Remote Fidelity of Container-Based Network Emulators

Submitted in fulfilment
of the requirements of the degree of

Master of Science

of Rhodes University

Schalk Willem Peach

ORCID 0000-0002-2451-2006
https://orcid.org/

Grahamstown, South Africa
January 2020

Abstract

This thesis examines if Container-Based Network Emulators (CBNEs) are able to instantiate emulated nodes that provide sufficient realism to be used in information security experiments. The realism measure used is based on the information available from the point of view of a remote attacker.

During the evaluation of a Container-Based Network Emulator (CBNE) as a platform to replicate production networks for information security experiments, it was observed that `nmap` fingerprinting returned Operating System (OS) family and version results inconsistent with that of the host Operating System (OS). CBNEs utilise Linux namespaces, the technology used for containerisation, to instantiate "emulated" hosts for experimental networks. Linux containers partition resources of the host OS to create lightweight virtual machines that share a single OS kernel. As all emulated hosts share the same kernel in a CBNE network, there is a reasonable expectation that the fingerprints of the host OS and emulated hosts should be the same.

Based on how CBNEs instantiate emulated networks and that fingerprinting returned inconsistent results, it was hypothesised that the technologies used to construct CBNEs are capable of influencing fingerprints generated by utilities such as `nmap`. It was predicted that hosts emulated using different CBNEs would show deviations in remotely generated fingerprints when compared to fingerprints generated for the host OS.

An experimental network consisting of two emulated hosts and a Layer 2 switch was instantiated on multiple CBNEs using the same host OS. Active and passive fingerprinting was conducted between the emulated hosts to generate fingerprints and OS family and version matches. Passive fingerprinting failed to produce OS family and version matches as the fingerprint databases for these utilities are no longer maintained. For active fingerprinting the OS family results were consistent between tested systems and the host OS, though OS version results reported was inconsistent. A comparison of the generated fingerprints revealed that for certain CBNEs fingerprint features related to network stack optimisations of the host OS deviated from other CBNEs and the host OS.

The hypothesis that CBNEs can influence remotely generated fingerprints was partially confirmed. One CBNE system modified Linux kernel networking options, causing a deviation from fingerprints generated for other tested systems and the host OS. The hypothesis was also partially rejected as the technologies used by CBNEs do not influence the remote fidelity of emulated hosts.

Acknowledgements

I would like to thank my family, especially my wife, for the support (and coffee) provided. Without you, I would not have been able to maintain focus and complete this journey.

I would like to thank my supervisor, Prof. Barry Irwin, for his insight, guidance, support, and patience throughout my research. Above all, I would like to thank Prof. Irwin for the suggestion of a rather obscure set of tests to be conducted, that turned out to reveal hidden secrets of computer systems.

ACM Computing Classification System Classification 2012

- **Security and privacy** → **Virtualization and security**
- **Networks** → **Network experimentation**
- **Networks** → **Network security**
- **Networks** → *Network simulations*
- **Networks** → *Network architectures*
- **Networks** → *Network protocols*

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API Application Programming Interface

ARP Address Resolution Protocol

BSD Berkeley Software Distribution

CBE Container-Based Emulator

CBNE Container-Based Network Emulator

CLI Command Line Interface

CORE Common Open Research Emulator

CPU Central Processing Unit

DMZ Demilitarised Zone

EDVAC Electronic Discrete Variable Automatic Computer

EMANE Extendable Mobile Ad-hoc Network Emulator

GENI Global Environment for Network Innovations

GNS3 Graphical Network Simulator 3

GUI Graphical User Interface

HAL Hardware Abstraction Layer

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

ICMP Internet Control Message Protocol

ICS Industrial Control System

IDPS Intrusion Detection and Prevention System

IMUNES Integrated Multiprotocol Network Emulator/Simulator

IOS Internetwork Operating System

IoT Internet of Things

IP Internet Protocol

ISA Instruction Set Architecture

KVM Kernel-Based Virtual Machine

L3 OSI Layer 3

L7 OSI Layer 7

LAN Local Area Network

LXC Linux Containers

MAC Media Access Control

MIPS Microprocessor without Interlocked Pipelined Stages

MSS Maximum Segment Size

Multics Multiplexed Information and Computing Service

NEP Network Experimentation Platform

NGFW Next Generation Firewall

NIC Network Interface Controller

OS Operating System

RAM Random Access Memory

RFC Request for Comment

RPC Remote Procedure Call

RTT Round Trip Time

SCADA Supervisory Control and Data Acquisition

SDN Software Defined Networking

SONAR Sound Navigation and Ranging

TCP Transmission Control Protocol

TTL Time To Live

UDP User Datagram Protocol

UI User Interface

UML User Mode Linux

UTM Unified Threat Management

VDE Virtual Distributed Ethernet

VM Virtual Machine

VMM Virtual Machine Monitor

VNUML Virtual Network User Mode Linux

VNX Virtual Networks over Linux

Chapter 1

Introduction

Anyone who considers protocol unimportant
has never dealt with a cat.

