it is clear that Google dorks are being used by attackers as a means to identify vulnerable web servers. It is assumed that because the honeypot was not given a registered domain that this resulted in the length of time that passed without Google's web crawling service discovering the honeypot on it's own and requiring that the honeypot be added manually to Google's search index.

Figure 3.5 provides a high-level overview of what attackers were targeting when attempting to access the honeypot. By order of frequency, the top five attempted exploits were:

1. Java servlets
2. Common Gateway Interface (CGI) scripts
3. Shared libraries in the system '/include' directory
4. Contents of the web server '/view/' directory
5. Various scripts contained in the server '/scripts/' directory

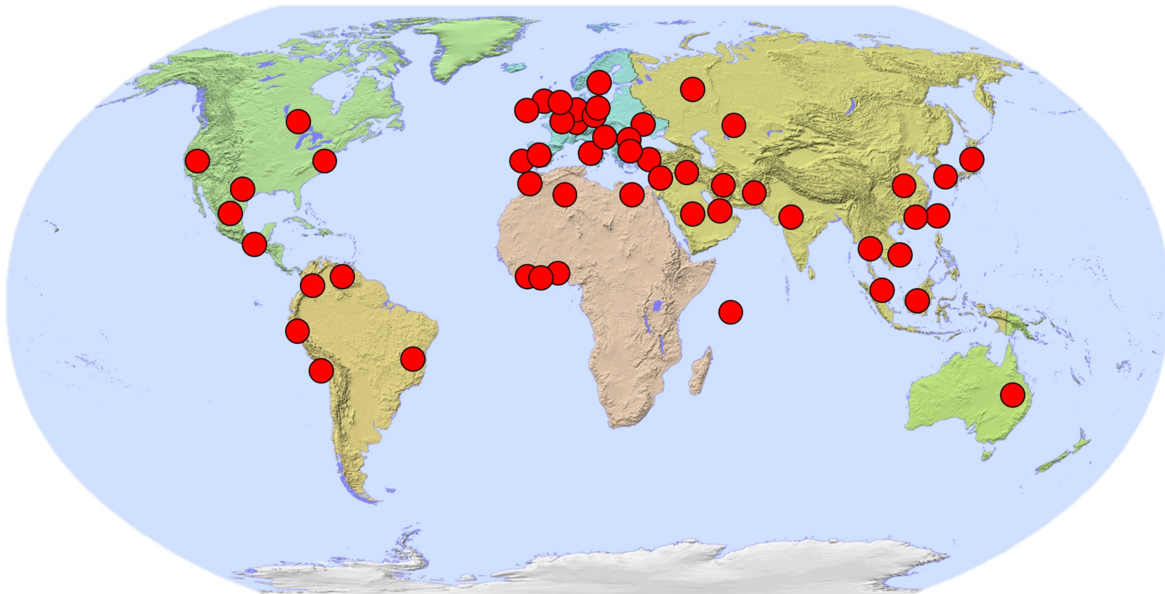

Figure 3.2: World map illustrating the countries of origin for attacks on *Glastopf* web-server honeypot.

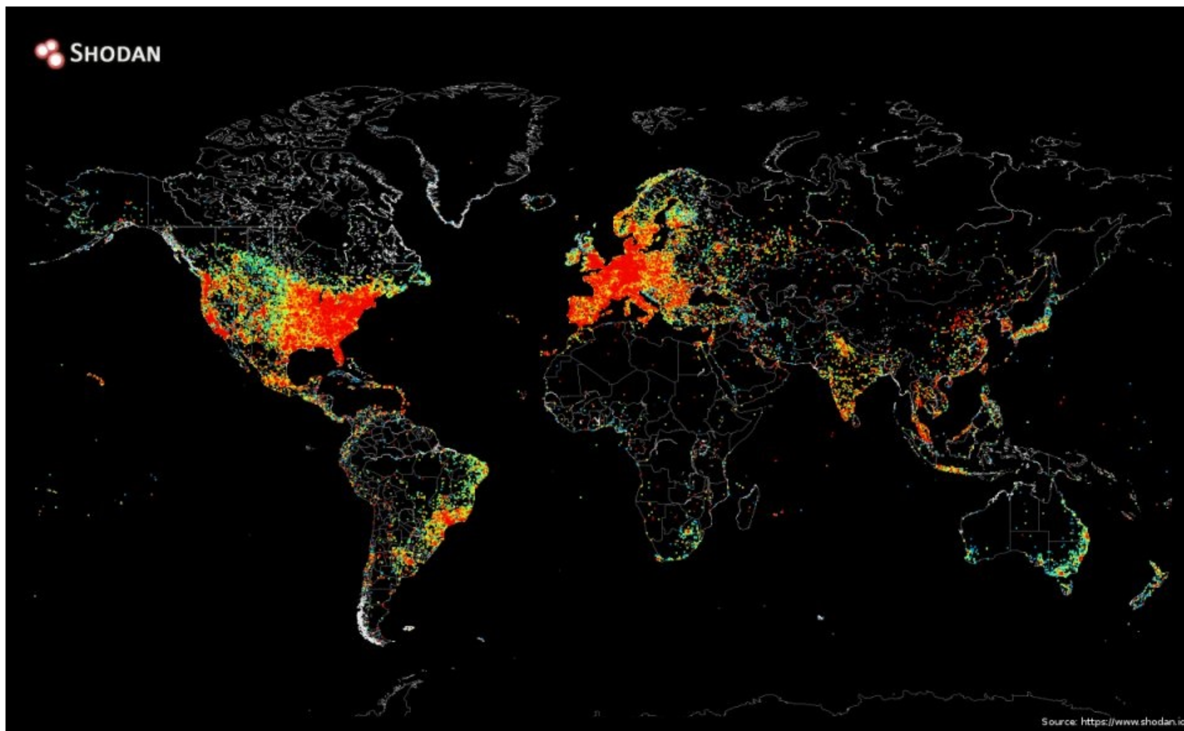

Figure 3.3: World map illustrating the Internet connectivity density [22].

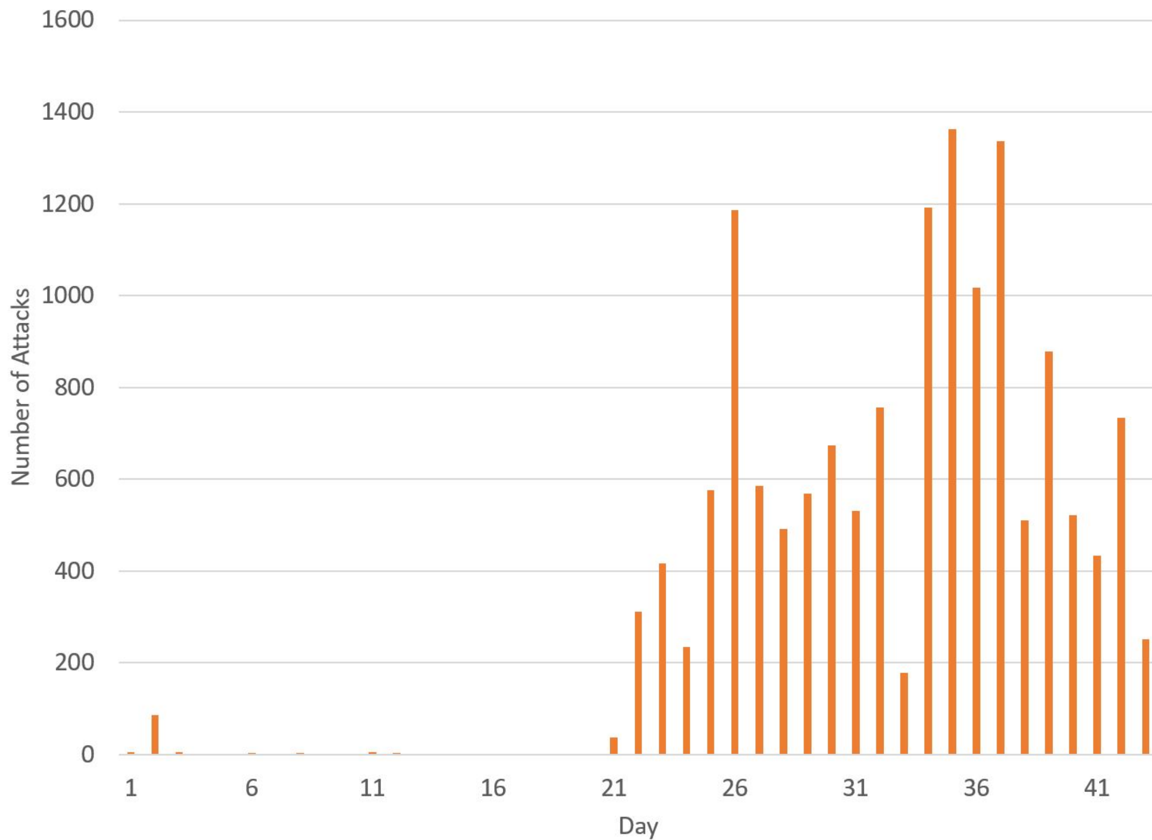

Figure 3.4: Illustrating the frequency of attacks on the *Glastopf* web-server honeypot. The day value is the number of days the honeypot was online for.

Java servlets are a method of providing dynamic content to websites through the use of Java code snippets being called from a container on the web server, normally in response to a particular type of HTTP request [24]. Exploits of Java servlets is a significant concern as they are used frequently throughout the World Wide Web; if an exploit can be found for a popular Java servlet, than that exploit can be leveraged against a large number of web servers.

Common Gateway Interface (CGI) scripts are another method to provide dynamic content to websites by providing scripts that are run on the server in order to generate content for websites on-demand. CGI defines a method for which a web page can provide inputs to a program or script that executes server-side, then have it's results placed into the web page that is delivered to the client browser [63]. CGI script use is somewhat deprecated in scenarios where efficiency is important due to CGI scripts being executed for every single request; however, they are still used frequently due to their past popularity of granting dynamic web page content and it's simple implementation to provide script execution when performance is not a significant factor.

The remaining exploits target shared libraries: Contents of the /view/ directory on a web server, and server scripts contained in the /scripts/ directory, all may contain commonly used code having security vulnerabilities. By servicing specially crafted HTTPRequests that seek to exploit a vulnerability against a targeted library or program, an attacker may be able to trigger malicious behavior which can range from disclosing unintended, server-side information to allowing remote code execution. This poses a legitimate threat to web servers as each of these tools are used extensively throughout the World Wide Web.

These glimpses into the attempted exploitations made against web servers are particularly useful for network administrators to identify what software is either most vulnerable or most targeted.

In addition to collecting attack logs, *Glastopf* does provide the ability to capture malicious binaries if the malicious URL includes some sort of Remote File Inclusion attack. During the case study, a particularly determined attacker from Chile attempted a Remote File Inclusion attack 88 times through an apparent exploit in a Google Map plugin. This malicious file was downloaded by *Glastopf* and contained for potential future analysis. It should be noted that this attack was not the result of a Google dork query as it happened on the second day of operation and accounts for the only significant amount of attack traffic prior to the honeypot being added to Google's search index.

The observations made above consist of only first-order analysis of the collected data over a relatively short period of time. A deeper review of the data, as well as conducting a longer-term study, can provide additional insight, such as changes in attack strings over time that might indicate large shifts in attacker exploitation focus across the landscape; matching identical attacks made by different attackers, possibly indicating separate, individual bots within the same botnet; or changes in attacks made by repeat offenders that might indicate shifts in exploitation focus for individual attackers.

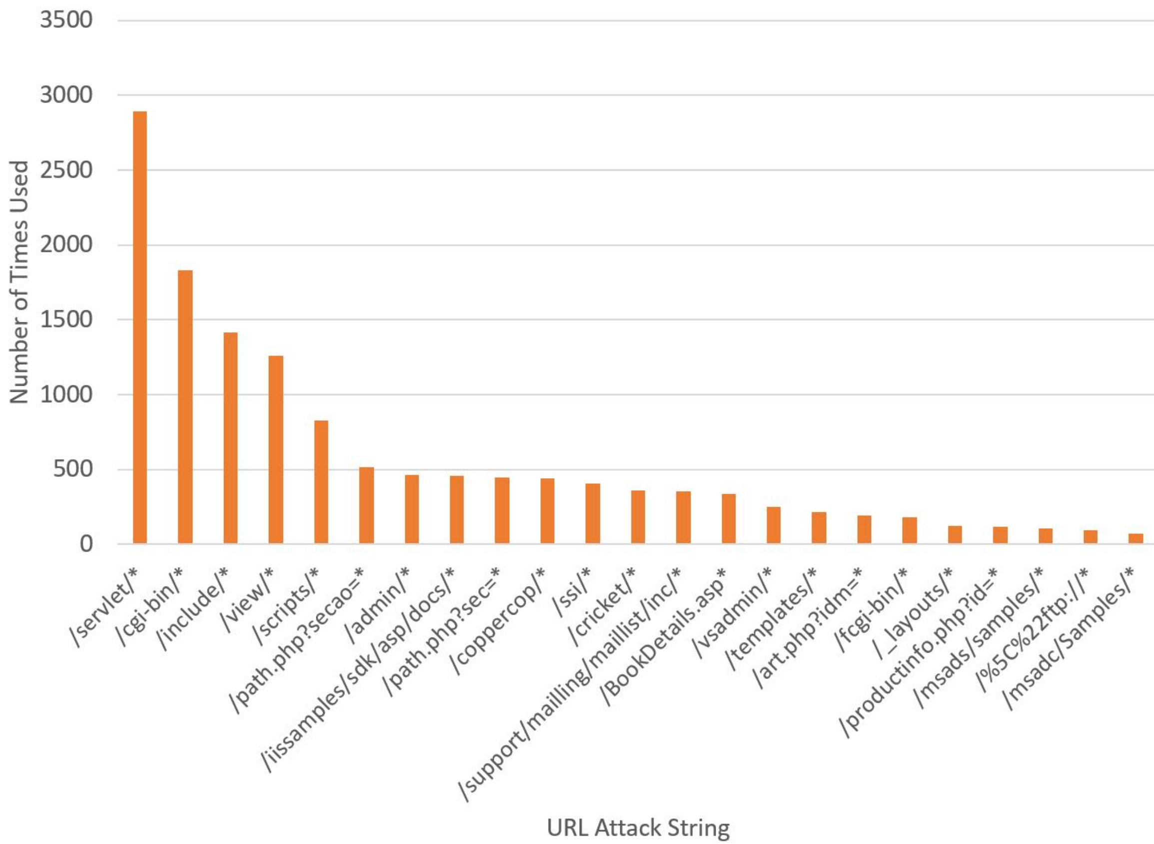

Figure 3.5: Top URL attack strings recorded by the *Glastopf* web-server honeypot and the number of times recorded during the case study

Security researchers from Lewis University performed a similar case study on the *Glastopf* honeypot in late 2015 to early 2016, operating it for approximately 40 days [62]. During their case study, they observed approximately 80,000 individual attack strings against their honeypot from a wide number of countries. The country that originated the most attacks was Indonesia, accounting for 39% of the attack traffic, with the United States having the second highest, accounting for 30%. The remaining countries accounted for 6% or less of the attack traffic.

Comparing the case study performed by Lewis University with the case study performed for this thesis demonstrates several similarities. Both case studies were operated for similar time periods of about 40 days and both kept the *Glastopf* honeypot configuration as default as possible. While the default configurations for this case study were used in order to generate as generic a case study as possible, the default configurations may potentially deter any observant attacker from attempting to exploit the honeypot; *Glastopf* can easily be recognized when using the default configuration by simply browsing to the honeypot's web address and viewing the presented website. *Glastopf's* default configuration generates a website that, while producing randomized text in order to prevent all default *Glastopf* pages from being duplicates, maintains the same layout with nonsensical text strings.

There are several differences, however, in both the set up of *Glastopf* honeypots between the two test cases as well as in the results collected. Lewis University set up it's *Glastopf* instance on an Internet connection being provided by a commercial ISP and registered the honeypot with a commercial domain registrar; the *Glastopf* instance configured for this case study was established on the Virginia Tech network and the honeypot was advertised as a link listed on a university-provided student web page. The differences in setup most likely account for the disparity between the time it took for Lewis University's honeypot to be indexed by Google's web crawler (indexed after being operational for 10 days) and the time that passed before this case study's honeypot was manually entered into Google's search index (manually added on day 20). Both honeypots showed small amounts of attack traffic prior to Google indexing them with the real volume of attack traffic beginning once the indexing was performed.

The Lewis University case study focused on identifying attack characteristics, such as attack frequency and duration, and less on the actual exploit each attack was attempting. Similar analysis can be performed on the data collected during this study if desired, due to the quality of information and statistics collected by *Glastopf*.

It should be noted that it is expected that the attack URL strings recorded by *Glastopf* in this study were not the only attacks being performed against the honeypot. As stated in [62], researchers from Lewis University identified that a significant number of attacks against their *Glastopf* honeypot were SSH based, with 7,143 login attempts performed. However, whether the number of SSH attacks performed against the Lewis University honeypot was influenced by the Google dorks that allow attackers to discover *Glastopf* honeypots, or if that number is representative of the number of SSH penetration attempts made against any

system on their network with an open SSH port was not discussed.

Glastopf is a very effective and clever honeypot, but it does have it's drawbacks. Every one of the recorded strings reflects an attempted penetration into whatever system the attacker believed the honeypot to be. Unfortunately, while *Glastopf* strives to respond appropriately to an attack string in order to fool the attacker into continuing with it's exploitation attempt, the logs collected during the case study don't reveal any follow-up interaction by attackers at all. Granted, *Glastopf* is recording only HTTP interactions and follow-up penetration attacks may be attempted on other protocols; however, it does illustrate the lack of data collection for subsequent steps performed by attackers. This is a hallmark limitation of low/medium-interaction honeypots that counterbalances their ease of implementation and minimal resource requirements.

3.1.2 Low/Medium-Interaction vs High-Interaction Honeypots

Low/medium-interaction honeypots require very minimal resources to operate, as their emulations of networks and protocols are generally lightweight. After reviewing the landscape of popular honeypots, it appeared that all of them are capable of running on a single-board, low-powered device, such as Raspberry Pi 3 or an Intel Atom based system. The code base for these honeypots always remain minimal and storage and system memory requirements are low, with CPU intensity generally low as well, allowing the honeypots to execute on low-powered devices without maximizing CPU utilization. Logging capabilities are built within the application and establishing additional monitoring tools is not required.

The availability of low/medium-interaction honeypots to run on low-cost systems is a particular benefit for security professionals who may need to deploy more than one honeypot in order to provide a more complete view of malicious activity on their network, as external threats never consist of the sole threat to major networks.

High-interaction honeypots typically require a more challenging and time-consuming setup than low/medium-interaction honeypots due to the fact that high-interaction honeypots are complete systems contained within a virtual machine. Such a system first requires the guest operating system to be installed and configured, followed by the installation and configuration of each service that will be employed, as well as the monitoring tools that will be leveraged to log system activities. Once the honeypot has been fully configured and established, the virtual machine hypervisor generally has the ability to store a copy of the virtual machine image that can be reinstated easily in the future, which is of supreme value.

The greatest limitation to high-interaction honeypots is the system resource overhead that is incurred, which dictates the minimum hardware requirements for the honeypot to operate at an acceptable performance level. For example, as discussed later in Section 7.1.3, instruction execution time suffered a minimum penalty of an additional 52% execution time with hardware assisted virtualization. This encumbrance may be prohibitively expensive in terms

System Resource Expense

Interaction		*Low*	*High*
	High		High-Interaction Honeypots
	Low	Low/Medium-Interaction Honeypots	

Table 3.2: System Resource Expense vs Honeypot Interaction Level

of system resources to operate a high-interaction honeypot on a low-power device which may be able to run most, if not all, services of a production system natively, but not with the additional resource overhead of running within a virtual machine.

With sufficient and properly configured monitoring tools, high-interaction honeypots provide the best data collection, but are time-consuming to set up and resource intensive to run. Additionally, as will be discussed in Section 4.2, a secondary concern is when malware penetrates executes on a high-interaction honeypot, it may be detect the honeypot as a virtual environment and refuse to proceed with it's attack. These concerns are what is compelling the development of low/medium-interaction honeypots.

The jump in resource intensity between low/medium-interaction honeypots and high-interaction honeypots is rather significant, leaving no middle ground for security professionals between the hardware investment for the greater deception available of high-interaction honeypots and the significantly lower-cost hardware investment needed for low/medium-interaction honeypots at the expense of deception capabilities.

3.1.3 Aggregate Honeypot Servers

In the implementation of low and medium interaction honeypots, there are some considerations that are specific to their usage. Due to the fact that low/medium-interaction honeypots are designed to emulate a single, specific interface, a varied population of these honeypots must be deployed on a single network in order to span monitoring across multiple services and gain a broader picture of the potential threats a network faces. As low and medium interaction honeypots are generally stand-alone systems, deploying a number of them requires constant monitoring of separate logs and activities in multiple locations within the network, which can be a time-consuming task depending on the number of honeypots deployed. In response to this situation, the security community has begun to develop honeypot aggregate servers that operate as back-end logging systems to which honeypots are able to report all activity to. A collection of deployed honeypots, combined with a central server for collecting logs, is commonly referred to as a *honeynet* [94][25][66][16]. The aggregation server is able

to compile the logs from all communicating honeypots and develop combined threat reports that are retrieved from a single location, potentially generating a significant savings in time over retrieving each report individually as well as packaging the reports in a manner that greatly aids the production of real-time threat analysis. Additionally, this combined pool of data is available for historical analysis of attack trends and provide relevant, ongoing threat assessments.

Due to the unique requirements of the security domain, a new communication protocol named *hpfeeds* was developed by Mark Schloesser specifically for the purpose of honeypots reporting to an aggregation server [33]. Originally written for Python and C, the *hpfeeds* community has expanded its support to a number of other languages such as Ruby, Go, and JavaScript in order to be incorporated into a greater number of honeypots. *Hpfeeds* is set up as a lightweight protocol with a publish-subscribe model with support for the delivery of binary payloads along with authentication being provided through the use of authentication keys. The protocol has also been tested for use with SSL/TLS for confidentiality. This makes *hpfeeds* a simple, purpose-built protocol that is ideal for use with honeypots with plaintext logging and captured malicious binaries.

The development of *honeynets* poises network security for the use of multiple low/medium-interaction honeypots deployed throughout a network as "sensors" that tie into a central, aggregate server for potentially unparalleled visibility of ongoing cyber-threats. Discussions have begun in academic research circles in the last few years exploring ideas on an ideal framework for deploying multiple honeypots either in a single network or over a large geographical location [66][16].

Modern Honey Network (MHN) is an open-source honeypot aggregate server developed and supported by Anomali and published on GitHub.com. A free and open-source platform, MHN supports the automated deployment of low/medium-interaction honeypots and the collection of their logs through *hpfeeds* which are viewable through a web-based, threat assessment front end. MHN was used during the case study of the *Glastopf* honeypot, discussed in Section 3.1.1, for both honeypot deployment and log collection.

The deployment of Glastopf from MHN was designed to be streamlined and near-fully autonomous. However, a number of script corrections needed to be made in order for Glastopf to successfully deploy on the Raspberry Pi due to differences in available libraries through Ubuntu's APT packet manager. Additionally, the Glastopf honeypot itself suffered from an error in it's code that resulted in corrupted Remote-File Inclusion attack reports. A very minor re-ordering of source code was able to resolve the issue.

The necessary script and source code modifications highlight the downside to the current state of honeypot aggregate servers: Aggregate server support is performed independently from honeypot developers, so there is an inherent likelihood of integration issues without a standardized interface specification between honeypots and aggregate servers. Using *hpfeeds* greatly aids in the passing of logs, which is the primary focus of *hpfeeds*, but ancillary concerns such as honeypot deployment or additional reporting features tend to break as

honeypot and aggregate server code is updated independently over time.

Additionally, a lack of standardized interface specifications between honeypots and aggregate servers limits the amount of honeypot configurations that can be performed by the aggregate server without manually coding for each specific honeypot (and possibly different versions of the same honeypot). However, as the use of aggregate servers for honeypot deployment and management gains momentum, it is hoped that a common specification will be adopted that allows for large numbers of varied honeypots to be deployed and managed by aggregate servers, requiring minimal knowledge of the specifics of each of the honeypots by the security administrators.

Additional discussion regarding the refactoring of existing honeypots to support aggregate server interfaces, such as the *hpfeeds* protocol, is included in Section 12.1.

3.2 Rise of the Raspberry Pi and the Single-Board Computer (SBC)

The capabilities and popularity of low-cost, low-power, single-board computers has risen dramatically since the release of the original Raspberry Pi in 2012 by the Raspberry Pi Foundation. The Raspberry Pi was designed for the purpose of promoting an interest in computer programming in children of developing countries [86] and has taken off as a versatile and affordable platform for programming enthusiasts of all ages around the world with 10 million Raspberry Pi's being sold in less than 5 years [85].

The original Raspberry Pi was equipped with at 700 MHz, single-core, 32-bit ARM-based processor and 256 MB of RAM; modest specifications for 2012, but sufficient to run many non-graphical applications. The current edition, the Raspberry Pi 3, has a 1.2 GHz, quad-core, 64-bit ARM-based processor and 1 GB of RAM; along with a host of additional improvements over the first edition, the Raspberry Pi 3 is a potent, bare-bones computer that retails for $35(USD). In addition, the Raspberry Pi Foundation introduced a new computer in 2015, called the Raspberry Pi Zero, which has similar specifications to the original Raspberry Pi, but retails for only $5. During the initial release, consumer interest in Raspberry Pi Zero has significantly outstripped manufacturer supply and quantities continued to remain limited one year later.

Use of Raspberry Pis as the hardware base for low/medium-interaction honeypots have risen in the last few years and it's application has begun to present itself in academic literature [16][87][48][55].

With the rise in popularity of the Raspberry Pi series of low-cost systems, other manufacturers are attempting to capitalize on it's success with their own variant of low-power single-board computers such as the NanoPi 2 Fire [59], the PixiePro [11], and most recently, the Asus Tinker Board [88]. Each of these boards attempt to offer different variants on the

ARM-based architecture offered by the Raspberry Pi with price points ranging from slightly less than the Raspberry Pi to nearly twice the price. Clearly the interest and demand for low-cost, low-power systems is significant.

Currently, there does not appear to be as many low-cost single-board devices offerings for the x86/x86-64 architecture as Intel has had slow gains in the low-power device market. Several of the x86-64 SBCs on the market are the Intel Atom-based Jaguarboard [53], the Intel Celeron/Pentium based UP-squared board [9], the Intel Celeron/Pentium/Atom based Udoo X86 [38], and the Intel Atom-based Minnowboard Turbot [23], with prices ranging from $80 - $259 (USD), considerably higher than the price of the Raspberry Pi.

Low power devices that support x86 compiled software have an advantage in the fact that most production systems that honeypots are designed to emulate are native to the x86 architecture. However, with the ability of many ARM-based low-powered devices being powerful enough to run many terminal-based Linux operating systems, combined with the recent surge in popularity of ARM-based systems prompting many services to be compiled for the ARM architecture, coupled with the lower cost of ARM single-board computers, x86-based systems loose many of it's benefits. Additionally, ARM-based servers are beginning to make an appearance in business networks and have begun to challenge an Intel dominated market [12][7], most likely promoting even more business network services to support ARM natively.

Combining the low-cost versatility afforded by low-power single-board computers with the emerging use of honeypot sensors in a *honeynet*, security researchers and administrators are given a data collection tool that has not previously existed - the ability to deploy cheap sensors throughout one or more networks to collect data on network penetration attempts and new exploits as they are used in the wild. The only limitation in such a *honeynet* is the reduced deception afforded by low/medium-interaction honeypots as the overhead imposed by virtual machines for high-interaction honeypots on single-board computers is prohibitive; most single-board computer processors are designed for low-power usage and are not equipped with hardware virtualization extensions to reduce the processing overhead.

Chapter 4

The Problem

4.1 Vital Role of Virtual Machines

Virtual machines provide an ideal environment for security researchers to analyze and study botnet behavior and code [5]. In terms of the discussion on honeypots, high-interaction honeypots are deployed within virtual environments for a number of reasons [80] [41] [15], but mainly for the purposes of containment and fast image restoration. This allows security researchers and administrators to monitor the attacker behavior throughout the entirety of the attack cycle in a contained environment and, once the attack and all data collection is finished, the honeypot can be quickly reset back to it's original state, ready to interact with the next attacker. Deception is able to be maintained indefinitely as the actual services and applications under attack are being run with no emulation or approximation that may alert an attacker that they are not interacting with a production system.

In addition to their use in high-interaction honeypots, virtual machines also provide a sandboxed environment for researchers to observe the behavior of any captured malicious binaries that require analysis. These binaries are free to behave as their author intended while being limited to within the virtual environment and its actions and activities monitored for analysis. These sandbox environments provide researchers the convenience of containment and quick machine restoration (by easily restoring a ready-to-execute machine image), as well as being able to maintain a cache of images configured to specific requirements. The benefit provided by virtual machines is significant to researchers, as managing the same set of tasks for bare metal machines can easily become overly burdensome to even the largest security research team.

Unfortunately, the ideal application of virtual machines for high-interaction honeypots and malicious binary analysis has not gone unnoticed by malware authors. These authors make great efforts to thwart analysis and study of their binaries by preventing the execution of malicious code in virtual environments [5]. There are a number of methods that botnet

authors have to detect such environments, which has already been studied at length in literature [27] [42] [69].

4.2 VM Aware Malware

In order to discuss how malware might detect the presence of a virtual environment, a brief discussion on virtual machines is necessary.

A virtual machine, in the basic sense, is a computer hardware configuration that doesn't exist physically. The virtual hardware components that makes up the virtual machine are either actual hardware components of the host system, managed by software or virtual hardware components that are emulated by software. The virtual machine is managed by the hypervisor (for this thesis, the term hypervisor also includes Virtual Machine Manager), that controls the relationship between the virtual machine and the actual hardware platform it is executing on and can manage multiple virtual machines simultaneously.

Traditionally, virtual machines are divided into Type I and Type II which varies with the location where the hypervisor is running. A Type I virtual machine has the hypervisor boot directly on the hardware and has the highest level of hardware access privilege. Any OS that will interact with users or execute programs are 'guests' that are virtual machines managed by the hypervisor. Type II virtual machines start with a 'host' OS that has the highest level of hardware access privilege, then runs the hypervisor and any subsequent virtual machines as programs. This is opposed to a non-virtual machine that simply has the OS execute on the system hardware. See Figure 4.1 for a graphical representation of where the hypervisor operates between the two virtual machine types. It should be noted that when the hardware CPU natively supports virtualization, these access privileges are modified somewhat in order to grant hypervisors special privileges, but further details are outside the scope of this discussion.

Fully emulated virtual environments are virtual machines in which all of the virtual system's hardware is implemented in software on the host machine. This type of virtual machine is the most flexible, as every aspect of the hardware can be tuned/changed in software; however, this type of virtual machine is also the slowest technique, as all operations performed by the virtual machine are done in software on emulated hardware or otherwise binary translated from the 'guest' architecture to match the hardware of the host system [64]. Popular virtual environments that provide fully emulated environments are Bochs and QEMU.

Full virtualization are virtual environments that match the host machine's hardware and do not need to perform hardware operations in software except privileged CPU instructions that can only be executed by the 'host' operating system that is interfacing directly with the system hardware. Any privileged instructions that the virtual environment attempts to perform are intercepted by the hypervisor and are either binary translated to non-privileged instructions before executing them or the hypervisor has the 'host' operating system perform

those instructions on the virtual environment's behalf. There is a considerable performance benefit over fully emulated virtual environments, but the hardware and architecture restrictions may not be feasible for all applications. This type of virtualization is also referred to as a Hybrid Virtual Machine [64].

Paravirtualized environments are virtual environments that are very similar to full virtualization except for the fact that the 'guest' operating system has been modified to not call any privileged instructions, instead interfacing with the hypervisor to perform the necessary operations. This provides additional performance benefits over full virtualization as the hypervisor does not need to trap and translate instructions, but it does require modifications to the 'guest' operating system [64]. Some modern operating systems, such as Windows and Ubuntu, provide built-in paravirtualization when it detects it is running in a virtual environment (through advertisement of hypervisor interfaces), which aids in the implementation of paravirtualization.

Wide-spread use of hardware assisted virtual environments is a relatively recent development with the introduction of Intel's VT-x processor extensions in 2005 and AMD's AMD-V processor extensions in 2006. VT-x and AMD-v provide privilege modes for virtual environments that support additional, hypervisor specific instructions to avoid the delay of trapping and translating privileged instructions that exists in full virtualization, but without the additional modifications of the guest operating system that exist in paravirtualization [64]. Hardware assisted virtualization provides the greatest performance, expecting to operate at near-native speeds, but requires both hardware support from the CPU as well as a hypervisor that utilizes the additional CPU instructions. In 2010, ARM announced extensions to their ARMv7-A architecture that supports hardware assisted virtualization [10].

Another relatively recent development for virtualization is what is being referred to as operating system level virtualization. The main difference between traditional virtualization and operating system level virtualization is that traditional virtual machines all run their own operating systems and compete for hardware resources at the hypervisor level (which may be competing for resources with the host OS, in the case of Type II hypervisors); instead, operating system level virtual machines all share the kernel of the host OS with isolation being managed by the OS through a variety of techniques [74]. This virtualization allows additional performance benefits over traditional virtual machine techniques by reducing the need to run distinct operating systems for each virtual environment and share resources at the OS level at the expense of restricted configurability (OS level virtualization is restricted to the configuration of the system hardware and kernel) and decreased security (a single environment crashing the kernel takes the entire system down).

Malware that tests for the presence of a virtual environment use a variety of methods to detect changes in the environment that have been introduced by the hypervisor. Some of these methods include detectable differences between emulated and actual CPUs, such as undocumented CPU behavior or lack of CPU registers that are not present in an emulated CPU. Other methods simply check the running environment or file system for artifacts that

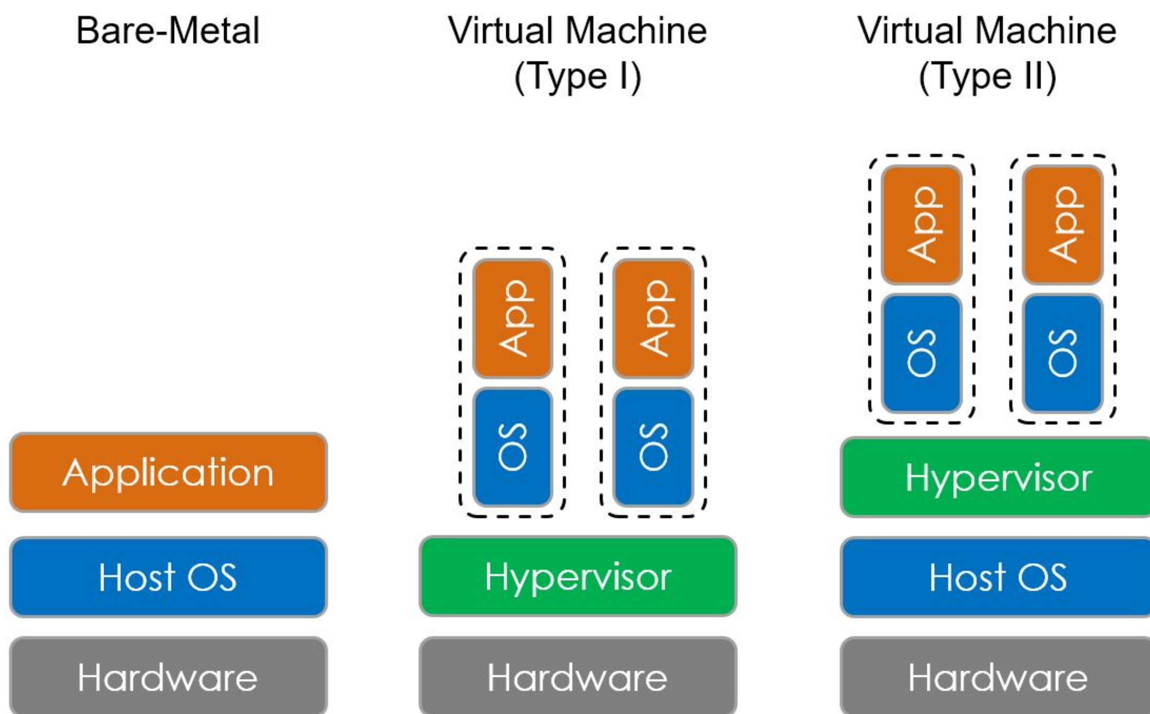

Figure 4.1: Hypervisor placement in Type I and Type II virtual machines compared with bare metal

indicate hypervisor software is either installed or running. Lastly, a very effective virtual environment detection method seeks to exploit the processing overhead of virtual machines that is such a barrier to low-powered devices using virtual environments by detecting the additional delay in execution that occurs when operating within a virtual environment [5].

To be fair, not every virtual environment testing method uses technologically clever or sophisticated techniques to identify if it's on a research system; malware was recently discovered to check for the presence of a requisite number of Microsoft Word documents. If the system lacks enough Word documents, the malware assumes it is currently in a testing environment and ceases it's malicious behavior [75].

Accounts have been made in 2008, that up to 40% of the malware in the wild altered their malicious behavior when being executed in a virtual environment or a debugger [14]. If this percentage is even loosely representative of the percentage of VM-aware malware in the wild today, it poses a significant concern to the security research community that much of a binary's malicious behavior may go unnoticed.

Unfortunate for malware authors (but fortunate for the security research community), there has yet to be a single, user space test that can reliability detect the presence of a virtual environment in all scenarios, a test that has been coined 'red pill' after security researcher Joanna Rutkowska published a method in 2004 that used several lines of code to detect Windows running in a virtual environment [72]. The 'red pill' test queried the location of the interrupt descriptor table register (IDTR) to determine if it was being stored in a non-standard memory location, indicating the current operating system is not the host OS. However, this test is no longer effective as the location of the IDTR is no longer predetermined for multi-core processors [27]. Similar developments in both CPU architectures and virtual environment implementations have prompted an every-changing landscape in virtual machine detection.

The difficulty in devising a general virtual environment detection method is due to the varied types of virtual environments and the different methods that are used to implement them. Type I and Type II hypervisors operate differently because of where the hypervisor resides in relationship to the hardware. Fully emulated virtual environments will behave differently than paravirtualized environments which will behave differently than hardware assisted virtual environments. These different approaches vary considerably on how they implement an ephemeral hardware environment which leaves little common ground between all implementations, thus it becomes a difficult problem to solve in order to detect them all with a generic method.

Despite there no longer being a 'red pill' that can be used for general virtual environment detection, malware authors have shown a high level of ingenuity in the variety of methods employed and there is no shortage of academic writings on the topic [27] [69] [29] [14]. Many of the environment detection methods discussed in literature focus on an inherent level of discoverability of many virtual environment implementations. In the case of full hardware emulation, attackers seek to exploit less-than-perfect emulation by executing CPU

instructions with undocumented side-effects that are often not included in the emulated CPU; other environment detection methods seek to have a hypervisor reveal itself by triggering a hypervisor event and detecting side-effects, such as cache flushes; malware may also seek to identify any communication that may be occurring between the guest OS, the hypervisor, and the host OS [27]. Capturing malware in the wild to analyze the virtual environment detection methods currently being employed is a non-trivial task and was beyond the scope of this thesis.

4.3 Related Work in Deception and VM Aware Malware

In 2005, researchers from the University of California, San Diego saw the beneficial role that honeypots play in network security, but were concerned with their scalability as the number of honeypots deployed grew. Vrable, et al., proposed a system call *Potemkin* [89] that addressed the inevitable trade-off between low-fidelity monitoring performed by low-interaction honeypots at a low resource cost and the high-fidelity monitoring that can be performed by high-interaction honeypots at a high resource cost. *Potemkin* addresses this trade-off by generating a virtual network consisting of thousands of ephemeral honeypot 'clients'. When a potential attacker interfaces with one of these clients, a virtual machine is quickly spun-up that provides the high level of deception of a high-interaction honeypot for the duration of the attack. Once the attack is completed, the virtual machine is terminated and it's resources are freed. The authors discuss the concern of VM-aware malware and ultimately, due to the persistent challenges and solutions required to defeat all possible environment checks, left it unaddressed.

A recent work leveraging the benefits of honeypots was proposed by Taylor, et al., in [78] in which a system was deployed 'on-the-wire' to detect malware shortly after it was downloaded by any client on the monitored network. Taylor, et al., extended the idea of a 'honeyclient', a security system that trolls the Internet, searching for any compromised server that attempts to infect the system with malware when the honeyclient interacts with it; rather than attempting to individually interface with the innumerable servers on the Internet, Taylor, et al.,'s on-the-wire honeyclient caches all HTTP exchanges clients on the protected network have with external servers. When an exchange is observed that contains potentially risky code or binaries, the on-the-wire honeyclient starts a sandboxed environment with a configuration matching the client that triggered the risky exchange. The HTTP cache is replayed on the sandbox, impersonating both the client and server responses while using the binaries that had been cached as part of the HTTP exchange in order to attempt to detect any malicious behavior performed by the binaries. The execution of the binaries within the sandbox are monitored for specific malicious behaviors, which include halting execution (assumes binaries detected virtual environment and terminated). The tested implementation of the proposed on-the-wire honeyclient resulted in a significantly higher detection rate than signature-based

IDS and caught numerous malware samples that were missed by the pre-existing network security system. However, the authors acknowledged that their implementation does suffer from limitations related to virtual-environment detection which they tried to compensate for with their variety of sandbox flags.

Much of the attention regarding VM-aware malware has focused on malicious binaries already loaded on to a system that, during execution, perform some environment checks in order to determine if it is operating in a virtual environment [27] [42] [69]. These environment checks can range from simply checking the file system for tell-tale indicators such as standard VMWare network adapter drivers, to testing for the presence of model-specific CPU errors that aren't replicated in a simulated system [42]. Probably the most well-known detection mechanism is related to timing, where specific operations are performed and the elapsed time is compared to an expected reference; if the operation took longer than expected or frequent executions of the operation have a wide variance in execution time, a program can assume it is operating in a virtual environment [69]. The reason for the delay is inherent to the virtualization technology, where, at minimum, several CPU instructions are either emulated in software or captured and translated to other instructions, thus creating an additional time overhead [29]. In the case of full system emulation, the CPU is emulated fully in software, which generates a substantial resource overhead. In full virtualization, the resource overhead is significantly reduced due to only a small number of privileged instructions that need to be intercepted and translated, however an overhead still remains. Hardware assisted virtualization reduces this overhead still further, yet it still requires hypervisor events that trigger the CPU to change privilege modes, which still results in a delay. As no technology is currently available to remove this inherent delay in instruction processing, the use of timing to detect virtual environments remains a popular method in virtual environment detection and an open problem to be solved [29].