Robert A. Heinlein

As long as communications technologies have been available, intrusions into the private communications of others have been a goal of malicious actors. The first recorded intrusion into secure communications through the exploitation of technology occurred during a demonstration of Guglielmo Marconi's wireless telegraphy technology (Marks, 2011) . As John Ambrose Fleming was in the final stages of preparing one side of a 300 mile transmission demonstration with Guglielmo Marconi, the letters "R A T S" repeatedly printed out on Fleming's machine. The perpetrator, Nevil Maskelyne, a competing inventor, had devised a method to interfere with Marconi's demonstration.

As communication technology became more sophisticated, attack methodologies on these technologies became more sophisticated, and the first known and recorded intrusion into a computer network occurred in 1967 (Falkoff, 1991). A group of students were given access to IBMs APL laboratory, and by learning the internals of the systems, the students were able to take control of major parts of the system. This intrusion into the computer network led to a testing methodology where hackers are used to test the security of a network. This methodology later became known as network penetration testing.

Conducting network penetration testing on production networks is risky and could disrupt services (Türpe and Eichler, 2009). An alternative solution is to use a testbed, which is a physical copy of the production network. Replicating large networks using physical hardware is impractical due to both size and cost constraints. However, advances in the

virtualisation of computer and network equipment has enabled physical machines to be partitioned into multiple virtual systems (Smith and Nair, 2005). Contemporary Network Experimentation Platforms (NEPs) - such as MiniNet (Heller, 2013) - have evolved to exploit these technologies, increasing the number of experimental network nodes that a single physical machine can instantiate. The introduction of kernel virtualisation and link emulation technologies in Linux and FreeBSD presented an opportunity to create NEPs using a single commodity desktop computer. These systems became known as Container-Based Emulators (CBEs) (Handigol *et al.*, 2012) and enabled experimentation with network protocols and distributed system design using consumer laptops. To prevent confusion with other types of emulators, these systems will be referred to as Container-Based Network Emulators (CBNEs).

A selection of open source CBNEs using Linux kernel virtualisation and network virtualisation technologies were selected as the test platforms for this study. An initial study of the technologies and architectures of the selected CBNEs indicated that a variety of open source containerisation and network virtualisation technologies that can have an influence on network traffic were used to construct the individual systems. To confirm that Operating System (OS) fingerprinting is used by remote attackers, four classes of network attack models were reviewed. From the reviewed literature it was established that there is a reasonable expectation for fingerprinting of OSs to be used in attacks targeted at specific machines and during penetration testing. The techniques and technologies used to construct a fingerprint for an OS from network traffic were investigated for the ability to detect changes brought on by abstracting a computer system. Most of the features extracted by fingerprint utilities relate to changes in the source code of an OS. Knowing that the construction of CBNEs can influence OS fingerprints and that fingerprinting utilities are capable of detecting certain changes, it was predicted that different CBNE platforms instantiating the same experimental network would deliver different fingerprints for hosts in the network.

1.1 Problem Statement

The scale and complexity of modern computer networks, as well as the diversity of software packages that make up such networks, presents a challenge for testing the security of the overall system. Conducting penetration tests on “live” networks may be detrimental to the functioning of the network. Testing the security of such large systems typically requires playgrounds or sandboxes that can replicate critical components and a large enough segment of the operational network to represent realistic operational environments. These

playgrounds and sandboxes must have the ability to accurately replicate the conditions of the network, taking into account the distribution of network and end-user device types. Testbeds and virtualisation systems are viable solutions in circumstances where the size or cost of constructing a physical sandbox is a prohibiting factor.

Simulated networks do not necessarily present a network penetration testing team with sufficient realism to conduct a network penetration test or train staff. A stochastic model of a computer system is not capable of accounting for variations and flaws that could creep in during the implementation of network protocols, performance metrics, and OSs (Rampfl, 2013; Sharif and Sadeghi-Niaraki, 2017). An alternative technology in the form of CBNEs - training and experimentation networks created using OS level virtualisation (Section 2.2.3) - presents a middle ground in terms of cost and realism when a Unix based network is subjected to a penetration test.

The primary question that arises for this research is whether or not a CBNE can emulate a network of Linux devices with sufficient realism to be used as a platform for information security experiments.

1.2 Research Outline

The research conducted was structured to address the following three research objectives:

1. Conduct literature surveys on the techniques and technologies used to:
   (a) create abstracted computer systems such as virtual machines and containers.
   (b) remotely fingerprint a computer system.
2. Develop a model that can measure the realism of an abstracted computer system using fingerprints generated by active and passive fingerprinting utilities.
3. Apply the model to a selection of open source CBNEs based on Linux namespaces and assess the suitability of these systems to be used as experimental platforms for information security research, education, and training.

1.3 Research Method

To answer the question of whether or not a CBNE can accurately emulate a computer network for information security experimentation, the research was conducted as a descriptive and explanatory study based on action and experimental research methods. A

literature review was conducted in three parts. The first literature review was conducted to assess the types of platforms that are used to create computer networks for research and experimentation. The second literature review was conducted to assess the current state of open source CBNEs that utilise virtualisation technologies of the Linux kernel. The third literature review was conducted to assess the techniques and technologies used to fingerprint computer systems from the perspective of a remote attacker. The information gained from the literature reviews were used to create an abstract model that attempts to define the fidelity of an abstracted machine as seen from the perspective of a remote attacker. Finally, an experimental platform was built to enable testing and validation of the research hypothesis and the model for remote fidelity.

1.4 Document Conventions

A `monospace` font will be used to indicate that an executable application is being referred to. For example, the active fingerprinting utility `nmap` will always be rendered in a monospace font.

An *italic* font will be used when referring to an item in a table or a component of a diagram.

Port number and protocol pairings will be written in the short-hand form of port/protocol. As an example when writing 22/tcp the port number is 22 and the Layer 2 protocol used is TCP. The same convention will be applied to Layer 7 protocols, for example 80/tcp refers to the OSI Layer 7 protocol HTTP over port number 80.

The term Container-Based Emulator (CBE) as defined by Handigol *et al.* (2012) will not be used. In its place the term Container-Based Network Emulator (CBNE) will be used to distinguish it from other types of emulator systems.

The term Network Experimentation Platform (NEP) will be used to refer to systems that are used to construct and instantiate computer networks for experimentation, irrespective of the abstraction technology used.

Throughout the document certain technical terms could have different meanings based on the context within which these terms are used. Utilisation of these terms will have a particular interpretation in Chapter 2, and a general interpretation in the rest of the document. These terms are defined below.

Emulation: In Chapter 2 the term emulator will be used to refer to Instruction Set Architecture (ISA) level virtualisation. Throughout the rest of the document the terms emulator and emulated will be used to refer to CBNEs and hosts instantiated by CBNEs, respectively.

Virtualisation: In Chapter 2 the term virtualisation and virtual machine will exclusively be used to refer to machines created using Type I and Type II Virtual Machine Monitors (VMMs). Throughout the rest of the document the terms virtualisation will be used to refer to any type of abstraction system or mechanism, and virtual machine will refer to any abstracted computer system instantiated using an abstraction system or mechanism. Simulated computer systems are excluded from this use.

1.5 Document Structure

The remainder of this thesis is structured as follows:

Chapter 2 introduces the concept of Network Experimentation Platforms (NEPs). NEPs exploit the hierarchical nature of computer systems and the opportunities for abstraction created by this hierarchical organisation. This chapter explores how different abstraction techniques influence the realism of abstracted hosts.

Chapter 3 presents an overview of a selection of open source CBNEs, a type of NEP that utilises Linux containerisation technologies. The architecture and technologies used to construct these systems are investigated for the ability to modify network traffic and influence remotely generated fingerprints.

Chapter 4 explores the techniques used by remote attackers to construct a fingerprint for a targeted machine. The techniques and technologies used by a remote attacker to generate fingerprints are then used to construct a model for measuring the realism of an abstracted host as seen from the perspective of a remote attacker.

Chapter 5 presents results obtained from generating active and passive fingerprints for selected open source CBNEs and findings on the ability of CBNEs to alter remotely generated fingerprints are reported.

Chapter 6 gives a conclusion on the work conducted during this research and discusses opportunities for future work related to specific discoveries made during experimentation.

Chapter 2

Network Experimentation Platforms

Two elements are needed to form a truth
—a fact and an abstraction.