A mechanism to implement several of the key features of virtual machines on a bare metal system was explored in [42] due to the emerging threat of VM-aware malware. The malware analysis framework outlined by Kirat, et al., allowed a state of the bare metal execution environment (consisting of memory and CPU register contents) to be captured and later restored in order to provide a "fast and rebootless system restore technique". The proposed framework contains many advantages and is anticipated by its authors to overcome all common virtual environment detection tests, however the framework is complicated and requires a modification of the running operating system that will be dependent upon the hardware utilized by the bare metal system. Additionally, in the preceding years following the publishing of the paper in 2011, there has been little effort to commoditize such a system to be made easily accessible to the security research community as a whole.

Attempts have been made to develop fully transparent virtual environments that, unlike traditional virtual environments that are designed for productivity, are designed to be invisible to software that attempt to detect a virtual environment; Cobra and Ether are two often cited examples of this attempt. Unfortunately, both virtual environments had ultimately fallen short of their goal; tests have been devised that reveal their presence either

through inaccurate CPU semantics or through timing tests with verification from outside sources [69] [65]. It has even been argued by Garfinkel, et al., that a fully transparent virtual environment is, in itself, impossible to achieve due to necessary deviations that virtual environment developers must make that are different from the hardware they are emulating [30]. Garfinkel outlines a number of categories in which virtual machines will differ from their target hardware, to include discrepancies with CPU implementation, resource discrepancies, and execution timing differences.

In [5], Balzarotti, et al. argued that rather than attempting to implement an entirely transparent virtual machine (while being possible, will most likely be extremely resource intensive), it is better to simply detect when software changes it's behavior when operating in a virtual environment. The authors implement a two-system behavioral comparison technique in which a binary is executed on a non-virtual reference machine while recording the system calls and parameters used to execute the malware. The same malware is then executing on a virtual machine using the same system calls performed on the reference machine and compare the outgoing system calls made by the binary between the two execution runs. The authors found their technique to be reliable and efficient at detecting what they referred to as a 'split personality', where malware behaves differently depending on the environment it is being run in.

In 2005, Symantec published a journal article discussing the increase in threats against virtual machines [91]. Symantec's article included discussions on VM-aware malware designed to alter their behavior when being analyzed by Symantec's security analysis systems, which rely heavily on virtualization, making it "difficult or impossible for an automated system to come to an accurate conclusion about the malware in a short timeframe." The article outlined a number of techniques their malware analysis team observed for binaries to detect the presence of a virtual environment to include network adapter MAC address checks, Windows registry keys alluding to virtual machine software, and checking for specific process and service names. Additional methods to bypass automated analysis techniques include the incorporation of some type of user interaction, such as waiting until a specific number of left mouse clicks have been performed before attempting malicious activity, or incorporating a delay before attempting malicious behavior that is longer than an automated process has before determining whether the binary's behavior is malicious by nature. Symantec's article ultimately provided no conclusive solution on handling VM-aware malware, merely just commenting on it's existence and discussing some of the behaviors that they have witnessed.

An interesting application of deception in regards to network security defense is the definition of the term 'honey-patching' made in [3]. The authors made the valid observation that many patches to security vulnerabilities reveal their presence by outright rejecting the traffic that was attempting to leverage the exploit. This gives attackers the advantage in reconnaissance by allowing them to quickly identify which systems were patched and which were not. Honey-patching is a term describing the process of patching a vulnerability, but instead of outright rejecting malicious traffic, the server process that receives it will be live migrated to an isolated environment in which the attacker can continue with their malicious behavior for

monitoring or observation, away from production services.

Older work in the area of security deception includes the practice of using 'honeyfiles', outline in [93]. Honeyfiles consist of special files that are not intended to be used by standard network users, but are designed to appear enticing to a network intruder. Whenever that file is accessed, details about the intrusion are recorded and an alert is made. The intent for honeyfiles is to be a means of validating a network's current security system; alerting security administrators to the presence of a possibly undetected intruder.

Chapter 5

Our Contribution

In Chapter 3, the value of honeypots was discussed, to include the preference towards high-interaction honeypots for prolonged deception and how those honeypots traditionally rely on virtual environments in order to provide isolation during an attack as well as quick restoration once an attack was completed. Chapter 4 discussed the arrival of VM-aware malware that is being written by authors in order to avoid/defeat analysis of their malicious binaries and the challenges presented in trying to create a virtual environment that is not susceptible to environment detection.

In Section 4.2, the idea of operating system level virtualization was introduced in which programs could run on a system in isolation from other programs and the sharing of system resources is managed by the operating system kernel rather than a traditional hypervisor. The last few years has seen a maturation of this style of software isolation and containment in Linux, being coined under the term *Linux Containers*. Some implementations of containers provide many of the functionalities that are essential for high-interaction honeypots and malware analysis without incurring the performance overhead or specific hardware requirements for efficient implementation of traditional virtual machines (further discussion of containers continues in Section 5.1). Given the security research benefit provided by honeypots in general (and high-interaction honeypots specifically), this recent development opens the possibility of deploying high-interaction honeypots on low-power devices. This is particularly enticing for the deployment of a honeynet in which multiple honeypots act as sensors throughout a large network or collection of networks which normally, due to hardware costs, is relegated to the use of low-cost computers, thus low/medium-interaction honeypots. This thesis proposes that Linux containers employed on low-power devices provide a viable and effective deception-based cybersecurity tool.

5.1 Linux Containers

A good, introductory treatment on Linux containers is given by Wang in [90]. In brief, Linux containers are the product of tying together two Linux technologies: namespaces and cgroups. Both of these technologies provide isolation and containment for one or more processes from the rest of the host in order to abstract applications away from the operating system. The initial development of containers is called Linux Containers (otherwise referred to as LXC), which allows multiple Linux systems to be run on the same host, sharing only the system kernel. Each of these containers 'feels' like it's own entire Linux system and are isolated from both each other and the host system. The more popular implementation of Linux containers, Docker, originated as a derivate of LXC with the goal of developing single-application containers rather than each container housing an entire system. As Docker matured, it began to focus on providing single-process containers, where multi-process applications would be made up of a collection of Docker containers, with each container being stateless and accessing a shared, mounted storage [17]. Although Docker has brought Linux containers into mainstream popularity, it's design focus does not match the security goals of this thesis; LXC (and it's newer management daemon, LXD), delivering full Linux systems in contained environments, better align with the goals of prolonged attacker deception with minimal resource overhead.

A significant component to the reduced overhead of running Linux containers as opposed to traditional VMs is that not only do containers lack the processing overhead incurred by the hypervisor with emulated, full virtualized, paravirtualized, or even hardware assisted virtualization, but there's less processing to do in general due to not needing duplicate kernels starting up and running for each VM in parallel. This generates a large amount of additional work when considering duplicate memory management, duplicate disc I/O, duplicate process management, and more.

Taking a sample from one of the CPU timing experiments demonstrated in Chapter 6 on the system Cindy, LXC added approximately 6% to the execution time of a particular code sample when compared to bare metal execution, where KVM added approximately 41%. While demonstrating the performance difference, this example is not a conclusive analysis on the CPU performance between the two isolation environments as there can be a significant number of influences on execution time between the two tests compared with the bare metal execution. While such an investigation and discussion is not within the scope of this thesis, such a comparison has started to be considered in literature [26].

Due to being abstracted away from the operating system, Linux containers have the added benefit of being managed very similarly to a virtual machine image. Containers can be started and stopped quickly (no kernel boot-up or shutdown required), they can have their state frozen and restored at a later time, and they can be copied to other physical hosts with minimal restrictions. Additional benefits are provided to containers depending on their implementation (ie, LXC vs Docker) and care must be exercised to ensure a selected Linux

container implementation matches the design goals of a specific project.

5.1.1 Linux Container Security Assumptions

The reason why Linux containers may be an effective alternative to virtual environments for isolation and containment of high-interaction honeypots is in being reasonably assured they are a 'safe-enough' platform. As describe by Hayden, et al. in [36], system administrators don't necessarily need to sacrifice security when implementing Linux containers, however there are some nuances unique to Linux containers that need to be addressed in order to provide reasonable assurance of security.

There are a number of matured mechanisms by which Linux containers can be deemed reasonably secure, chief among them are the employment of AppArmor and SELinux. AppArmor is a Linux security system that allows administrators to place restrictions on what programs can do, what they can access, and the type of access they are permitted. AppArmor was applied to Linux Containers as a means to avoid an attacker from accessing resources on the host system that the container has not been designated access to [34].

SELinux is another Linux access control system that is similar to AppArmor in that it restricts container access from resources it is not authorized to interact with. However, SELinux provides access control through the use of labels for system resources and is designed to implement policies for how various resources are authorized to interact with other resources at a very granular level [36].

Linux namespaces and cgroups, which allow Linux containers to operate by providing the isolation mechanism, also provide their own security measures. Linux namespaces allow a kernel to hide various resources from a container and allow the container to believe it has full access to the system. As an example, namespaces grants UID isolation in which someone logged into an unprivileged container may have root access with a UID of 0 within the container; however, the kernel has actually assigned that user the UID of a normal, unprivileged user. In the event an attacker discovers a means to break out of a container, they will find themselves acting without root privilege.

Linux cgroups allows the system kernel to place restrictions on the amount of resources a container is allowed to consume; this includes CPU, system memory, and I/O resources. Cgroups ensures that no single container is allowed to consume all resources available on the host and thereby perform a DoS attack against other containers running on the system. An example of this would be a cgroup policy that limits the number of processes a container may spawn; by placing this restriction, if limited low enough, cgroups will prevent an attacker from performing a fork bomb (an interested read on using cgroups to prevent a fork bomb in a Linux container can be found here [61]).

5.2 Linux containers as a Viable Alternative to Virtual Environments on Low Performance Machines

When exploring the question whether Linux containers are a viable alternative to using virtual environments on resource restricted machines, such as the Raspberry Pi, for the use of high-interaction honeypots, it must be identified what the requirements are.

The functionality provided by virtual environments that predisposes their use in high-interaction honeypots are:

1. Containment to prevent an attacker from breaking out of the honeypot and onto the production network
2. State restoration that allows administrators to return the honeypot to its original state after an attack has been performed
3. System state monitoring that allows researchers to observe the actions being performed by the attacker

Requirement 1 was discussed in Subsection 5.1.1 and provides reasonable assurances based on matured security mechanisms that are part of the mainline Linux kernel.

Container management discussed in Section 5.1 mentions the portability of Linux containers due to their abstraction away from the host operating system and being managed similarly to those of traditional virtual machines. This allows the state of Linux containers to be maintained and restored efficiently and quickly, oftentimes within seconds, depending on the host system and the container configuration, addressing Requirement 2.

Requirement 3 is simply satisfied by noting that Linux containers operate as a collection of processes that may be isolated and contained from within the container, but are fully visible by the host system. As such, containers monitoring is available by any set of tools that allow the monitoring of processes behavior, the tool STRACE being an example. For monitoring the behavior of VMs, it has been mentioned in [40] and [42] that security researchers have to make a trade-off between out-of-host monitoring (observing the state of the virtual machine from the outside) but at a lower fidelity, and in-host monitoring, which allows greater fidelity in behavior monitoring, but is more prone to discovery (or modification) due to occurring within the same environment as the malware.

By leveraging the lower overhead cost of using Linux containers instead of virtual machines for high-interaction honeypots, it becomes feasible to employ high-interaction honeypots on low-performance machines, such as the Raspberry Pi, opening a new category of high-interaction and low system resource expense, as demonstrated in Table 5.1.

		System Resource Expense	
		Low	*High*
Interaction	*High*	**High-Interaction Honeypots with Containers**	High-Interaction Honeypots
	Low	Low/Medium-Interaction Honeypots	

Table 5.1: System Resource Expense vs Honeypot Interaction Level with Containers

5.3 Experiment Proposal

While Linux containers appear to be technologically capable of replacing virtual environments for the purposes of high-interaction honeypots, in fact, there appears to be some advantages Linux containers may possess over virtual machines, specifically environment detection techniques utilized by VM-aware malware to discover the presence of a virtual environment may potentially be ineffective against container environments. VM-aware malware is a serious security concern as potentially not all malicious behavior is being identified by current commercial security systems. However, due to the variability in techniques utilized by malware authors in an attempt to discover the presence of a virtual environment, this statement requires some investigation. Defeating environment detection in virtual environments remains an open-ended question, but some preliminary discovery work will be performed for Linux containers.

It is proposed that the use of Linux containers will allow security researchers to observe and monitor the desired behavior of VM-Aware malware, while remaining in an isolated and contained environment, by defeating the primary methods that virtual environments are detected by.

Methods by which a binary can identify the presence of a virtual environment have been frequently discussed in academia [14] [45] [5] [69] [27] as well as across the Internet at large. It would be unfeasible to perform a comprehensive analysis on each of the identified methods. Rather, detection methods have been grouped into categories and one or two example detection methods from each category were implemented and tested against an array of system configurations, both to validate the ability of the test to detect a virtual environment as well as test it's detection of a Linux Container. The selected categories of virtual environment detection methods are:

- CPU
- File system

- Hardware attributes
- System Memory Performance
- Network

Unless otherwise noted, this experiment is performed through the lens of a remote attacker who has penetrated a system's initial defenses in order to plant a payload binary that is about to execute.

Chapter 6

The Experiment

As stated previously in Section 5.3, in order to establish the use of Linux Containers as a viable alternative to Virtual Environments for the purposes of malware detection, it first must be shown that Linux Containers are resistant to the environment detection methods that VM-Aware malware perform. An experiment was conducted to perform a number of virtual environment detection tests on both virtual environments and Linux Containers and their results were reviewed and analyzed.

6.1 Design

Virtual environment detection methods can come in a variety of ways, each seeking to identify different features of the executing environment that are abnormal for bare metal systems. The tests performed in this study are intended to sample different aspects about an operating environment and only contain a small portion of the range of tests that can and have been performed when trying to detect a virtual environment. Each explored category has the following tests being performed:

- CPU
  - CPU clock variability
  - CPU information
  - Execution time of privileged instructions
- File system
  - Artifacts in the File System
- Hardware Attributes

    - Artifacts in system hardware attributes

- System Memory Performance
    - Increase in memory latency following hypervisor event
- Network
    - Variability in TCP timestamp clock skew

These tests are meant to either explore information about the system, such as what is present in the file system, what the system reports about its own hardware, or gain information about how the system executes operations. Additionally, these tests are intended to operate with user-level privileges, thereby mimicking operations that can be performed by malware prior to conducting any malicious activities, such as attempting privilege escalation.

As source code for malware environment detection methods were not utilized for experiment test development, these tests are both preliminary and unrefined. They are testing both the validity of the suggested virtual environment detection methods discussed in literature as well as observing the test behavior on bare metal systems, various virtual environments, and within Linux containers. These tests are performed with default OS installations and with minimal isolation efforts; while this approximates the vulnerability of a basic system, it does not allow for throughout scientific analysis of test results. Rather, these results provide a preliminary look into a number of areas in which virtual environments are reportedly detectable.

6.1.1 CPU-based Environment Detection Tests

CPU clock variability seeks to sample the various system clocks that are available on modern CPUs a high number of times, then determine the amount of variability in the clock samples by calculating a Mean and Standard Deviation. This is a test devised from an online discussion regarding 'red pill' virtual environment detection [73]. The test code is based on previously published code designed for execution on x86 architectures. A version of the test that was modified for this study on ARM architecture systems uses system "tick" counts as its timing source due to ARM not supporting the same internal clocks as x86 processors. Due to the sharing of system resources and the need to handle hardware and software interrupts on both the guest and host system, it is anticipated that virtual machines may demonstrate a variability in their clock timing sources that is not present in bare metal operating environments.

CPU information testing merely queries the system for information on the CPU and attempts to determine if there are any abnormalities such as a non-standard CPU name or an unexpected number of CPU cores. This test has been identified in [27] as an effective means

to detecting certain virtual environments due to how the hypervisor, or the configuration of the hypervisor, presents the physical attributes of the bare metal system to the virtual environment. An additional test, demonstrated in code [51], shows an effective means of detecting hypervisors that advertise themselves by the CPUID instruction on x86 processors. When executing CPUID, the 31st bit of the ECX register can (optionally) return a value of '1' if a hypervisor is present [82]. It should be noted that no academic literature found to this point has identified this as a virtual environment detection method, which may indicate that academic literature is somewhat lacking when compared to the number of tests found on the Internet.

Execution time of instructions was outlined in several papers [14] [30] [65] [69] as a candidate for virtual environment detection. Expanding on the concept, but taking into account hardware-assisted VMs that execute all instructions natively except a small number that trigger a hypervisor event before executing on the CPU, privileged instructions were targeted for this implementation. As a security method, CPUs limit a core number of instructions to be executable only by the primary operating system, preventing programs from performing operations that can affect the stability and integrity of a system. Because virtual environments operate as guest sessions and are not the primary operating system, in the case of full virtualization, the hypervisor is required to intercept any instructions the guest OS believes it can execute and either run the command on behalf of the guest OS, or translate the privileged instruction into non-privileged ones that the guest OS is capable of executing; in the case of hardware assisted virtualization, the guest operating system has an additional set of instructions it can execute rather than the standard privileged ones, but a hypervisor event is still triggered, adding a delay. This act of intercepting and possibly translating instructions results in a delay in execution that is not present in bare metal systems. By performing a privileged instruction a high number of times and tracking its execution time, it is expected that virtual environments will demonstrate a slower execution than bare metal systems, even in the case of hardware-assisted virtualization. For this project, memory mapping was identified as a privileged instruction and selected for implementation [8].

6.1.2 File System-based Environment Detection Tests

Artifacts in the file system merely scans the system drive looking for files that may indicate whether the program is executing in a virtual environment. While not an advanced test, this has been identified as a likely method for VM detection and continues to show up in the wild [45]. As in the case of the previously discussed virtual environment detection method, the lack of having a sufficient number of Microsoft Word documents saved to the file system would indicate that it is not a production system and is likely intended for malware analysis [75].

6.1.3 Hardware Attribute-based Environment Detection Tests

Artifacts in system hardware attributes queries the system hardware through the operating system and attempts to identify any markers that are common for virtual environments, as outlined in [14]. For this project, the hardware attributes analyzed are the MAC addresses of the available network interface cards. There are several well-known vendor codes that are used by popular hypervisors which are searched for. It is also frequent that virtual machines implement a MAC address that is selected randomly in order to avoid being identified as a virtual machine while minimizing the chance of repeat MAC addresses. To account for this scenario, in the event that a MAC address wasn't identified as belonging to a virtual environment, the test passes the MAC address to an online MAC vendor database to determine if the vendor code of the MAC address belongs to any registered vendor. If not, the MAC address is identified as being locally administered and the system is identified as possibly a virtual environment.

6.1.4 System Memory Performance-based Environment Detection Tests

It has been proposed in [27] that unprivileged users operating on an x86 processor can execute the sensitive CPUID instruction that, while not affecting system memory, triggers a hypervisor event that flushes the CPU's Translation Lookaside Buffer (TLB). The test devised for this study takes a simplified approach to the method outlined by Ferrie, et al. and simply tracks the amount of time taken to retrieve items from memory. If CPUID requests trigger a hypervisor event that results in a flush of the TLB, additional delays and variations are expected in memory access times.

6.1.5 Network-based Environment Detection Tests

Much less explored is the idea that malware may attempt to determine whether a potential target is a virtual environment before it is even loaded on to the target host. Traditionally, in order to detect the presence of a virtual environment, a malicious binary must already be loaded and executing on the potential victim system. Two methods of remote virtual environment detection are explored in [14] and [29]. By using these techniques, malware may be able to identify systems that are likely virtual machines, thus potential monitoring systems, and choose to not continue in the attack. With malware terminating attacks even before they attempt infection, security researchers are left with little to analyze and must pursue analysis by other methods. Only [14] discusses true remote detection that does not rely on the execution of code on the potential target, but through information gleaned by the timestamps in TCP packets.

Variability in TCP timestamp clock skew is suggested to be a test that malware can perform

remotely in order to determine whether to proceed with an attack, as outlined in [14]. In contrast to bare metal machines, virtual-machines tend to demonstrate a distinct variation from the clock skew in its TCP timestamp values due to their additional layer of virtualized hardware abstraction implemented in software. This variation in TCP timestamp measurements can then be used to detect whether the remote host is a virtual environment.

6.1.6 Linux Container Expected Performance

As Linux Containers isolate system resources and execute on the bare metal kernel, they are not anticipated to display the same artifacts that a virtual environment will generate, thus it is expected the test results to be very similar to bare metal machines. However, due to the additional layers of isolation in Linux Containers, along with the feasibility of using Linux containers for VM-aware malware testing, this is not a foregone conclusion and should be verified in a laboratory environment.

6.2 Experiment Lab

The lab systems utilized to conduct this experiment consist of:

- 2 x Third Generation Raspberry Pi Model B single-board computer; Hostnames “Amy” and “Betty”
- 1 x Minnowboard Turbot x86 64-bit Single-Board Computer; Hostname “Cindy”
- 1 x “Desktop Class” Intel Core i5-2400 system; Hostname “Dani”
- 1 x “Server Class” Intel Xeon E5320 system; Hostname “Emma”

Each system is installed with Ubuntu 16.04 as it’s operating system with the exception of the Raspberry Pi system Betty, which has Ubuntu Core 16. Systems Dani and Emma have the standard desktop installation of Ubuntu, while systems Amy and Cindy are installed with the server variant, identical to the desktop version but lacks the X Windows GUI. All systems operate using Linux kernel version 4.4, with the exception of Amy, which is using a modified kernel version 4.1.19.

The virtual environment software included in this study consisted of:

- VMWare Workstation, Ver 12.5.2
- QEMU, Ver 2.5.0
- KVM, Ver 2.5.0

- Xen Paravirtualized (PV), Ver 4.6.0
- Xen Hardware Assisted (HVM), Ver 4.6.0

6.2.1 Hardware

Third Generation Raspberry Pi Model B

The Raspberry Pi is single-board, ARM computer intended for use as a low-cost education platform. The specifications of the third generation model B are [84]:

- 1.2GHz 64-bit quad-core ARM Cortex-A53 CPU
- Integrated 802.11n Wi-Fi, 10/100 Mbit/s Ethernet, and Bluetooth 4.1
- Broadcom BMC2837 SoC architecture
- 1GB RAM
- Bootable MicroSD card for operating system and system storage

Minnowboard Turbot

The Minnowboard Turbot is an x86 64-bit single-board computer using Intel's Atom-series processor. Highlights of the system specifications are [23]:

- Intel Atom E3826 1.46GHz 64-bit dual-core SoC
- Integrated 10/100 Mbit/s Ethernet
- Intel VT-x virtualization extension capable
- 2GB RAM
- Bootable MicroSD card for operating system and system storage

Figure 6.1: Raspberry Pi 3 Model B

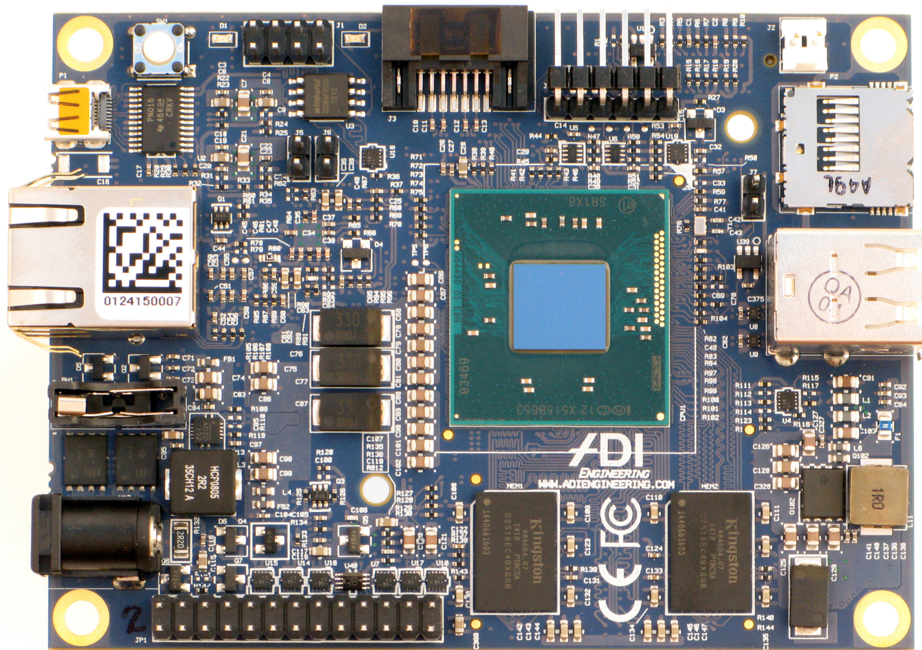

Figure 6.2: Minnowboard Turbot

Desktop Class System

The desktop system included in the study is a Dell OptiPlex 990 with the following specifications:

- Intel Core i5-2400 3.1GHz 64-bit quad-core CPU
- Intel VT-x virtualization extension capable
- 4GB RAM

Server Class System

The server system is a Dell Precision WorkStation 490 with an Intel Xeon chipset in a full-tower configuration with the following specifications. It is being labeled as a server class system for this study due to the Xeon chipset being utilized in a significant number of commercial servers.

- Intel Xeon E5320 1.86GHz 64-bit octo-core CPU
- Intel VT-x virtualization extension capable
- 4GB RAM

6.2.2 Operating System and Software

All systems were operating with Ubuntu 16.04.02, with the exception of Betty. Software and operating system updates were stopped on around November 2016 due to an unknown update breaking LXD on Amy; additionally, in order to minimize system performance/behavior changes during the course of the study, minimal updates were applied prior to that. The low-powered devices had Ubuntu Server installed rather than Ubuntu Desktop; the only difference being the lack of the X Windows environment.

Due to the third generation Raspberry Pi chipset not being fully supported by the mainline Linux kernel, a custom installation of Ubuntu Server 16.04 was installed on Amy from the website: https://ubuntu-pi-flavour-maker.org/.

Following an unknown software update change, LXD no longer ran on the Raspberry Pi 3 using a modified Ubuntu 16.04. Amy was kept as an Ubuntu Server installation while Betty was converted to Ubuntu Core, Canonical's variant of Ubuntu designed for embedded systems. Through the use of Ubuntu Core on Betty, LXD was able to be installed and containers could execute and run as expected.

System	Architecture	OS
Amy	ARM	Ubuntu 16.04 Server (Modified)
Betty	ARM	Ubuntu Core 16
Cindy	x86_64	Ubuntu 16.04 Server
Dani	x86_64	Ubuntu 16.04 Desktop
Emma	x86_64	Ubuntu 16.04 Desktop

Table 6.1: Table of lab hosts and installed operating systems.

6.2.3 Ubuntu Core

Ubuntu Core is an additional variant of Ubuntu designed specifically for embedded or IoT-type devices. It operates on the concept of 'snaps' rather than standard Linux software packages, where snaps are sandbox environments that contain applications, it's dependencies, and meta-data used by Ubuntu Core to control how the snap executes [52]. The philosophy behind snaps is to create independent application packages that have no dependencies on other snaps, yet can still communicate with other snaps if needed, as well as contain all of their dependencies within them, rather than having those dependencies resolved elsewhere in the operating system. This allows Ubuntu Core to operate at a very light-weight level as each application snap should already contain everything that is needed for execution. Ubuntu Core appears to be the only Ubuntu variant officially supported by Canonical to operate on embedded or low-power devices, such as the Raspberry Pi. Due to the unknown application or operating system update that broke LXD functionality on Raspberry Pis using an unofficial variant of Ubuntu, this was the only available option to retain Linux container capabilities on the Raspberry Pi without exploring other Linux distributions. Unfortunately, due to the fact that Ubuntu Core requires all software to be installed through snaps and the relative newness of snaps on the Linux landscape, this places a significant constraint on what software can be installed on a Raspberry Pi that is running Ubuntu Core.

It has been announced that the mainline Linux kernel version 4.8 has added support for the Broadcom BCM2837 SoC, which should allow standard Linux distributions to support the Raspberry Pi 3 natively without requiring modified kernels [44]. Unfortunately, due to the timing of the announcement, this kernel version was not adopted to the test systems; however, this indicates that the Raspberry Pi 3 will be able to utilize a wider range of Linux distributions and software without requiring OS modifications specific to it's architecture.

Table 6.1 identifies lab hosts with their installed operating systems.

6.2.4 Virtual Environment Software

As stated earlier, the virtualization software utilized in this study consisted of VMWare Workstation, QEMU, KVM, and Xen. The intentions of the virtual environments selected

System	Installed Hypervisors
Cindy	KVM
Dani	QEMU, KVM, VMWare
Emma	QEMU, KVM, VMWare, Xen PV, Xen HVM

Table 6.2: Table of systems and installed hypervisors.

were to remain Linux-based (as Linux containers are currently restricted to Linux) and popular choices for security researchers and enthusiasts. It is expected that these virtual environments are most likely to consist of the majority utilized in high-interaction honeypots.

VMWare Workstation is a powerful Type II hypervisor that is free for non-commercial use. It is widely employed on both Windows and Linux for single instance virtual machines.

QEMU is a useful and unique Type II hypervisor for researchers who are analyzing binaries across different architectures. QEMU emulates all of its system hardware using dynamic translation, which, while versatile and highly configurable, results in slow execution. QEMU is unable to operate on Amy, Betty, or Cindy due to excessive slowness of system boot resulting in boot failures.

KVM is a variant of QEMU that, rather than emulating the processor, uses hardware assisted virtualization to employ the bare metal CPU. While this limits the architecture of the virtual system to match the architecture of the host, hardware assistance greatly accelerates the virtual environment to near bare metal performance. KVM requires the presence of hardware assisted virtualization extensions in the system processor, which are not present on the Raspberry Pi, but are present for the processors of Cindy, Dani, and Emma.

Xen is an open source Type I hypervisor traditionally known for larger scale virtualization. It supports both hardware assisted virtualization (HVM) and paravirtualization (PV) and can be employed with a variety of guest operating systems. It is utilized by a number of medium to large scale commercial and professional networks.

Table 6.2 outlines which hypervisors were installed on which hardware systems for this study.

LXD

LXD is the latest iteration of LXC in that it delivers LXC through the use of a container management daemon. LXD grants easier container creation, management, and maintenance, as well as streamlining additional functionality of LXC, such as container live migration, and also providing a REST API for software to interface with.

All systems were installed with LXD version 2.12 with the exception of Cindy, which was installed with version 2.09. It was not determined what was preventing the successful update of LXD from 2.09 to 2.12 during the course of this study.

LXD was installed on each of the lab hosts using the Personal Package Archive (PPA) "ubuntu-lxc/lxd-git-master". Upon installation, LXD was assigned the following configuration through the use of the "lxd init" command:

- Storage backend: dir
- LXD not available over network
- LXD uses a network bridge (lxdbr0)
- Configured for IPv4 with random subnet
- Containers use NAT for IPv4 traffic
- Not setting up IPv6

6.3 Experiment Tests

6.3.1 CPU-based Environment Detection Test Details

Each of the tests within the CPU-based categories attempts to explore different aspects about a virtual machine's CPU that may allow malware to identify whether it is executing in a virtual environment. The selected areas tested were internal system clocks, information reported by the CPU, and instruction execution timing.

CPU Clock Variability

This test attempts to explore whether hypervisor management or increased resource contention results in greater variability that may be detected in order to identify a virtual environment. The test is derived from code written by Bill Torpey in order to test various system clocks [81] by collecting a specific number of timestamps in a loop, then measuring the time elapsed between each timestamp. Excerpted from the original code is the identification of available system clocks and the code that samples each of those clocks and reports basic statistical properties on the sampling.

For this study, the test was modified to include adding a timing source that is usable on ARM systems, namely system 'tick' counts, as ARM processors lack the same internal clocks that x86 processors possess.

The output for the test includes min and max values for each of the tested clocks, as well as average and median values, plus the standard deviation. For the purposes of examining

clock variation as a marker to detect a virtual environment, the standard deviation for each clock is the most significant feature.

Source code for this test can be found in Appendix A.1.

CPU Information

There are three types of information collected for this test: CPU model name from the Linux kernel in '/proc/cpuinfo', processor name and info reported by the 'platform' Python module, and the 31st bit of the ECX register when executing the CPUID instruction. The CPUID instruction test is only available on x86 processors.

CPU model name is a simple 'cat' and 'grep' from '/proc/cpuinfo' that returns the CPU name programmed into the system BIOS, which may be non-standard on virtual systems. Processor information reported by the Python 'platform' module reveals additional information about the processor, specifically the reported number of CPU cores, which may be less than the number of cores that a particular CPU model possesses.

Executing the CPUID instruction required the use of an assembly code example from the Linux kernel. CPUID uses several system registers for both input and output; when calling the instruction with EAX set to 1, CPUID returns processor information and reports whether some processor features are enabled. The 31st bit in the EAX register is available for hypervisors to optionally report whether they are present; a flag in that register bit is a clear indicator that the test is being executed in a virtual environment.

Source code for this test can be found in Appendix A.2.

Execution Time of Privileged Instructions

The execution timing test performs memory mapping a specific number of times and records how long to took to perform those operations. As some virtual machines attempt to defeat this type of test by modifying their internal clocks in order to disguise the actual execution time, this test also includes the ability to use an external timing source, specifically an NTP server from "us.pool.ntp.org", as a comparison to the internal clocks. The functions for getting timestamps from the NTP server were provided by code examples by David Lettier [47].

This test performs either a single run of 99,999 memory mapping and unmapping operations or executes several runs at increasing iterations at 999, 9,999, and 99,999 times in order to compare execution times at increasing scales.

Source code for this test can be found in Appendix A.3.

6.3.2 File System

Artifacts in the File System

This test searches the names of files in the system or names of device drivers that might indicate the presence of a virtual environment. Currently, the test searches for the presence of files named either 'qemu' or 'bochs'. The test then runs LSPCI -NN, asking the system to list all networking devices on the PCI bus, then checks if any of those devices contain the strings 'VirtualBox' or 'VMWare'. Additionally, due to an oddity noted when performing the test, it appears that for default configurations Xen PV virtual machines do not contain a PCI network device. As such, if the command LSPCI -NN fails to return any results, then that may be an indicator of a Xen virtual environment.

Source code for this test can be found in Appendix A.4.

6.3.3 Hardware Attributes

Artifacts in System Hardware Attributes

The check for artifacts in hardware attributes focuses on MAC addresses of network cards found in the system. There are specific manufacturer's strings that can be identified for network cards configured with the default settings for VMWare, Xen, QEMU, and Bochs virtual environments; if those strings are detected, then it indicates a virtual environment. As a check against network card MAC addresses that have been generated randomly as an attempt to avoid this type of check, the test uses a PHP script provided by MACVendors.com to also verify the MAC address against an online database [54]. If the MAC address doesn't posses a known vendor string, it is assumed to be a locally generated MAC address and an indicator of a virtual environment.

Source code for this test can be found in Appendix A.5.

6.3.4 System Memory Performance

The system memory performance test seeks to identify delays in memory access on virtual machines that are the result of TLB flushes occurring when the CPUID instruction is executed, triggering a hypervisor event. The test is expected to be particularly sensitive to slight performance variances as hypervisor events do not cause a clearing of the CPU cache, but the CPU's memory-management unit cache that retains recent translations from virtual to physical memory. Ideally, when CPUID is called and the TLB is flushed, the next memory read will result in a TLB miss; however, the system should still find the memory page containing the requested data in memory, resulting in only a slight access delay due to the

need to check for the presence of the page.

An arbitrary 40% was chosen as allowable additional latency caused by normal system variance. Any memory accesses that take longer than 40% of the pre-hypervisor event access will identify the system as a virtual environment.

Source code for this test can be found in Appendix A.6.

6.3.5 Network

A technique described by Chen, et al., in [14] allows an attacker to compare the clocks between the attacker's system clock and the target's TCP clock through the use of TCP timestamps. As the comparison is being made between two different clock sources, a clock skew is expected with a continual drift at a linear rate as the two time sources get further and further apart. Chen states that a target using a bare metal system will provide TCP timestamps that correspond to the expected clock skew with little variability. However, if the target is using a virtual environment, much greater variation is expected in the TCP timestamps. Using this, an attacker is able to identify remote targets as virtual environments without ever having to interact directly with the target.

This style of detection is capable of being performed passively, so long as the attacker resides in the same network segment as the target system and is able to capture TCP packets transmitted from the target. However, as the TCP timestamp is an optional field, not all TCP packets contain a timestamp; depending on the traffic of the victim, it may take considerable time for an attacker to collect enough TCP timestamps in order to determine whether the target system is a virtual environment. Due to this fact, the test used in this study establishes an active TCP session between the attacker and target with crafted TCP packets containing the TCP timestamp being sent between the two systems. The tools used to maintain the TCP session transmits a TCP packet containing a timestamp to the target; in accordance with the TCP timestamp RFC 1323, the remote target responds to the packet with a timestamp of its own, providing the data the attacker requires for analysis.

As RFC 1323 does not place any requirements on what TCP timestamps need to reflect, only that they increment proportionally, the following equations outlined in [14] are utilized in order to perform an analysis on the environment of a remote host. These equations factor out the attacker's local clock frequency:

$$TCP\,Timestamp\,Received\,from\,Remote\,Target = T_i \tag{6.1}$$

$$Initial\,TCP\,Timestamp\,Received = T_0 \tag{6.2}$$

$$Local\,Time\,Instance = t_i \tag{6.3}$$

$$Local\,Start\,Time = t_0 \tag{6.4}$$

$$Remote\,Target\,Frequency = (\frac{T_1 - T_2}{t_1 - t_2}) \tag{6.5}$$

$$Time\,Elapsed\,Locally\,At\,Time\,i : x_i = (t_i - t_0) \tag{6.6}$$

$$Time\,Elapsed\,at\,Target\,at\,Time\,i :$$

$$w_i = (\frac{T_i - T_0}{Remote\,Target\,Frequencey}) \tag{6.7}$$

$$Clock\,Skew\,at\,Time\,i : y_i = x_i - w_i \tag{6.8}$$

$$Linear\ Least\,Squares\,Fit\,Line : f(x) = Sx + q \tag{6.9}$$

$$Squared\,Error\,(SE) : SE = (f(x_i) - y_i)^2 \tag{6.10}$$

$$Mean\,Squared\,Error\,(MSE) :$$

$$MSE = (\frac{\Sigma(SE_i)}{Number\,of\,Timestamps}) \tag{6.11}$$

$$Theoretical\,Remote\,Target\,Frequency$$

$$= (\frac{1}{Remote\,Target\,Frequency})^2/12 \tag{6.12}$$

Chen, et al., states that in Linux, the TCP timestamps are merely a truncation of the system hardware clock, which will be running at the frequency calculated by equation 6.5; thus the timestamps are updated once every 1/(*Remote Target Frequency*). Chen, et al. then, assuming a uniform distribution of timestamp variance on bare metal systems, and applies equation 6.12 to derive a theoretical MSE that represents a variance that is uniformly distributed across a period. This value is used as a reference for the MSE that is expected to be found on a bare metal system.

Using the TCP timestamps collected from the target system, the local time on the attackers system, and the equations listed above, an attacker is able to perform the following calculations:

Equation 6.5, allows the attacker to calculate the clock frequency of the target system. With that, equation 6.7 allows the attacker to use the initial TCP timestamp received from the target and a later one to calculate how much time has passed on the target system between two TCP timestamps. Using equation 6.6, the attacker calculates how much time has passed locally, then uses equation 6.8 to determine the clock skew of the target from their own system at a particular moment in time.

After collecting a number of clock skew samples, the attacker can then calculate a linear least squares fit line. Generating a list of squared errors (SE) for each of the clock skew plots allows the calculation of a Mean Squared Error (MSE) to measure the overall fit of the

trend line to the collected plots. Once a Mean Squared Error (MSE) is calculated for the clock skew of a remote target, it is compared with the baseline expectation of the theoretical MSE.

A significant difference between the calculated MSE and the theoretical MSE of a remote target (by an order of magnitude, for example) would indicate that the remote target is a virtual machine.

This test was re-implemented with assistance from Andres Pico Chavez from the Virginia Tech CS department. Three types of systems were used when testing the suitability of Linux Containers to defeat this remote detection method: bare metal machines, virtual machines, and Linux containers. The experiments were conducted to validate the effectiveness of the detection method against virtual machines, and then to compare the results of Linux containers against the results of bare metal systems.

The capture and analysis portion of the detection method occurs on the attacker's machine. A script was written to customize each attack and pass the necessary arguments to the dependent code.

The primary capture and analysis code was derived from the program *sniffex*, written in C, by the Tcpdump Group. The use of the pcap library to capture raw IP packets and the initial packet analysis was mostly unmodified. However, once the code identified the captured packet as TCP, the detection and extraction of the timestamp from the packet header, along with the subsequent timestamp analysis, was code original to this thesis in an attempt to replicate the process outlined in [14].

Once a number of clock skew samples have been collected (Chen, et al. states that they collected clock skew samples until they reached convergence, but no convergence value was provided for reference in [14]), the c library, *libsgl* is used to calculate a Linear Least Squares Fit Line across all of the clock skew samples. The squared error for each clock skew sample from the fit line is calculated using equation (6.10) and the average of all the squared errors (the MSE), is calculated from equation (6.10). Comparing the calculated MSE from the theoretical MSE, an indication should be provided whether the remote target is a bare metal system or a virtual environment.

As knowledge of the remote target system's OS is required to calculate the theoretical MSE described, the remote OS fingerprinting capability of the tool *nmap* is used and incorporated into the test output.

Source code for this test can be found in Appendix A.7.

Chapter 7

The Results

7.1 Experiment Results

7.1.1 CPU-based Environment Detection Tests

CPU Clock Variability

Upon reviewing the variability reports from the numerous clock sources available on an x86 process, it appeared that reading the Time Stamp Counter after the execution of a CPUID instruction, referred to as CPUID+RDTSC, provided sufficient variance that may lead to predictability. Using a rough approximation of standard deviation as a measurement for variance, it was shown that the standard deviation of the CPUID+RDTSC timing source does increase in most virtualization, except on QEMU. This is most likely due to QEMU having the distinct advantage of fully emulating it's CPU; so despite the expection of significant variability in actual execution performance, QEMU maintains it's own CPU clock rate and generates test results based on that as Time Stamp Counters do not use time-of-day; slow emulation is not revealed. The other tested hypervisors do not perform full CPU emulation, so they must rely on the bare metal CPU as the external timing source. The impact that average value has on standard deviation was omitted for brevity and is not expected to significantly impact observed behavior.

As can be seen in Figure 7.1, both bare metal and container execution values remained approximately equivalent while the standard deviation for CPUID+RDTSC sampling in KVM increased by approximately 2 points.