Remy de Gourmont

Securing computer networks, from small scale Local Area Networks (LANs) to global Content Distribution Networks, is a daunting yet crucial task. Techniques and technologies used by network penetration testers and security researchers to secure computer networks against attacks have the capability to disrupt entire networks, even when used with proper care (Türpe and Eichler, 2009). Network Experimentation Platforms (NEPs) (Pediaditakis *et al.*, 2014) originated as physical test beds of computer and networking equipment that replicated production networks to assist developers in testing systems and applications in "real world" conditions. These systems have found an application in education and network security testing (van Heerden *et al.*, 2013; Browne *et al.*, 2018) to combat the disruption of network services during experimentation and testing. By using experimentation platforms that replicate the topologies and configurations of production networks, security professionals can apply testing procedures to replicated networks without fear of disrupting critical services. Testing of networks can be done using more aggressive techniques, increasing the probability of finding weaknesses that will only present themselves under extreme circumstances. Mitigations to weaknesses and vulnerabilities discovered in the replicated network can then be applied to the production network.

Replicating an enterprise network using physical hardware is often impractical and prohibitively expensive. Advances made in hardware and software technologies that abstract computer systems, such as virtualisation (Section 2.2.2) and containerisation (Section

2.2.3), have enabled NEPs to increase the density (number of abstracted machines per physical machine) at which experimentation networks can be built. Virtualisation technologies have enabled single physical machines to be partitioned into multiple abstracted machines and network virtualisation technologies have enabled physical networking devices to replicate the functions and behaviour of many of networked devices (Handigol *et al.*, 2012).

In Section **2.1**, three models for abstracting a computer system at hardware layer are introduced. These abstractions enable a single physical machine to be partitioned into multiple abstract machines. Each of these abstracted machines can replicate a subset of the features of the original machine.

Section **2.2** expands on the opportunities for abstracting a computer system by exploring the software systems that realise abstracted machines. Using a layered model, six abstraction techniques are investigated. The layered virtualisation model captures the different techniques employed by programmers to create software systems capable of emulating partial or whole computer architectures.

Four broad classes of NEP construction methodologies are introduced in Section **2.3**. Each of the four types of classical NEP was designed to address specific requirements for testing and experimentation activities in computer networks. In this section the construction and motivation for the development of each type of NEP is discussed, including how classical NEPs employ abstraction techniques to realise experimental networks.

In Section **2.4** the influence that abstracting a computer system has on realism is investigated. The metrics and parameters used to define realism, within the context of NEPs, are explored based on available literature. This chapter concludes with a summary in Section **2.5**.

2.1 A System of Abstractions

Computer systems are designed as a hierarchical system of abstractions (Smith and Nair, 2005). In this section three abstraction models are introduced. These abstraction models are explored as enablers for the virtualisation technologies used by NEPs to construct experimental networks. The first model to be explored is the von Neumann architecture (von Neumann, 1945), a model designed to capture the output of designing the Electronic Discrete Variable Automatic Computer (EDVAC). The second model, the Privilege Ring

model, presents the abstraction used in Operating Systems (OSs) where the hardware, the operating system, and applications a user executes are separated into different privilege levels, ensuring that management and control systems of the OS cannot be modified by user applications. The third model is the Popek and Goldberg model for Instruction Set Architecture (ISA) virtualisation. The Popek and Goldberg model ensures that a virtual computer is in full control of the virtual hardware, that it is properly isolated from the host machine, and that the virtual machine runs efficiently. The combination of these three models enable OSs to be designed to make provisioning for further levels of abstraction.

2.1.1 Von Neumann Architecture

The EDVAC computer (von Neumann, 1945; Gluck, 1953), delivered in 1945, was designated to replace the Electronic Numerical Integrator and Computer (ENIAC) (Goldstine and Goldstine, 1946) and was based on binary instead of decimal mathematics (Koons and Lubkin, 1949). This new computer system required a rethink of the base architecture of the machine to reduce the bottlenecks of previous computer systems. An enhanced input-output architecture was required to reduce the time required to program the system. The resultant architecture was named after its designer: John von Neumann. The architecture, shown in Figure 2.1, broke the design of the machine down into four logical units: input devices, output devices, a Memory Management Unit (MMU), and a Central Processing Unit (CPU) composed of a control unit and an Arithmetic/Logic Unit (ALU).