Figure 7.2 shows the standard deviation results for CPUID+RDTSC sampling on Dani. This figure demonstrates the similarity of QEMU's standard deviation to that of bare metal execution. Interestingly, an increase in standard deviation was observed for container execution by approximately 2 points, similar to the point increase shown for KVM execution

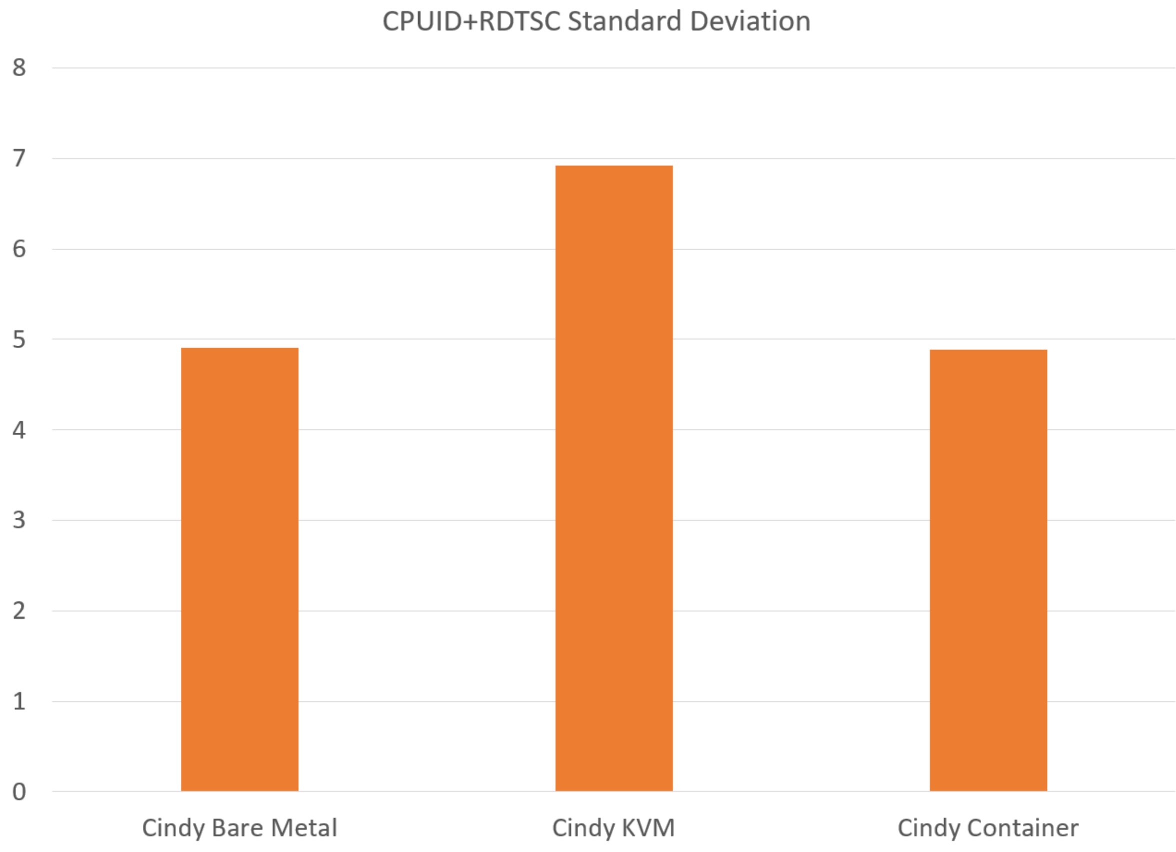

Figure 7.1: Standard deviation of CPUID+RDTSC timing source on Cindy. X-axis is environment of execution. Y-axis is standard deviation.

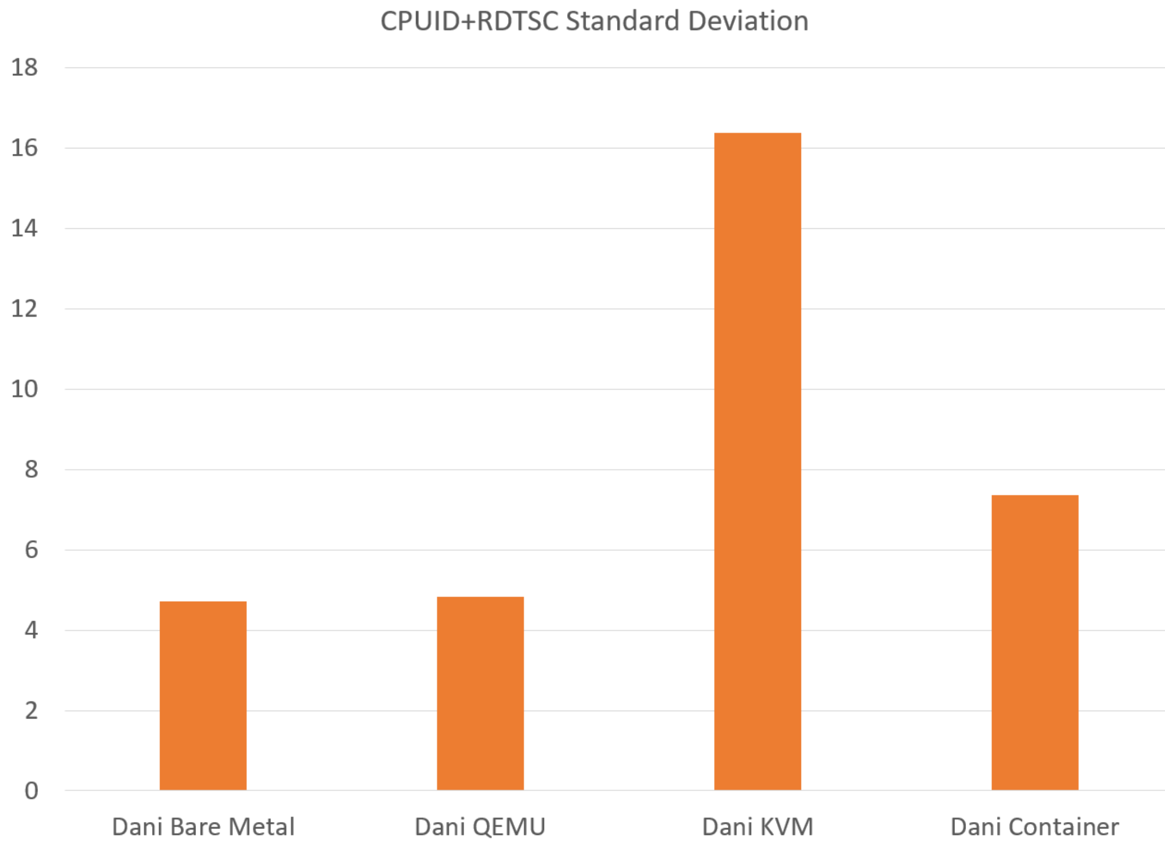

Figure 7.2: Standard deviation of CPUID+RDTSC timing source on Dani. X-axis is environment of execution. Y-axis is standard deviation.

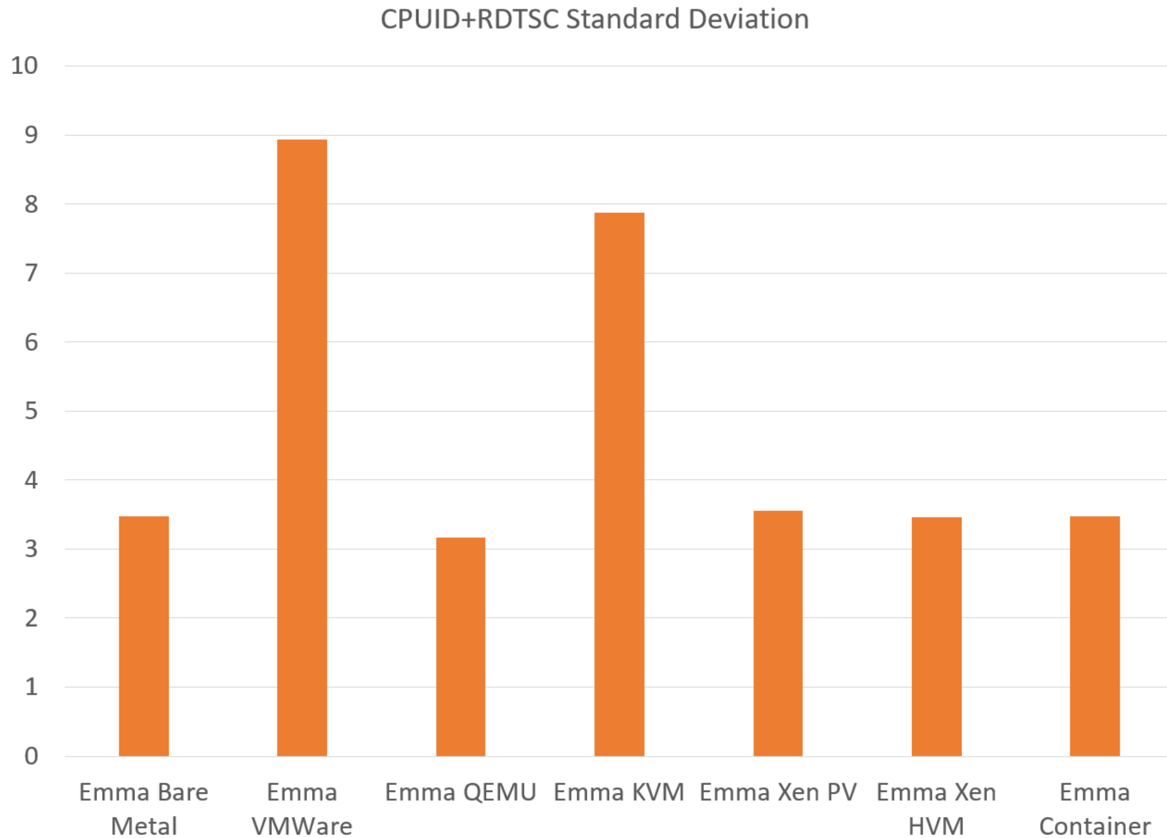

Figure 7.3: Standard deviation of CPUID+RDTSC timing source on Emma. X-axis is environment of execution. Y-axis is standard deviation.

on Cindy. Results for VMWare were omitted from this figure as they would skew the chart scale heavily, owning a standard deviation of approximately 1600.

Figure 7.3 illustrates the standard deviation results for the CPUID+RDTSC test on Emma. These results demonstrate that the traditional Type II hypervisors, KVM and VMWare Workstation, show an increase in standard deviation by over 4 points. However, not only does QEMU report the same standard deviation as bare metal execution, both Xen virtual environments do as well.

While observable impacts are made to clock sampling variance by many hypervisors, it would be difficult to translate these results into a simple reference test to detect virtual environments without knowing the bare metal performance metrics for a targeted platform beforehand. As such, a comparison was made for the average execution time between clock samplings on each platform.

Observing average execution time between clock samples on Cindy, as demonstrated in Figure 7.4, KVM shows a stark increase in average compared to bare metal and container execution

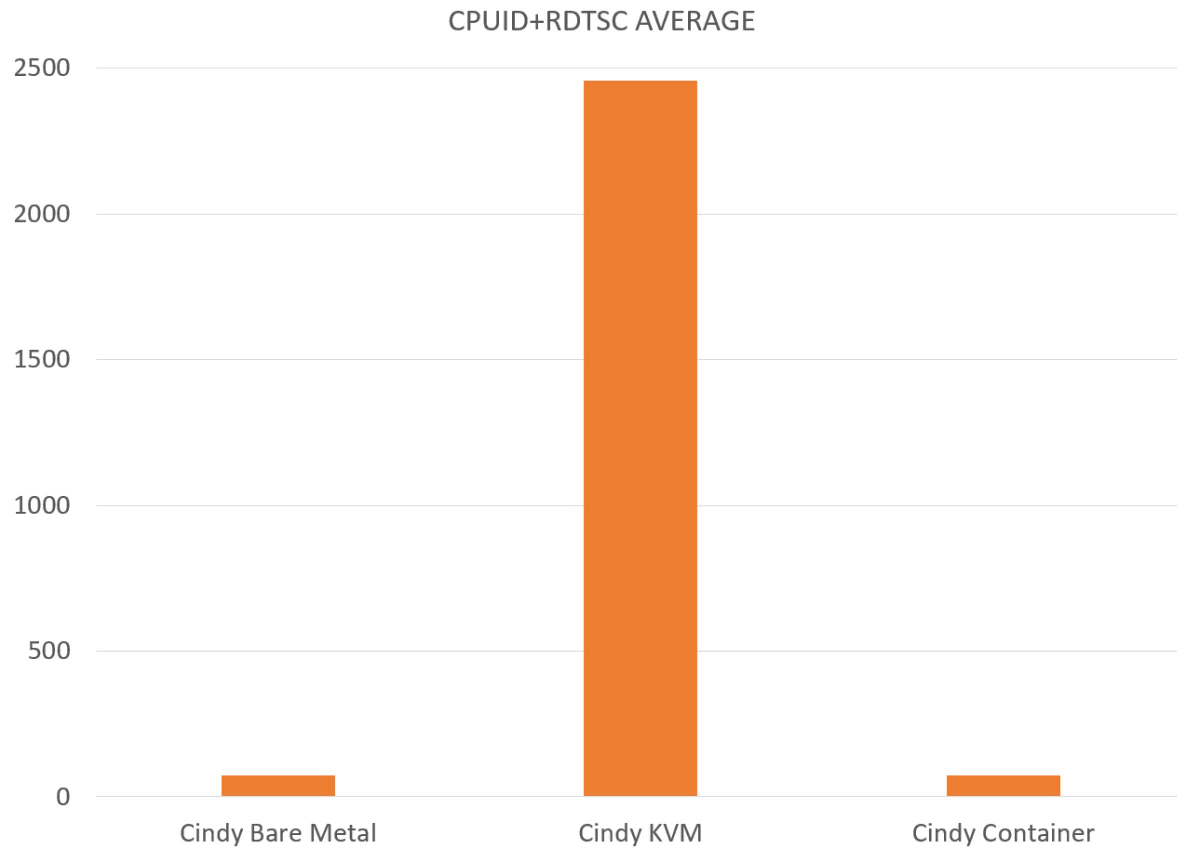

Figure 7.4: Average time between CPUID+RDTSC timing polls on Cindy. X-axis is environment of execution. Y-axis is average number of clock ticks.

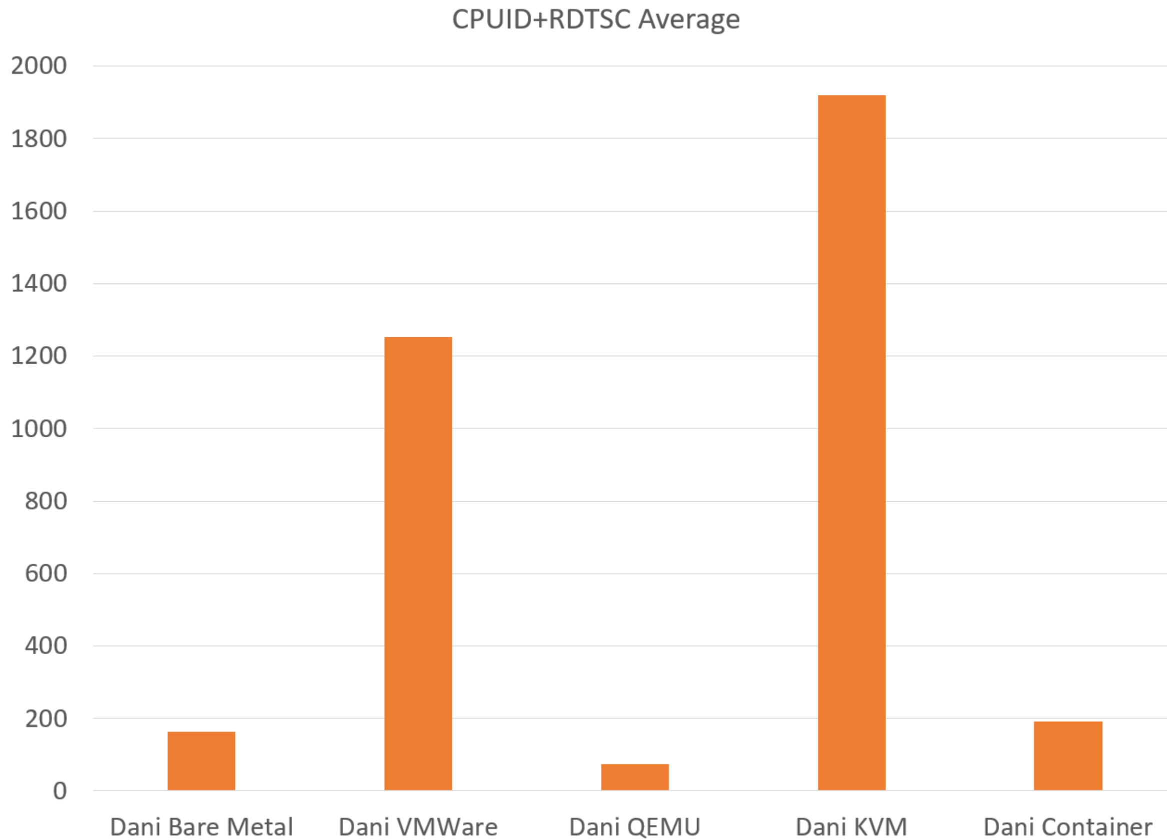

Figure 7.5: Average time between CPUID+RDTSC timing polls on Dani. X-axis is environment of execution. Y-axis is average number of clock ticks.

environments.

Figures 7.5 and 7.6 show the average execution time on Dani and Emma, demonstrating the marked increase in average value for all virtual environments, except for QEMU and Xen HVM.

QEMU is historically known to be a slow virtual environment due all hardware components being emulated in software; as such, by incorporating a timing source employing time of day, discrepancies in execution time can be easily identifiable and the virtual environment easily detected.

Due to the closeness in many of the reported standard deviation values, it is unlikely that standard deviation of CPU timing sources would result in a reliable virtual environment detection method. This may simply mean that standard deviation is a poor approximation of variability for timing sources and that a better method can be devised and implemented. However, more readily available for use as a virtual detection method would be average execution time between clock samples.

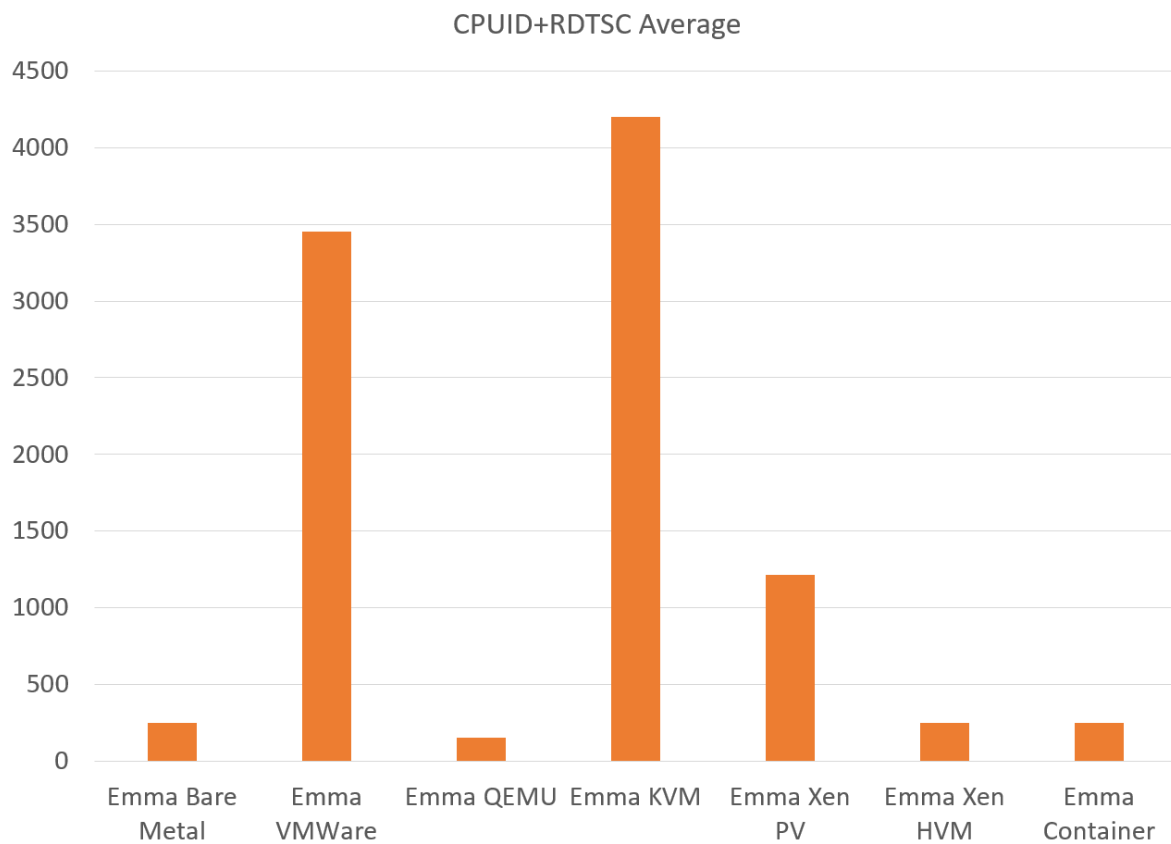

Figure 7.6: Average time between CPUID+RDTSC timing polls on Emma. X-axis is environment of execution. Y-axis is average number of clock ticks.

All virtual environments, with the exception of QEMU and Xen HVM, demonstrated an increase in average value by at least an order of magnitude. This feature can easily be tested without requiring reference values and would make a straightforward virtual environment detection method. As shown in the referenced figures, Linux containers repeatedly scored similarly to the bare metal systems in both standard deviation and average values and would not be revealed by such a detection method. It is possible that a virtual machine is capable of modifying the results outputted by tests such as this if it is anticipated that similar tests will be used for detection. However, it is expected that such a modification would require significant sophistication and effort to mask these results and is unlikely, thus this could be a candidate test for malware to perform in the wild.

Unfortunately, the results for this test cannot be compared against a bare metal Raspberry Pi and Linux container as the container implementation tested is running Ubuntu Core, which is unable to perform the clock variance test.

7.1.2 CPU Information

When polling the test systems' CPU information, it was revealed that most of the virtual systems returned non-standard CPU results as illustrated in table 7.1. QEMU and KVM hypervisors simply outputted "QEMU Virtual CPU," which clearly stated the operating environment was not only virtual, but being executed within QEMU or KVM. Both VMWare and Xen PV returned the name of the bare metal CPU the virtual machine was executing on, however the number of cores was not consistent with manufacturer specifications. Intel's core i5-2400 processors have always supported at least 2 cores, while Intel's Xeon E5320 have 4 cores. Both of these non-standard results can indicate to malware that it is not executing in a bare metal environment so long as the malware has a reference for the proper number of cores for each CPU model. Additionally, the QEMU, KVM, and VMWare hypervisors advertise their presence within the CPUID registers, which quickly and efficiently identifies those environments as virtual; neither Xen PV nor Xen HVM advertise their presence in CPUID. Only Xen HVM remained undetected by this test by both correctly relaying all appropriate CPU information as well as not advertising it's presence in CPUID.

As Linux containers poll the system information directly from the kernel, they returned results matching those of the bare metal system, except in the case of Betty, running Ubuntu Core, which prevented the container from viewing /proc/cpuinfo.

This test, in its present form, can be overcome by correctly configuring the virtual environment CPU to properly match the appropriate specification and to prevent hypervisor advertisement in the optional CPUID register field. This requires some moderate effort on the part of the system administrator. As using CPU specifications for detection relies on malware maintaining a reference on numerous processors, this is unlikely to be conducted by malware in the wild. However, the hypervisor flag in CPUID can be easily checked and may catch some security administrators off-guard who may not be aware of the flag's presence

System	CPU Information	CPUID Flag	VM Detected
Amy	ARMv7 Processor (v7l) x 4	N/A	No
Betty Container	armv7l	N/A	Yes
Cindy	Atom E3826 @ 1.46GHz x 2	No Hypervisor	No
Cindy KVM	QEMU Virtual CPU	Hypervisor Present	Yes
Cindy Container	Atom E3826 @ 1.46GHz x 2	No Hypervisor	No
Dani	i5-2400 @ 3.10GHz x 4	No Hypervisor	No
Dani VMWare	i5-2400 @ 3.10GHz x 1	Hypervisor Present	Yes
Dani QEMU	QEMU Virtual CPU	Hypervisor Present	Yes
Dani KVM	QEMU Virtual CPU	Hypervisor Present	Yes
Dani Container	i5-2400 @ 3.10GHz x 4	No Hypervisor	No
Emma	Xeon E5320 @ 1.86Ghz x 8	No Hypervisor	No
Emma VMWare	Xeon E5320 @ 1.86Ghz x 1	Hypervisor Present	Yes
Emma QEMU	QEMU Virtual CPU	Hypervisor Present	Yes
Emma KVM	QEMU Virtual CPU	Hypervisor Present	Yes
Emma Xen PV	Xeon E5320 @ 1.86Ghz x 2	No Hypervisor	Yes
Emma Xen HVM	Xeon E5320 @ 1.86Ghz x 8	No Hypervisor	No
Emma Container	Xeon E5320 @ 1.86Ghz x 8	No Hypervisor	No

Table 7.1: Results of CPU Information Test

and the VM's default behavior to advertise itself, thus this detection method is most likely conducted in the wild.

7.1.3 Execution time of privileged instructions

The execution timing test provides distinguishable results between several bare metal and virtual systems, as shown in figure 7.7. KVM resulted in an average execution time increase of 89%. VMWare on Dani took approximately 52% longer to perform the test where VMWare on Emma took 166% longer. Both Xen PV and Xen HVM took approximately twice as long to perform the test. Dani QEMU and Emma QEMU were omitted from the figure due to their extremely slow performance times; execution times were at least 18 times greater than that of bare metal performance.

A noticeable discrepancy is the increased performance time between Amy and Betty's Linux container. Container execution on Cindy, Dani, and Emma increased only an average 5% compared to bare metal, where execution on Betty's container took 129% longer than Amy's. As the hardware between the two Raspberry Pis is identical, the only accountable factor can be that Amy executes on a modified Ubuntu Server installation and Betty is running Ubuntu Core. Most likely, a modification in the handling of memory mapping operations creates an additional overhead on Ubuntu Core systems, possibly as a result of the additional resource

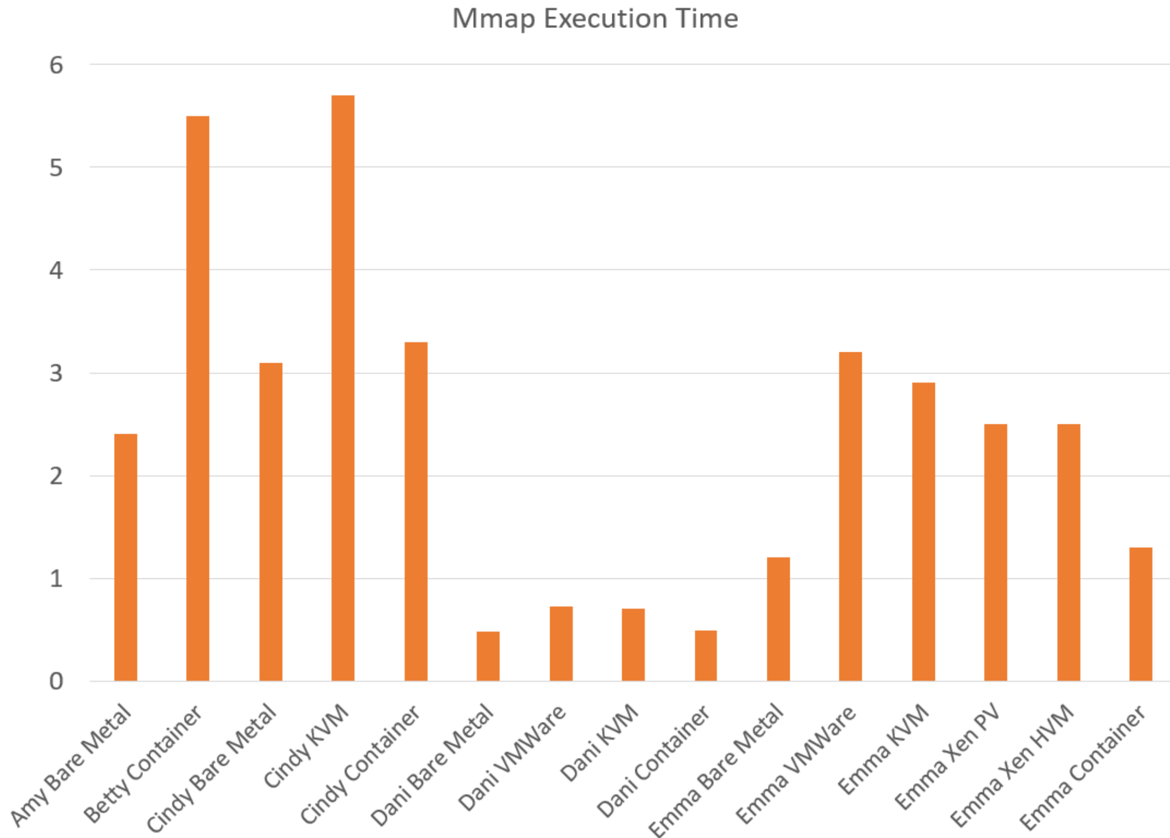

Figure 7.7: Execution time of 10,000 memory map operations. X-axis is environment of execution. Y-axis is execution time in seconds.

isolation mechanisms built into the operating system.

Similar to the CPU clock variance and execution times, only QEMU's performance values would be useful for virtual environment detection without a reference value specific to the target system. While the presence of a virtual environment had a clear impact on test execution across the board, it would be challenging to identify the performance loss without knowing the bare metal performance beforehand. As an example, VMWare running on Dani completed the test sooner than Emma on bare metal. However, as alluded to in [30], a test may be devised that compares the execution performance between two instructions in order to reveal the presence of a virtual environment.

It is expected that this test would require a highly sophisticated countermeasure to mask the test results by skewing the system clock; additionally, if such a countermeasure were devised, it could simply be defeated by the use of an external time reference. As such, it is highly likely that such a test will be performed in the wild in order to detect the presence of a virtual environment.

System	Clock Var	CPU Info	Exec Time	File Sys	H/W Attrib	Mem Lat	TCP Clock
Amy	No	No	No	No	No	N/A	No
Betty Container	No	Yes	Yes	No	Yes	N/A	No
Cindy	No	No	No	No	No	Yes	N/A
Cindy KVM	Yes	Yes	Yes	No	Yes	Yes	N/A
Cindy Container	No	No	No	No	Yes	No	N/A
Dani	No	No	No	No	Yes	Yes	No
Dani VMWare	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Dani QEMU	No	Yes	Yes	No	Yes	No	Yes
Dani KVM	Yes	Yes	Yes	No	Yes	Yes	N/A
Dani Container	No	No	No	No	Yes	No	No
Emma	No	No	No	No	Yes	No	No
Emma VMWare	Yes	Yes	Yes	Yes	Yes	Yes	N/A
Emma QEMU	No	Yes	Yes	No	Yes	No	N/A
Emma KVM	Yes	Yes	Yes	No	Yes	Yes	N/A
Emma Xen PV	Yes	Yes	Yes	Yes	Yes	No	No
Emma Xen HVM	No	No	Yes	No	Yes	No	N/A
Emma Container	No	No	No	No	Yes	No	N/A

Table 7.2: Summary of Virtual Environment Detection Test Results.

7.1.4 Artifacts in the File System

This test illustrated that a number of virtual machine files are already present in a fresh install of Ubuntu 16.04 which contains files for both QEMU and Bochs. As a result, searching for such files would be an insufficient method for detecting virtual environments. More telling in terms of detecting virtual environment presence were the names of installed system drivers: both VMWare and VirtualBox install and use system drivers that clearly state so in their names. Xen PV is a more interesting case where polling the system for network interface cards returns nothing. As it can be easily determined that the system is indeed on a network by using a number of methods, this mismatch of network connectivity without the presence of PCI bus network interface card drivers can be attributed to operating within a virtual environment.

Several virtual environments were able to go undetected due to differing reasons. QEMU and KVM's default network configuration emulates an Intel e1000 Network Interface Card which operates with standard network card drivers. As such, QEMU and KVM did not return any abnormal results. While malware could potentially link the presence of an e1000 network card to operating in a virtual environment, after a brief online search, it appears that the e1000 network interface card driver is compatible with a large number of network cards and may be legitimately used in bare metal systems. Likewise, Xen KVM did not present any identified artifacts that were tested for.

The results returned by Linux containers matched those of a bare metal system and they were not detected by this test. Unfortunately, due to differences in SoC architectures compared with standard desktop and servers, Raspberry Pi does not operate with a PCI bus. Attempting to list PCI devices clearly reveals this fact and has the potential of revealing to an attacker that they are interacting with an embedded-style device.

It is expected that with only a moderate amount of effort, the names of files and drivers can be masked in a virtual environment, thereby defeating this test. However, it is assumed this test can still provide a reasonable virtual environment detection method with a minimal amount of effort for an attacker to code and is very likely to be present in the wild.

7.1.5 Artifacts in System Hardware Attributes

Testing for artifacts in system hardware resulted in the detection of a virtual environment in most systems, bare metal execution and containers included. While this test provided the greatest number of false positives in terms of whether the code was being executed *inside* of a virtual environment, it was the only test that detected whether a system was configured to execute virtual environments, whether they were running or not.

The main reason behind the results is that nearly all systems had network bridges configured that are used by the virtual environments and visible from the bare metal system. These network bridges are either configured automatically by the hypervisor or configured by the administrator using instructions provided by the hypervisor documentation. As a result, the default configuration for the network bridges contain MAC addresses whose vendor IDs indicate that they are virtual adapters. If a MAC address was selected randomly to avoid this type of detection, the test revealed the lack of a proper vendor code and identified it as a virtual environment. While using randomly generated MAC addresses is not proof-positive of virtual environment presence, it may be telling enough for cautious malware authors to be wary of.

LXC does generate virtual bridges that communicate with the host hardware adapter, leaving it susceptible to this form of detection. It was discovered that the default configuration of LXC generates network card MAC addresses with a Xen vendor ID, which were identified by the test.

With only minimal effort, this test can be defeated by careful selection of MAC addresses that don't match known virtual adapter vendor IDs but still contain legitimate ones. However, due to the simplicity of the test and its ability to detect virtual environments that are not carefully configured, it is expected that this detection method is most likely present in the wild.

7.1.6 System Memory Performance

The results of this test remain somewhat inconclusive. Some virtual environments were not detected, while several bare metal systems were identified as virtual. The results indicate that the test correctly identified KVM virtual environments while not detecting QEMU. This may be attributed, as earlier, to QEMUs full processor emulation which does not have an interposing hypervisor that would result in TLB flush, where KVM utilizes the bare

metal CPU. Likewise, VMWare appeared to be successfully detected in each instance. Xen, however, was not identified in either PV or HVM mode.

This test could not be performed on the Raspberry Pi as the architecture lacks the CPUID instruction that is used to trigger the hypervisor event.

Interestingly, no containers were identified as virtual environments, even if bare metal execution was identified as such. Clearly, using CPUID to trigger TLB flushes in virtual environments and allow detection through increased memory latency is not a straightforward test. More investigation and tuning is required for this implementation to have greater accuracy; however, the motivation of the test has been previously mentioned in literature and it is very likely a similar detection method is employed by malware in the wild.

7.1.7 Variability in TCP Timestamp Clock Skew

The results of remote virtual environment detection are, unfortunately, less clear than the results of the previously discussed tests. After conducting nearly 150 tests, the results obtained did not display the differences between physical hosts and virtual machines that Chen, et al., indicated. In fact, the mean squared error (MSE) in many tests of both physical hosts and virtual machines showed values as small as the expected in bare metal machines, and as high as the values expected in virtual machines resulting in a high frequency of both false-positives and false-negatives. Some results are illustrated in Figures 7.8 and 7.9.

Considering that the observed behavior could be influenced by workload of the target system, several tests were redone while placing the systems under a stress test using the program 'stress-ng'. Interestingly, while placed under a load of CPU, I/O, and virtual memory tasks, VMWare began exhibiting the behavior suggested by Chen, et al. by showing well defined differences between bare metal and virtual machines clock skew variances, illustrated in Figure 7.11. However, predictable elevated variance could only be generated for the VMWare hypervisor; Xen and QEMU hypervisors could still produce bare metal results, even when performing significant CPU, I/O, and virtual memory operations.

While greater investigation should be performed regarding this type of virtual environment remote detection, to include testing under a greater variety of virtual environments, the experiment did demonstrate that TCP clock skew variance within a Linux container behaved the same as bare metal performance as can be seen in Figure 7.10. If such a virtual environment detection method is being successfully used in the wild, a significant portion of sophisticated malware may be going undetected.

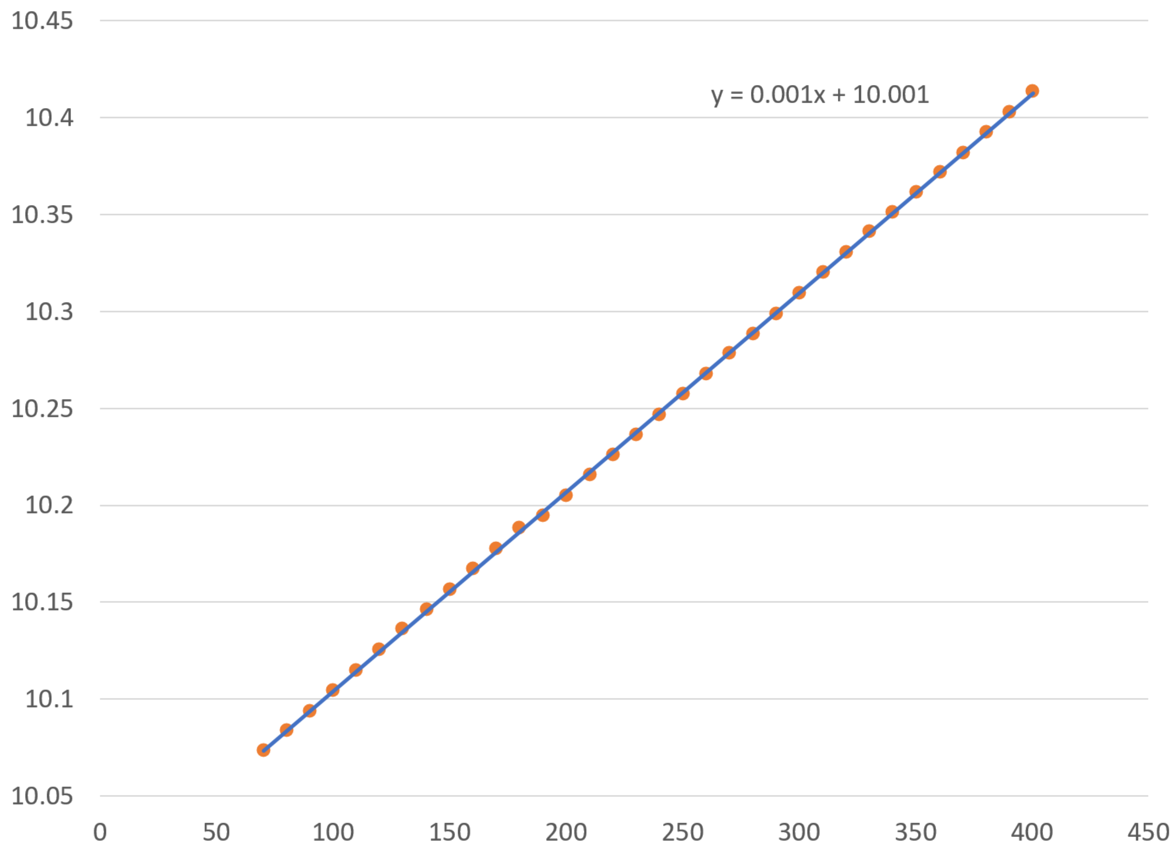

Figure 7.8: Clock skew variance for Emma, Bare Metal with least linear squares fit line and equation and low MSE. X-axis is time since start of packet capture (in seconds). Y-axis is clock skew offset (in seconds).

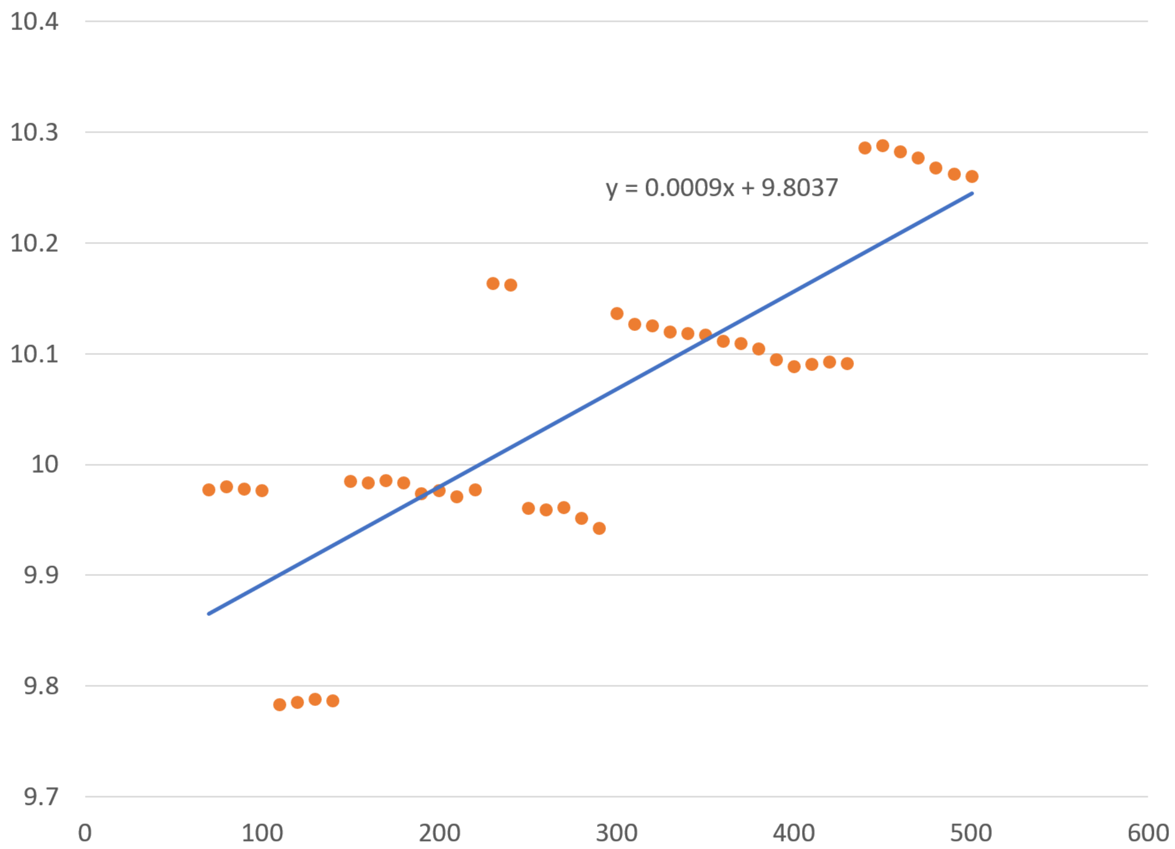

Figure 7.9: Clock skew variance for Xen VM running on Emma with least linear squares fit line and equation and high MSE. X-axis is time since start of packet capture (in seconds). Y-axis is clock skew offset (in seconds).

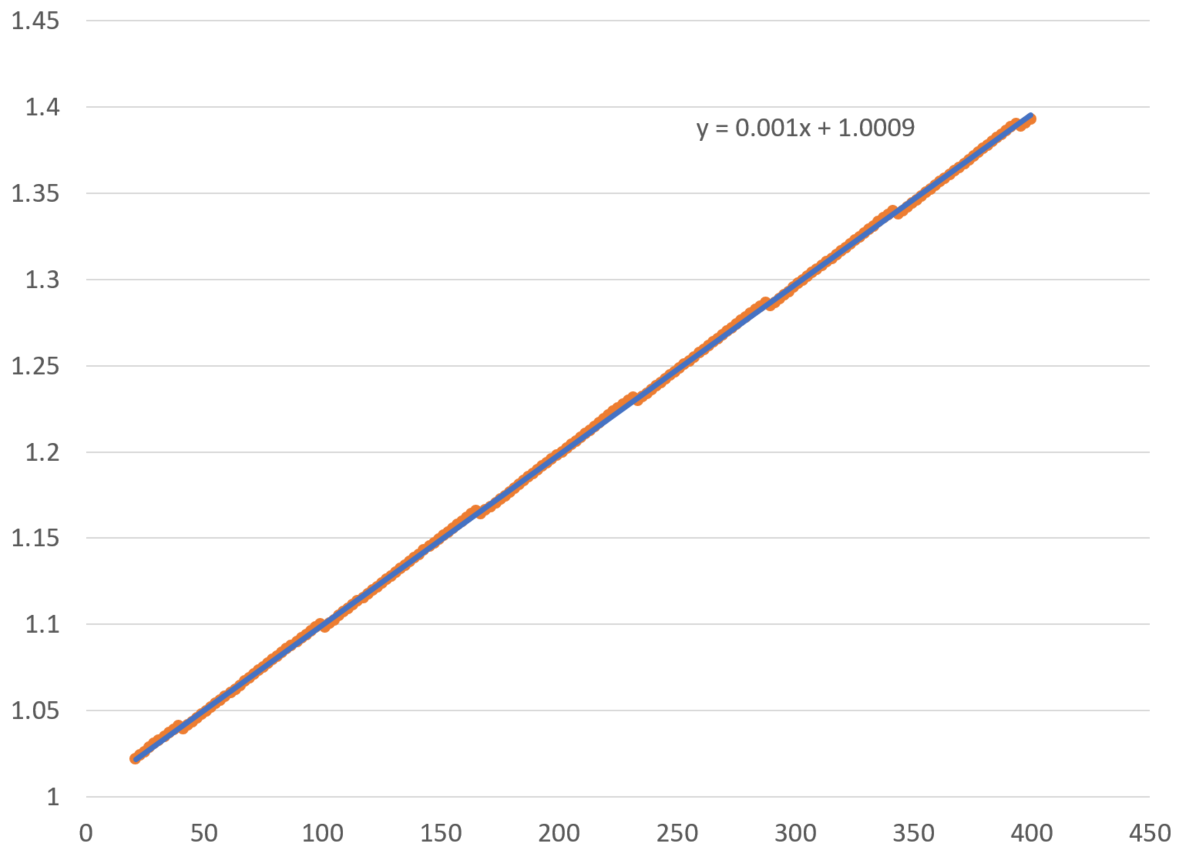

Figure 7.10: Clock skew variance for Linux container running on Dani with least linear squares fit line and equation and low MSE. X-axis is time since start of packet capture (in seconds). Y-axis is clock skew offset (in seconds).

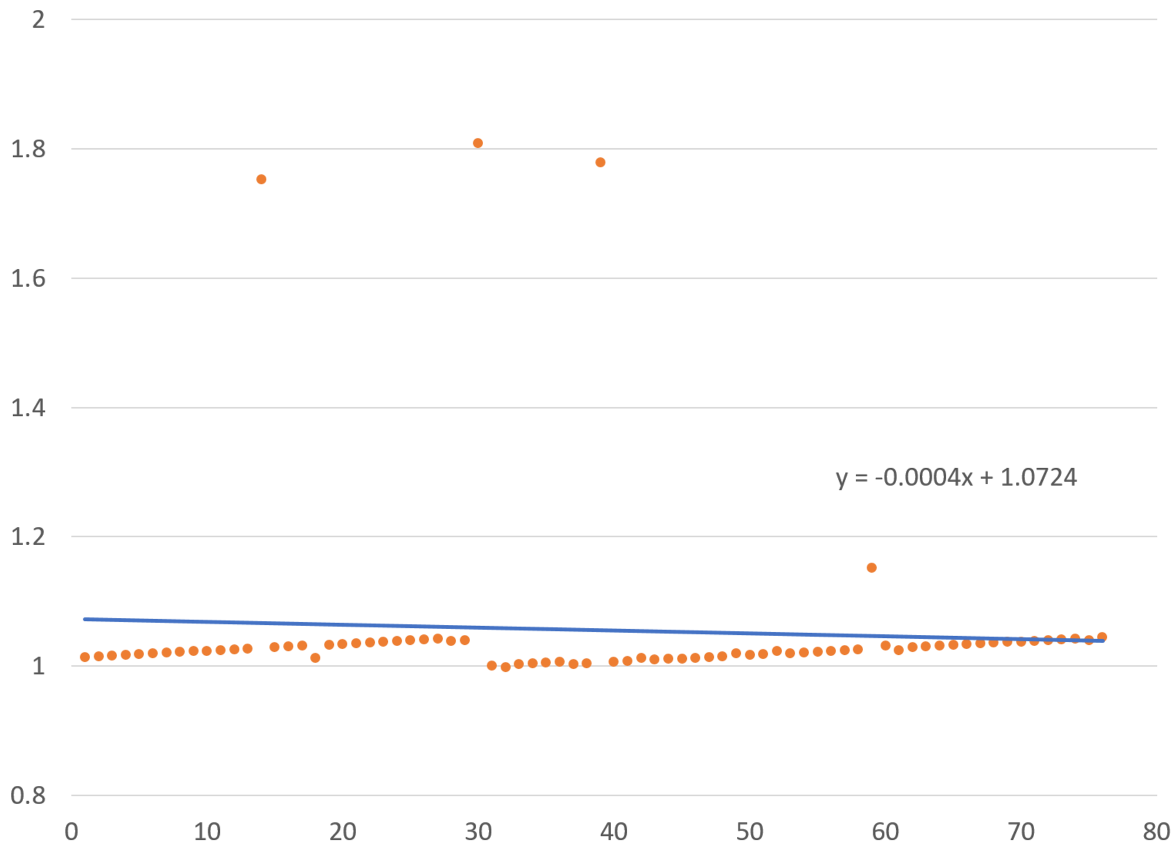

Figure 7.11: Clock skew variance for VMWare VM running on Dani with least linear squares fit line and equation and high MSE. X-axis is time since start of packet capture (in seconds). Y-axis is clock skew offset (in seconds).

7.1.8 Experiment Conclusion

After reviewing results from a variety of virtual environment detection methods on several different systems, a few observations can be made. Firstly, generic virtual environment detection is a not a simple or straightforward operation. As demonstrated in the previous section, some virtual environments are inherently detectable to some methods while others are not. Additionally, virtual environments may also be made more transparent by masking some of their tell-tale signs, such as default MAC addresses or hypervisor advertisement returned by the CPUID instruction. However, some tests may be significantly more difficult to defeat, such as instruction execution timing, which leaves virtual environments inherently detectable to sophisticated detection methods.

In regards to the ability of Linux containers defeating such environment detection methods, these test results are promising. Aside from a tell-tale network bridge default MAC address, containers operating on x86 systems were able defeat each of the attempted detection methods. Due to the realities of freshly maturing software on niche hardware, Linux containers running on Ubuntu Core flagged a number of detection methods. However, this does not appear to be an inherent drawback of Linux containers given their performance on x86 systems; rather a limitation of Ubuntu Core as it has slightly different environment goals than standard systems. As recent Raspberry Pi models are now supported by the mainline Linux kernel, an unmodified Ubuntu server operating system is expected to bring Linux containers back to Raspberry Pis without the need for Ubuntu Core, with expected performance in defeating detection methods.

As stated earlier in the thesis, the tests performed in this experiment are not derived from the source code of VM-aware malware, but instead were written to be representative of a large number of possible tests that could be performed against specific areas of virtualization. As such, these tests have not be rigorously analyzed for correctness, but the source code is available in Appendix A for reference and review.

It should be noted that several aspects of the work Chen, et al., presented in [14] regarding virtual environment detection using TCP packet clock skew are concerning. Due to the inconsistent results and inherent difficulty in recreating their employed method, it is believed that Chen, et al., did not include all necessary details of their experiment, such as software and operating system versions, virtual machine configurations, and the target system state during the test. As the tests performed in this study did not produce the expected results, it is difficult to determine if the discrepancy is resulting from an incorrect re-implementation of Chen, et al.'s experiment, if operating system and virtual machine technology sufficiently evolved since the paper's publication to no longer make this test possible, or if the results illustrated by Chen, et al.'s experiment were not representative of actual system behavior. Additional research needs to be performed in this area to identify the underlying disconnect.

Chapter 8

Benchmarking Variability

Recent investigations have begun in understanding the inherent variability of computer systems across a number of performance metrics. In support of this study, benchmarks were performed on various systems to determine if changes in performance variability were able to reveal information that may be useful to an attacker. The two areas that have been explored up to this point are disk I/O throughput and CPU cache latency.

8.1 Disk I/O Throughput

The disk I/O throughput benchmarking tool *iozone* [39] was used to collect raw disk performance data of a Raspberry Pi and compare it to the disk performance data of commercial servers collected by Uday Ananth. Visualization of results were generated using visualization tools developed by Thomas Lux.

Work is still ongoing and no conclusive features can yet be drawn from the collected data as disk performance variability is a challenging task to study statistically. However, while the statistical models are still being developed, a preliminary analysis of the data from a deception-defeating stand-point can provide a few indicators where an attacker could possibly identify the type of system their malware is executing on based on disk I/O performance. This is a significant concern in scenarios where the true nature of a system should not be disclosed.

Initial observations of the data revealed that disk reads demonstrated greater variability than disk writes. Due to this fact, preliminary focus is placed on the *fread* benchmarks performed by iozone. *Fread* simply performs disc reads in records of an assigned size until it has finished reading the file of an assigned size.

There are some inherent hardware differences between the two system architectures that provide clear explanations as to why the systems perform differently, such as disk media

and I/O bus speed; however, the goal of this section is to analyze differences in performance characteristics that may allow an attacker to identify the platform they are executing on. Specifically, if an attacker has penetrated a high-interaction honeypot running on a Raspberry Pi within a Linux container and cannot use traditional virtual environment detection methods to reveal the existence of the honeypot, what performance characteristics of the system can be used to defeat the honeypot's deception?

A quick comparison between Figures 8.1 and 8.3 shows that Raspberry Pi *fread* throughputs don't break 3MB/s where the baseline server *fread* throughput exceed 5 MB/s. Generating an *fread* throughput benchmark would allow an attacker to identify the top throughput of the system disk and, if below a specific threshold, assume they are on a low-power device such as the Raspberry Pi.

A straight forward way to defeat detection by throughput measurement would be to modify the system clock during I/O operations, similar to how some VMs may adjust their system time while executing certain instructions to defeat a timing test. While this would be significantly more difficult to implement for a Linux container than a virtual machine as the container is using the host system clock for reference, it remains a viable method to defeat this form of detection.

However, besides simple throughput differences, another feature can be noted. Comparing the differences in I/O performance characteristics across three different record sizes with a file size of 1024kB, distinct features can be identified. Figure 8.5 shows that on the baseline server, significant overlap occurs in throughput with record sizes of 32 and 128 kB, with some overlap occurring with record sizes of 512 kb. However, on the Raspberry Pi, as shown in Figure 8.6, no overlap in throughput occurs between the three different record sizes. No simple modifying of the system clock during I/O operations would be able to mask the differing performance characteristics. As such, this becomes a viable method to defeat platform-related deception.

The causes of these varying performance characteristics hasn't yet been sufficiently explored and simple modifications made to the Raspberry Pi, such as using an external, USB hard disk drive for storage, may mask some of these tell-tale signs.

8.2 CPU Cache Latency

Side channel attacks have been an area of unique interest to security researchers, as they provide an avenue for information to be leaked from a system using a method that is wholly unintended for such purposes. These side channels vary from analyzing power consumption of a CPU [37], to listening to the sounds a computer makes with a cellphone microphone [31]. These types of side-channel attacks are incredibly difficult to defend against because it involves artifacts introduced into the system as a byproduct of running the code - even if the software is entirely secure in it's implementation.

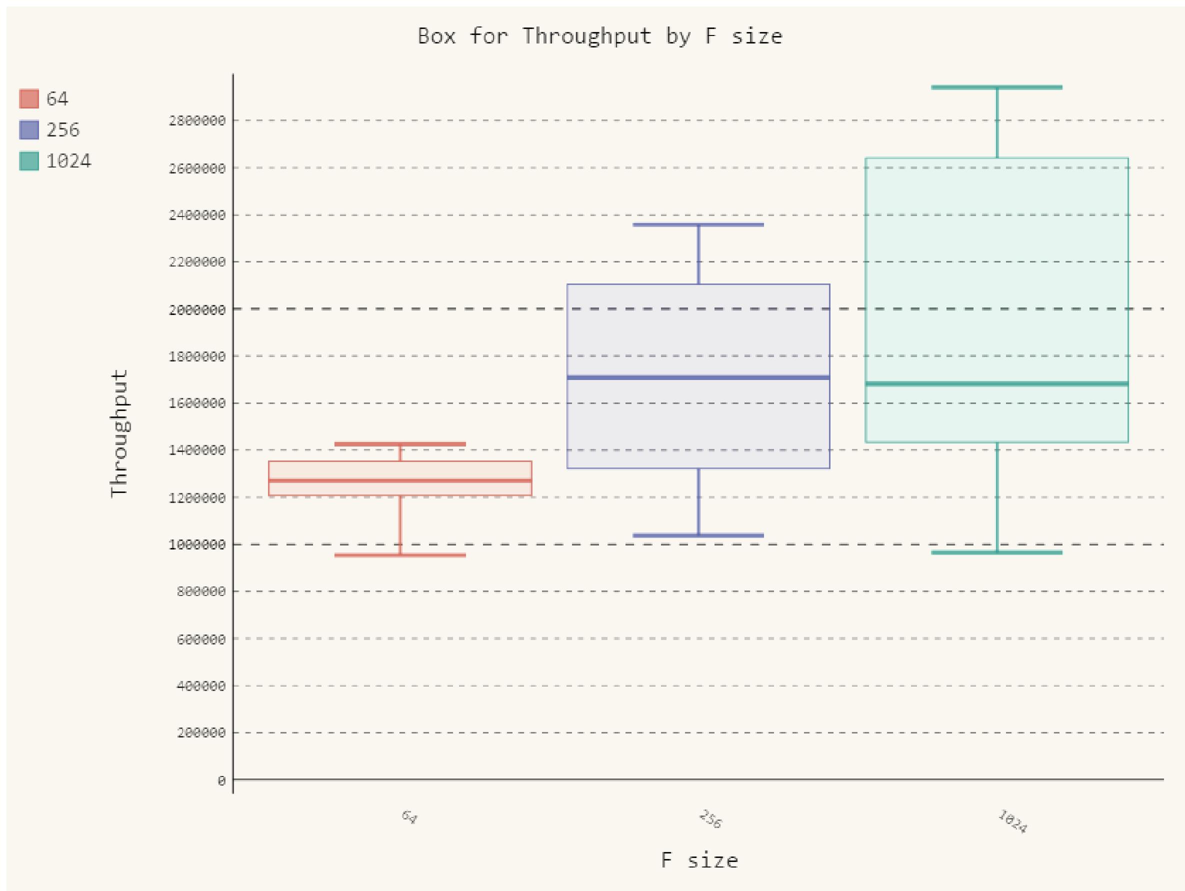

Figure 8.1: Box Plot for iozone FRead throughput benchmark by file size on Raspberry Pi 3 running with 2 threads at CPU frequency of 1.2GHz. System is using Deadline I/O scheduler. X-axis is file size in kB. Y-axis is throughput in kB/s.

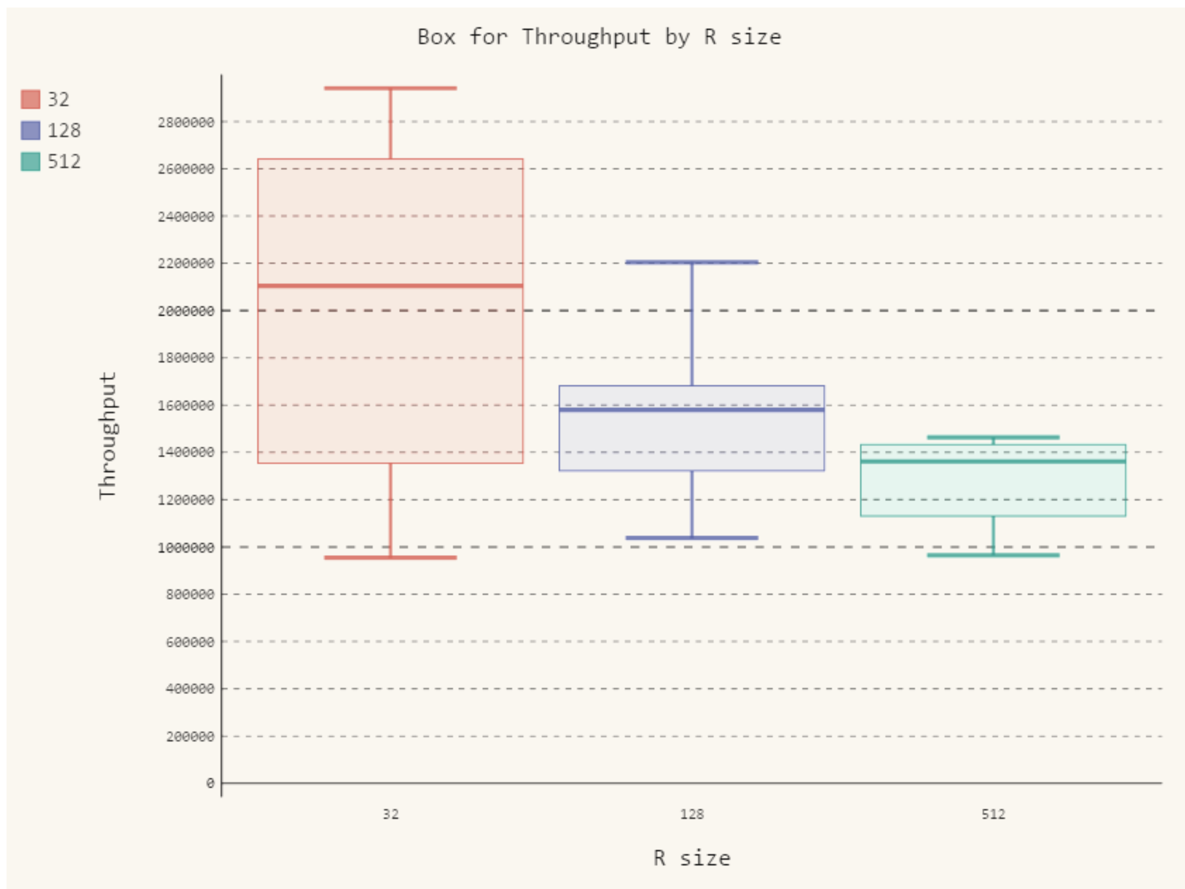

Figure 8.2: Box Plot for iozone FRead throughput benchmark by record size on Raspberry Pi 3 running with 2 threads at CPU frequency of 1.2GHz. System is using Deadline I/O scheduler. X-axis is record size in kB. Y-axis is throughput in kB/s.

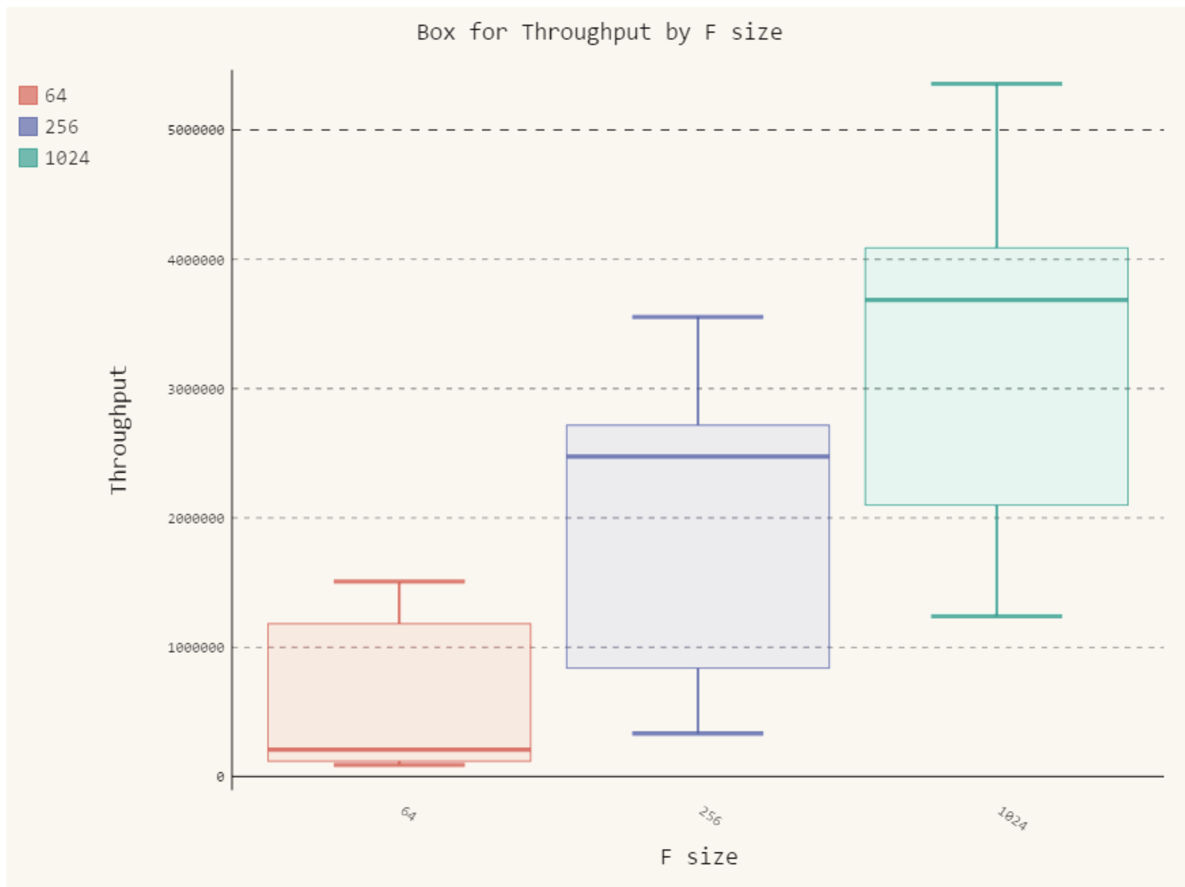

Figure 8.3: Box Plot for iozone FRead throughput benchmark by record size on server running with 2 threads at CPU frequency of 1.2GHz. Both Hypervisor and VM are using Deadline I/O Scheduler. X-axis is file size in kB. Y-axis is throughput in kB/s.

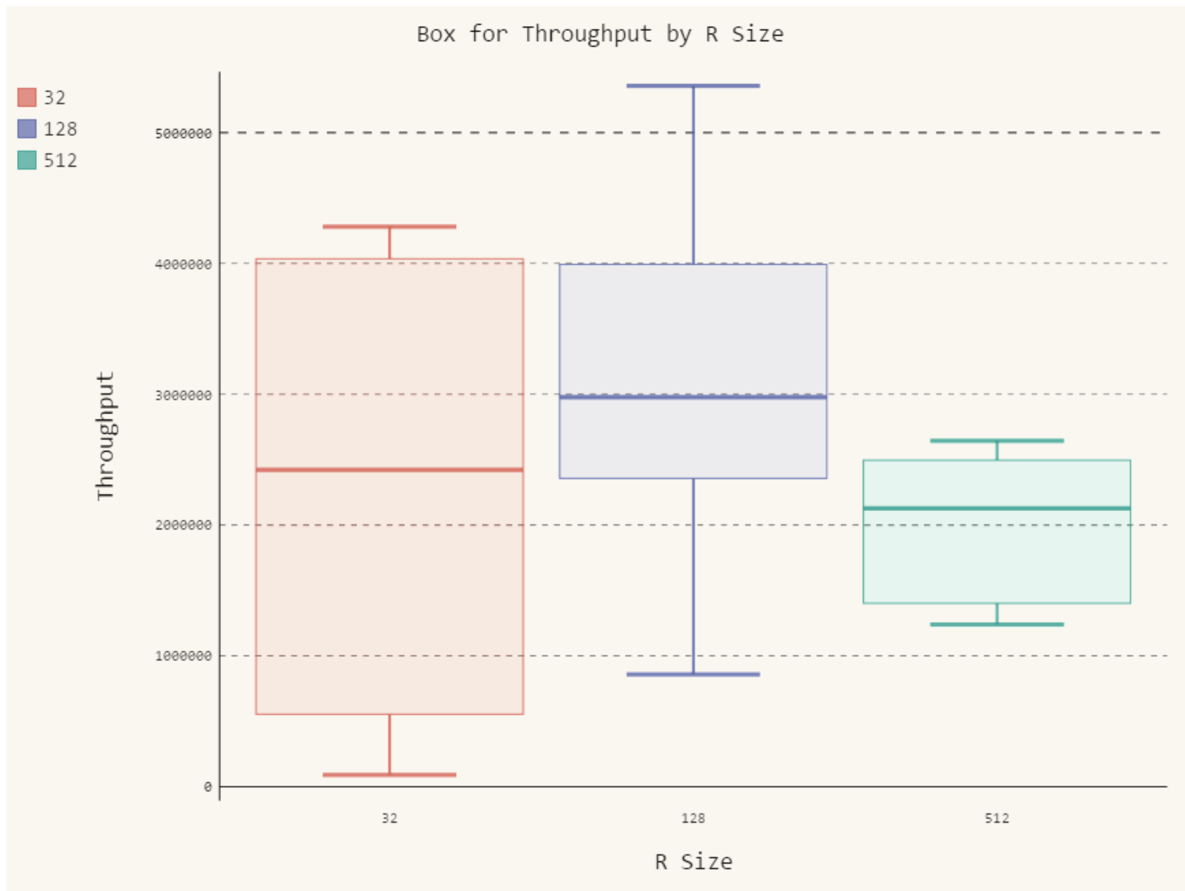

Figure 8.4: Box Plot for iozone FRead throughput benchmark by record size on server running with 2 threads at CPU frequency of 1.2GHz. Both Hypervisor and VM are using Deadline I/O Scheduler. X-axis is record size in kB. Y-axis is throughput in kB/s.

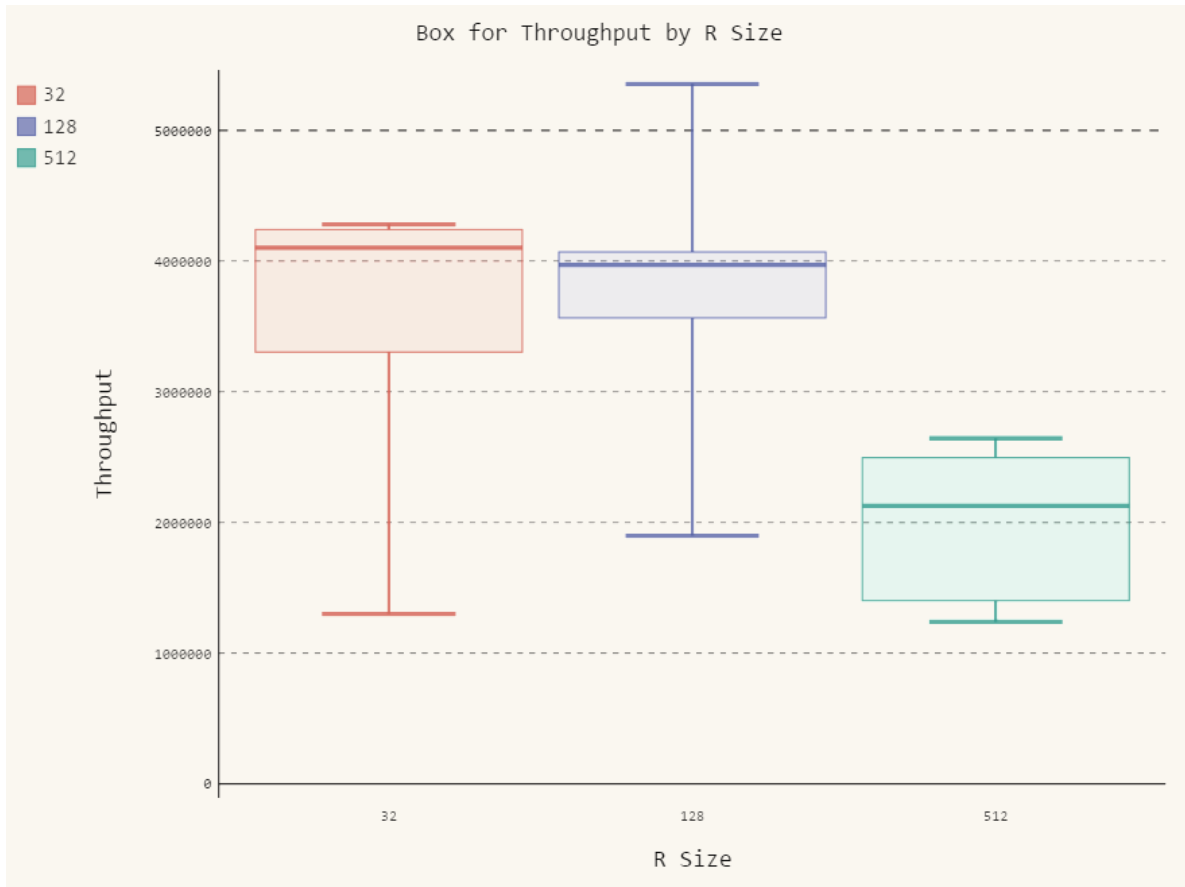

Figure 8.5: Box Plot for iozone FRead throughput benchmark by record size on server running with 2 threads at CPU frequency of 1.2GHz with filesize of 1,024kB. Both Hypervisor and VM are using Deadline I/O Scheduler. X-axis is record size in kB. Y-axis is throughput in kB.

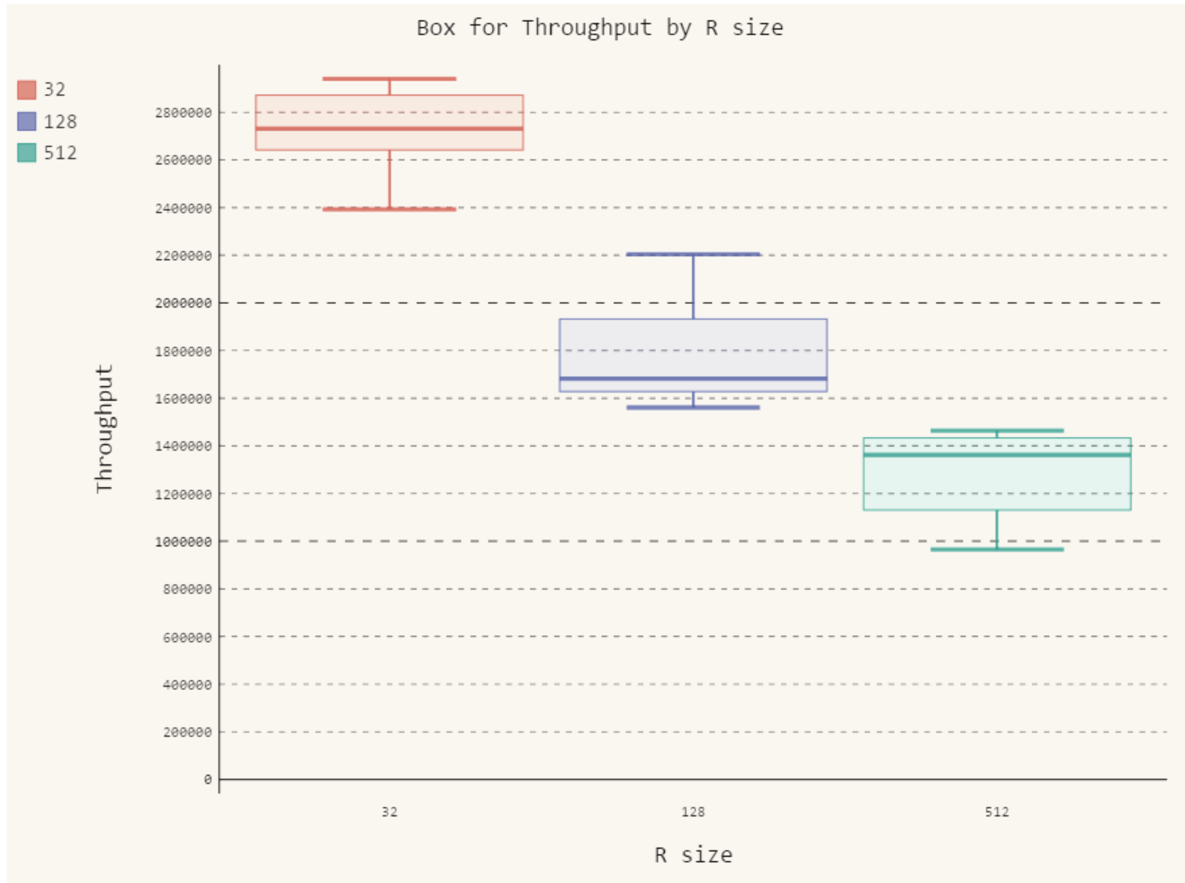

Figure 8.6: Box Plot for iozone FRead throughput benchmark by record size on Raspberry Pi 3 running with 2 threads at CPU frequency of 1.2GHz with filesize of 1,024kB. Both Hypervisor and VM are using Deadline I/O Scheduler. X-axis is record size in kB. Y-axis is throughput in kB.

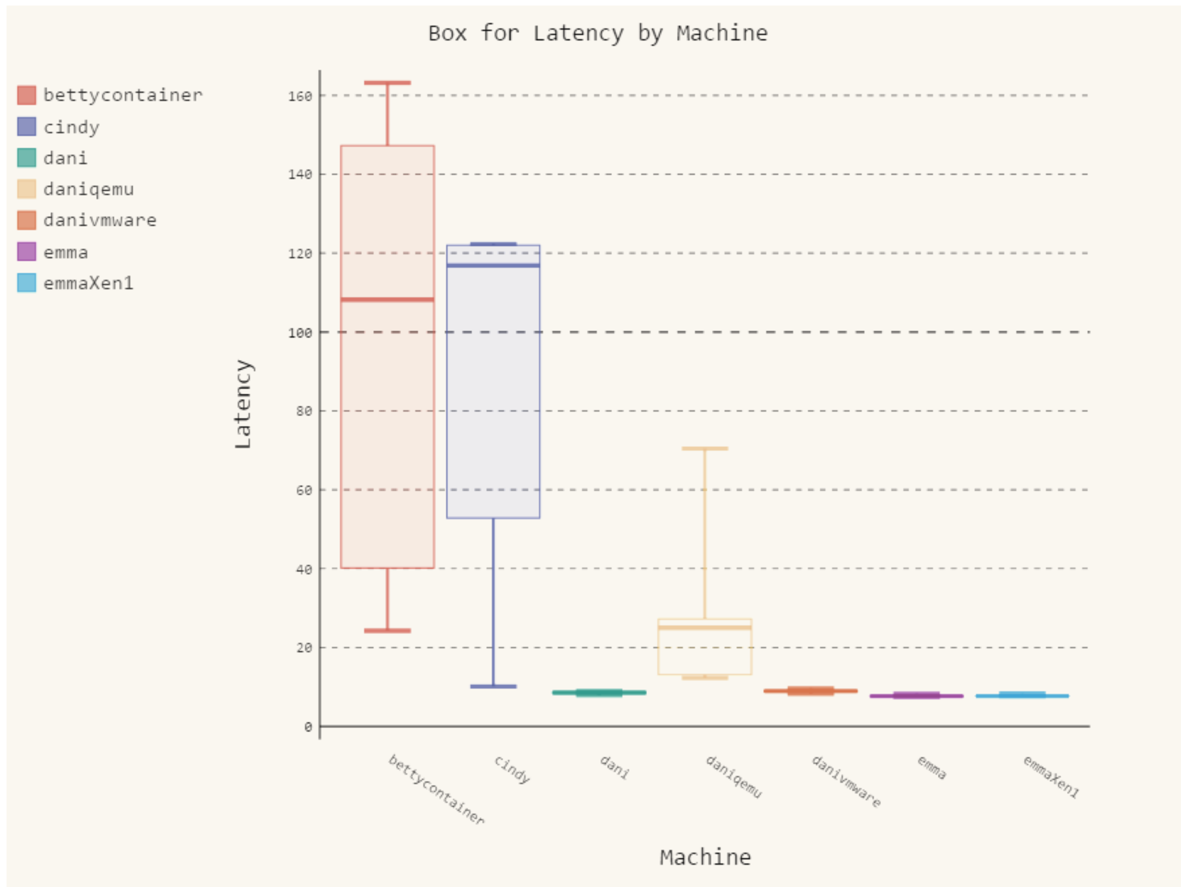

Figure 8.7: Box Plot illustrating variance in CPU cache latency by machine. X-axis is the system/VM. Y-axis is latency in nanoseconds.

A study was initiated to investigate whether cache latency benchmarks may be able to reveal the presence of operating within a virtual environment and if variability characteristics may be able to identify the hypervisor in use. Visualization of results were generated using visualization tools developed by Thomas Lux.

For these benchmarks, the tool *lmbench* was used to collect data sets. Despite it's age, the robustness of it's tests and the ease of implementation continue to make it a popular choice when measuring system cache performance.

Making a wide survey on a select subset of benchmark results illustrated in Figure 8.7, it can be seen that there is a wide spread both in cache latency performance and variance across all of the tested systems. This data subset is for data arrays of between .35MB and 1.5MB in size. Data of this size is typically serviced by a system's L2 cache.

Comparing Dani bare metal cache latency performance against a VMWare virtual machine running on Dani, a noticeable difference in performance can be noted, as illustrated in Figure

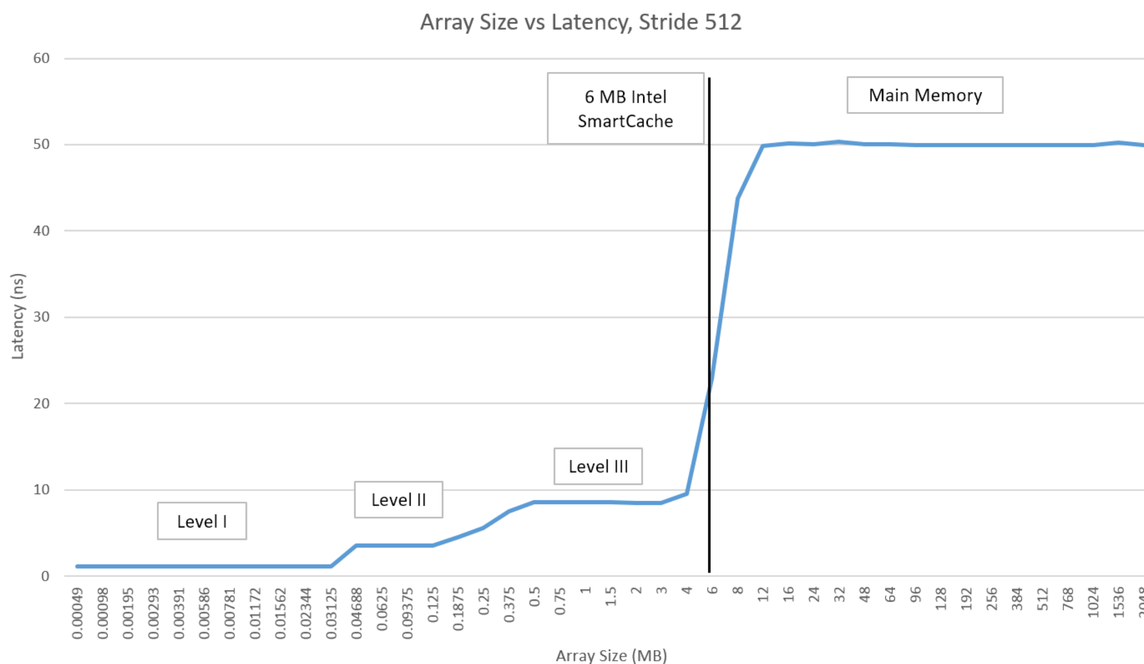

Figure 8.8: Line graph illustrating cache latency over range of array sizes with read stride length of 512kB on Dani. X-axis is the array size in MB. Y-axis is latency in nanoseconds.

8.9. While only an approximate half nanosecond average increase is gained while running VMWare, a feature that may be very difficult to detect reliably, VMWare does show a number of cache latencies values that are nearly a full nanosecond slower than the average bare metal latency performance. Additionally, comparing the slowest VMWare cache latency values with the fastest bare metal latencies, there is an increase in over 1.5 nanoseconds, potentially a much more detectable difference.