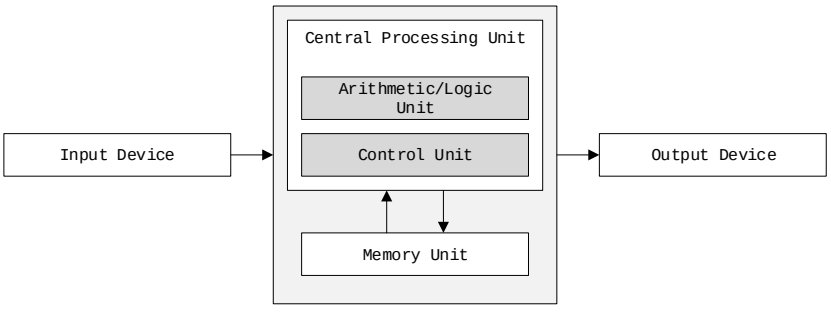


Figure 2.1: Von Neumann Architecture[1], von Neumann (1945)

In *How to Build a Working Digital Computer*, Alcosser *et al.* (1967) guides the curious reader in building a functional programmable computer system from "household" components. The base architecture for the computer system being built is the von Neumann

[1]Adapted from image by Kapooht [CC BY-SA 3.0], from Wikimedia Commons

architecture. Computer systems based on the von Neumann architecture are commonly referred to as von Neumann machines. The detailed interaction between the components of the home-made computer can be seen in Figure 2.2. These interactions are the same as those of a von Neumann machine.

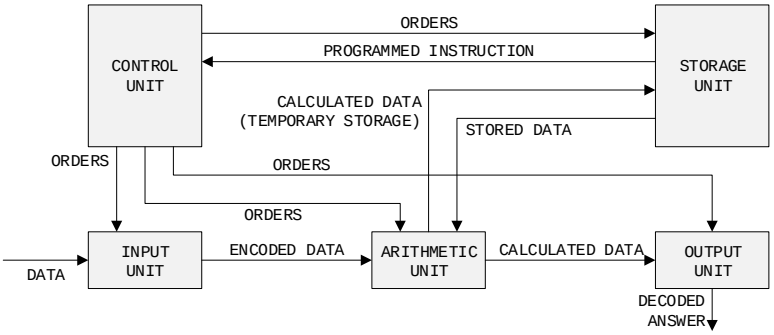


Figure 2.2: A Typical Computer's Functional Diagram, After Alcosser *et al.* (1967)

Modern microprocessors, as presented to the OS, are von Neumann machines. Internally, the processor could be designed using either the von Neumann or Harvard (Aiken and Hopper, 1946a,b,c) architectures. The primary difference between the architectures is in the memory unit. The Harvard architecture (Figure 2.3) requires separate data and program memory whereas the von Neumann architecture stores program code and data in the same memory.

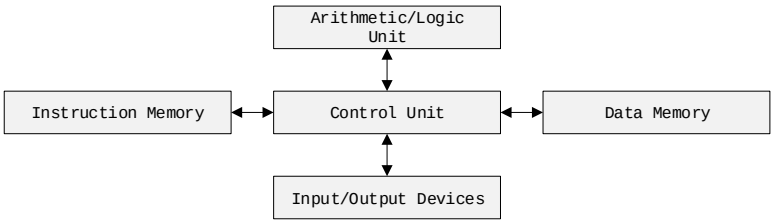


Figure 2.3: Harvard Architecture[2]

The von Neumann architecture defines the basic components that an abstracted machine should provide: input and output mechanisms, program and data storage, and a way to execute machine code. At any level of abstraction, the abstracted machine must expose these elements to create a realistic environment that a user can interact with.

[2]Adapted from image by Nessa los [CC BY-SA 3.0], from Wikimedia Commons

2.1.2 The Protection Ring Model

The concept of separating resource access in a computer system according to privilege originated within the Multiplexed Information and Computing Service (Multics) OS (Corbató and Vyssotsky, 1965). In the build up to the Protection Ring model Dennis (1965) developed methods and mechanisms to segment machine resources and Graham (1968) developed a software layer that implemented an early Protection Ring model. The experiments within the Multics OS development team to increase security by enforcing access rules at the hardware layer resulted in the Protection Ring model (Schroeder and Saltzer, 1972). The original Protection Ring model as proposed by Graham (1968) provisioned for an arbitrary number of rings, as shown in Figure 2.4.