Another data excerpt from the ongoing study compares the cache latency performances between Emma and a paravirtualized Xen virtual machine running on the same system, showing in Figure 8.10. This comparison appears to be more stark in its appearance as the Xen virtual environment not only shifts the average cache latency higher, but also appears to change the distribution of cache latency values. On bare metal Emma, there is a strong clustering of cache latencies between 7.5 and 7.7 nanoseconds with a small number of outliers with higher values; however, the 95th percentile of collected values remains below 7.7 nanoseconds. The values collected on the Xen virtual environment indicate a split clustering, with all lower values strongly centered around 7.68 nanoseconds and all higher values strongly centered around 8.32. Visual inspection indicates that the distribution of the data from the Xen virtual environment follows a significantly different model than the data collected from bare metal Emma. While the timing differences in cache latency appear to all fall within a single nanosecond, the disparity between the two distributions makes a

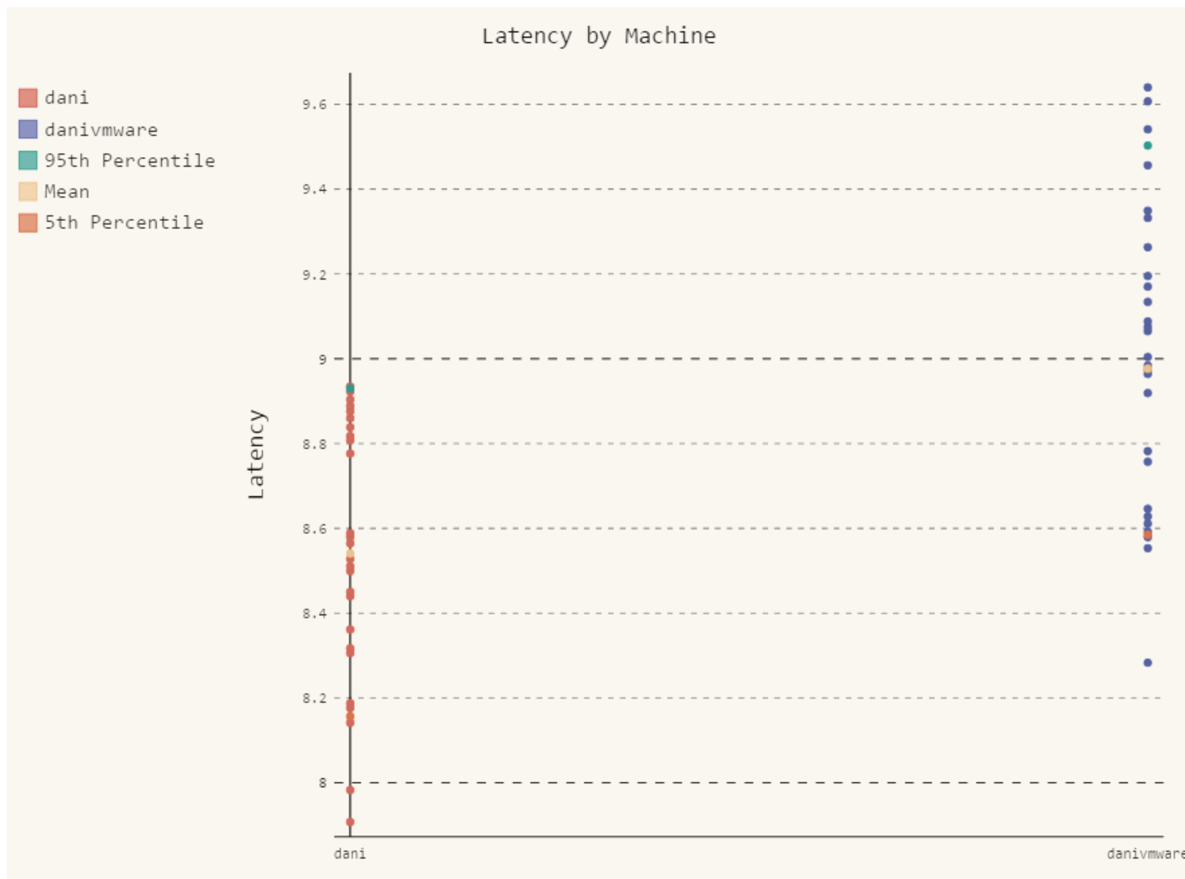

Figure 8.9: Plot comparing cache latency between bare metal Dani and VMWare virtual machine running on Dani with read stride length of 512kB. X-axis is the machine. Y-axis is latency in nanoseconds.

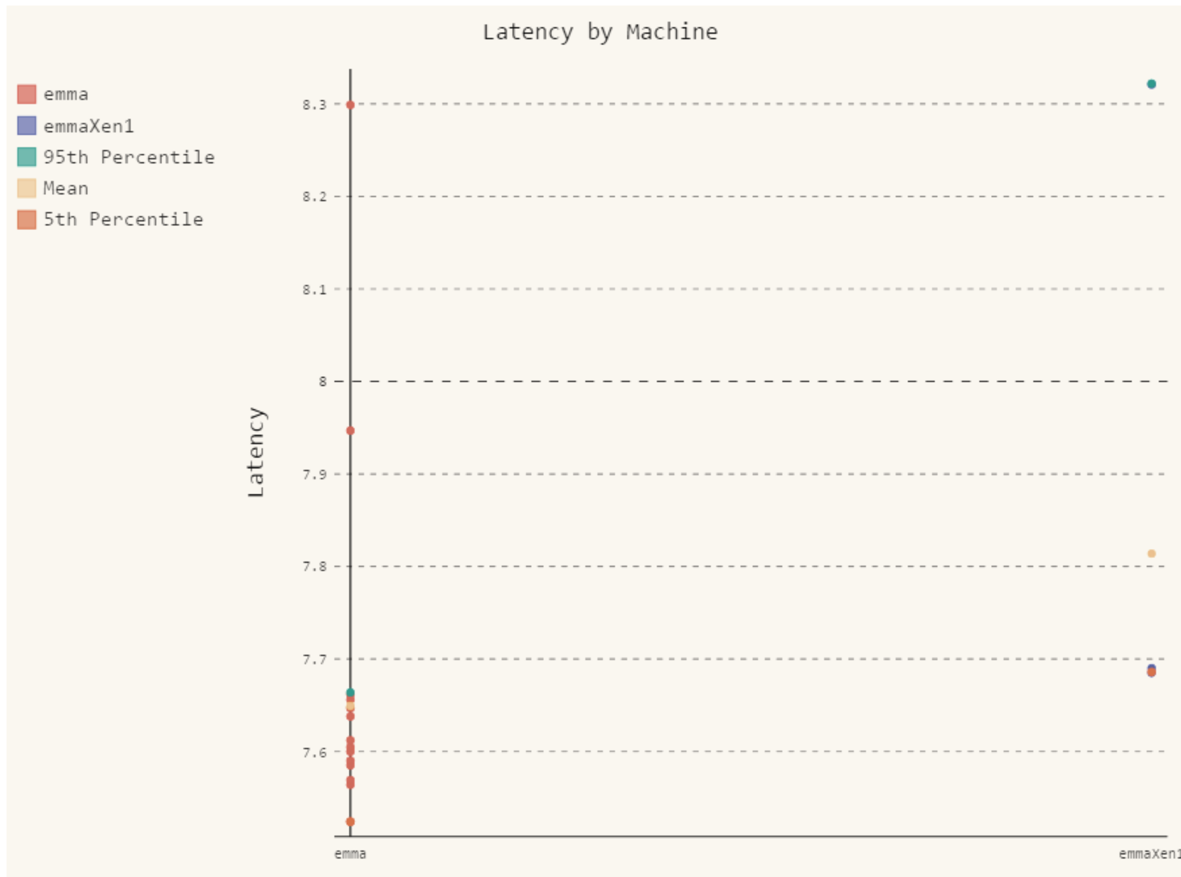

Figure 8.10: Plot comparing cache latency between bare metal Emma and Xen virtual machine (PV) running on Emma with read stride length of 512kB. X-axis is the machine. Y-axis is latency in nanoseconds.

telling feature that could potentially be detectable by an attacker.

Much more statistical analysis needs to be performed on the collected data, however the data currently gathered indicates the presence of a VM impacting cache latency in a detectable manner. If this conjecture is supported with additional study, it provides malware authors with a generic, cross-platform approach to discovering the presence of a virtual machine. While this detection method is not as simple as a 'red pill', due to cache latency reference data needing to be included in the detection test for comparison, it is assumed to be an extremely difficult virtual environment marker to mask. A long term goal of this study is whether the impact virtual machines have on cache latency is unique to each virtual machine; if so, that makes virtual machine software potentially identifiable, providing a side-channel information leakage that may assist attackers in crafting targeted attacks against specific virtual environment implementations.

Chapter 9

Container Detection

Adopting Linux container usage as a means to defeat virtual environment detection tests leads to a follow-on question: What about Linux container detection tests? This is a valid point and bears strong consideration. As virtual environment detection tests generally focus on discovering disparities between what a bare metal machine will look and behave like versus what a virtual machine looks and behaves like, Linux container detections tests would seek to identify traits or characteristics that would be different when executed inside a container versus bare metal. Several of the previously discussed virtual environment detection tests seek to identify features such as:

- Do CPU instructions take longer than they're suppose to?
- Does the CPU provide information that is different than what was expected?
- Does the system hardware contain identification that is only found in virtual machines?

However, as Linux containers are managed by the operating system kernel and execute directly on the hardware, many of the virtual environment detection methods outline in Chapter 6 are not effective against Linux containers. What attackers will need to focus their attention on are possible gaps in the isolation methods employed by Linux containers in order to identify possible 'chinks in the armor'.

9.1 Linux Namespaces

A perfect example of such a chink was identified in late 2016 being employed by the VLANY LD_PRELOAD rootkit malware source code that was published online [57]. The VLANY rootkit is designed to contain a large number of stealth and anti-detection features, one of which is a technique to identify if the code is running within an LXC container by means of

comparing the number of processes listed by the PS command and the number of processes provided by SYSINFO.

This test identified a lapse in coverage by Linux kernel namespaces in that PS lists only the processes owned by the container, yet still enumerating the total number of processes running on the host with the SYSINFO command. If PS provides a significantly smaller process count than SYSINFO, then it is assumed that PS is restricted by user namespaces and hiding the true number of total processes. The results of the LXC detection test are given in Table 9.1 and can be seen to accurately identify when it is being run within a Linux container on a standard Ubuntu installation. Interestingly, the test is not accurate when being run within a Linux container on Ubuntu Core, as the total number of system process returned by SYSINFO is significantly lower than that of a standard Ubuntu installation and does not provide a wide enough gap with the number of processes reported by PS to be identified as a Linux container. It is assumed the differences in process count reported by SYSINFO in Ubuntu Core and standard Ubuntu is that Ubuntu Core is a heavily stripped down variant of Ubuntu, designed to run on low-performance or embedded systems, and has far fewer number of processes running in the background automatically.

It is anticipated that SYSINFO would need to be updated to account for kernel namespaces in order to prevent this type of discovery.

9.2 Permissions

Another area in which Linux containers may be revealed is in user permissions. Through the use of containers, there arises a unique condition in which a user may be in possession of root privileges in a container, but the container itself is unprivileged. While within the container namespace, the user may be operating with a UID of 0, the user's actual UID may be something like 3000. This generates a scenario in which a user with root privileges in a container may attempt to perform an operation on the system that they expect to be able to perform, but are prevented.

In order to test this detection method, the previously stated restriction on unprivileged detection tests will be loosened as the premise of this detection method is to identify when a user doesn't have root access when the system is telling them they do.

The Linux command DMIDECODE was selected as a means to test for actual root permission. DMIDECODE decodes and dumps the Desktop Management Interface (DMI) table and output it to the console. As the DMI table accesses system info, the kernel requires root privileges in order to access those resources. A privileged user with root access, when executing the command DMIDECODE will be given a large output of text outlining various characteristics of the system. A user without root access will simply be informed 'Permission denied'.

A non-standard response occurs when running DMIDECODE on Ubuntu Core as it does not

System	Namespace Test	Permission Test
Amy	No	No
Betty Container	No	Yes
Cindy	No	No
Cindy KVM	No	No
Cindy Container	No	Yes
Dani	No	No
Dani VMWare	No	No
Dani QEMU	No	No
Dani KVM	No	No
Dani Container	Yes	Yes
Emma	No	No
Emma VMWare	No	No
Emma QEMU	No	No
Emma KVM	No	No
Emma Xen PV	No	No
Emma Xen HVM	No	No

Table 9.1: Results of Container Detection Tests

operate using the standard features and privileges expected of a standard Linux OS. As such, a container running Ubuntu 16.04 isn't provided with a /dev/mem file; attempting to run the command DMIDECODE fails. By testing for this scenario not only is the presence of a container revealed, but the existence of Ubuntu Core as the host operating system as well.

Defeating permissions-related container detection tests remains an open problem. A possible solution would be for privileged users within an unprivileged container to be granted read-only permissions in order to access system resources. Unfortunately this will not be an effective counter-measure for permissions checks that seek to modify system aspects such as the CPU governor, which a user with root access would expect to be able to do. As such, at this time there is no effective means to defeat this Linux container environment test for an attacker with root access.

As shown in Table 9.1, the tests specific to revealing the presence of containers appear to be mostly accurate, particularly if the namespace test was adjusted to not allow such a large disparity in reported process numbers between PS and SYSINFO that allows minimal installation OSs with low numbers of running processes to go undetected. Checking for permissions was demonstrated to have the best accuracy for container detection. Fortunately, performing such a check would require an attacker to already have root privileges; in that case, if a privilege escalation attack was performed, then the container would already have observed at least a portion of the attack before the deception was defeated.

Source code for these tests is provided in Appendix A.9.

9.3 Raspberry Pi Detection

The popularity of Raspberry Pi use for low-interaction honeypots was one of the primary reasons prompting the research performed in this study; however, such popularity has a cost as continued use of Raspberry Pis for honeypots, particularly employing Linux containers to enable High-Interaction honeypots, will most likely result in them becoming a targeted platform by attackers to unmask honeypot deception.

Given a fictitious scenario where all operating system markers flagging the honeypot as operating on Raspberry Pi are somehow masked, there are still performance differences inherent to the system hardware, as discussed in Chapter 8, that may still be used to reveal the honeypot's identity.

One such example, as mentioned in Section 7.1.5, the Raspberry Pi lacks a PCI bus to generate output for the command LSCPU, making for a very quick detection.

While not specific to a Raspberry Pi, similar low-power devices will most likely employ microSD cards as their primary storage devices. In order to detect such a storage device, two observations from Chapter 8 were translated into detection tests: disk read throughput and comparison between disk read throughput of different record sizes.

Fread throughput of a specific record size appears to be characteristically slower on the Raspberry Pi than on a standard server, and, if tested results are slower than a defined threshold, can flag the use of a microSD card as the system storage drive.

However, as disk throughput can potentially be altered by controlling the system clock during I/O processes, a more sophisticated test was devised to compare the difference in reported throughput between disk reads using different record sizes. Figure 9.1 shows that on the Raspberry Pi there is no overlap between the throughput of disk reads of record size 32kB and 128kB when reading from a 1 MB file, with smaller record size reads always having higher throughput. Figure 9.2 shows the server demonstrating an inverse on throughput performance between the two record sizes.

Raspberry Pi detection test results are listed in Table 9.2. Disk throughput measurement demonstrated itself as an accurate detection method, even detecting Cindy's use of microSD card for storage, with false positives reported only for QEMU emulators; again, this is most likely attributed to QEMU slow system performance and that the test used 'gettimeofday' rather than CPU ticks as the timing source. Unfortunately, comparing the throughput for different record sizes did not prove itself nearly as a reliable. This is attributed to additional I/O variance being introduced in the detection test that was not present in iozone's benchmarking. Further investigation in this detection method may reveal more accurate reporting thresholds and grant a microSD card storage detection test that is not susceptible to altered system clocks.

Source code for these tests is provided in Appendix A.10.

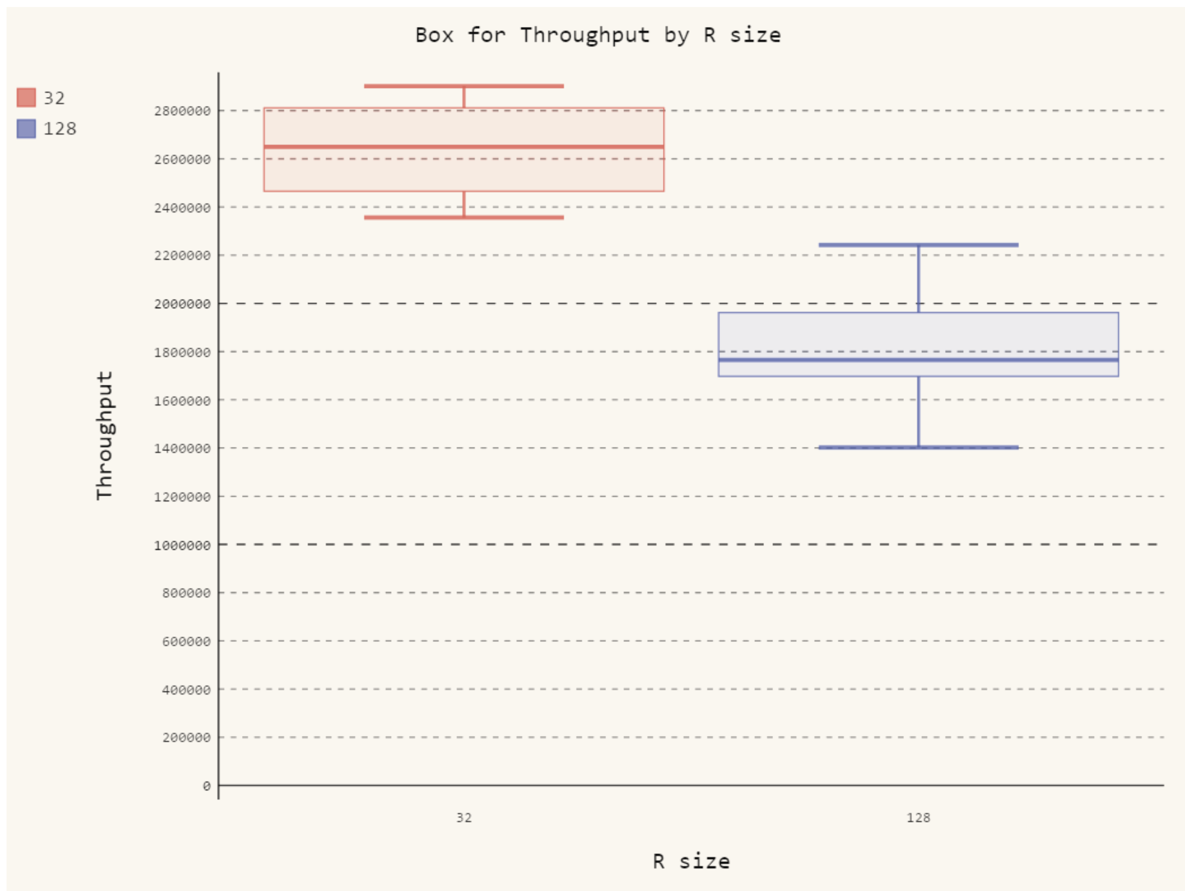

Figure 9.1: Box Plot for iozone FRead throughput benchmark by record size on Raspberry Pi with 1 thread at CPU frequency of 1.2GHz with filesize of 1,024kB. Kernel is using Deadline I/O Scheduler. X-axis is record size in kB. Y-axis is throughput in kB.

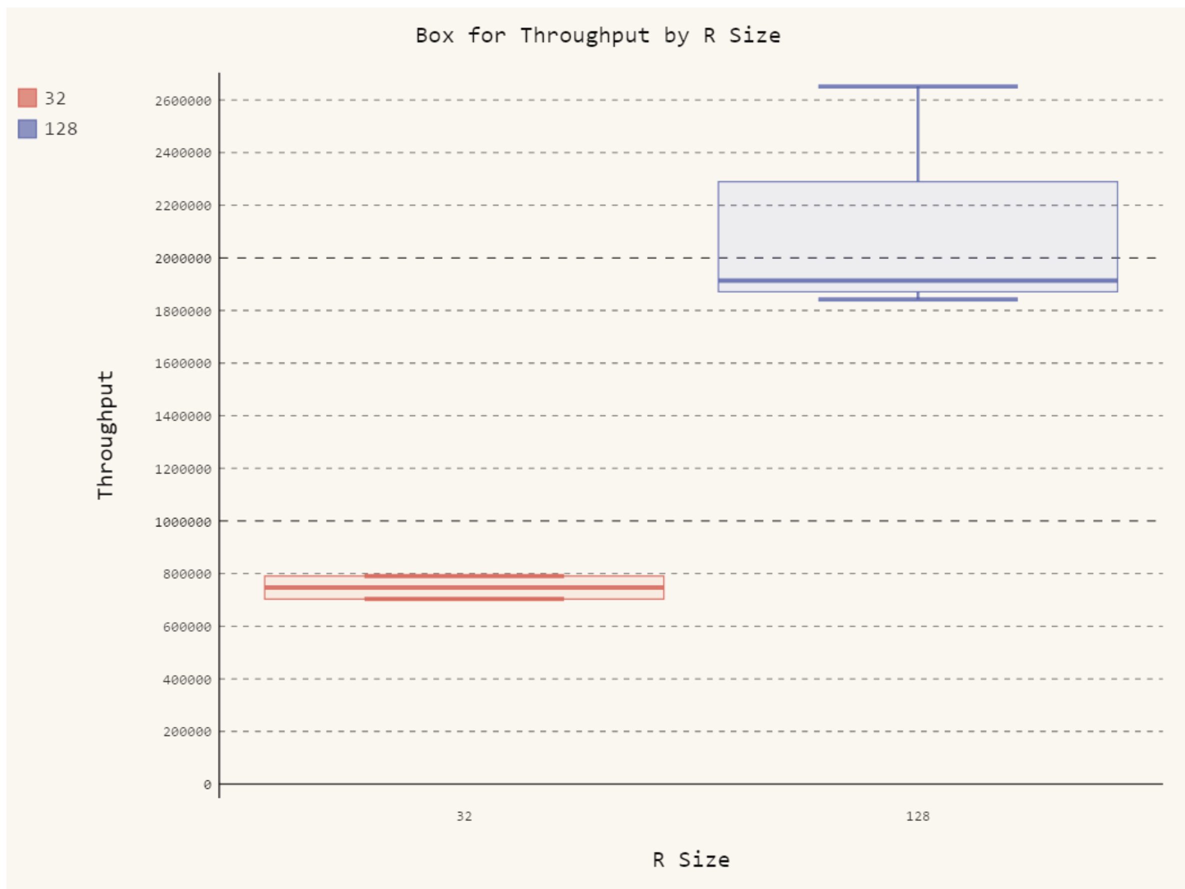

Figure 9.2: Box Plot for iozone FRead throughput benchmark by record size on server with 1 thread at CPU frequency of 1.2GHz with filesize of 1,024kB. Both Hypervisor and VM are using Deadline I/O Scheduler. X-axis is record size in kB. Y-axis is throughput in kB.

System	Disk Throughput	Record Size Throughput
Amy	Yes	No
Betty Container	Yes	No
Cindy	Yes	No
Cindy KVM	Yes	No
Cindy Container	Yes	No
Dani	No	No
Dani VMWare	No	No
Dani QEMU	Yes	No
Dani KVM	No	No
Dani Container	No	No
Emma	No	No
Emma VMWare	No	No
Emma QEMU	Yes	No
Emma KVM	No	No
Emma Xen PV	No	No
Emma Xen HVM	No	No

Table 9.2: Results of Raspberry Pi Detection Tests

9.4 Analysis Tool Detection

Discussions have been made regarding malware testing for the presence of debugger software in conjunction with, or in replacement of, virtual environment detection methods [5] [14] in an attempt to thwart monitoring and analysis of their attacks. This is a logical approach as the use of virtual environments has grown tremendously in professional applications; to avoid attacking virtual environments would be to limit an attacker from going after the potentially most profitable targets.

In [40], Jiang, et al., discussed the trade-off with “in-host” malware detection and monitoring compared with “out-of-the-box” analysis. As “in-host” monitoring occurred within the virtual system being attacked, the analysis tools were able to retrieve the greatest amount of semantic detail, but were susceptible to detection or modification, while “out-of-the-box” monitoring occurred out of the targeted virtual environment, being far less likely to detect and modify, but potentially losing a significant amount of semantic detail about the attack. Jiang, et al., proposed a solution where the hypervisor is able to reconstruct information “out-of-the-box” without monitoring occurring “in-host”. While the presented results appeared promising, in the years since the paper’s publication (2007), this type of “out-of-the-box” malware analysis is not yet freely available for use by the security community.

Fortunately, Linux containers, by use of their unique features, may present a much less technologically challenging solution than what was presented in [40], but with similar results.

As an example of a useful tool available for debuggers is the command STRACE, which allows the kernel to output the system calls a process and it's children make. This is particularly useful for malware analysis as it allows researchers to monitor the behavior of a malicious binary at the system call level, providing a great amount of detail on what actions the malware is taking.

Unfortunately, a very simple test is available for malware to determine whether it is being monitored by STRACE - attempting to use STRACE on itself or one of it's child processes. Due to an implementation restriction in the tool, STRACE is not able to be called recursively and fails, alerting the malware to the fact it is being monitored.

Linux containers, through the means it provides isolation from the host, is able to circumvent the recursive STRACE detection method. From the host, STRACE may be called on any process executing within the container or on the container itself and all of it's children (providing the most abundant amount of information, but with staggering amounts of non-malicious system calls that would need to be sifted through) in order to monitor the system calls made by that process. Due to container isolation, if the process being monitored attempts to call STRACE on itself or one of it's children, STRACE will execute normally and falsely indicate to the malware that it is not under analysis.

Another debug detection mechanism that is available to malware is reading information present in the Linux /proc/self/status file in the process file system. By searching that file for the text string "TracerPid:", it's presence will indicate to a malicious binary that a debugging tool, such as GDB or STRACE is currently monitoring it's execution. A debugger detection test was written using a slightly modified code example provided by Sam Liao in [50]. The detection method was tested for both GDB and STRACE and correctly identified the presence of a monitoring tool. However, when using STRACE from the host system to monitor the process within a Linux container, the tool did not detect the presence of the process tracer.

These preliminary tests show great promise in using Linux containers to defeat debugger and monitoring tool detection methods, which are an identified and present threat to security analysis. Using the terminology of Jiang, et al., this unanticipated benefit from the use of Linux containers grants researchers "in-host" semantic detail of malware behavior with "out-of-host" detection prevention.

Source code for these tests is provided in Appendix A.8.

Chapter 10

Deception Enabled by Containers

Due to the abstraction of Linux containers away from the operating system, there is great opportunity for deception at the software level, where containers can be built using different Linux distributions, different software installations, and different software versions. This type of software-level deception is ideal when attempting to observe malware using exploits that are dependent on a system's software configuration.

Additionally, the question of whether Linux containers could provide deception regarding system aspects of a honeypot was explored. The ultimate goal would be for a low-cost Raspberry Pi honeypot to appear to an attacker as any system a researcher could desire, such as a high-end commercial server. This would allow malware research using cheap or generic hardware to explore attacks on different architectures and systems.

While being a lofty goal, the current implementation of Linux containers does not allow this type of behavior. As the design of containers is for programs to execute directly on the host kernel, isolated from only other host processes, those programs will behave and respond as if they were executed directly on the host. This results in any command used to identify the hardware of the host system to report accurately (which is an advantage when defeating hardware attribute virtual environment detection tests). Linux containers are also without any sort of virtual hardware, with the exception of a virtual network bridge, so additional hardware components cannot be added to a container.

While it is possible that a host system may be able to intercept a number of system information related commands, such as SYSINFO, and modify the response before providing it to the requesting container, the behavior of the system hardware itself can reveal it's true nature.

Using data discussed in Chapter 8 as an example, consider the cache latency performance on a Raspberry Pi 3 with an *Fread* stride of 512kb, illustrated in Figure 10.1. A user can identify, approximately, the number of CPU cache levels and their sizes; in this example, a user can identify two cache levels with a maximum cache size approximately around 512kB. If a container was intercepting LSCPU commands to report the CPU as an ARM Cortex-A73

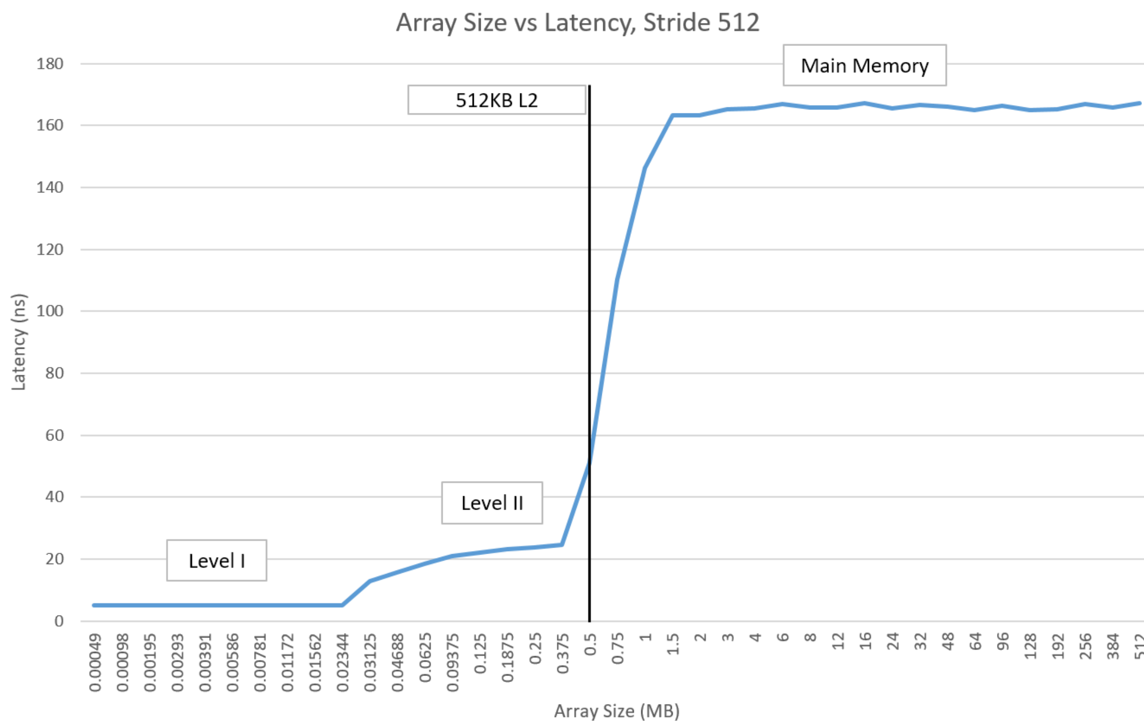

Figure 10.1: Line graph illustrating cache latency over range of array sizes with read stride length of 512kB on Betty. X-axis is the array size in MB. Y-axis is latency in nanoseconds.

(a high-performance ARM processor that implements the same ARMv8-A instruction set as the Raspberry Pi CPU), an attacker may know that the Cortex-A73 has a programmable L2 cache that is a minimum of 1 MB, inconsistent with the collected latency values and revealing the deception. Countless other examples can be made, all dealing with unprivileged user commands.

It should be noted that even virtual environments are susceptible to such types of system performance detection tests. Even though a virtual environment may be able to simulate all of it's hardware to match whatever platform is desired, the performance characteristics of that platform's hardware simply cannot be simulated in software, thus are susceptible to benchmarking-based detection methods if they are using external time sources.

Unfortunately, using a combination of virtual environments and Linux containers does not address the issue. Indeed, it can potentially make a system even more susceptible to environment detection methods. A virtual environment that runs Linux containers within it can be detected by both virtual environment *and* container detection methods. At best, a Linux container running a virtual environment will still remain detectable using virtual environment tests.

Such hardware-based deception will have to remain, for the time being, in the imperfect realm of virtual environments.

Chapter 11

Future Work

As this study was just a preliminary look into the types of cybersecurity deception that can be performed on low-power devices, a large number of topics were discussed with only a minimal amount of depth explored for each topic. Many of the areas included in this research are deserving of a much greater treatment in an effort to develop tools for security researchers that are new, affordable, flexible in deployment arrangements, and provide new avenues of data collection in the ever-changing landscape of botnet threats.

While research is ongoing into the capabilities of Raspberry Pi honeypot sensors [48] [16], these studies are focusing only on low and medium interaction honeypots. With the recent availability to deploy high interaction honeypots on Raspberry Pis, even greater distributed, deceptive sensors can be deployed, which in turn can gather an even richer pool of malware related data.

Unique features of Linux containers, such as container live migration, offer interesting avenues of research and how more agile honeynets may be developed, where honeypots are no longer restricted to the current host, but can traverse within a network or even between networks.

Of considerable import is the additional work that needs to be performed for container detection methods, such as those discussed in Chapter 9, in order to close all gaps that may remain for attackers to discover the nature of the container as well as to identify what best practices may be adopted in order to maximize the deception afforded by Linux containers.

Ongoing exploration in to the deception that is enabled by containers is also in need of greater research. Despite the limitations outlined in Chapter 10, there is new territory to be explored with possibly surprising outcomes. Abstraction away from the operating system can present many opportunities to clever security researchers.

Chapter 12

Additional Discussion

This chapter includes any remaining discussions that weren't fully addressed earlier in the thesis due to the topics not quite fitting into the thesis structure.

12.1 Refactoring Honeypots for Aggregate Server Interfaces

Advancing the discussion started in Section 3.1.3, regarding the utility and recent maturity of aggregate honeypot servers, most, if not all, honeypots would benefit from being able to report to a back-end collector. However, as these aggregate servers are a relatively new development, a large number of existing honeypots do not have the necessary interface support in their code base to connect with a back-end server. While a number of new honeypots being developed have the requisite interfaces implemented, oftentimes pre-existing ones do not. This section proposes a basic refactoring workflow that can be utilized in a mechanical process that allows honeypot authors or contributors to refactor their code base to allow the addition of alternate honeypot interfaces in a much more simple and straightforward manner.

12.1.1 Refactoring

One of the pioneers of modern refactoring, William Griswold, stated the use of updating the structure of a code-base specifically for the intention of improving maintenance can decrease the difficulty and burden of future software updates [35]. It has been many years since that paper's publishment and since then, many tools have been developed for a large number of programming languages for the explicit purpose outlined by Griswold. Advancing refactoring beyond mere tool development, fully automated, program-wide refactoring has

also been explored, such as the search-based refactoring method proposed by Mark O'Keeffe in [60] in which the author developed a template that maximized program maintainability by refactoring software in order to more closely adhere to object oriented programming's core principles.

Narrowing the scope from general maintainability, the security realm has several domain-specific tasks that generate their own particular challenges. Specifically, for the purposes of this discussion, the refactoring operations considered are program-wide in scope, yet highly targeted at a specific set of function calls; mainly, those function calls that deal with logging operations as that is the primary intent of *hpfeed* traffic. Additionally, the application of refactoring operations that are being proposed is intended to span multiple platforms and languages.

For the benefit of the discussion, a refactoring workflow is developed in which honeypot developers are able to mechanically implement into their code-base, relieving much of the effort required in refactoring as well as minimize potential code disruption.

Why is refactoring a good choice?

As a considerable code-base of useful and relevant low and medium interaction honeypots already exists, refactoring existing solutions to allow easy enhancement of their capabilities by aggregate back-end servers appears to be a very economical solution. Refactoring also allows for the greatest flexibility for aggregate server platforms of choice; once the refactoring has been performed, multiple interface additions are relatively straightforward, avoiding the honeypot from being tied to a specific interface due platform dependent code changes.

Additionally, as honeypots span multiple platforms as well as multiple languages, using a refactoring workflow allows for a language and platform agnostic approach, provided the language the honeypot is written in supports class inheritance.

Honeypots are uniquely poised for this type of refactoring due to the recent attraction of object oriented, high-level scripting languages, specifically Python. It is assumed that Python's high-level of abstraction and focus on rapid-prototyping makes it an attractive language for non-commercial software authors to use for home-brew honeypots. By using an object oriented approach, interfaces can more readily be abstracted into interface objects with subclasses specific for each interface.

12.1.2 Refactoring Workflow Template

Due to the highly specialized domain that this refactoring is targeting and the "home-brew" nature of many open source honeypot projects, no single automated tool can be developed that will address more than a small subset of honeypots that meet the criteria for the proposed refactoring. Different programming styles, heterogeneous program structures and

implementations, as well as different programming languages fall within the focus of this discussion. The resulting solution of the highly varied target population is a novel refactoring implementation method referred to as a refactoring workflow template.

While refactoring is a suitable approach to address the need to enhance preexisting honeypots, a single automated refactoring tool is not suitable to the wide target code base due to the variety of honeypots this refactoring addresses. The proposed solution consists of a workflow template that provides a basic, systematic approach on applying the target refactoring using existing refactoring tools that are available in may IDEs. The workflow template is intended to mechanize the refactoring process that a developer follows and thereby reducing the number of potential side-effects by minimizing the actions performed to only a few, highly focused steps. Similar to following a cooking recipe, a developer who adheres to a refactoring workflow template on a code base that fits the application criteria can expect a successful refactoring without too many unforeseen obstacles.

12.1.3 Developing a Refactoring Template

The following criteria was established to identify honeypot programs as candidates for the application of refactoring:

- Open Source
- Low/medium-interaction honeypot
- Active project (activity within the last two years)
- Written in programming language that supports class inheritance

The requirement for the software to be open source is an obvious one. Additionally, low and medium interaction honeypots are the topic of discussion due to their targeted approaches and the need for a varied population of honeypots to be deployed on a network in order to gain greater visibility on a variety of network threats. The criteria for a project to have shown activity in the last two years was imposed to ensure any refactorings considered are for relevant honeypots that are still actively maintained. Finally, the language restriction is imposed as the proposed refactoring workflow requires the use of class abstraction.

Once identified through the use of the primary criteria, the following secondary requirements were used to identify the honeypots that would benefit from refactoring:

- Honeypot does not already support remote logging to an aggregate server
- Honeypot does not handling logging in an efficient manner

While interface refactoring may still benefit a honeypot that already supports logging to an aggregate server, particularly in the case when multiple interfaces may wish to be configured, honeypots already supporting back-end logging were not considered.

To be considered an inefficient implementation of logging, the honeypot would require code modifications in multiple locations and/or multiple files in order to add additional interfaces. This was best illustrated in honeypots code-bases that did not implement interfaces using a class-based approach or an existing logging API.

After conducting an online web search for open source honeypots, 19 honeypots met the list of requirements to be identified as initial candidates. Out of those 19 candidates, six were identified as also meeting the secondary requirements.

Name	**Description**
Artillery	Monitors specific ports for unauthorized activity
RDPY	RDP client/server and honeypot
HoneyPy	Framework with plug-ins for expanded capabilities
HoneyNTP	NTP server honeypot
Mailoney	SMTP honeypot
UDPot	DNS honeypot/sinkhole

The general guideline of the proposed refactoring was to implement multiple interfaces through the use of class-based hierarchy with an Abstract Base Class serving as a template that the interface subclasses would need to match. All pre-existing logging operations would be implemented by the interface sub-classes and at the completion of the refactoring, the semantic behavior of the honeypot would be unchanged, while the structure of the code would be such that additional logging interfaces could be implemented in a single location as a new interface sub-class and the section of the program that instantiates the appropriate interface class would be updated to check for the updated configuration settings.

Many IDEs have refactoring tools to assist with each of these operations, such as method extraction and method signature updates.

12.1.4 Case Study

The honeypot *Artillery* was selected as a case study to apply the general interface refactoring template and to further refine the workflow after an initial application. *Artillery* is a low-interaction honeypot that monitors specific ports for network traffic and sends alerts for and blacklists any unauthorized traffic. It's current implementation is in Python 3 and runs on Windows/Unix-based systems (although it is not fully implemented in Windows). The

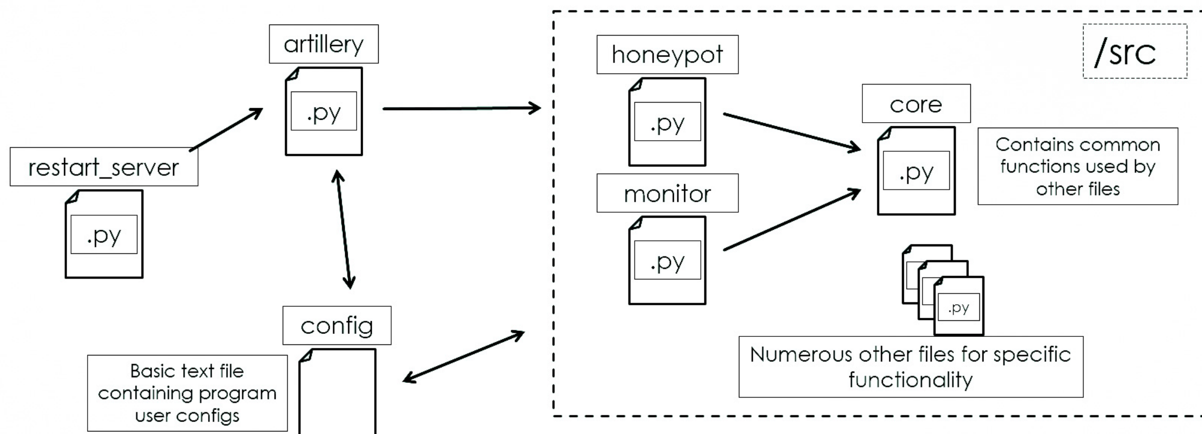

Figure 12.1: An overview of the file structure and module interaction of Artillery.

current version of *Artillery* supports logging output to a local file as well as local or remote syslog for Unix-based platforms.

Artillery is one of the larger Low Interaction honeypots and is fairly structured, with a code-base spanning 16 files. The basic structure of *Artillery* is depicted in Figure 12.1.

The file src\core.py contains most of the functions that are utilized by other modules within *Artillery* and are where the logging functions reside. The primary method by which *Artillery* performs it's logging is through calls to the function write_log(). When write_log() is called with an alert message passed as a string argument, the function first verifies the environment; if it is running on a Windows OS, write_log() will simply write the alert message to a local log file. If *Artillery* is running on a Unix-based system, the alert is passed to the function syslog() as shown in Listing 12.1.

Listing 12.1: write_log(alert) code excerpt.

```
def write_log(alert):
    if is_posix():
        syslog(alert)

    if is_windows():
        #send alert to local file
        ...
```

Once a message is passed to syslog(), the function checks the configuration file to properly direct the output. From here, the message is either sent to a remote syslog, sent to the local syslog, or written as a local file as determined by the configuration file, illustrated in Listing 12.2.