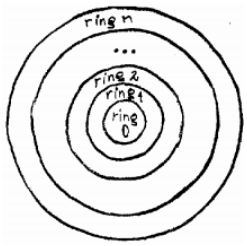


Figure 2.4: Original Protection Rings Sketch, Graham (1968)

Contemporary processors based on the Intel x86 architecture implement the concepts of the original Protection Ring model, but limits the number of rings to 4 (Intel Corporation, 2019, Section 5.5). An annotated illustration of the processes that occupy a particular ring in the x86 architecture is shown in Figure 2.5. Ring 0 is assigned the most privileges and allows modification of management and control systems within the CPU. Ring 0 is where the kernel of the OS resides, allowing the OS to exercise control over all aspects of the system. Rings 1 and 2 are commonly used for device drivers and system services, enabling interaction between user space applications and the underlying hardware. Applications that the user interacts with reside in Ring 3, the lowest privilege level.

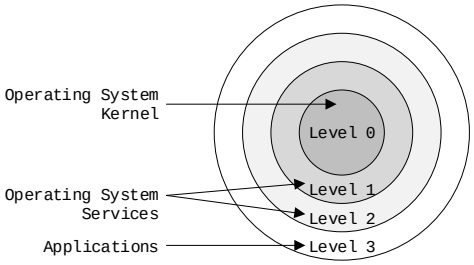


Figure 2.5: Intel x86 Architecture Privilege Rings, After Intel Corporation (2019)

2.1.3 Hardware Assisted Virtualisation

Computer virtualisation is the process of creating a virtual instance of a computer running on another computer (Section 2.2.2). Early Virtual Machines (VMs) were implemented in software. These software systems were responsible for creating a virtual environment that is indistinguishable from real hardware. The OS executing in the virtual environment could exact full control over the virtual environment just as it would on real hardware. Software-based VMs are expensive in terms of system resources and clock cycles of the CPU. To enable more efficient utilisation of the host hardware by guest VMs, hardware-assisted virtualisation was introduced (IBM, 1972a).

Hardware-assisted virtualisation enabled guest OSs to control some aspects of the host hardware. Virtual Machine Monitors (VMMs), systems that manage and control VMs, had to intercept and safely handle instructions issued by guest OSs that requested a state change of the host hardware that would conflict with the operation of the host OS. In an attempt to solve this conflict between the host OS and guest OSs, Popek and Goldberg defined a set of requirements for the host computer's ISA that would enable hardware assisted virtualisation capable of dealing with guest OSs attempting to set sensitive machine states (Popek and Goldberg, 1974).

The protection ring model (Section 2.1.2) introduced boundaries between the OS and applications executed by a user. Attempting to create a VM in such an environment creates a conflict, where both the host OS and guest OSs expect full Ring 0 privileges. Early solutions to this conflict had the host OS intercept (trap) any sensitive calls that required Ring 0 privileges and handled these in software (Adams and Agesen, 2006), however the trap system is expensive in terms of CPU clock cycles. In 2005 Intel introduced extensions to the x86 architecture with the release of the Intel Pentium 4 models 662 and 672 that enabled hardware virtualisation comparable to the requirements of Popek and Goldberg.

The introduction of the Intel VT-x extensions moved the VMM from executing in Ring 0 to executing in a special mode called hypervisor mode (Asada, 2013), which was designed specifically for VMMs, thus solving the conflict between the host and guest OSs. Recent work on the ARM ISA (Penneman *et al.*, 2013; Shuja *et al.*, 2016) has enabled hardware virtualisation for certain versions of the architecture.

Combining the various abstractions that exist in a computer system enables the creation of various types of abstract machines. The abstraction of the minimal devices required to create a computer system (Section 2.1.1) enables whole computer systems to be implemented in software (Section 2.2.1). By abstracting the execution model of a computer

system, multiple operating systems can run on a single hardware platform (Section 2.2.2). And finally, by providing hardware mechanisms to isolate executing processes' access to resources, virtual systems that execute as part of the OS kernel (Section 2.2.3) can be created.

An example of how the various abstractions mechanisms can be combined to create virtual machines can be found in Kata Containers. Kata Containers[3] (a continuation of Intel Clear Containers[4]) utilises virtualisation technologies to instantiate lightweight abstracted machines called containers (Section 2.2.3). Kata Containers instantiate an optimised VM based on Quick Emulator (QEMU) and Kernel-Based Virtual Machine (KVM) called `qemu-lite`[5] that serves as an isolated runtime for containers. Kata Containers enable the end-user to instantiate containers that have isolation comparable to that of VMs (Section 2.2.2).