Listing 12.2: syslog(message) code excerpt.

```
def syslog(message):
    type = read_config("SYSLOG_TYPE").lower()

    if type == "remote":

        #send message to remote syslog
        ...

    if type == "local":

        #send message to local syslog
        ...

    if type == "file":

        #send message to local file
        ...
```

While this particular logging structure is relatively straightforward and easy to understand in a small code-base, it doesn't easily allow for the expansion of additional interfaces, particularly if the additional interface is connection/session based, which would require additional code in, at a minimum, core.py and artillery.py for each additional interface. As mentioned in the previous section, the refactoring template proposed is based on the use of an overall interface Abstract Base Class and subclasses for each implemented interface. The goal of this template is for the Abstract Base Class to define the virtual methods that are anticipated by the rest of the program and are overridden by each of the subclass interfaces. This allows all of the code specific for each used interface to reside in a single location with a single, polymorphic logging method that is used by the rest of the program. This approach also improves data encapsulation regarding the interface methods.

As the logging implementation in *Artillery* is not particularly complex, the only required virtual method in the interface Abstract Base Class is write_log(). Listing 12.3 depicts the simple Abstract Base Class. Print statements are included as illustration during the program execution and are not required for the implementation of the refactoring template. As a requirement of a pure Abstract Base Class, any calls made to the Abstract Base Class's write_log() method result in an exception as this is a pure virtual method and not meant to be called during normal program execution.

Listing 12.3: Abstract Base Class.

```
class Interface:

    def __init__(self):
```

```
        print("Interface set to ABC")

    def write_log(self, alert):
        raise NotImplementedError("...")
```

The code for each of the interfaces already supported in *Artillery* was extracted into it's own subclass. As an example, Listing 12.4 shows the subclass for the Local syslog interface. Due to the code for each interface being extracted into it's own write_log() method, the syslog() function was no longer needed and was removed.

Listing 12.4: Local syslog subclass.

```
class Local_Syslog_Interface(Interface):

    def __init__(self):
        print("Interface set to Local Syslog")

    def write_log(self, message):
        print("Local Syslog interface write_log() called")
        #send message to local syslog
        ...
```

This completed the refactoring of the built-in interfaces that *Artillery* was initially configured for. In keeping with the original implementation of the write_log() function, each time write_log() is called, the program reads it's configuration to determine the proper interface to use, creates an instance of the appropriate interface subclass, and calls that subclass's write_log() method.

However, given that the additional interfaces that are intended to be implemented communicate across a network, there is a requirement for some procedure to maintain a connection/session based interface. A communication session will need to be established at the start of program execution and maintained throughout. Session establishment can be performed simply enough in the subclass initialization method, however the instance of the interface object that creates that session will need to be passed to every utilizing function rather than creating a new instance of the interface every time it is needed as this will generate a new session at every object instantiation.

The simplest solution would be to generate a new interface instance at the beginning of code execution as a global variable and have every function that calls write_log() simply access that global object. Unfortunately, aside from violating the object oriented principle of encapsulation, this type of global variable access doesn't work in Python in the required manner; each module accesses a call-by-value instance of the object rather than the necessary call-by-reference.

The preferred method, which maintains encapsulation, is passing the instance of the object

as a parameter to each function that calls write_log(). This step is relatively straight forward; however the way in which *Artillery* executes code in each of it's modules presents a complication as the code for each module is executed implicitly when the module is imported by artillery.py, rather than using explicit function calls, which can pass parameters from artillery.py (as shown in Listing 12.5) to honeypot.py (as shown in Listing 12.6).

Listing 12.5: Implicit module code execution.

```
# imports honeypot module and immediately
    executes main() within the module
import src.honeypot
```

Listing 12.6: Original honeypot.main() execution receiving port and interface variables.

```
def main(ports, bind_interface):
    ...
    for port in ports:
        thread.start_new_thread(listen_server,
            (port, bind_interface,))

# launch the application
main(ports, bind_interface)
```

To resolve this complication, the automatic code execution in each module was removed and module imports were modified in artillery.py in order to explicitly call the module's main function and pass the object instance as shown in the example of artillery.py for executing the code within honeypot.py in Listing 12.7 and the updated code in honeypot.py in Listing 12.8.

Listing 12.7: Explicit module code execution with interface object passing.

```
#import src.honeypot
from src.honeypot import honeypot_main
honeypot_main(interface)
```

Listing 12.8: Updated honeypot_main() receiving interface object and using global variables.

```
def honeypot_main(interface):
    global ports
    global bind_interface
    ...
    for port in ports:
        thread.start_new_thread(listen_server,
            (port, bind_interface, interface))
```

Once the module is executed with a reference to the interface object, the interface can now easily be passed to any functions that need it. Listing 12.9 demonstrates where the interface

object is passed to the function warn_the_good_guys() and the write_log() method for that interface object is subsequently called.

Listing 12.9: Interface object parameter passing.

```
def warn_the_good_guys(subject, alert,
        interface):
    #if email alerts enabled, send email
    ...
    interface.write_log(alert)
```

12.1.5 Case Study Results

After completing the case study on refactoring *Artillery* for multiple interfaces, a few observations were made. Firstly, logging operations in *Artillery* were able to be successfully abstracted into class objects without altering program behavior. As a result, additional interfaces can now be added at a single location with minimal risk of implementation side-effects. However, in order to successfully complete the refactoring, additional manipulations of how module code was being executed were required - specifically, transforming it from implicit code execution made through simply importing the module, to explicit function calls made from artillery.py.

12.1.6 Refined Refactoring Workflow

As a result of conducting a case study, the refactoring workflow template was refined in order to incorporate the additional hurdles met during refactoring *Artillery*. The updated refactoring workflow is as follows:

1. Create an Abstract Base Class.
2. Create subclasses for each type of interface.
3. Extract methods into appropriate subclasses for pre-existing interfaces (using method extraction refactoring tools).
4. Add appropriate interface object instantiation.
5. Add interface object parameter passing (using method signature modification refactoring tools).
6. Modify any implicit module method calls to be explicit calls that pass the interface object.

This final step may be unique only to honeypots written in Python, however as there may be similar mechanisms in other languages that may require the same code modification, it was added as the final potential step in the workflow.

12.1.7 Code Review Results

After conducting a code review of the remaining five refactoring candidates, it was identified that four of the candidates could be refactored by following the steps outlined in the refactoring workflow. The single unique case of *HoneyPy* would require some additional refactoring steps outside of the developed workflow mainly due to it's light use of a logging API. Using logging APIs was a disqualifying criteria for consideration to apply the refactoring workflow approach, however due to the API use not being fully implemented by *HoneyPy*, it was considered as a refactoring candidate.

12.1.8 Considered Solutions

There are potential alternatives for implementing back-end server logging of honeypots for developers who choose to not refactor the source-code of their project or for honeypots in which the refactoring template is not suited (namely, projects written in a non-object oriented language). One of the more direct approaches for a linux-based honeypot would be utilizing the Dynamic Library Linker "LD_PRELOAD". By using LS_PRELOAD, Linux allows users to dynamically modify library functions a program is requesting and replacing those function calls with custom functions, all without needing access to the target program's source code.

The simplest approach for using LD_PRELOAD would be to configure the honeypot to write logs to a local file (if that option is available), then write custom open() and close() functions that establish and disconnect a network connection to the aggregate server and write a custom write() function that transmits the output to the connected server.

This approach is highly project dependent and may not be viable for honeypots that use the system file I/O functions for tasks not related to logging.

As an alternative to LD_PRELOAD or manually refactoring the code-base, developers could explore a honeypot's abstract syntax tree in order to modify the program's executable without changing the source code. This option is available for honeypot's written in C-based languages by using the Clang front-end tool. Honeypots written in Python are able to do likewise with the Python AST module. A manual inspection of the abstract syntax tree would be necessary in order to identify logging functions and all of their dependencies. These logging functions could then be modified to direct their output to an aggregate server once a connection to the server has been established. Through a careful application of this method, all dependencies on logging operations can be identified and the program can be modified in

confidence that no other execution behavior will be impacted.

As in the approach utilizing LD_PRELOAD, the task of expanding a honeypot's existing interfaces through the process of reviewing and modifying it's abstract syntax tree is highly dependent on the honeypot code-base itself and requires careful comparison of the abstract syntax tree against the source code in order to correctly modify the executable. This approach is only advisable when neither refactoring the source code nor use of LD_PRELOAD is available.

12.1.9 Refactoring Conclusions

This discussion determined that interface refactoring is a useful and practical way to extend and enhance the capabilities of existing security tools through the introduction of a refactoring workflow template. Refactoring is not simply a task reserved for larger code-bases, but, given a developed workflow, can be followed simply and mechanically by a developer in order to stage the software for easy implementation of enhancements.

The criteria developed for determining whether a honeypot is a potential candidate for this refactoring template did eliminate many potential honeypots; specifically the requirement that the honeypot be written in a language that supports class inheritance. However, it was found that for honeypots that met the criteria, their code-bases fit the workflow template well.

Due to the unique characteristics of all potential honeypots that could be enhanced by applying this interface refactoring workflow, such as the varied languages they are implemented in and being written by developers with varied backgrounds who may or may not focus on implementing the fundamentals of object oriented programming, there does not appear to be a tool that can highly-automate the process of conducting the refactoring workflow. However, there are many refactoring tools available in many major IDEs that can aid a developer in the basic refactoring tasks in order to reduce the time required for refactoring as well as assist in avoiding typos or introduce other errors within the code during the refactoring.

12.1.10 Future Work of Honeypot Refactoring

The current refactoring workflow template does not account for issues where a multithreaded honeypot may be accessing an interface concurrently. While this is a concern unique to the specific interface and whether it can be considered thread-safe, the general refactoring template should be further developed to identify standard locations to implement proper synchronization. Additionally, for honeypots written in Python, a template should be developed that incorporates Python's rich logging API. There are multiple benefits to using the API but the features specifically important to this discussion are the ease in which logging objects can be referenced (possibly removing the necessity for passing the interface as a

parameter) as well as thread-safe logging operations.

The novel approach of developing refactoring workflow templates has potential applications in other, highly-focused refactoring efforts that may impact a wide range of code-bases that require a similar set of internal structure changes, specifically in the domain of system security where discovered vulnerabilities may required a common fix for a large population of programs. To aid in this effort and to enhance the legitimacy of refactoring workflow templates, this section proposes the development of a 'guided refactoring' template tool that can be developed as a plug-in for popular IDEs, such as Eclipse, which can load a designed template and assist developers in performing the steps of the workflow while suggesting the refactoring tools best suited to assist the task at hand. While this is not an automated tool, it can provide relevant prompts and useful suggestions while keeping the developer focused on the specific refactoring task being performed. Great care must be taken when developing refactoring workflow templates as the quality of content developed for the template directly impacts the usefulness of the guided refactoring tool.

The overall goal of this section was to provide a means in which pre-existing and useful security tools can be enhanced in order to provide an even greater benefit to the security community. Many software authors and contributors are hesitant to implement significant changes to their code-base for software enhancements out of concern of breaking their code; however, this study demonstrates the process of generating a refactoring workflow template that is written to address a very specific task and can be applied to different code bases in a mechanical fashion that eliminates the majority of concerns developers have regarding unintended side-effects and poises their code-base for easy implementation of future enhancements. Botnet developers are not shy with implementing new features in order to change the security landscape and the security community should not be shy of doing the same with their set of tools.

Chapter 13

Conclusion

The ever-evolving nature of botnets have made constant malware collection an absolute necessity for security researchers in order to analyze and investigate the latest, nefarious means by which bots exploit their targets and operate in concert with each other and their bot master.

In that effort of on-going data collection, honeypots have established themselves as a useful and potent tool for deception-based security, diligently luring and monitoring attackers in the endeavor to discover their tactics and capture their dangerous payloads for later analysis. The value of honeypots has been well established both in practice and in literature and their employment techniques continue to develop and mature. Low-powered devices, such as the Raspberry Pi, have found a natural home with some categories of honeypots and their use is being embraced by the honeypot community. Due to the low cost of these devices, new techniques are being explored to employ multiple honeypots within a network to act as distributed sensors, collecting activity reports and captured malicious binaries to send to back-end servers for later analysis and network threat assessments. While these techniques are just beginning to gain their stride within the security community, they are held back by the minimal amount of deception a traditional honeypot on a low-powered device is capable of delivering.

This thesis has made a preliminary investigation into the viability of using Linux containers to greatly expand the deception possible on low-powered devices by providing isolation and containment of full system images with minimal resource overhead. Indeed, employing Linux containers on low-powered devices has enabled an entire category of honeypots previously unavailable on such hardware platforms. In addition to granting formerly unavailable high interaction with honeypots on low-power devices, the use of Linux containers grants unique advantages that have not previously been explored by security researchers, such as the ability to defeat many types of virtual environment and monitoring tool detection methods.

During the investigation it has been identified that there are limitations to the deception such

systems are capable of and security researchers and administrators need to be aware of such limitations in order to make informed decisions on what deception tactics are appropriate for low-powered devices.

However, in the ever-evolving cat-and-mouse game between botnet authors and security researchers, the combination of honeypots, low-power devices, and Linux containers make a potent blend, enabling and enhancing the strengths that each technology offers. With this new tool, the academic community stands to potentially gain even greater insight into the most hardened of botnets.

Bibliography

[1] Eggheads. https://www.eggheads.org/, 2016. 2017-2-28.

[2] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 41–52. ACM, 2006.

[3] Frederico Araujo, Kevin W Hamlen, Sebastian Biedermann, and Stefan Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 942–953. ACM, 2014.

[4] Michael Bailey, Evan Cooke, Farnam Jahanian, Yunjing Xu, and Manish Karir. A survey of botnet technology and defenses. In *Conference For Homeland Security, 2009. CATCH'09. Cybersecurity Applications & Technology*, pages 299–304. IEEE, 2009.

[5] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In *NDSS*, 2010.

[6] Sakshi Bansal, Mir Qaiser, Shefali Khatri, and Anchit Bijalwan. Botnet forensics framework: Is your system a bot. In *Advances in Computing and Communication Engineering (ICACCE), 2015 Second International Conference on*, pages 535–540. IEEE, 2015.

[7] Dina Bass and Ian King. Microsoft pledges to use arm server chips, threatening intel's dominance. https://www.bloomberg.com/news/articles/2017-03-08/microsoft-pledges-to-use-arm-server-chips-threatening-intel-s-dominance, 2017. 2017-14-3.

[8] Andy Bavier. Os structures and system calls. https://www.cs.princeton.edu/courses/archive/fall10/cos318/lectures/ OSStructure.pdf, 2010. 2016-10-25.

[9] Up Board. Up board—power up your ideas. http://www.up-board.org/upsquared/specifications-up2/, 2015. 2017-14-3.

[10] David Brash. Extensions to the armv7-a architecture. http://www.hotchips.org/wp-content/uploads/hc_archives/hc22/HC22.23.220-1-Brash-ARMv7A.pdf, 2010.

[11] Eric Brown. Open-spec i.mx6 quad sbc is bursting with wireless i/o. http://linuxgizmos.com/open-spec-i-mx6-quad-sbc-is-bursting-with-wireless-io/, 2015. 2017-14-3.

[12] Jeffrey Burt. Arm server chips challenge x86 in the cloud. https://www.nextplatform.com/2017/02/01/arm-server-chips-challenge-x86-cloud/, 2017. 2017-2-2.

[13] Wentao Chang, An Wang, Aziz Mohaisen, and Songqing Chen. Characterizing botnets-as-a-service. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 585–586. ACM, 2014.

[14] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 177–186. IEEE, 2008.

[15] WY Chin, Evangelos P Markatos, Spiros Antonatos, and Sotiris Ioannidis. Honeylab: large-scale honeypot deployment and resource sharing. In *Network and System Security, 2009. NSS'09. Third International Conference on*, pages 381–388. IEEE, 2009.

[16] Andreas Christoforou, Harald Gjermundrød, and Ioanna Dionysiou. Honeycy: a configurable unified management framework for open-source honeypot services. In *Proceedings of the 19th Panhellenic Conference on Informatics*, pages 161–164. ACM, 2015.

[17] Mike Coleman. Containers are not vms. https://blog.docker.com/2016/03/containers-are-not-vms/, 2016.

[18] Angelo Dell'Aera. Thug. https://github.com/buffer/thug, 2011. 2017-28-2.

[19] desaster. Kippo. https://github.com/desaster/kippo, 2009. 2017-28-2.

[20] David Dittrich and Sven Dietrich. New directions in peer-to-peer malware. In *Sarnoff Symposium, 2008 IEEE*, pages 1–5. IEEE, 2008.

[21] David Dittrich and Sven Dietrich. P2p as botnet command and control: a deeper insight. In *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*, pages 41–48. IEEE, 2008.

[22] Pamela Engel. This world map shows every device connected to the internet. http://www.businessinsider.com/this-world-map-shows-every-device-connected-to-the-internet-2014-9, 2014. 2017-8-3.

[23] ADI Engineering. Minnowboard turbot. http://www.adiengineering.com/products/minnowboard-turbot/, 2017. 2017-14-3.

[24] Jayson Falkner and Kevin W. Jones. Java servlets — what servlets are and why you would want to use them. http://www.informit.com/articles/article.aspx?p=170963, 2004. 2017-8-10.

[25] Maryam Feily, Alireza Shahrestani, and Sureswaran Ramadass. A survey of botnet and botnet detection. In *Emerging Security Information, Systems and Technologies, 2009. SECURWARE'09. Third International Conference on*, pages 268–273. IEEE, 2009.

[26] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 171–172. IEEE, 2015.

[27] Peter Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange*, 55, 2007.

[28] Dennis Fisher. Storm, nugache lead dangerous new botnet barrage. http://searchsecurity.techtarget.com/news/1286808/Storm-Nugache-lead-dangerous-new-botnet-barrage, 2007. 2017-28-2.

[29] Jason Franklin, Mark Luk, Jonathan M McCune, Arvind Seshadri, Adrian Perrig, and Leendert Van Doorn. Remote detection of virtual machine monitors with fuzzy benchmarking. *ACM SIGOPS Operating Systems Review*, 42(3):83–92, 2008.

[30] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: Vmm detection myths and realities. In *HotOS*, 2007.

[31] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. In *International Cryptology Conference*, pages 444–461. Springer, 2014.

[32] Pierce M Gibbs. Botnet tracking tools. 2014.

[33] Johannes Gilger. Using hpfriends - the social data sharing platform. http://heipei.github.io/2013/05/11/Using-hpfriends-the-social-data-sharing-platform/, 2013. 2016-05-03.

[34] Stephane Graber. Lxc 1.0: Security features [6/10]. https://www.stgraber.org/2014/01/01/lxc-1-0-security-features/, 2014. 2016-12-07.

[35] William G Griswold. Program restructuring as an aid to software maintenance. *University of Washington, Ph.D Thesis*, 1991.

[36] Major Hayden and Richard Carbone. Securing linux containers. *GIAC (GCUX) Gold Certification, Creative Commons Attribution-ShareAlike 4.0 International License*, 2015.

[37] Naofumi Homma, Sei Nagashima, Yuichi Imai, Takafumi Aoki, and Akashi Satoh. High-resolution side-channel attack using phase-based waveform matching. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 187–200. Springer, 2006.

[38] Seco USA Inc. Udoo x86 - udoo. http://www.udoo.org/udoo-x86/, 2016. 2017-14-3.

[39] Iozone. Iozone filesystem benchmark. http://www.iozone.org/, 2016. 2016-10-9.

[40] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138. ACM, 2007.

[41] John P John, Fang Yu, Yinglian Xie, Arvind Krishnamurthy, and Martín Abadi. Heat-seeking honeypots: design and experience. In *Proceedings of the 20th international conference on World wide web*, pages 207–216. ACM, 2011.

[42] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 403–412. ACM, 2011.

[43] Bitdefender Labs. Gameover zeus variants targeting ukraine, us. https://labs.bitdefender.com/2014/08/gameover-zeus-variants-targeting-ukraine-us/, 2014. 2016-03-16.

[44] Michael Larabel. Raspberry pi's bcm2837 soc now supported by mainline linux 4.8. https://www.phoronix.com/scan.php?page=news_item&px=ARM-Platforms-Linux-4.8, 2016. 2017-26-3.

[45] Boris Lau and Vanja Svajcer. Measuring virtual machine detection in malware using dsd tracer. *Journal in Computer Virology*, 6(3):181–195, 2010.

[46] Martin Lee. Four ways cybercriminals profit from botnets. https://www.symantec.com/connect/blogs/four-ways-cybercriminals-profit-botnets, 2010. 2017-1-5.

[47] David Lettier. Ntp client. https://github.com/lettier/ntpclient/blob/master/source/c/main.c, 2014. 2016-15-9.

[48] Jason Lewis. The effectiveness of using a raspberry pi as a honeypot sensor. In *IEEE Conference on Communications and Network Security 2016*, 2015.

[49] Chao Li, Wei Jiang, and Xin Zou. Botnet: Survey and case study. In *Innovative Computing, Information and Control (ICICIC), 2009 Fourth International Conference on*, pages 1184–1187. IEEE, 2009.

[50] Sam Liao. How to detect if the current process is being run by gdb. http://stackoverflow.com/questions/3596781/how-to-detect-if-the-current-process-is-being-run-by-gdb, 2014. 2017-4-1.

[51] LordNoteworthy. Al-khaser. https://github.com/LordNoteworthy/al-khaser/tree/ff8d53891709b407cbf43a323abc302730504fae, 2016. 2017-28-3.

[52] Canonical Ltd. snapcraft - snaps are universal linux packages. https://snapcraft.io/?from=corepage, 2017. 2017-25-2.

[53] Jaguar Electronic H.K. Co. Ltd. Jaguarboard–industry first x86-based. http://www.jaguarboard.org/index.php/products/product_show/jaguarboard-industry-first-x86-based.html, 2015. 2017-14-3.

[54] MACVendors.com. Api — the simplest mac vendor lookup api. https://macvendors.com/api, 2017. 2016-11-9.

[55] Surendra Mahajan, Akshay Mhasku Adagale, and Chetna Sahare. Intrusion detection system using raspberry pi honeypot in network security. *International Journal of Engineering Science*, 2792, 2016.

[56] Dhia Mahjoub. Monitoring a fast flux botnet using recursive and passive dns: A case study. In *eCrime Researchers Summit (eCRS), 2013*, pages 1–9. IEEE, 2013.

[57] mempodippy. Vlany. https://raw.githubusercontent.com/mempodippy/vlany/ master/misc/detect_lxc.c, 2016. 2016-11-3.

[58] Hamid Mohammadzadeh, Masood Mansoori, and Ian Welch. Evaluation of fingerprinting techniques and a windows-based dynamic honeypot. In *Proceedings of the Eleventh Australasian Information Security Conference-Volume 138*, pages 59–66. Australian Computer Society, Inc., 2013.

[59] NanoPi. Nanopi 2 fire. http://nanopi.io/nanopi2-fire.html, 2015. 2017-14-3.

[60] Mark O'Keeffe and Mel O Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.

[61] Kyle Olivo. Using lxd and cgroups. http://kyleolivo.com/dev/2016/08/21/lxd-and-cgroups/, 2016. 2017-23-3.

[62] Safwan Omari. Web attack threat analysis, experiences with glastopf honeypot. Centers of Academic Excellence Tech Talk - February 2016, 2016.

[63] Keith Parkansky. What is cgi and how to use your cgi-bin cgi tutorial and perl introduction for beginners. http://www.parkansky.com/tutorials/bdlogcgi.htm, 2004. 2017-8-10.

[64] Michael Pearce, Sherali Zeadally, and Ray Hunt. Virtualization: Issues, security threats, and solutions. *ACM Computing Surveys (CSUR)*, 45(2):17, 2013.

[65] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. nether: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the Fourth European Workshop on System Security*, page 3. ACM, 2011.

[66] Peter Pisarčík and Pavol Sokol. Framework for distributed virtual honeynets. In *Proceedings of the 7th International Conference on Security of Information and Networks*, page 324. ACM, 2014.

[67] Niels Provos. honeyd. http://www.honeyd.org/, 2004. 2017-28-2.

[68] Niels Provos et al. A virtual honeypot framework. In *USENIX Security Symposium*, volume 173, pages 1–14, 2004.

[69] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In *International Conference on Information Security*, pages 1–18. Springer, 2007.

[70] rep. Dionaea. https://github.com/rep/dionaea, 2010. 2017-28-2.

[71] Lukas Rist. Glastopf. https://github.com/mushorg/glastopf, 2008. 2017-28-2.

[72] Joanna Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction. https://blog.invisiblethings.org/papers/, 2004.

[73] SecuriTeam. Red pill... or how to detect vmm using (almost) one cpu instruction. http://www.securiteam.com/securityreviews/6Z00H20BQS.html, 2004. 2016-09-14.

[74] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[75] Tom Spring. Malware evades detection with novel technique. https://threatpost.com/malware-evades-detection-with-novel-technique/120787/, 2016. 2016-09-25.

[76] Binary Defense Systems. Artillery. https://github.com/BinaryDefense/artillery, 2012. 2017-28-2.

[77] Govind Singh Tanwar and Vishal Goar. Tools, techniques & analysis of botnet. In *Proceedings of the 2014 International Conference on Information and Communication Technology for Competitive Strategies*, page 92. ACM, 2014.

[78] Teryl Taylor, Kevin Z Snow, Nathan Otterness, and Fabian Monrose. Cache, trigger, impersonate: Enabling context-sensitive honeyclient analysis on-the-wire. 2016.

[79] Government Technology. Top 10 countries where cyber attacks originate. http://www.govtech.com/security/204318661.html, 2013. 2017-8-3.

[80] Ritu Tiwari and Abhishek Jain. Improving network security and design using honeypots. In *Proceedings of the CUBE International Information Technology Conference*, pages 847–852. ACM, 2012.

[81] Bill Torpey. Code to test various clocks under linux (and osx). https://github.com/btorpey/clocks, 2016. 2016-26-9.

[82] Linux Torvalds. Linux kernel source tree, cpufeature.h. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/include/asm/cpufe 2011. 2017-28-3.

[83] Amit Kumar Tyagi and G Aghila. A wide scale survey on botnet. *International Journal of Computer Applications*, 34(9):10–23, 2011.

[84] Eben Upton. Raspberry pi 3 on sale now at $35. https://www.raspberrypi.org/blog/raspberry-pi-3-on-sale/, 2016. 2017-29-3.

[85] Eben Upton. Ten millionth raspberry pi, and a new kit. https://www.raspberrypi.org/blog/ten-millionth-raspberry-pi-new-kit/, 2016. 2017-14-3.

[86] Eben Upton and Gareth Halfacree. *Raspberry Pi User Guide*. Wiley, 1 edition, 2012.

[87] Emmanouil Vasilomanolakis, Shankar Karuppayah, Panayotis Kikiras, and Max Mühlhäuser. A honeypot-driven cyber incident monitor: lessons learned and steps ahead. In *Proceedings of the 8th International Conference on Security of Information and Networks*, pages 158–164. ACM, 2015.

[88] James Vincent. The tinker board is a more powerful raspberry pi rival from asus. http://www.theverge.com/circuitbreaker/2017/1/24/14368622/raspberry-pi-alternative-tinker-board-asus-4k, 2017. 2017-14-3.

[89] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C Snoeren, Geoffrey M Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 148–162. ACM, 2005.

[90] Chenxi Wang. Containers 101: Linux containers and docker explained. http://www.infoworld.com/article/3072929/linux/containers-101-linux-containers-and-docker-explained.html, 2016. 2017-1-31.

[91] Candid Wueest. Threats to virtual environments. *Symantec Security Response. Version*, 1, 2014.

[92] Sagar A Yeshwantrao and Vilas J Jadhav. Threats of botnet to internet security and respective defense strategies.

[93] Jim Yuill, Mike Zappe, Dorothy Denning, and Fred Feer. Honeyfiles: deceptive files for intrusion detection. In *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*, pages 116–122. IEEE, 2004.

[94] Zhaosheng Zhu, Guohan Lu, Yan Chen, Zhi Judy Fu, Phil Roberts, and Keesook Han. Botnet research survey. In *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*, pages 967–972. IEEE, 2008.

[95] Cliff C Zou and Ryan Cunningham. Honeypot-aware advanced botnet construction and maintenance. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 199–208. IEEE, 2006.

Appendix

Appendix A

Implemented Code

A.1 CPU Clock Variability Test

A.1.1 ClockTest.sh

```
#!/bin/bash
#
# Copyright 2014 by Bill Torpey. All Rights Reserved.
# This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs
3.0 United States License.
# http://creativecommons.org/licenses/by-nc-nd/3.0/us/deed.en
#
if [[ ${OSTYPE} == *linux* ]]; then
  LIBRT="-lrt"
  TASKSET="taskset -c 0"
  JAVAINC=linux
  SOEXT=so
  # does this machine have rdtscp instruction?
  if [[ $(grep rdtscp /proc/cpuinfo | wc -l) -gt 0 ]] ; then
    RDTSCP=" -DRDTSCP=1 "
  fi
elif [[ ${OSTYPE} == *darwin* ]]; then
  # assume RDTSCP -- not sure what else to do
  RDTSCP=" -DRDTSCP=1 "
  JAVAINC=darwin
  SOEXT=jnilib
fi
## show available clocks
```

```
echo;echo "clocks.c"
gcc -o clocks clocks.c ${LIBRT} && ./clocks
# benchmark clocks
# cpp side
echo;echo "ClockBench.cpp"
if [[ $(lscpu | grep "Architecture" | grep "arm" | wc -l) -gt 0 ]] ; then
  echo "Running ARM variant"
  g++ -O3 -ggdb ${LIBRT} ${RDTSCP} -o ClockBenchARM ClockBenchARM.cpp && sudo
-S ${TASKSET} ./ClockBenchARM $*  #ARM specific ClockBench
else
  echo "Running x86 variant"
  g++ -O3 -ggdb ${LIBRT} ${RDTSCP} -o ClockBench ClockBench.cpp && ${TASKSET}
./ClockBench $* #x86 specific ClockBench
fi
```

A.1.2 clocks.c

```
#include <time.h>
#include <sys/time.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
// http://c-faq.com/stdio/commaprint.html
#include <locale.h>
char *commaprint(unsigned long n)
{
static int comma = '\0';
static char retbuf[30];
char *p = &retbuf[sizeof(retbuf)-1];
int i = 0;
if(comma == '\0') {
struct lconv *lcp = localeconv();
if(lcp != NULL) {
if(lcp->thousands_sep != NULL &&
*lcp->thousands_sep != '\0')
comma = *lcp->thousands_sep;
else comma = ',';
}
}
*p = '\0';
do {
```

```
if(i%3 == 0 && i != 0)
*--p = comma;
*--p = '0' + n % 10;
n /= 10;
i++;
} while(n != 0);
return p;
}
// macro to call clock_gettime w/different values for CLOCK
#define do_clock(CLOCK) \
do { \
   struct timespec x; \
   clock_getres(CLOCK, &x); \
   printf("%25s\t%15s", #CLOCK, commaprint(x.tv_nsec)); \
   clock_gettime(CLOCK, &x); \
   printf("\t%15s", commaprint(x.tv_sec)); \
   printf("\t%15s\n", commaprint(x.tv_nsec)); \
} while(0)
int main(int argc, char** argv )
{
   printf("%25s\t%15s\t%15s\t%15s\n", "clock", "res (ns)", "secs", "nsecs");
   struct timeval tv;
   gettimeofday(&tv, NULL);
   printf("%25s\t%15s\t%15s\t", "gettimeofday","1,000", commaprint(tv.tv_sec));
   printf("%15s\n", commaprint(tv.tv_usec*1000));

#if _POSIX_TIMERS > 0
   #ifdef CLOCK_REALTIME
   do_clock(CLOCK_REALTIME);
   #endif
   #ifdef CLOCK_REALTIME_COARSE
   do_clock(CLOCK_REALTIME_COARSE);
   #endif
   #ifdef CLOCK_REALTIME_HR
   do_clock(CLOCK_REALTIME_HR);
   #endif
   #ifdef CLOCK_MONOTONIC
   do_clock(CLOCK_MONOTONIC);
   #endif
   #ifdef CLOCK_MONOTONIC_RAW
   do_clock(CLOCK_MONOTONIC_RAW);
   #endif
```

```
   #ifdef CLOCK_MONOTONIC_COARSE
   do_clock(CLOCK_MONOTONIC_COARSE);
   #endif

#endif
   return 0;
}
```

A.1.3 ClockBench.cpp

```
/ Copyright 2014 by Bill Torpey. All Rights Reserved.
// This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs
3.0 United States License.
// http://creativecommons.org/licenses/by-nc-nd/3.0/us/deed.en
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <cmath>
using namespace std;
#define ONE_BILLION  1000000000L
// NOTE: if you change this value you prob also need to adjust later code that
does "& 0xff" to use modulo arithmetic instead
const int BUCKETS = 256;        // how many samples to collect per iteration
const int ITERS = 100;          // how many iterations to run
double CPU_FREQ = 1;
inline unsigned long long cpuid_rdtsc() {
  unsigned int lo, hi;
  asm volatile (
     "cpuid \n"
     "rdtsc"
   : "=a"(lo), "=d"(hi) /* outputs */
   : "a"(0)             /* inputs */
   : "%ebx", "%ecx");     /* clobbers*/
  return ((unsigned long long)lo) | (((unsigned long long)hi) << 32);
}
inline unsigned long long rdtsc() {
  unsigned int lo, hi;
  asm volatile (
     "rdtsc"
```

```
   : "=a"(lo), "=d"(hi) /* outputs */
   : "a"(0)             /* inputs */
   : "%ebx", "%ecx");     /* clobbers*/
  return ((unsigned long long)lo) | (((unsigned long long)hi) << 32);
}
inline unsigned long long rdtscp() {
  unsigned int lo, hi;
  asm volatile (
     "rdtscp"
   : "=a"(lo), "=d"(hi) /* outputs */
   : "a"(0)             /* inputs */
   : "%ebx", "%ecx");     /* clobbers*/
  return ((unsigned long long)lo) | (((unsigned long long)hi) << 32);
}
// macro to call clock_gettime w/different values for CLOCK
#define do_clock(CLOCK) \
do { \
   for (int i = 0; i < ITERS * BUCKETS; ++i) { \
      struct timespec x;  \
      clock_gettime(CLOCK, &x); \
      int n = i % (BUCKETS); \
      timestamp[n] = (x.tv_sec * ONE_BILLION) + x.tv_nsec; \
   } \
   deltaT x(#CLOCK, timestamp); \
   x.print(); \
} while(0)
struct deltaT   {
   deltaT(const char* name, vector<long> v) : sum(0), sum2(0), min(-1), max(0),
avg(0), median(0), stdev(0), name(name)
   {
      // create vector with deltas between adjacent entries
      x = v;
      count = x.size() -1;
      for (int i = 0; i < count; ++i) {
         x[i] = x[i+1] - x[i];
      }
      for (int i = 0; i < count; ++i) {
         sum    += x[i];
         sum2   += (x[i] * x[i]);
         if (x[i] > max) max = x[i];
         if ((min == -1) || (x[i] < min)) min = x[i];
      }
```

```
      avg = sum / count;
      median = min + ((max - min) / 2);
      stdev =  sqrt((count  * sum2 - (sum * sum)) / (count * count));
   }
   void print()
   {
      printf("%25s\t", name);
      printf("%ld\t%7.2f\t%7.2f\t%7.2f\t%7.2f\t%7.2f\n", count, min, max, avg,
median, stdev);
   }
   void dump(FILE* file)
   {
      if (file == NULL)
         return;
      fprintf(file, "%s", name);
      for (int i = 0 ; i < count; ++i ) {
         fprintf(file, "\t%ld", x[i]);
      }
      fprintf(file, "\n");
   }
   vector<long> x;
   long   count;
   double sum;
   double sum2;
   double min;
   double max;
   double avg;
   double median;
   double stdev;
   const char* name;
};
int main(int argc, char** argv)
{
   if (argc > 1) {
      CPU_FREQ = strtod(argv[1], NULL);
   }
   FILE* file = NULL;
   if (argc > 2) {
      file = fopen(argv[2], "w");
   }
   vector<long>    timestamp;
   timestamp.resize(BUCKETS);
```

```
   printf("%25s\t", "Method");
   printf("%s\t%7s\t%7s\t%7s\t%7s\t%7s\n", "samples", "min", "max", "avg", "median",
"stdev");
#if _POSIX_TIMERS > 0
   #ifdef CLOCK_REALTIME
   do_clock(CLOCK_REALTIME);
   #endif
   #ifdef CLOCK_REALTIME_COARSE
   do_clock(CLOCK_REALTIME_COARSE);
   #endif
   #ifdef CLOCK_REALTIME_HR
   do_clock(CLOCK_REALTIME_HR);
   #endif
   #ifdef CLOCK_MONOTONIC
   do_clock(CLOCK_MONOTONIC);
   #endif
   #ifdef CLOCK_MONOTONIC_RAW
   do_clock(CLOCK_MONOTONIC_RAW);
   #endif
   #ifdef CLOCK_MONOTONIC_COARSE
   do_clock(CLOCK_MONOTONIC_COARSE);
   #endif
#endif
   {
      for (int i = 0; i < ITERS * BUCKETS; ++i) {
         int n = i & 0xff;
         timestamp[n] = cpuid_rdtsc();
      }
      for (int i = 0; i < BUCKETS; ++i) {
         timestamp[i] = (long) ((double) timestamp[i] / CPU_FREQ);
      }
      deltaT x("cpuid+rdtsc", timestamp);
      x.print();
      x.dump(file);
   }
   // will throw SIGILL on machine w/o rdtscp instruction
   #ifdef RDTSCP
   {
      for (int i = 0; i < ITERS * BUCKETS; ++i) {
         int n = i & 0xff;
         timestamp[n] = rdtscp();
      }
```

```
      for (int i = 0; i < BUCKETS; ++i) {
         timestamp[i] = (long) ((double) timestamp[i] / CPU_FREQ);
      }
      deltaT x("rdtscp", timestamp);
      x.print();
      x.dump(file);
   }
   #endif
   {
      for (int i = 0; i < ITERS * BUCKETS; ++i) {
         int n = i & 0xff;
         timestamp[n] = rdtsc();
      }
      for (int i = 0; i < BUCKETS; ++i) {
         timestamp[i] = (long) ((double) timestamp[i] / CPU_FREQ);
      }
      deltaT x("rdtsc", timestamp);
      x.print();
      x.dump(file);
   }
   printf("Using CPU frequency = %f\n", CPU_FREQ);
   return 0;
}
```

A.1.4 ClockBench.cpp

```
// Copyright 2014 by Bill Torpey. All Rights Reserved.
// This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs
3.0 United States License.
// http://creativecommons.org/licenses/by-nc-nd/3.0/us/deed.en
/*
Compile with:
g++ -O3 -ggdb -o ClockBenchARM ClockBenchARM.cpp
*/
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
```

```
#include <vector>
#include <cmath>
using namespace std;
#define ONE_BILLION  1000000000L
#define BCM2708_ST_BASE 0x3F003000 /* Updated address for RP3 */
// NOTE: if you change this value you prob also need to adjust later code that
does "& 0xff" to use modulo arithmetic instead
const int BUCKETS = 256;         // how many samples to collect per iteration
const int ITERS = 100;           // how many iterations to run
double CPU_FREQ = 1;
volatile unsigned *TIMER_registers;
void sys_timer_init() {
    /* open /dev/mem */
    int TIMER_memFd;
    if ((TIMER_memFd = open("/dev/mem", O_RDWR/*|O_SYNC*/) ) < 0)
    {
        printf("can't open /dev/mem - need root ?\n");
        exit(-1);
    }
    /* mmap BCM System Timer */
    void *TIMER_map = mmap(
        NULL,
        4096, /* BLOCK_SIZE */
        PROT_READ /*|PROT_WRITE*/,
        MAP_SHARED,
        TIMER_memFd,
        BCM2708_ST_BASE
    );
    close(TIMER_memFd);
    if (TIMER_map == MAP_FAILED)
    {
        printf("mmap error %d\n", (int)TIMER_map);
        exit(-1);
    }
    TIMER_registers = (volatile unsigned *)TIMER_map;
}
unsigned int SysTick() {
    return TIMER_registers[1];
}
// macro to call clock_gettime w/different values for CLOCK
#define do_clock(CLOCK) \
do { \
```

```
   for (int i = 0; i < ITERS * BUCKETS; ++i) { \
      struct timespec x;  \
      clock_gettime(CLOCK, &x); \
      int n = i % (BUCKETS); \
      timestamp[n] = (x.tv_sec * ONE_BILLION) + x.tv_nsec; \
   } \
   deltaT x(#CLOCK, timestamp); \
   x.print(); \
} while(0)
struct deltaT   {
   deltaT(const char* name, vector<long> v) : sum(0), sum2(0), min(-1), max(0),
avg(0), median(0), stdev(0), name(name)
   {
      // create vector with deltas between adjacent entries
      x = v;
      count = x.size() -1;
      for (int i = 0; i < count; ++i) {
         x[i] = x[i+1] - x[i];
      }
      for (int i = 0; i < count; ++i) {
         sum    += x[i];
         sum2   += (x[i] * x[i]);
         if (x[i] > max) max = x[i];
         if ((min == -1) || (x[i] < min)) min = x[i];
      }
      avg = sum / count;
      median = min + ((max - min) / 2);
      stdev =  sqrt((count  * sum2 - (sum * sum)) / (count * count));
   }
   void print()
   {
      printf("%25s\t", name);
      printf("%ld\t%7.2f\t%7.2f\t%7.2f\t%7.2f\t%7.2f\n", count, min, max, avg,
median, stdev);
   }
   void dump(FILE* file)
   {
      if (file == NULL)
         return;
      fprintf(file, "%s", name);
      for (int i = 0 ; i < count; ++i ) {
         fprintf(file, "\t%ld", x[i]);
```

```
      }
      fprintf(file, "\n");
   }
   vector<long> x;
   long   count;
   double sum;
   double sum2;
   double min;
   double max;
   double avg;
   double median;
   double stdev;
   const char* name;
};
int main(int argc, char** argv)
{
   if (argc > 1) {
      CPU_FREQ = strtod(argv[1], NULL);
   }
   FILE* file = NULL;
   if (argc > 2) {
      file = fopen(argv[2], "w");
   }
   vector<long>    timestamp;
   timestamp.resize(BUCKETS);
  sys_timer_init();
   printf("%25s\t", "Method");
   printf("%s\t%7s\t%7s\t%7s\t%7s\t%7s\n", "samples", "min", "max", "avg", "median",
"stdev");
   for (int i = 0; i < ITERS * BUCKETS; ++i) {
     int n = i & 0xff;
     timestamp[n] = SysTick();
   }
   for (int i = 0; i < BUCKETS; ++i) {
     timestamp[i] = (long) ((double) timestamp[i] / CPU_FREQ);
   }
   deltaT x("SysTick", timestamp);
   x.print();
   x.dump(file);
#if _POSIX_TIMERS > 0
   #ifdef CLOCK_REALTIME
   do_clock(CLOCK_REALTIME);
```

```
   #endif
   #ifdef CLOCK_REALTIME_COARSE
   do_clock(CLOCK_REALTIME_COARSE);
   #endif
   #ifdef CLOCK_REALTIME_HR
   do_clock(CLOCK_REALTIME_HR);
   #endif
   #ifdef CLOCK_MONOTONIC
   do_clock(CLOCK_MONOTONIC);
   #endif
   #ifdef CLOCK_MONOTONIC_RAW
   do_clock(CLOCK_MONOTONIC_RAW);
   #endif
   #ifdef CLOCK_MONOTONIC_COARSE
   do_clock(CLOCK_MONOTONIC_COARSE);
   #endif
#endif
   printf("Using CPU frequency = %f\n", CPU_FREQ);
   return 0;
}
```

A.2 CPU Information Test

A.2.1 CPUInfo.sh

```
#!/bin/bash
echo "/proc/cpuinfo:"
cat /proc/cpuinfo | grep "model name"
echo
echo "Python platform module:"
python CPUInfo.py
if [[ $(lscpu | grep "Architecture" | grep "x86" | wc -l) -gt 0 ]] ; then
  echo
  echo "CPUID test"
  gcc -o cpuid CPUID.c
  ./cpuid
fi
```

A.2.2 CPUInfo.py

```
import platform
print("Processor Name: %s" % platform.processor())
system = platform.uname()
for i in system:
    print("Platform info: %s" % i)
```

A.2.3 CPUID.c

```
/*
Compile with:
gcc -o cpuid cpuid.c
Using code example from:
https://github.com/LordNoteworthy/al-khaser/blob/master/al-khaser/Anti%20VM/Generic.cpp
*/
#include <stdio.h>
static inline void native_cpuid(unsigned int *eax, unsigned int *ebx,
                                unsigned int *ecx, unsigned int *edx)
{
        /* ecx is often an input as well as an output. */
        asm volatile("cpuid"
            : "=a" (*eax),
              "=b" (*ebx),
              "=c" (*ecx),
              "=d" (*edx)
            : "0" (*eax), "2" (*ecx));
}
int main(int argc, char **argv)
{
  unsigned eax, ebx, ecx, edx;
  eax = 1; /* processor info and feature bits */
  native_cpuid(&eax, &ebx, &ecx, &edx);

  int hypervisor = (ecx >> 31) & 1;
  if (hypervisor == 1) {
    printf("Hypervisor detected\n");
  }
  else {
    printf("No hypervisor detected\n");
  }
```

```
}
```

A.3 CPU Execution Time Test

A.3.1 timer_test.c

```
/*
Compile with:
gcc -g -O0 -o timer_test timer_test.c -lrt
Examples of priveleged instructions:
- Memory address mapping
- Flush or invalidate data cache
- Invalidate TLB entries
- Load and read system registers
- Change processor modes from kernel to user
- Change the voltage and frequency of processor
- Halt a processor
- Reset a processor
- Perform I/O operations
https://www.cs.princeton.edu/courses/archive/fall10/cos318/lectures/OSStructure.pdf
-----------------------------
NTP client functions from:
David Lettier (C) 2014.

http://www.lettier.com/

NTP client.
https://github.com/lettier/ntpclient/blob/master/source/c/main.c
-----------------------------
*/
#include <time.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
```

```
#include <sys/mman.h>
void error( char* msg )
{
    perror( msg ); // Print the error message to stderr.
    exit( 0 ); // Quit the process.
}
double get_ext_timestamp()
{
int sockfd, n; // Socket file descriptor and the n return result from writing/reading
from the socket.
int portno = 123; // NTP UDP port number.
char* host_name = "us.pool.ntp.org"; // NTP server host-name.
// Structure that defines the 48 byte NTP packet protocol.
typedef struct
{
unsigned li   : 2;       // Only two bits. Leap indicator.
unsigned vn   : 3;       // Only three bits. Version number of the protocol.
unsigned mode : 3;       // Only three bits. Mode. Client will pick mode 3 for
client.
uint8_t stratum;         // Eight bits. Stratum level of the local clock.
uint8_t poll;            // Eight bits. Maximum interval between successive messages.
uint8_t precision;       // Eight bits. Precision of the local clock.
uint32_t rootDelay;      // 32 bits. Total round trip delay time.
uint32_t rootDispersion; // 32 bits. Max error aloud from primary clock source.
uint32_t refId;          // 32 bits. Reference clock identifier.
uint32_t refTm_s;        // 32 bits. Reference time-stamp seconds.
uint32_t refTm_f;        // 32 bits. Reference time-stamp fraction of a second.
uint32_t origTm_s;       // 32 bits. Originate time-stamp seconds.
uint32_t origTm_f;       // 32 bits. Originate time-stamp fraction of a second.
uint32_t rxTm_s;         // 32 bits. Received time-stamp seconds.
uint32_t rxTm_f;         // 32 bits. Received time-stamp fraction of a second.
uint32_t txTm_s;         // 32 bits and the most important field the client cares
about. Transmit time-stamp seconds.
uint32_t txTm_f;         // 32 bits. Transmit time-stamp fraction of a second.
} ntp_packet;                // Total: 384 bits or 48 bytes.
// Create and zero out the packet. All 48 bytes worth.
ntp_packet packet = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
memset( &packet, 0, sizeof( ntp_packet ) );
// Set the first byte's bits to 00,011,011 for li = 0, vn = 3, and mode = 3. The
rest will be left set to zero.
*( ( char * ) &packet + 0 ) = 0x1b; // Represents 27 in base 10 or 00011011 in
base 2.
```

```
// Create a UDP socket, convert the host-name to an IP address, set the port number,
// connect to the server, send the packet, and then read in the return packet.
struct sockaddr_in serv_addr; // Server address data structure.
struct hostent *server;      // Server data structure.
sockfd = socket( AF_INET, SOCK_DGRAM, IPPROTO_UDP ); // Create a UDP socket.
if ( sockfd < 0 )
error( "ERROR opening socket" );
server = gethostbyname( host_name ); // Convert URL to IP.
if ( server == NULL )
error( "ERROR, no such host" );
// Zero out the server address structure.
bzero( ( char* ) &serv_addr, sizeof( serv_addr ) );
serv_addr.sin_family = AF_INET;
// Copy the server's IP address to the server address structure.
bcopy( ( char* )server->h_addr, ( char* ) &serv_addr.sin_addr.s_addr, server->h_length
);
// Convert the port number integer to network big-endian style and save it to
the server address structure.
serv_addr.sin_port = htons( portno );
// Call up the server using its IP address and port number.
if ( connect( sockfd, ( struct sockaddr * ) &serv_addr, sizeof( serv_addr) ) <
0 )
error( "ERROR connecting" );
// Send it the NTP packet it wants. If n == -1, it failed.
n = write( sockfd, ( char* ) &packet, sizeof( ntp_packet ) );
if ( n < 0 )
error( "ERROR writing to socket" );
// Wait and receive the packet back from the server. If n == -1, it failed.
n = read( sockfd, ( char* ) &packet, sizeof( ntp_packet ) );
if ( n < 0 )
error( "ERROR reading from socket" );
// These two fields contain the time-stamp seconds as the packet left the NTP
server.
// The number of seconds correspond to the seconds passed since 1900.
// ntohl() converts the bit/byte order from the network's to host's "endianness".
packet.txTm_s = ntohl( packet.txTm_s ); // Time-stamp seconds.
packet.txTm_f = ntohl( packet.txTm_f ); // Time-stamp fraction of a second.
/* Doing some conversions to convert timestamp seconds and fractions to float
 */
double comb_secs = (packet.txTm_f * 0.0000000001); //adjusting fraction value
to correct decimal place
comb_secs += (u_int)packet.txTm_s;
```

```
return comb_secs;  /*  This returns value in seconds */
}
/* Using code from:
http://www.guyrutenberg.com/2007/09/22/profiling-code-using-clock_gettime/ */
struct timespec calc_time_diff(struct timespec start, struct timespec end)
{
  struct timespec temp;
  if ((end.tv_nsec-start.tv_nsec)<0) {
    temp.tv_sec = end.tv_sec-start.tv_sec-1;
    temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
  } else {
    temp.tv_sec = end.tv_sec-start.tv_sec;
    temp.tv_nsec = end.tv_nsec-start.tv_nsec;
  }
  return temp;
}
/* Tracks system time at the microsecond level  */
double int_timer(int scale, void (*functionPtr)(int))
{
        struct timeval start, end;
        gettimeofday(&start, NULL);
/*      Perform some operation */
        functionPtr(scale);
/*      End of operation*/
        gettimeofday(&end, NULL);
        double start_sec, end_sec, start_usec;
        start_sec = start.tv_sec + (start.tv_usec*.000001);
        end_sec = end.tv_sec + (end.tv_usec*.000001);
return end_sec - start_sec; /* Returns the time in seconds  */
}
/* Gets time from external NTS*/
double ext_timer(int scale, void (*functionPtr)(int))
{
        double start_stamp, end_stamp;
        start_stamp = get_ext_timestamp();
/*      Perform some operation */
        functionPtr(scale);
/*      End of operation */
        end_stamp = get_ext_timestamp();
return (end_stamp - start_stamp) * .000001;  /* Convert seconds to microseconds
*/
}
```

```
void both_timers(double *int_timer, double *ext_timer, int scale, void (*functionPtr)(in
{
struct timeval start, end;
double start_stamp, end_stamp;
start_stamp = get_ext_timestamp();
        gettimeofday(&start, NULL);
functionPtr(scale);
gettimeofday(&end, NULL);
end_stamp = get_ext_timestamp();
        double start_sec, end_sec, start_usec;
        start_sec = start.tv_sec + (start.tv_usec*.000001);
        end_sec = end.tv_sec + (end.tv_usec*.000001);
        *int_timer = end_sec - start_sec;
*ext_timer = end_stamp - start_stamp;
}
double get_timestamp_offset()
{
        double stamp1, stamp2;
        stamp1 = get_ext_timestamp();
        get_ext_timestamp();
        stamp2 = get_ext_timestamp();
        return stamp2 - stamp1;
}
double remove_timestamp_offset(double ext_timer, double offset)
{
        return ext_timer - (offset * 2);  //Removes twice the offset as each external
timing requires two time requests
}
void intense_function1()
{
        int a = 0;
        for (int i = 0; i < 100000000; i++) {
        a += 1;
        }
        printf("%i\n", a);
}
void mmap_function1(int scale)
{
for (int i = 0; i < (10 * scale); i++) {
/* Mapping to an anonomous region not connected to a file */
void * provided_region = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED
| MAP_ANON, -1, 0);
```

```
if (provided_region == MAP_FAILED)
printf("Memory Mapping Failed\n");
memset(provided_region, 1, sizeof(int)); //writes 1's to the entire memory region
if (msync(provided_region, sizeof(int), 0) != 0) //forces OS to ensure values
are written to memory.  May not be necessary...
printf("Issue with calling msync\n");
if (munmap(provided_region, sizeof(int)) != 0)
printf("Memory Unmapping Failed\n");
}
}
#ifndef __arm__
void call_CPUID()
{
    int i;
    unsigned int index = 0;
    unsigned int regs[4];
    int sum;
    char temp;
    __asm__ __volatile__(
#if defined(__x86_64__) || defined(_M_AMD64) || defined (_M_X64)
        "pushq %%rbx     \n\t" /* save %rbx */
#else
        "pushl %%ebx     \n\t" /* save %ebx */
#endif
        "cpuid            \n\t"
        "movl %%ebx ,%[ebx]  \n\t" /* write the result into output var */
#if defined(__x86_64__) || defined(_M_AMD64) || defined (_M_X64)
        "popq %%rbx \n\t"
#else
        "popl %%ebx \n\t"
#endif
        : "=a"(regs[0]), [ebx] "=r"(regs[1]), "=c"(regs[2]), "=d"(regs[3])
        : "a"(index));
    for (i=4; i<8; i++) {  //Don't need to output the CPUID info 1000 times...
        //printf("%c" ,((char *)regs)[i]);
        temp=((char *)regs)[i];
    }
    for (i=12; i<16; i++) {
        //printf("%c" ,((char *)regs)[i]);
        temp+=((char *)regs)[i];
    }
    for (i=8; i<12; i++) {
```

```
        //printf("%c" ,((char *)regs)[i]);
        temp+=((char *)regs)[i];
        temp++;
    }
    //printf("\n");
}
#else
void call_CPUID(){
return;
}
#endif
void CPUID_function1(int scale)
{
for (int i = 0; i < (10 * scale); i++) {
call_CPUID();
}
}
int main(int argc, char **argv)
{
int ext_timer_check = 1;
int scaling_check = 1;
if (argc > 1) {
for (int i = 1; i < argc; i++) {
if (!strcmp(argv[i], "-n"))
ext_timer_check = 0;
if (!strcmp(argv[i], "-s"))
scaling_check = 0;
if (!strcmp(argv[i], "-h")) {
printf("Usage: timer_test <option>\n-n: No external time check\n-s: No comparison
at increasing scales\n");
exit(1);
}
}
}
void (*functionPtr)(int) = &mmap_function1;
//void (*functionPtr)(int) = &CPUID_function1;
printf("Using mmap_function1\n");
printf("All values are in seconds\n\n");
/* Calculate the time it takes to acquire a timestamp using external timing*/
/* I don't think this is an accurate representation because sometimes the tested
function executes in less time than the offset */
double int_timer_val, ext_timer_val;
```

```
int scale = 10000;
/* Timing single execution at a scale of 10000  */
printf("Single execution at scale of %i (ie, %i-1 runs)\n", scale, scale);
if (ext_timer_check == 1) {
//both_timers(&int_timer_val, &ext_timer_val, 1000, &mmap_function1);
both_timers(&int_timer_val, &ext_timer_val, scale, functionPtr);
printf("Internal Timer: %.8f\n", int_timer_val);
printf("External Timer: %.8f\n", ext_timer_val);  // TO DO: see if we have to
account for the time it takes to collect a timestamp
printf("Difference of: %.8f\n", ext_timer_val - int_timer_val);
}
else {
printf("Internal Timer: %.8f\n", int_timer(scale, functionPtr));
}
/* Looking at difference in growing the scale */
if (scaling_check == 1) {
printf("Execution at scale of 100, 1,000, and 10,000 and the ratio of execution
time / scale\n");
        if (ext_timer_check == 1) {
int scale;
double int_scale_ratio, ext_scale_ratio;
scale = 100;
both_timers(&int_timer_val, &ext_timer_val, scale, functionPtr);
int_scale_ratio = int_timer_val / scale;
ext_scale_ratio = ext_timer_val / scale;
                printf("Internal Timer:\n  Execution at scale of %i / scale: %.8f\n",
scale, int_scale_ratio);
                printf("External Timer:\n  Execution at scale of %i / scale: %.8f\n",
scale, ext_scale_ratio);
                printf("Difference of: %.8f\n", ext_scale_ratio - int_scale_ratio);
                scale = 1000;
                both_timers(&int_timer_val, &ext_timer_val, scale, functionPtr);
                int_scale_ratio = int_timer_val / scale;
                ext_scale_ratio = ext_timer_val / scale;
                printf("Internal Timer:\n  Execution at scale of %i / scale: %.8f\n",
scale, int_scale_ratio);
                printf("External Timer:\n  Execution at scale of %i / scale: %.8f\n",
scale, ext_scale_ratio);
                printf("Difference of: %.8f\n", ext_scale_ratio - int_scale_ratio);
                scale = 10000;
                both_timers(&int_timer_val, &ext_timer_val, scale, functionPtr);
                int_scale_ratio = int_timer_val / scale;
```

```
                ext_scale_ratio = ext_timer_val / scale;
                printf("Internal Timer:\nExecution at scale of %i / scale: %.8f\n",
scale, int_scale_ratio);
                printf("External Timer:\nExecution at scale of %i / scale: %.8f\n",
scale, ext_scale_ratio);
                printf("Difference of: %.8f\n", ext_scale_ratio - int_scale_ratio);
        }
        else {
int scale;
double scale_ratio;
printf("Internal Timer:\n");
scale = 100;
scale_ratio = int_timer(scale, functionPtr) / scale;
         printf("Execution at scale of %i / scale: %.8f\n", scale, scale_ratio);
         scale = 1000;
         scale_ratio = int_timer(scale, functionPtr) / scale;
         printf("Execution at scale of %i / scale: %.8f\n", scale, scale_ratio);
         scale = 10000;
         scale_ratio = int_timer(scale, functionPtr) / scale;
            printf("Execution at scale of %i / scale: %.8f\n", scale, scale_ratio);
        }
}
return 0;
}
```

A.4 Artifacts in the File System Test

A.4.1 file_search.sh

```
#!/bin/bash
#
VMFILES='false'
#check for qemu files
if [[ $(locate qemu | wc -l) -gt 0 ]] ; then
  echo "Found qemu software files - can be standard in some Linux distros"
#  VMFILES='true'
fi
#check for bochs files
if [[ $(locate bochs | wc -l) -gt 0 ]] ; then
  echo "Found bochs software files - can be standard in some Linux distros"
```

```
#  VMFILES='true'
fi
#check for VirtualBox drivers
if [[ $(lspci -nn | grep -c -i VirtualBox) -gt 0 ]] ; then
  echo "Found VirtualBox drivers - Possibily VM"
  VMFILES='true'
fi
#on Betty Container, lspci gives error - possibly because RPis dont have PCI bus
if [[ $(lspci 2>&1 | grep -c Cannot) -gt 0 ]] ; then
  echo "lspci unable to view PCI bus, may be Raspberry Pi"
else
  #check for VMWare drivers
  if [[ $(lspci -nn | grep -c -i VMWare) -gt 0 ]] ; then
    echo "Found VMWare drivers - Possibily VM"
    VMFILES='true'
  fi
  #on Xen, lspci -nn returns nothing, which is odd. So test for it
  if [[ "$(lspci -nn)" == "" ]] ; then
    echo "lspci -nn not returning any values, - Possibly Xen"
    VMFILES='true'
  fi
fi
#test "$VMFILES" != 'true' && (echo No VM files found; exit 1)
if [[ "$VMFILES" != 'true' ]] ; then
  echo; echo "Did not detect Virtual Environment"
else
  echo; echo "Detected Virtual Environment"
fi
exit 1
```

A.5 Artifacts in System Hardware Attributes Test

A.5.1 hw_attribute_check.sh

```
#!/bin/bash
#
arm="0"
if [[ $(lscpu | grep "Architecture" | grep "arm" | wc -l) -gt 0 ]] ; then
  arm="1"
fi
```

```
if [[ $arm == "0" ]] ; then
  #Check to make sure lmbench package is installed
  #Check to ensure all required packages are installed
  if [[ $(dpkg-query -l php7.0-cli 2>&1 | grep -c ’no packages found’) -gt 0 ]]
; then
    echo "Missing php7.0 command line interface"
    echo "install with: sudo apt-get install php7.0-cli"
    exit
  fi
  if [[ $(dpkg-query -l php-curl 2>&1 | grep -c ’no packages found’) -gt 0 ]]
; then
    echo "Missing curl for php"
    echo "install with: sudo apt-get install php-curl"
    exit
  fi
fi
IS_VM=’false’
#check for VirtualBox NIC MAC
#check for VMWare NIC MAC
if [[ $(lshw -quiet -class network | grep serial | grep -i -c -E ’00:1c:14|00:0c:29|00:5
-gt 0 ]] ; then
  echo "Found VMWare NIC MAC address"
  IS_VM=’true’
fi
#check for Xen NIC MAC
if [[ $(lshw -quiet -class network | grep serial | grep -i -c -E ’00:16:3e’) -gt
0 ]] ; then
  echo "Found Xen NIC MAC address"
  IS_VM=’true’
fi
#check for QEMU NIC MAC
#QEMU source indicates this is the default MAC if non is specified
if [[ $(lshw -quiet -class network | grep serial | grep -i -c -E ’52:54:00’) -gt
0 ]] ; then
  echo "Found default QEMU NIC MAC address"
  IS_VM=’true’
fi
#check for bochs NIC MAC
#Bochs documentation recommends using b0:c4 for MAC and fe:fd:00... for ethertap
mode
#if [[ $(ifconfig | grep HWaddr | grep -i -c -E ’b0:c4|fe:fd:00’) -gt 0 ]] ; then
if [[ $(lshw -quiet -class network | grep serial | grep -i -c -E ’b0:c4|fe:fd:00’)
```

```
-gt 0 ]] ; then
  echo "Found a Bochs default NIC MAC address"
  IS_VM='true'
fi
if [[ $arm == "0" ]] ; then
  #if nothing has flagged for VM yet, then build program that pulls the system
mac and tests it against online database
  if [[ "$IS_VM" != 'true' ]] ; then
    LOOP_CNT=$(lshw -quiet -class network | grep serial: | grep -c "[A-Za-z0-9][A-Za-z0-
    echo "$LOOP_CNT MAC Addresses Found"
    REM_STRING=$(lshw -quiet -class network | grep serial: | grep -o "[A-Za-z0-9][A-Za-z
    for ((i=1; i<=$LOOP_CNT; i++))
    do
      CUR_STRING=$(echo $REM_STRING | grep -o "^[A-Za-z0-9][A-Za-z0-9]\:[A-Za-z0-9][A-Za
      echo "Checking MAC Address: $CUR_STRING"
      if [[ $(php -f ./checkMAC.php $CUR_STRING) == "Vendor: Vendor not found"
]] ; then
        echo "Locally Administered MAC: $CUR_STRING.  Possibly a VM"
        #if comes back as locally administered address and not tied to a non-VM
vendor, then flag
        IS_VM='true'
      fi
      #This is removing the Current String (and it's trailing space, if present)
from the Remaining String.
      REM_STRING=$(echo $REM_STRING | sed -r "s/\b$CUR_STRING[ ]?\b//g")
    done
  fi
fi
if [[ "$IS_VM" != 'true' ]] ; then
  echo; echo "Did not detect Virtual Environment"
else
  echo; echo "Detected Virtual Environment is present (although may not be running
in it)"
fi
exit 1
\end{spbervatim}

\subsection{checkMAC.php}
\begin{spverbatim}
<?php
  if(empty($argv[1])) {
    echo "Must include MAC";
```

```
    exit();
  } else {
    $mac_address = $argv[1];
  }
  $url = "http://api.macvendors.com/" . urlencode($mac_address);
  $ch = curl_init();
  curl_setopt($ch, CURLOPT_URL, $url);
  curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
  $response = curl_exec($ch);
  if($response) {
    echo "Vendor: $response";
  } else {
    echo "Server Not Found";
  }
?>
```

A.6 System Memory Performance Test

A.6.1 CPUID_TLB_flush.c

```
/*
Compile with:
gcc -g -o CPUID_cache_flush CPUID_cache_flush.c -lrt
Testing for presence of hardware-assisted VM by seeing if CPUID instruction results
in L2 cache being flushed from a hypervisor event.
This test was outlined in "Attacks on more Virtual Machine Emulators - 2007"
Going to load up memory with array of 0.05megs (50,000 Bytes) and a stride access
of 512 Bytes
L2 access times should range from 3.5 ns (Intel SmartCache) to 24.6 (ARM7)
Code for CPUID execution copied from:
https://en.wikipedia.org/wiki/CPUID#Calling_CPUID
!! TODO !!  Change this to straight variance check!
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define BUFFER_SIZE 50000
#define STRIDE 12500
void call_CPUID()
{
```

```
    int i;
    unsigned int index = 0;
    unsigned int regs[4];
    int sum;
    __asm__ __volatile__(
#if defined(__x86_64__) || defined(_M_AMD64) || defined (_M_X64)
        "pushq %%rbx     \n\t" /* save %rbx */
#else
        "pushl %%ebx     \n\t" /* save %ebx */
#endif
        "cpuid            \n\t"
        "movl %%ebx ,%[ebx]  \n\t" /* write the result into output var */
#if defined(__x86_64__) || defined(_M_AMD64) || defined (_M_X64)
        "popq %%rbx \n\t"
#else
        "popl %%ebx \n\t"
#endif
        : "=a"(regs[0]), [ebx] "=r"(regs[1]), "=c"(regs[2]), "=d"(regs[3])
        : "a"(index));
    for (i=4; i<8; i++) {
        printf("%c" ,((char *)regs)[i]);
    }
    for (i=12; i<16; i++) {
        printf("%c" ,((char *)regs)[i]);
    }
    for (i=8; i<12; i++) {
        printf("%c" ,((char *)regs)[i]);
    }
    printf("\n");
}
char *get_and_prep_buffer(int buffer_size)
{
  char *buffer = malloc(sizeof(char) * buffer_size); //this is our 50,000 byte
buffer
  if (buffer == NULL){
    printf("Failed to allocate buffer.  Exiting...\n");
    exit(1);
  }
  printf("Buffer allocated and prepped\n");
  return buffer;
}
//Read the entire buffer to get it loaded into cache
```

```
void buffer_warmup (char *buffer, int buffer_size)
{
  char temp;
  for (int i=0; i < buffer_size; i++) {
    temp=buffer[i];
    temp++;  //doing somethign with the data so the compiler doesn't just disregard
these reads
  }
}
/* Using code from:
http://www.guyrutenberg.com/2007/09/22/profiling-code-using-clock_gettime/ */
struct timespec calc_time_diff(struct timespec start, struct timespec end)
{
  struct timespec temp;
  if ((end.tv_nsec-start.tv_nsec)<0) {
    temp.tv_sec = end.tv_sec-start.tv_sec-1;
    temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
  } else {
    temp.tv_sec = end.tv_sec-start.tv_sec;
    temp.tv_nsec = end.tv_nsec-start.tv_nsec;
  }
  return temp;
}
int main(int argc, char **argv)
{
  //printf("Size of int: %lu\n", sizeof(int));  //this is 4 bytes on  both intel
and ARM systems
  //remember, a char is always 1 byte, so perhaps stick with that
  //Going to load up memory with array of 0.05megs (50,000 Bytes) and a stride
access of 512 Bytes
  int intervals = 5;
  int i;
  printf("Buffer size: %i\n", BUFFER_SIZE);
  printf("Stride size: %i\n", STRIDE);
  int pre_access_times[intervals];
  int post_access_times[intervals];
  for (i = 0; i < intervals ; i++) {
    char *buffer = get_and_prep_buffer(BUFFER_SIZE);
    char temp;
    //use CLOCK_MONOTONIC to track elapsed time
    struct timespec start_pre_time, end_pre_time, elapsed_pre_time, start_post_time,
end_post_time, elapsed_post_time;
```

```
    //clocking the latency before the L2 cache is warmed up
    //Do a stride read before filling the buffer
    clock_gettime(CLOCK_MONOTONIC, &start_pre_time);
    //Do the stride read here
    for (int i=0; i < BUFFER_SIZE; i+=STRIDE) {
      temp=buffer[i];
      temp++;  //doing somethign with the data so the compiler doesn't just disregard
these reads
    }
    //Trying the stride read in reverse in order to try to avoid stride-aware
pre-caching
    //Had about the same results as forward stride...  Perhaps the buffer is too
small to avoid being pre-fetched.
    clock_gettime(CLOCK_MONOTONIC, &end_pre_time);
    elapsed_pre_time = calc_time_diff(start_pre_time, end_pre_time);  //Calc elapsed
time
    //Going to assume it took less than a second
    printf("Elapsed Pre-CPUID time average before buffer warmup:  %f(ns)\n", elapsed_pre
/(float)(BUFFER_SIZE/STRIDE));
    //prep the L2 cache here
    buffer_warmup (buffer, BUFFER_SIZE);
    clock_gettime(CLOCK_MONOTONIC, &start_pre_time);
    //Do the stride read here
    for (int i=0; i < BUFFER_SIZE; i+=STRIDE) {
      temp=buffer[i];
      temp++;  //doing somethign with the data so the compiler doesn't just disregard
these reads
    }
    clock_gettime(CLOCK_MONOTONIC, &end_pre_time);
    elapsed_pre_time = calc_time_diff(start_pre_time, end_pre_time);  //Calc elapsed
time
    //Going to assume it took less than a second
    pre_access_times[i] = elapsed_pre_time.tv_nsec / (float)(BUFFER_SIZE/STRIDE);
    printf("Elapsed Pre-CPUID time average:  %f(ns)\n", elapsed_pre_time.tv_nsec
/ (float)(BUFFER_SIZE/STRIDE));
    call_CPUID();  //Execute CPUID instruction
    clock_gettime(CLOCK_MONOTONIC, &start_post_time);
    //Do the stride read here
    for (int i=0; i < BUFFER_SIZE; i+=STRIDE) {
      temp=buffer[i];
      temp++;  //doing somethign with the data so the compiler doesn't just disregard
these reads
```

```
    }
    clock_gettime(CLOCK_MONOTONIC, &end_post_time);
    elapsed_post_time = calc_time_diff(start_post_time, end_post_time);  //Calc
elapsed time
    //Again, I'm assuming it took less than a second
    post_access_times[i] = elapsed_post_time.tv_nsec / (float)(BUFFER_SIZE/STRIDE);
    printf("Elapsed Post-CPUID time average:  %f(ns)\n", elapsed_post_time.tv_nsec
/ (float)(BUFFER_SIZE/STRIDE));
    free(buffer);
  }
  int vm_presence = 0;
  for (i = 0; i < intervals; i++) {
    printf("post_access_time: %d\n", post_access_times[i]);
    printf("pre_access_time + .4: %f\n", pre_access_times[i] + (pre_access_times[i]
* .4));
    if (post_access_times[i] > (pre_access_times[i] + (pre_access_times[i] * .4)))
{ //this allows for slight variance on bare metal machines
      vm_presence = 1;
      break;
    }
  }
  if (vm_presence == 0) {
    printf("No Virtual Environment Detected\n");
  }
  else {
    printf("Virtual Environment Detected\n");
  }
  return 0;
}
```

A.7 Network Test

A.7.1 collect_timestamps.sh

```
#!/bin/bash
#Get all the parameters for timestamp capture job
target=""
while [ "$target" == "" ]
do
echo "Enter IP of target (XXX.XXX.XXX.XXX):"
```

```
read target
done
echo "Enter total number of TCP packets to capture: (100)"
read packet_cnt
if [[ "$packet_cnt" == "" ]] ; then
  packet_cnt="100"
fi
echo "Seconds between packet transmissions: (1)"
read packet_freq
if [[ "$packet_freq" == "" ]] ; then
  packet_freq="1"
fi
echo "Enter number of iterations: (note: I'm not validating input) (1):"
read iter
if [[ "$iter" == "" ]] ; then
  iter="1"
fi
echo "Enter filename for output (timestamp_capture_output.csv):"
read file
if [[ "$file" == "" ]] ; then
  file="timestamp_capture_output.csv"
fi
echo "Fingerprint target host with nmap? (y/N):"
read nmap
if [[ "$nmap" == "" ]] ; then
  nmap="n"
fi
#Do we want to attempt to fingerprint remote host Operating System?
if [[ "$nmap" == y* ]] ; then
  echo "Remotely fingerprinting OS with nmap"
  OS=$(sudo /usr/bin/nmap -O $target | grep "OS CPE")
  if [[ $(grep linux <<< "$OS" | wc -l) -gt 0 ]] ; then
    echo "Target is running Linux"
    echo "OS,Linux" > $file
  elif [[ $(grep windows <<< "$OS" | wc -l) -gt 0 ]] ; then
    echo "Target is running Windows"
    echo "OS,Windows" > $file
  else
    echo "Don't know OS type"
    echo $OS
    echo "OS,Unknown" > $file
  fi
```

```
else
  echo "OS, OS Detection Skipped" > $file
fi
#Now run packet capture against target
echo
#Compiles and checks to see if gcc/linker complains of any missing libraries
#if [[ $(gcc -Wall -g -o timestamp_capture timestamp_capture.c -lpcap -lgsl -lgslcblas
| grep "No such file" | wc -l) -gt 0 ]] ; then
#  echo "Error:  Missing libraries.  Need to install pcap/gsl?"
#  exit
#fi
errormsg=$(gcc -Wall -g -o timestamp_capture timestamp_capture.c -lpcap -lgsl
-lgslcblas 2>&1)
if [[ $(echo $errormsg | grep -c "No such file") -gt 0 ]] ; then
  echo $errormsg ; echo "Error:  Missing libraries.  Tools needed included: gcc,
pcap library, and GSL libraries"
  echo "Can be installed with:  apt-get install build-essential libpcap0.8-dev"
  echo "GSL libraries downloaded from: http://www.dvlnx.com/software/gnu/gsl/"
  echo "After compiling and installing GSL libraries, may need to run ldconfig"
  exit
elif [[ "$errormsg" != "" ]] ; then
  echo $errormsg
  exit
fi
echo "Capturing packets -- output being saved to $file; please wait"
#Start packet transmission in the background
echo
echo "Starting packet tranmission in background"
gcc -o tcpClientLinux tcpClientLinux.c -lnsl -lm
./tcpClientLinux "$target" "$packet_freq" &
packet_xmit_pid=$!
#Run timestamp_capture for the specified number of iterations and concatenating
results to timestamp_capture_output.csv
for ((i=1; i<=$iter; i++))
do
  echo "Iteration $i running"
  sudo ./timestamp_capture "$target" "$packet_cnt" >> $file
done
#Outputting results; unfortunately this will kind of be jumbled together -- any
formatting options for grep?
echo
replace_string=','
```

```
echo "Iteration Results:" >> $file
string=$(printf '%q\n' "$(cat $file | grep "Time taken")")
new_string=$(echo "${string//'\n'/$replace_string}")
echo -e $new_string >> $file
string=$(printf '%q\n' "$(cat $file | grep "Calculated MSE")")
new_string=$(echo "${string//'\n'/$replace_string}")
echo -e $new_string >> $file
string=$(printf '%q\n' "$(cat $file | grep "Theoretical Linux")")
new_string=$(echo "${string//'\n'/$replace_string}")
echo -e $new_string >> $file
string=$(printf '%q\n' "$(cat $file | grep "Theoretical Windows")")
new_string=$(echo "${string//'\n'/$replace_string}")
echo -e $new_string >> $file
string=$(printf '%q\n' "$(cat $file | grep "accurate")")
new_string=$(echo "${string//'\n'/$replace_string}")
echo -e $new_string >> $file
string=$(printf '%q\n' "$(cat $file | grep "too quickly")")
new_string=$(echo "${string//'\n'/$replace_string}")
echo -e $new_string >> $file
printf '%s\n' "$(cat $file | grep -A6 "Iteration Results")"
echo
#Stop the packet transmission
echo "Finished all iterations, stopping packet transmission (may need to enter
sudo passw)"
sudo kill 15 "$packet_xmit_pid"
```