2.2 Layered Virtualisation Model

In *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*, Hwang *et al.* (2013) present a layered model of computer virtualisation. The model categorises virtualisation technologies based on the level to which a computer system is abstracted. By analysing the common abstraction mechanisms used by virtualisation systems, they created the model shown in Figure 2.6. Each layer in the model describes an aspect of a computer system's hardware and software hierarchy that can be exploited to create an abstracted machine. For the purposes of this study, the model presented in Figure 2.6 has been extended to include simulation as a final layer of abstraction. The model presented by Hwang *et al.* (2013) was chosen as the basis for this study due to the match between the abstractions presented and the technologies and techniques used to to construct experimental networks using abstracted machines.

In the context of NEPs, simulated computer systems replicate the behaviour of a computer system interacting with network traffic. By leveraging the abstractions of the Layered Model, NEPs can be built to achieve higher node densities. The level at which each component in the experimental network is abstracted is based on the "realism" requirement of the component. Components that require high realism can be instantiated using physical machines, while components with lower realism requirements can be instantiated

[3] `https://katacontainers.io/`
[4] `https://github.com/clearcontainers`
[5] `https://github.com/intel/qemu-lite`

using simulation. Examples of how virtualisation technologies and system can by utilised by NEPs are shown in Section 2.2.7.

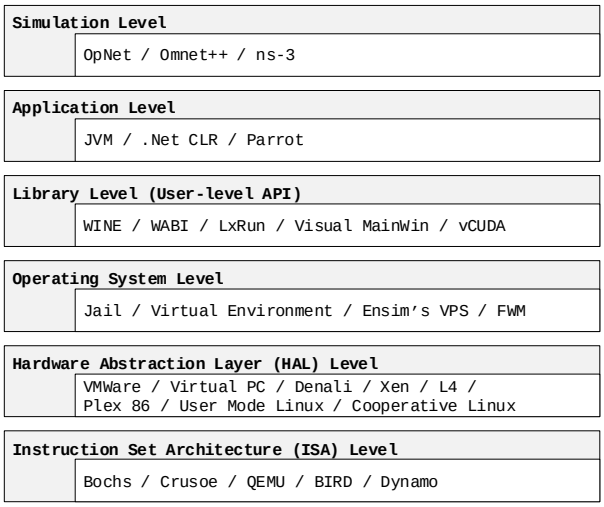


Figure 2.6: Layered Model of Computer Virtualisation, After Hwang *et al.* (2013)

2.2.1 Instruction Set Architecture Virtualisation

Instruction Set Architecture (ISA) emulators can operate in two modes: ISA interpretation, and dynamic ISA translation. Interpreters provide software-based implementations of the target ISA and hardware components such as Random Access Memory (RAM), storage, and I/O peripherals in a software defined virtual machine. These software-based computer systems replicate the design of the von Neumann architecture (Section 2.1.1). Interpreters re-implement the operations of the original hardware's opcodes[6] and apply the operations to the software implementation of the machine. Interpreters are not dependent on the host ISA. Translators provide abstractions for RAM, storage, and I/O peripherals similar to interpreters. Translators replicate the operations of the source opcodes in the target machine's opcodes. The translated opcodes then apply the necessary operations to segments of the host's RAM dedicated to the virtual machine. Translators are dependent on the host ISA and requires new translations when ported to a new target ISA. A conceptual illustration of how systems emulated through ISA level virtualisation relate to the host machine (Jones, 2011) is shown in Figure 2.7.

Bochs[7] (Lawton, 1996), an interpreter, started as a project to run the DOS (an x86 based OS) and a set of applications on a SPARC ISA based computer. Bochs originally

[6]Operation Code, a mnemonic and human readable form of a machine language instruction.
[7]`http://bochs.sourceforge.net/`

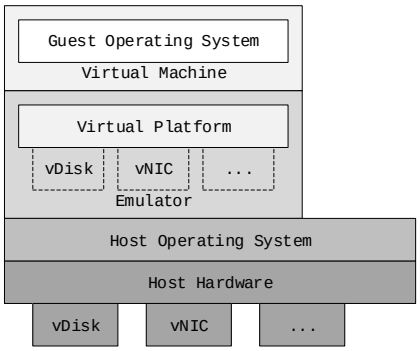


Figure 2.7: Device Virtualisation Through Emulation, After Jones (2011)

implemented a full Intel 286 machine and various peripherals in software. Over time Bochs has been extended to include the x86-64 ISA as source architecture and has been ported to multiple host OSs. Another example is Dynamips[8] (Fillot, 2005) which was created to provide a software implementation of the hardware required to run the Cisco Internetwork Operating System (IOS) range of OSs. Dynamips interprets the Microprocessor without Interlocked Pipelined Stages (MIPS) ISA on which classical Cisco IOS devices were based. Neither Bochs nor Dynamips is host architecture dependent and can be compiled for a wide array of host architectures.