A.7.2 timestamp_capture.c

```
/*
 * sniffex.c
 *
 * Sniffer example of TCP/IP packet capture using libpcap.
 *
 * Version 0.1.1 (2005-07-05)
 * Copyright (c) 2005 The Tcpdump Group
 *
 * This software is intended to be used as a practical example and
 * demonstration of the libpcap library; available at:
 * http://www.tcpdump.org/
 *
 ****************************************************************************
```

```
*
* This software is a modification of Tim Carstens' "sniffer.c"
* demonstration source code, released as follows:
*
* sniffer.c
* Copyright (c) 2002 Tim Carstens
* 2002-01-07
* Demonstration of using libpcap
* timcarst -at- yahoo -dot- com
*
* "sniffer.c" is distributed under these terms:
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
* 4. The name "Tim Carstens" may not be used to endorse or promote
*    products derived from this software without prior written permission
*
* THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
* <end of "sniffer.c" terms>
*
* This software, "sniffex.c", is a derivative work of "sniffer.c" and is
* covered by the following terms:
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
```

```
* 1. Because this is a derivative work, you must comply with the "sniffer.c"
*    terms reproduced above.
* 2. Redistributions of source code must retain the Tcpdump Group copyright
*    notice at the top of this source file, this list of conditions and the
*    following disclaimer.
* 3. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
* 4. The names "tcpdump" or "libpcap" may not be used to endorse or promote
*    products derived from this software without prior written permission.
*
* THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.
* BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY
* FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW.  EXCEPT WHEN
* OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES
* PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED
* OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
* MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS
* TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU.  SHOULD THE
* PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
* REPAIR OR CORRECTION.
*
* IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
* WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
* REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
* INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
* OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED
* TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
* YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
* PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGES.
* <end of "sniffex.c" terms>
*
****************************************************************************
*
* Below is an excerpt from an email from Guy Harris on the tcpdump-workers
* mail list when someone asked, "How do I get the length of the TCP
* payload?" Guy Harris' slightly snipped response (edited by him to
* speak of the IPv4 header length and TCP data offset without referring
* to bitfield structure members) is reproduced below:
*
* The Ethernet size is always 14 bytes.
```

```
*
* <snip>...</snip>
*
* In fact, you *MUST* assume the Ethernet header is 14 bytes, *and*, if
* you're using structures, you must use structures where the members
* always have the same size on all platforms, because the sizes of the
* fields in Ethernet - and IP, and TCP, and... - headers are defined by
* the protocol specification, not by the way a particular platform's C
* compiler works.)
*
* The IP header size, in bytes, is the value of the IP header length,
* as extracted from the "ip_vhl" field of "struct sniff_ip" with
* the "IP_HL()" macro, times 4 ("times 4" because it's in units of
* 4-byte words).  If that value is less than 20 - i.e., if the value
* extracted with "IP_HL()" is less than 5 - you have a malformed
* IP datagram.
*
* The TCP header size, in bytes, is the value of the TCP data offset,
* as extracted from the "th_offx2" field of "struct sniff_tcp" with
* the "TH_OFF()" macro, times 4 (for the same reason - 4-byte words).
* If that value is less than 20 - i.e., if the value extracted with
* "TH_OFF()" is less than 5 - you have a malformed TCP segment.
*
* So, to find the IP header in an Ethernet packet, look 14 bytes after
* the beginning of the packet data.  To find the TCP header, look
* "IP_HL(ip)*4" bytes after the beginning of the IP header.  To find the
* TCP payload, look "TH_OFF(tcp)*4" bytes after the beginning of the TCP
* header.
*
* To find out how much payload there is:
*
* Take the IP *total* length field - "ip_len" in "struct sniff_ip"
* - and, first, check whether it's less than "IP_HL(ip)*4" (after
* you've checked whether "IP_HL(ip)" is >= 5).  If it is, you have
* a malformed IP datagram.
*
* Otherwise, subtract "IP_HL(ip)*4" from it; that gives you the length
* of the TCP segment, including the TCP header.  If that's less than
* "TH_OFF(tcp)*4" (after you've checked whether "TH_OFF(tcp)" is >= 5),
* you have a malformed TCP segment.
*
* Otherwise, subtract "TH_OFF(tcp)*4" from it; that gives you the
```

```
* length of the TCP payload.
*
* Note that you also need to make sure that you don't go past the end
* of the captured data in the packet - you might, for example, have a
* 15-byte Ethernet packet that claims to contain an IP datagram, but if
* it's 15 bytes, it has only one byte of Ethernet payload, which is too
* small for an IP header.  The length of the captured data is given in
* the "caplen" field in the "struct pcap_pkthdr"; it might be less than
* the length of the packet, if you're capturing with a snapshot length
* other than a value >= the maximum packet size.
* <end of response>
*
****************************************************************************
*
* Example compiler command-line for GCC:
*   gcc -Wall -o timestamp_capture timestamp_capture.c -lpcap
*
*   NEED THE GSL LIBRARY INSTALLED!
*   gcc -Wall -g -o timestamp_capture timestamp_capture.c -lpcap -lgsl -lgslcblas
*
****************************************************************************
*
* Code Comments
*
* This section contains additional information and explanations regarding
* comments in the source code. It serves as documentaion and rationale
* for why the code is written as it is without hindering readability, as it
* might if it were placed along with the actual code inline. References in
* the code appear as footnote notation (e.g. [1]).
*
* 1. Ethernet headers are always exactly 14 bytes, so we define this
* explicitly with "#define". Since some compilers might pad structures to a
* multiple of 4 bytes - some versions of GCC for ARM may do this -
* "sizeof (struct sniff_ethernet)" isn't used.
*
* 2. Check the link-layer type of the device that's being opened to make
* sure it's Ethernet, since that's all we handle in this example. Other
* link-layer types may have different length headers (see [1]).
*
* 3. This is the filter expression that tells libpcap which packets we're
* interested in (i.e. which packets to capture). Since this source example
* focuses on IP and TCP, we use the expression "ip", so we know we'll only
```

```
 * encounter IP packets. The capture filter syntax, along with some
 * examples, is documented in the tcpdump man page under "expression."
 * Below are a few simple examples:
 *
 * Expression Description
 * ---------------------
 * ip Capture all IP packets.
 * tcp Capture only TCP packets.
 * tcp port 80 Capture only TCP packets with a port equal to 80.
 * ip host 10.1.2.3 Capture all IP packets to or from host 10.1.2.3.
 *
 ****************************************************************************
 *
 */
#define APP_NAME "sniffex"
#define APP_DESC "Sniffer example using libpcap"
#define APP_COPYRIGHT "Copyright (c) 2005 The Tcpdump Group"
#define APP_DISCLAIMER "THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM."
#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <sys/time.h>
#include <gsl/gsl_fit.h>
/* default snap length (maximum bytes per packet to capture) */
#define SNAP_LEN 1518
/* ethernet headers are always exactly 14 bytes [1] */
#define SIZE_ETHERNET 14
/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN 6
/* Ethernet header */
struct sniff_ethernet {
        u_char  ether_dhost[ETHER_ADDR_LEN];    /* destination host address */
        u_char  ether_shost[ETHER_ADDR_LEN];    /* source host address */
        u_short ether_type;                     /* IP? ARP? RARP? etc */
```

```
};
/* IP header */
struct sniff_ip {
        u_char  ip_vhl;                 /* version << 4 | header length >> 2 */
        u_char  ip_tos;                 /* type of service */
        u_short ip_len;                 /* total length */
        u_short ip_id;                  /* identification */
        u_short ip_off;                 /* fragment offset field */
        #define IP_RF 0x8000            /* reserved fragment flag */
        #define IP_DF 0x4000            /* dont fragment flag */
        #define IP_MF 0x2000            /* more fragments flag */
        #define IP_OFFMASK 0x1fff       /* mask for fragmenting bits */
        u_char  ip_ttl;                 /* time to live */
        u_char  ip_p;                   /* protocol */
        u_short ip_sum;                 /* checksum */
        struct  in_addr ip_src,ip_dst;  /* source and dest address */
};
#define IP_HL(ip)               (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip)                (((ip)->ip_vhl) >> 4)
/* TCP header */
typedef u_int tcp_seq;
struct sniff_tcp {
        u_short th_sport;               /* source port */
        u_short th_dport;               /* destination port */
        tcp_seq th_seq;                 /* sequence number */
        tcp_seq th_ack;                 /* acknowledgement number */
        u_char  th_offx2;               /* data offset, rsvd */
#define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
        u_char  th_flags;
        #define TH_FIN  0x01
        #define TH_SYN  0x02
        #define TH_RST  0x04
        #define TH_PUSH 0x08
        #define TH_ACK  0x10
        #define TH_URG  0x20
        #define TH_ECE  0x40
        #define TH_CWR  0x80
        #define TH_FLAGS        (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
        u_short th_win;                 /* window */
        u_short th_sum;                 /* checksum */
        u_short th_urp;                 /* urgent pointer */
//uint32_t th_opts; /* first 32 bits of options field */
```

```
};
/* TCP Timestamp Option Field struct.
 * Use __packed__ to prevent (disastrous!) compiler padding. */
struct __attribute__((__packed__)) tcp_timestamp_s {
char length;
uint32_t timestamp;
uint32_t timestamp_echo;
};
typedef struct tcp_timestamp_s tcp_timestamp_t;
/* Global variables for comparing packet captures */
struct timeval time_start;
uint32_t timestamp_start;
uint32_t timestamp_prev;
uint32_t last_timestamp_seen;
double sec_prev;
/*
Array for packet timestamp data and MSE calculation
Column 1: Elapsed Local Time
Column 2: Observed Clock Skew
Column 3: Squared Error */
#define MAX_PACKETS_CAPTURED 500
double data_array[3][MAX_PACKETS_CAPTURED];
int data_array_index = 0;
/* Mean of the squared error */
double mean_se;
/* Number of packets we'll capture to calculate the remote target clock frequency*/
#define FREQ_CACHE_MAX 5
/* Global variables for calculating remote target clock frequency*/
double freq_cache[FREQ_CACHE_MAX];
int freq_cache_cnt = 0;
double target_freq = 0;
/* Value to determine whether convergence was found for linear fit */
#define CONV_VALUE 0.0001
void
got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet);
void
print_payload(const u_char *payload, int len);
void
print_hex_ascii_line(const u_char *payload, int len, int offset);
void
print_app_banner(void);
void
```

```
print_app_usage(void);
/*
 * app name/banner
 */
void
print_app_banner(void)
{
printf("timestamp_capture.c, based on code by:\n");
printf("%s - %s\n", APP_NAME, APP_DESC);
printf("%s\n", APP_COPYRIGHT);
printf("%s\n", APP_DISCLAIMER);
printf("\n");
return;
}
/*
 * print help text
 */
void
print_app_usage(void)
{
printf("Usage: %s [interface]\n", APP_NAME);
printf("\n");
printf("Options:\n");
printf("    interface    Listen on <interface> for packets.\n");
printf("\n");
return;
}
/*
 * print data in rows of 16 bytes: offset   hex   ascii
 *
 * 00000   47 45 54 20 2f 20 48 54  54 50 2f 31 2e 31 0d 0a   GET / HTTP/1.1..
 */
void
print_hex_ascii_line(const u_char *payload, int len, int offset)
{
int i;
int gap;
const u_char *ch;
/* offset */
printf("%05d   ", offset);
/* hex */
ch = payload;
```

```
for(i = 0; i < len; i++) {
printf("%02x ", *ch);
ch++;
/* print extra space after 8th byte for visual aid */
if (i == 7)
printf(" ");
}
/* print space to handle line less than 8 bytes */
if (len < 8)
printf(" ");
/* fill hex gap with spaces if not full line */
if (len < 16) {
gap = 16 - len;
for (i = 0; i < gap; i++) {
printf("   ");
}
}
printf("   ");
/* ascii (if printable) */
ch = payload;
for(i = 0; i < len; i++) {
if (isprint(*ch))
printf("%c", *ch);
else
printf(".");
ch++;
}
printf("\n");
return;
}
/*
 * print packet payload data (avoid printing binary data)
 */
void
print_payload(const u_char *payload, int len)
{
int len_rem = len;
int line_width = 16; /* number of bytes per line */
int line_len;
int offset = 0; /* zero-based offset counter */
const u_char *ch = payload;
if (len <= 0)
```

```
return;
/* data fits on one line */
if (len <= line_width) {
print_hex_ascii_line(ch, len, offset);
return;
}
/* data spans multiple lines */
for ( ;; ) {
/* compute current line length */
line_len = line_width % len_rem;
/* print line */
print_hex_ascii_line(ch, line_len, offset);
/* compute total remaining */
len_rem = len_rem - line_len;
/* shift pointer to remaining bytes to print */
ch = ch + line_len;
/* add offset */
offset = offset + line_width;
/* check if we have line width chars or less */
if (len_rem <= line_width) {
/* print last line and get out */
print_hex_ascii_line(ch, len_rem, offset);
break;
}
}
return;
}
/*
 * dissect/print packet
 */
void
got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
/* declare pointers to packet headers */
__attribute__((unused)) const struct sniff_ethernet *ethernet;  /* The ethernet
header [1] */
const struct sniff_ip *ip;              /* The IP header */
const struct sniff_tcp *tcp;            /* The TCP header */
__attribute__((unused)) const char *payload;                    /* Packet payload
*/
int size_ip;
int size_tcp;
```

```
__attribute__((unused)) int size_payload;
/* define ethernet header */
ethernet = (struct sniff_ethernet*)(packet);
/* define/compute ip header offset */
ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
size_ip = IP_HL(ip)*4;
if (size_ip < 20) {
printf("   * Invalid IP header length: %u bytes\n", size_ip);
return;
}
/* determine protocol */
switch(ip->ip_p) {
case IPPROTO_TCP: /* It's a TCP packet, so proceed with processing */
break;
case IPPROTO_UDP:
printf("   Protocol: UDP\n");
return;
case IPPROTO_ICMP:
printf("   Protocol: ICMP\n");
return;
case IPPROTO_IP:
printf("   Protocol: IP\n");
return;
default:
printf("   Protocol: unknown\n");
return;
}
/*
 *  OK, this packet is TCP.
 */
/* define/compute tcp header offset */
tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
size_tcp = TH_OFF(tcp)*4;
if (size_tcp < 20) {
printf("   * Invalid TCP header length: %u bytes\n", size_tcp);
return;
}
/* define/compute tcp payload (segment) offset */
payload = (char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
/* compute tcp payload (segment) size */
size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);
if (size_tcp > 20) { /* If there are options in the TCP header, check to see what
```

```
they are */

char *opt_addr = (char *)tcp + 0x14;  //seperation of 14 between start of tcp
header and options
char *opt_kind_addr = opt_addr;
int hdr_left = size_tcp - 20;
opt_kind_addr = opt_kind_addr + 1;
hdr_left--;
int i = 3;  /* Only going to check the first three options fields for timestamps
*/
while (hdr_left > 0 && i > 0) {
if (*opt_kind_addr == 0x08) { /* this tcp packet has a timestamp */
/* Define the location of the timestamp field*/
tcp_timestamp_t *timestamp_field = (tcp_timestamp_t *)(opt_kind_addr + 0x1);
/* Avoid analyzing packets that have already been captured */
if (last_timestamp_seen == ntohl(timestamp_field->timestamp))
break;

printf("%u,", ntohl(timestamp_field->timestamp)); // TCP Packet Timestamp
struct timeval time_now;
gettimeofday(&time_now, NULL);
double start_sec, now_sec;
start_sec = time_start.tv_sec + (time_start.tv_usec*.000001); /* Convert microseconds
to seconds */
now_sec = time_now.tv_sec + (time_now.tv_usec*.000001); /* Convert microseconds
to seconds */
/* Save the first and previous timestamps for later calculations */
if (timestamp_start == 0) {
timestamp_start = ntohl(timestamp_field->timestamp);
last_timestamp_seen = timestamp_prev = ntohl(timestamp_field->timestamp);
sec_prev = now_sec;
printf("\n"); //Return carriage for blank entry at end of line
break;
}
/* Target freq already determined*/
if (target_freq != 0) {
                 //double local_time_elapsed = now_sec - start_sec;  /* Calc elapsed
number of seconds from start of capture*/
                 double time_diff = now_sec - start_sec;  /* Calc elapsed number
of seconds from start of capture*/
                 double time_since_prev_packet = now_sec - sec_prev; /* Calc elapsed
number of seconds since previous packet*/
```

```
                    //double target_time_elapsed = (ntohl(timestamp_field->timestamp)
- timestamp_start) / target_freq;
                    //double clock_skew = time_diff - target_time_elapsed;
                    double target_time_elapsed = (ntohl(timestamp_field->timestamp)
- timestamp_start) / target_freq;
                    double clock_skew = time_diff - target_time_elapsed;
uint32_t timestamp_diff = ntohl(timestamp_field->timestamp) - timestamp_prev;

            /* Filling up the data array for later calculations */
if (data_array_index < MAX_PACKETS_CAPTURED) {
         data_array[0][data_array_index] = time_diff;
data_array[1][data_array_index] = clock_skew;
data_array_index++;

/* Printing results to console */
printf("%f,", timestamp_diff / time_since_prev_packet); //Current raw TCP clock
freq
printf("%d,", (int)target_freq); // calculated target freq
                        printf("%f,", time_diff); //Local time elapsed
                        printf("%f\n", clock_skew);  //The clock skew
                    }
                    /* Setting values for next comparison */
last_timestamp_seen = timestamp_prev = ntohl(timestamp_field->timestamp);
sec_prev = now_sec;
}
/* Estimating host TCP clock freq */
else if ((timestamp_prev != 0) && ((now_sec - sec_prev) > 3)) {
uint32_t timestamp_diff = ntohl(timestamp_field->timestamp) - timestamp_prev;
double time_since_prev_packet = now_sec - sec_prev; /* Calc elapsed number of
seconds since previous packet*/
printf("%f,", timestamp_diff / time_since_prev_packet); //Current raw TCP clock
freq
printf(","); /* blank entry for calc target freq*/
printf("%f,", now_sec - start_sec); //Local time elapsed
printf("\n"); /* blank entry for clock skew */
freq_cache[freq_cache_cnt] = timestamp_diff / time_since_prev_packet;
freq_cache_cnt++;
if (freq_cache_cnt > (FREQ_CACHE_MAX - 1)) { /* We have enough samples to generate
target clock freq */
/* Find the mean of the collected estimates */
double freq_cache_sum = 0;
int i;
```

```
for (i = 1; i < FREQ_CACHE_MAX; i++) { /* We're skipping the first captured clock
freq since it's always 0 */
freq_cache_sum += freq_cache[i];
}
target_freq = freq_cache_sum / (FREQ_CACHE_MAX - 1);
/* Now round the raw target_freq to the nearest 50Mhz */
int ceil = 75;
while (target_freq > ceil) {
ceil += 50;
}
target_freq = ceil - 25; /* We now have our target host frequency */
}

/* Setting values for next comparison */
last_timestamp_seen = timestamp_prev = ntohl(timestamp_field->timestamp);
sec_prev = now_sec;
}

/* This isn't the first timestamp captured, but not enough time has passed to
use this timestamp to calculate the target frequency */
else {
last_timestamp_seen = ntohl(timestamp_field->timestamp);
printf("\n");  //Return carriage for blank entry at end of line
}
break; // finished processing timestamp option so stop looking for more options
}
opt_kind_addr = opt_kind_addr + 1;
hdr_left--;
i--;
}
}
return;
}
/* String concatenation:  returns NULL if failure, else if success */
/* Format:  result = concat(&source1, &source2)*/
/* Used code from here as guide: http://stackoverflow.com/questions/8465006/how-to-conca
char* concat(const char *s1, const char *s2) {
char *result = malloc(strlen(s1)+strlen(s2)+1); // +1 for zero-terminator
if (result == NULL)
return NULL;
strcpy(result, s1);
strcat(result, s2);
```

```
return result;
}
int main(int argc, char **argv)
{
char *dev = NULL; /* capture device name */
char errbuf[PCAP_ERRBUF_SIZE]; /* error buffer */
pcap_t *handle; /* packet capture handle */
char init_filter_exp[] = "tcp and src host ";
char *filter_exp;
struct bpf_program fp; /* compiled filter program (expression) */
bpf_u_int32 mask; /* subnet mask */
bpf_u_int32 net; /* ip */
int num_packets = 100; /* number of packets to capture */
print_app_banner();
/* Initialize Global Var to be tested later */
timestamp_prev = 0;
/* get target ip from command-line*/
if (argc < 2) {
fprintf(stderr, "error: use format: timestamp_capture <target ip> <number of packets>\n\
exit(EXIT_FAILURE);
}
else if (argc > 3) {
fprintf(stderr, "error: unrecognized command-line options\n\n");
exit(EXIT_FAILURE);
}
else {
/* concatenate original filter with host passed by command-line*/
filter_exp = concat(init_filter_exp, argv[1]);
num_packets = atoi(argv[2]);
/* find a capture device */
dev = pcap_lookupdev(errbuf);
if (dev == NULL) {
fprintf(stderr, "Couldn't find default device: %s. sudo?\n",
    errbuf);
exit(EXIT_FAILURE);
}
}

/* get network number and mask associated with capture device */
if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
fprintf(stderr, "Couldn't get netmask for device %s: %s\n",
    dev, errbuf);
```

```
net = 0;
mask = 0;
}
/* print capture info */
printf("Device: %s\n", dev);
printf("Number of total tcp packets being collected: %d\n", num_packets);
printf("Filter expression: %s\n", filter_exp);

/* open capture device */
handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
if (handle == NULL) {
fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
exit(EXIT_FAILURE);
}
/* make sure we're capturing on an Ethernet device [2] */
if (pcap_datalink(handle) != DLT_EN10MB) {
fprintf(stderr, "%s is not an Ethernet\n", dev);
exit(EXIT_FAILURE);
}
/* compile the filter expression */
if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
fprintf(stderr, "Couldn't parse filter %s: %s\n",
    filter_exp, pcap_geterr(handle));
exit(EXIT_FAILURE);
}
/* apply the compiled filter */
if (pcap_setfilter(handle, &fp) == -1) {
fprintf(stderr, "Couldn't install filter %s: %s\n",
    filter_exp, pcap_geterr(handle));
exit(EXIT_FAILURE);
}
/*Print out column headers */
printf("\ntimestamp, raw target freq, calc target freq, elapsed local time, clock
skew\n"); //Title for nearly full output
struct timeval time_end;
gettimeofday(&time_start, NULL);
/* now we can set our callback function */
pcap_loop(handle, num_packets, got_packet, NULL);
gettimeofday(&time_end, NULL);
/* cleanup */
pcap_freecode(&fp);
pcap_close(handle);
```

```
printf("\nCapture complete\n");
if (target_freq == 0) { /* a target frequency was not calculated */
printf("Did not calculate target host clock frequency:  Not capture enough packets
or packet capture completed too quickly?\n");
exit(EXIT_FAILURE);
}
    double start_sec, end_sec;
    start_sec = time_start.tv_sec + (time_start.tv_usec*.000001);
    end_sec = time_end.tv_sec + (time_end.tv_usec*.000001);
printf("Time taken for capture: %.f (s)\n", end_sec - start_sec);
if ((end_sec - start_sec) < 60)
printf("**Capture completed quickly, MSE may not be accurate**\n");
        /* Linear Curve Fitting of the plots */
printf("Number of captured packets for analysis: %d\n", data_array_index);
        int i;
//        int sumx = 0, sumy = 0, sumxy = 0, sumx2 = 0;
        float slope, y_intercept;
//        for (i = 0; i < data_array_index; i++) {
//                sumx += data_array[0][i];
//                sumx2 += (data_array[0][i] * data_array[1][i]);
//                sumy += data_array[1][i];
//                sumxy = sumxy + (data_array[0][i] * data_array[1][i]);
//        }
/* Calculating linear regression using GSL library (because doing it by hand was
generating poor fit) */
/* int gsl_fit_linear (const double * x, const size_t xstride, const double *
y, const size_t ystride, size_t n, double * c0, double * c1, double * cov00, double
* cov01, double * cov11, double * sumsq) */
/* Personal note: stride is the seperation between two elements in an array in
increments of size double.  If it's an array of doubles, the stride is 1*/
double c0, c1, cov00, cov01, cov11, sumsq;
/* Finding fit over total number of packets collected */
gsl_fit_linear (data_array[0], 1, data_array[1], 1, data_array_index, &c0, &c1,
&cov00, &cov01, &cov11, &sumsq);
printf("Fitted line using gsl: y = %.5fx + %.5f\n", c1, c0);
slope = c1;
y_intercept = c0;
    /* Calculate the Squared Error: SE_i = [f(x_i)-y_i]^2 */
    for (i = 0; i < data_array_index; i++) {
        float error = (((slope * data_array[0][i]) + y_intercept) - data_array[1][i]);
        data_array[2][i] = error * error;
    }
```

```
    /* Calculate Sample Mean of Squared Error (MSE): sum(all SEs) / number of
SEs */
    float prevSum_SE = 0;
    for (i = 0; i < (data_array_index - 1); i++) {
        prevSum_SE += data_array[2][i];
    }
    int cur_pack_index = (data_array_index - 1);
double conv_check = (prevSum_SE + (data_array[2][cur_pack_index]) / prevSum_SE);
printf("prevSum_SE: %.12f\nNext SE: %.12f\nconv_check: %.12f\n", prevSum_SE, data_array[
conv_check);
if (conv_check >= CONV_VALUE)
printf("Convergence check value of %.12f is greater than required value of %.12f.
 May not have good fit\n", conv_check, CONV_VALUE);
else
printf("Convergence check value of %.12f is less than required value of %.12f.
 Should have good fit\n", conv_check, CONV_VALUE);
    float sum_SE = prevSum_SE + data_array[2][cur_pack_index];
    float MSE = sum_SE / data_array_index;
printf("Calculated MSE = %f ms^2\n", MSE * 1000000);
/* Theoretical MSE */
/* Linux Hosts */
double h = 1 / target_freq;
printf("Theoretical Linux MSE: %f ms^2\n", ((h * h) / 12.0) * 1000000);
/* Windows is fixed?*/
printf("Theoretical Windows MSE: 1667 ms^2\n");
return 0;
}
```

A.7.3 tcpServerLinux.c

```
//====================================================== file = tcpServer.c =====
//=  A message "server" program to demonstrate sockets programming
=
//===============================================================================
//=  Notes:
=
//=    1) This program compiles for BSD as appropriate.
=
//=    2) This program serves a message to program tcpClient running on
=
//=       another host.
```

```
=
//=    3) The steps #'s correspond to lecture topics.
=
//=------------------------------------------------------------------------------=
//=  Build: gcc -o tcpServerLinux tcpServerLinux.c -lnsl for BSD
                       =
//================================================================================
#define  BSD
//----- Include files ------------------------------------------------------------
#include <stdio.h>          // Needed for printf()
#include <string.h>         // Needed for memcpy() and strcpy()
#include <stdlib.h>         // Needed for exit()
#include <sys/types.h>      // Needed for sockets stuff
#include <netinet/in.h>     // Needed for sockets stuff
#include <sys/socket.h>     // Needed for sockets stuff
#include <arpa/inet.h>      // Needed for sockets stuff
#include <fcntl.h>          // Needed for sockets stuff
#include <netdb.h>          // Needed for sockets stuff
#include <linux/net_tstamp.h>
#include <unistd.h>
#include <netinet/tcp.h>
//----- Defines ------------------------------------------------------------------
#define  PORT_NUM   1050    // Arbitrary port number for the server
//===== Main program =============================================================
int main()
{
  // Important links
  // https://www.freebsd.org/cgi/man.cgi?query=setsockopt
  // https://www.freebsd.org/cgi/man.cgi?query=tcp&sektion=4&apropos=0&manpath=FreeBSD+1
  // https://www.freebsd.org/cgi/man.cgi?query=getprotoent&sektion=3&apropos=0&manpath=F
  int                  welcome_s;       // Welcome socket descriptor
  struct sockaddr_in   server_addr;     // Server Internet address
  int                  connect_s;       // Connection socket descriptor
  struct sockaddr_in   client_addr;     // Client Internet address
  struct in_addr       client_ip_addr;  // Client IP address
  int                  addr_len;        // Internet address length
  char                 out_buf[4096];   // Output buffer for data
  char                 in_buf[4096];    // Input buffer for data
  int                  retcode;         // Return code
  int        value = 1; // This is the optval used in setsockopt()
  // >>> Step #1 <<<
  // Create a welcome socket
```

```
//   - AF_INET is Address Family Internet and SOCK_STREAM is streams
//welcome_s = socket(AF_INET, SOCK_STREAM, 0);
welcome_s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (welcome_s < 0)
{
  printf("*** ERROR - socket() failed \n");
  exit(-1);
}
// >>> Step #2 <<<
// Fill-in server (my) address information and bind the welcome socket
server_addr.sin_family = AF_INET;                 // Address family to use
server_addr.sin_port = htons(PORT_NUM);           // Port number to use
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);  // Listen on any IP address
retcode = bind(welcome_s, (struct sockaddr *)&server_addr,
  sizeof(server_addr));
if (retcode < 0)
{
  printf("*** ERROR - bind() failed \n");
  exit(-1);
}
// >>> Step #3 <<<
// Listen on welcome socket for a connection
listen(welcome_s, 1);
// >>> Step #4 <<<
// Accept a connection.  The accept() will block and then return with
// connect_s assigned and client_addr filled-in.
printf("Waiting for accept() to complete... \n");
addr_len = sizeof(client_addr);
connect_s = accept(welcome_s, (struct sockaddr *)&client_addr, &addr_len);
if (connect_s < 0)
{
  printf("*** ERROR - accept() failed \n");
  exit(-1);
}
// Copy the four-byte client IP address into an IP address structure
memcpy(&client_ip_addr, &client_addr.sin_addr.s_addr, 4);
// Print an informational message that accept completed
printf("Accept completed (IP address of client = %s  port = %d) \n",
  inet_ntoa(client_ip_addr), ntohs(client_addr.sin_port));
// >>> Step #5 <<<
// Sets socket TCP option to TCP_NODELAY so that there is no delay
// TCP packets will be sent as soon as there is data available
```

```
  /* Example Below
  setsockopt(sock, SOL_SOCKET, SO_TIMESTAMPING, &timestamp_flags, sizeof(timestamp_flags
< 0) {
        die("setsockopt()");*/
  retcode = setsockopt(connect_s, IPPROTO_TCP, TCP_NODELAY,
    (char *)&value, sizeof(int));
  if (retcode < 0)
  {
    printf("*** ERROR - setsockopt() failed \n");
    exit(-1);
  }
  /*
  // >>> Step #5 <<<
  // Send to the client using the connect socket
  strcpy(out_buf, "This is a message from SERVER to CLIENT");
  retcode = send(connect_s, out_buf, (strlen(out_buf) + 1), 0);
  if (retcode < 0)
  {
    printf("*** ERROR - send() failed \n");
    exit(-1);
  }
  */
  // >>> Step #6 <<<
  // Receive from the client using the connect socket
  //int counter;
  //for (counter = 0; counter < 10; counter++) {
  while (1) {
    retcode = recv(connect_s, in_buf, sizeof(in_buf), 0);
    if (retcode < 0)
    {
      printf("*** ERROR - recv() failed \n");
      exit(-1);
    }
    //printf("Received from client: %s \n", in_buf);
    printf("%s \n", in_buf);
    in_buf[0] = 'X';
  }
  // >>> Step #7 <<<
  // Close the welcome and connect sockets
  retcode = close(welcome_s);
  if (retcode < 0)
  {
```

```
    printf("*** ERROR - close() failed \n");
    exit(-1);
  }
  retcode = close(connect_s);
  if (retcode < 0)
  {
    printf("*** ERROR - close() failed \n");
    exit(-1);
  }
  // Return zero and terminate
  return(0);
}
```

A.7.4 tcpClientLinux.c

```
//===================================================== file = tcpClient.c =====
//=  A message "client" program to demonstrate sockets programming
=
//==============================================================================
//=  Notes:
=
//=    1) This program compiles for BSD sockets.
=
//=    2) This program needs tcpServer to be running on another host.
=
//=       Program tcpServer must be started first.
=
//=    3) This program assumes that the IP address of the host running
=
//=       tcpServer is defined in "#define IP_ADDR"
=
//=    4) The steps #'s correspond to lecture topics.
=
//=----------------------------------------------------------------------------=
//=  Build: gcc -o tcpClientLinux tcpClientLinux.c -lnsl -lm for BSD
                           =
//==============================================================================
#define  BSD
//----- Include files ---------------------------------------------------------
#include <stdio.h>          // Needed for printf()
#include <string.h>         // Needed for memcpy() and strcpy()
```

```
#include <stdlib.h>         // Needed for exit()
#include <sys/types.h>      // Needed for sockets stuff
#include <netinet/in.h>     // Needed for sockets stuff
#include <sys/socket.h>     // Needed for sockets stuff
#include <arpa/inet.h>      // Needed for sockets stuff
#include <fcntl.h>          // Needed for sockets stuff
#include <netdb.h>          // Needed for sockets stuff
#include <linux/net_tstamp.h>
#include <unistd.h>
#include <netinet/tcp.h>
#include <math.h>
#include <time.h>
//----- Defines -------------------------------------------------------------
#define  PORT_NUM         1050   // Port number used at the server
//#define  IP_ADDR    "127.0.0.1"  // IP address of server (*** HARDWIRED ***)
#define  IP_ADDR    "172.30.206.30" //"128.173.239.245"  // IP address of server
(*** HARDWIRED ***)
char *target_ip_addr;
double tcpFreq;
struct timespec sleep_time;
//===== Assign String =======================================================
void assignString(char *bufferCount, int *count) {

  int mod = *count % 5;
  switch(mod) {
    case 0:
      *count = 1;
      bufferCount[0] = '5';
      break;
    case 1:
      *count += 1;
      bufferCount[0] = '1';
      break;
    case 2:
      *count += 1;
      bufferCount[0] = '2';
      break;
    case 3:
      *count += 1;
      bufferCount[0] = '3';
      break;
    case 4:
```

```
      *count += 1;
      bufferCount[0] = '4';
      break;
    default:
      printf("*** ERROR - assignString() failed \n");
      break;
  }
}
//===== Main program ======================================================
int main(int argc, char **argv)
{
  // Important links
  // https://www.freebsd.org/cgi/man.cgi?query=setsockopt
  // https://www.freebsd.org/cgi/man.cgi?query=tcp&sektion=4&apropos=0&manpath=FreeBSD+1
  // https://www.freebsd.org/cgi/man.cgi?query=getprotoent&sektion=3&apropos=0&manpath=F
  int                  client_s;        // Client socket descriptor
  struct sockaddr_in   server_addr;     // Server Internet address
  char                 out_buf[4096];   // Output buffer for data
  char                 in_buf[4096];    // Input buffer for data
  int                  retcode;         // Return code
  int        value = 1; // This is the optval used in setsockopt()
 if (argc < 3) {
   printf("*** ERROR - usage: tcpClientLinux <target ip> <seconds per tcp packet>\n");
   exit(-1);
 }
 else {
   target_ip_addr = argv[1];
   tcpFreq = atof(argv[2]);
 }
 sleep_time.tv_sec = floor(tcpFreq);
 int mod = 1;
 if (sleep_time.tv_sec > mod)
   mod = sleep_time.tv_sec;
 sleep_time.tv_nsec = fmod(tcpFreq, mod) * 1000000000;
 printf("Transmitting packets to %s every %d second(s) and %ld nano second(s)\n",
target_ip_addr, (int)sleep_time.tv_sec, sleep_time.tv_nsec);
  // >>> Step #1 <<<
  // Create a client socket
  //   - AF_INET is Address Family Internet and SOCK_STREAM is streams
  //client_s = socket(AF_INET, SOCK_STREAM, 0);
  //while(1) {
  client_s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
if (client_s < 0)
{
  printf("*** ERROR - socket() failed \n");
  exit(-1);
}
// >>> Step #2 <<<
// Fill-in the server's address information and do a connect with the
// listening server using the client socket - the connect() will block.
server_addr.sin_family = AF_INET;                 // Address family to use
server_addr.sin_port = htons(PORT_NUM);           // Port num to use
server_addr.sin_addr.s_addr = inet_addr(target_ip_addr); // IP address to use
retcode = connect(client_s, (struct sockaddr *)&server_addr,
  sizeof(server_addr));
if (retcode < 0)
{
  printf("*** ERROR - connect() failed \n");
  exit(-1);
}
// >>> Step #3 <<<
// Sets socket TCP option to TCP_NODELAY so that there is no delay
// TCP packets will be sent as soon as there is data available
retcode = setsockopt(client_s, IPPROTO_TCP, TCP_NODELAY,
  (char *)&value, sizeof(int));
if (retcode < 0)
{
  printf("*** ERROR - setsockopt() failed \n");
  exit(-1);
}
/*
// >>> Step #3 <<<
// Receive from the server using the client socket
retcode = recv(client_s, in_buf, sizeof(in_buf), 0);
if (retcode < 0)
{
  printf("*** ERROR - recv() failed \n");
  exit(-1);
}
// Output the received message
printf("Received from server: %s \n", in_buf);
*/
// >>> Step #4 <<<
// Send to the server using the client socket
```

```
  //strcpy(out_buf, "This is a reply message from CLIENT to SERVER");
  int count = 1;
  char bufferCount[2];
  while(1)
  {
    assignString(bufferCount, &count);
    //strcpy(out_buf, "This is a message from CLIENT to SERVER");
    strcpy(out_buf, bufferCount);
    retcode = send(client_s, out_buf, (strlen(out_buf) + 1), 0);
    if (retcode < 0)
    {
      printf("*** ERROR - send() failed \n");
      exit(-1);
    }
    nanosleep(&sleep_time, (struct timespec *)NULL);
  }
  // >>> Step #5 <<<
  // Close the client socket
  retcode = close(client_s);
  if (retcode < 0)
  {
    printf("*** ERROR - close() failed \n");
    exit(-1);
  }
  //}
  // Return zero and terminate
  return(0);
}
```

A.8 Analysis Tool Detection Test

A.8.1 self_status.c

```
/*
code example from:
http://stackoverflow.com/questions/3596781/how-to-detect-if-the-current-process-is-being
Sam Liao
edited Dec 10 '14 at 16:32
compile with:
gcc -o self_status self_status.c
```

```
*/
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char **argv)
{
    char buf[1024];
    int debugger_present = 0;
    int status_fd = open("/proc/self/status", O_RDONLY);
    if (status_fd == -1)
        return 0;
    size_t num_read = read(status_fd, buf, sizeof(buf));
    if (num_read > 0)
    {
        static const char TracerPid[] = "TracerPid:";
        char *tracer_pid;
        buf[num_read] = 0;
        tracer_pid    = strstr(buf, TracerPid);
        if (tracer_pid)
            debugger_present = !!atoi(tracer_pid + sizeof(TracerPid) - 1);
    }
    if (debugger_present)
        printf("Debugger detected in /proc/self/status \n");
    else
        printf("No debugger detected in /proc/self/status \n");
    return 0;
}
```

A.9 Container Detection Tests

A.9.1 detect_lxc.c

```
// This very small application is able to differentiate between a normal Linux
environment and an LXC environment.
// This is because output from ps does not correlate consistently with results
from sysinfo().
// By comparing the output of both ps and sysinfo(), we can differentiate between
```

```
a regular environment and an LXC environment.
/*
source:  https://raw.githubusercontent.com/mempodippy/vlany/master/misc/detect_lxc.c
Jean-Philippe states it's "Part of the LD_PRELOAD-based vlany rootkit"
compile with:
gcc -g -o detect_lxc detect_lxc.c
*/
#include <stdio.h>
#include <sys/sysinfo.h>
int main(int argc, char *argv[])
{
    int count;
    FILE *ps = popen("ps axHo lwp,cmd|wc -l", "r");
    fscanf(ps, "%d", &count);
    pclose(ps);
    struct sysinfo si;
    sysinfo(&si);
    printf("# of sysinfo processes: %i\n", si.procs);
    printf("# of ps processes: %i\n\n", count);
    if(si.procs - 200 > count && argc == 1) { printf("[!] definitely in LXC\n");
return 1; }
    printf("[+] either not in LXC or an LXC environment was not detected\n");
return 0;
}
```

A.9.2 perm_test.sh

```
#!/bin/bash
#attempts to output various hardware characteristics that require root privileges
to access
#checking for root
if [[ "$(whoami)" != "root" ]] ; then
  echo "Must be root to run"
  exit
fi
output=$(dmidecode 2>&1)
if [[ $(echo $output | grep -c 'Permission denied') -gt 0 ]] ; then
  echo "Unable to access system info areas.  Probably container"
elif [[ $(echo $output | grep -c 'No such file or directory') -gt 0  ]] ; then
  echo "Probably container in Ubuntu Core"
else
```

```
  echo "Probably not container"
fi
```

A.10 Raspberry Pi Detection Tests

A.10.1 io_perf.c

```
/*
compile with:
gcc -o io_perf io_perf.c
!! Change this to a simple throughput check since it appears that there's a common
ratio for the jump in record size regardless of medium
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
struct eight_megs {
double contents[1024];  //A double is 8 bytes and I want the struct to be 8kB
};
struct small_record {  //this is the 32kB record size
double contents[4];
};
struct large_record {  //this is the 128kB record size
double contents[16];
};
void purge_buffer_cache()
{
//TODO:  Write buffer cache purge
//iozone does this by unmounting drive - can't do that
//try to read/write unintialized files a whole bunch...
for (int i = 0; i <= 15; i++) {
struct eight_megs *eight_megs = malloc(sizeof(struct eight_megs));
FILE *fp = fopen("cache_purge_tmp", "w");
if (fwrite(eight_megs, 1, sizeof(struct eight_megs), fp) != sizeof(struct eight_megs))
{
         printf("Error writing temp file\n");
                exit(1);
        }
        fclose(fp);
        fp = fopen("cache_purge_tmp", "r");
```

```
        unsigned long num_chunks = sizeof(struct eight_megs) / sizeof(struct small_recor
        unsigned long read_size = sizeof(struct small_record);
        int chunks_read = fread(eight_megs, read_size, num_chunks, fp);
fclose(fp);
        fp = fopen("cache_purge_tmp", "r");
        fseek(fp, SEEK_SET, 0);  //restart at the beginning of the file
        num_chunks = sizeof(struct eight_megs) / sizeof(struct large_record);
        read_size = sizeof(struct large_record);
        chunks_read = fread(eight_megs, read_size, num_chunks, fp);
fclose(fp);
        }
        if (remove("cache_purge_tmp")) {
                perror("remove");
                exit(1);
}
}
int main(int argc, char *argv[])
{
//MAYBE TRY TO CLEAR DISK CACHE BEFORE DOING TEST?
FILE *fp = fopen("io_perf_temp", "w");
struct eight_megs *eight_megs = malloc(sizeof(struct eight_megs));
if (fwrite(eight_megs, 1, sizeof(struct eight_megs), fp) != sizeof(struct eight_megs))
{
printf("Error writing temp file\n");
exit(1);
}
fclose(fp);
fp = fopen("io_perf_temp", "r");
unsigned long num_chunks = sizeof(struct eight_megs) / sizeof(struct small_record);
unsigned long read_size = sizeof(struct small_record);
printf("Reading file in %lu chunks of %lu bytes\n", num_chunks, read_size);
purge_buffer_cache();
struct timeval start, end;
gettimeofday(&start, NULL);
int chunks_read = fread(eight_megs, read_size, num_chunks, fp);
gettimeofday(&end, NULL);
if (chunks_read != num_chunks) {
printf("Error reading temp file\n");
exit(1);
}
double start_sec, end_sec, start_usec;
start_sec = start.tv_sec + (start.tv_usec*.000001);
```

```
end_sec = end.tv_sec + (end.tv_usec*.000001);
double elapsed_sec = end_sec - start_sec;
double small_speed = (sizeof(struct eight_megs)/elapsed_sec)/1000;
printf("Small record read at %.0f kbytes/sec\n", small_speed);
fseek(fp, SEEK_SET, 0);  //restart at the beginning of the file
num_chunks = sizeof(struct eight_megs) / sizeof(struct large_record);
read_size = sizeof(struct large_record);
printf("Reading file in %lu chunks of %lu bytes\n", num_chunks, read_size);
purge_buffer_cache();
        gettimeofday(&start, NULL);
        chunks_read = fread(eight_megs, read_size, num_chunks, fp);
        gettimeofday(&end, NULL);
        if (chunks_read != num_chunks) {
                printf("Error reading temp file\n");
                exit(1);
        }
        start_sec = start.tv_sec + (start.tv_usec*.000001);
        end_sec = end.tv_sec + (end.tv_usec*.000001);
        elapsed_sec = end_sec - start_sec;
double large_speed = (sizeof(struct eight_megs)/elapsed_sec)/1000;
        printf("large record read at %.0f kbytes/sec\n", large_speed);
printf("ratio of small/large: %f\n", small_speed / large_speed);
if (large_speed < 1000000) {
printf("*** Probably a SBC ***\n");
}
else {
printf("*** Probably not a SBC ***\n");
}
fclose(fp);
if (remove("io_perf_temp")) {
perror("remove");
exit(1);
}
}
```

A.10.2 io_perf2.c

```
/*
compile with:
gcc -o io_perf2 io_perf2.c
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
struct eight_megs {
double contents[1048576];  //A double is 8 bytes and I want the struct to be 8192kB
}
;
struct one_meg {
double contents[131072];  //A double is 8 bytes and I want the struct to be 1024kB
};
struct quarter_meg {
double contents[32768];  //A double is 8 bytes and I want the struct to be 64kb
};
struct small_record {  //this is the 32kB record size
double contents[4096];
};
struct large_record {  //this is the 128kB record size
double contents[16384];
};
void purge_buffer_cache()
{
//TODO:  Write buffer cache purge
//iozone does this by unmounting drive - can't do that
//try to read/write unintialized files a whole bunch...
for (int i = 0; i <= 15; i++) {
struct eight_megs *eight_megs = malloc(sizeof(struct eight_megs));
FILE *fp = fopen("cache_purge_tmp", "w");
if (fwrite(eight_megs, 1, sizeof(struct eight_megs), fp) != sizeof(struct eight_megs))
{
         printf("Error writing temp file\n");
                exit(1);
        }
        fclose(fp);
        fp = fopen("cache_purge_tmp", "r");
        unsigned long num_chunks = sizeof(struct eight_megs) / sizeof(struct small_recor
        unsigned long read_size = sizeof(struct small_record);
        int chunks_read = fread(eight_megs, read_size, num_chunks, fp);
fclose(fp);
        fp = fopen("cache_purge_tmp", "r");
        fseek(fp, SEEK_SET, 0);  //restart at the beginning of the file
        num_chunks = sizeof(struct eight_megs) / sizeof(struct large_record);
        read_size = sizeof(struct large_record);
```

```
        chunks_read = fread(eight_megs, read_size, num_chunks, fp);
fclose(fp);
        }
        if (remove("cache_purge_tmp")) {
                perror("remove");
                exit(1);
}
}
int main(int argc, char *argv[])
{
//MAYBE TRY TO CLEAR DISK CACHE BEFORE DOING TEST?
int read_iter = 25;  //the number of iterations performed for reading
double small_read_speed[read_iter];
double large_read_speed[read_iter];
FILE *fp = fopen("io_perf_temp", "w");
//struct eight_megs *eight_megs = malloc(sizeof(struct eight_megs));
struct one_meg *one_meg = malloc(sizeof(struct one_meg));
struct quarter_meg *quarter_meg = malloc(sizeof(struct quarter_meg));
if (fwrite(one_meg, 1, sizeof(struct one_meg), fp) != sizeof(struct one_meg))
{
printf("Error writing temp file\n");
exit(1);
}
fclose(fp);
fp = fopen("io_perf_temp", "r");
unsigned long num_chunks = sizeof(struct one_meg) / sizeof(struct small_record);
unsigned long read_size = sizeof(struct small_record);
printf("Reading file in %lu chunks of %lu bytes\n", num_chunks, read_size);
int chunks_read;
double start_sec, end_sec, start_usec, elapsed_sec;
int i;
struct timeval start, end;
for (i = 0; i < read_iter; i++) {
//   purge_buffer_cache();
  gettimeofday(&start, NULL);
  chunks_read = fread(one_meg, read_size, num_chunks, fp);
  gettimeofday(&end, NULL);
  if (chunks_read != num_chunks) {
  printf("Error reading temp file\n");
  exit(1);
  }
  start_sec = start.tv_sec + (start.tv_usec*.000001);
```

```
  end_sec = end.tv_sec + (end.tv_usec*.000001);
  elapsed_sec = end_sec - start_sec;
  double small_speed = (sizeof(struct one_meg)/elapsed_sec)/1000;
  printf("Small record read at %.0f kbytes/sec\n", small_speed);
  small_read_speed[i] = small_speed;
  fseek(fp, SEEK_SET, 0);  //restart at the beginning of the file
}
num_chunks = sizeof(struct one_meg) / sizeof(struct large_record);
read_size = sizeof(struct large_record);
printf("Reading file in %lu chunks of %lu bytes\n", num_chunks, read_size);
for (i = 0; i < read_iter; i++) {
//   purge_buffer_cache();
  gettimeofday(&start, NULL);
  chunks_read = fread(one_meg, read_size, num_chunks, fp);
          gettimeofday(&end, NULL);
          if (chunks_read != num_chunks) {
                  printf("Error reading temp file\n");
                  exit(1);
          }
          start_sec = start.tv_sec + (start.tv_usec*.000001);
          end_sec = end.tv_sec + (end.tv_usec*.000001);
          elapsed_sec = end_sec - start_sec;
  double large_speed = (sizeof(struct one_meg)/elapsed_sec)/1000;
          printf("large record read at %.0f kbytes/sec\n", large_speed);
  large_read_speed[i] = large_speed;
  fseek(fp, SEEK_SET, 0);  //restart at the beginning of the file
}
//Compare small and large read speeds
//If any performances of large reads are lower than the lowest performance of
small read, then raspberry pi
int sbc_found = 1;
double slowest_small_read = small_read_speed[1]; //starting at 1 to disregard
the first access
for (i = 1; i < read_iter; i++) {
  if (small_read_speed[i] < slowest_small_read)
slowest_small_read = small_read_speed[i];
}
for (i = 1; i < read_iter; i++) {
  if (large_read_speed[i] > slowest_small_read) {
    sbc_found = 0;
    printf("Large record read speed of %f is faster than slowest small record
read speed of %f\n", large_read_speed[i], slowest_small_read);
```

```
    break;
  }
}
if (sbc_found == 1) {
printf("*** Probably a SBC ***\n");
}
else {
printf("*** Probably not a SBC ***\n");
}
//Clean up
fclose(fp);
if (remove("io_perf_temp")) {
perror("remove");
exit(1);
}
free(one_meg);
free(quarter_meg);
}
```