A contemporary example of an ISA level virtualisation system that makes use of dynamic translation is QEMU (Bellard, 2005; Chen *et al.*, 2018). Originally created as a full system emulator, QEMU has been extended to become a multi-method virtualisation system. QEMUs user-mode emulation can execute a target ISA binary on a different host ISA, provided the host OS is the same. QEMU intercepts system calls made by the binary and passes these calls on to the host OS while the source machine code of the binary is translated to the target machine code of the host. QEMU virtualisation mode makes use of the KVM and Xen virtualisation systems and acts as a front end to these systems if the source and target ISAs are the same. In full system emulation mode, QEMU provides a full software defined system with pluggable peripherals, and uses dynamic translation to emulate the CPU component of the source machine on the target machine.

2.2.2 Hardware Abstraction Layer Virtualisation

Systems that are known today as Virtual Machines (VMs) originated as ISA level simulators. When a new computer was on the horizon, developers would implement a simulated version of the new machine on the machine that was to be replaced. The software of the current machine could then be ported to the architecture of the new machine (Goldberg,

[8] `https://github.com/GNS3/dynamips/`

1973, 1974) and reduce lead time to utilise the new machine. The IBM System/370 introduced hardware level support for machine virtualisation (IBM, 1972a,b). A user would access the System/370 from a terminal, authenticate, and then be presented with a "virtual" instance of the machine. The *IBM Virtual Machine Facility/370: Planning Guide* introduced the concept of a Virtual Machine (VM) as follows:

> A virtual machine, as implemented by VM/370, is the functional equivalent of an IBM System/370 and its associated devices. - (IBM, 1972b)

In contemporary usage, the systems that abstract the interface between the OS and the hardware to enable multiple OSs to execute on the same hardware are commonly referred to as VMMs or hypervisors. VMMs replicate the hardware of a complete computer system (King *et al.*, 2003) and present a system to the guest OS that is nearly or fully identical to the host system. The emulated hardware as seen by the guest OS is in most cases identical to that of the host. Virtualisation on the x86 and x86-64 platforms is hardware assisted by extensions to the x86 architecture. Architectural extensions such as VT-x on Intel CPUs and AMD-V on AMD CPUs provide hardware virtualisation comparable (Adams and Agesen, 2006) to the virtualisation requirements of Popek and Goldberg (Section 2.1.3).

VMMs are split into two categories: Type I and Type II. Type I VMMs do not rely on a host OS and provide all the required functions and interfaces for system management and machine virtualisation. Type II VMMs require a host OS and provides only the functions and interfaces necessary for machine virtualisation. The differences in the architectures of Type I and Type II VMMs are shown in Figure 2.8. Examples of open-source VMMs are Xen (Barham *et al.*, 2003) (Type I) and VirtualBox[9] (Type II). Examples of commercial VMMs are VMWare ESXi[10] (Type I) and Parallels Desktop[11] (Type II).

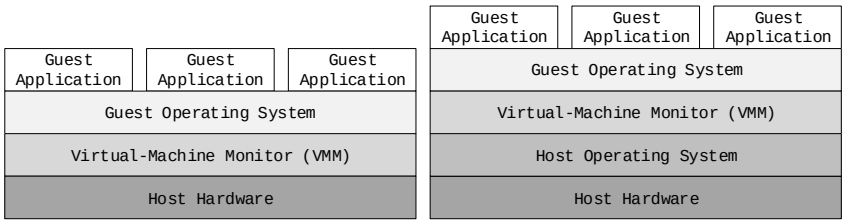


Figure 2.8: Virtual-Machine Monitor Types, After King *et al.* (2003)

[9] https://www.virtualbox.org/
[10] https://www.vmware.com/products/esxi-and-esx.html
[11] https://www.parallels.com/products/desktop/

2.2.3 Operating System Level Virtualisation

OS level virtualisation, commonly referred to as containerisation, is a technology used to create lightweight VMs (containers) by partitioning access to resources exposed by the host OS. Containers do not dedicate blocks of resources to a single VM, instead access to resources is controlled through resource management structures in the kernel. This structural organisation results in VMs with very little overhead (Vaughan-Nichols, 2006). Figure 2.9 shows the architecture of OS level virtualisation. Containers are created by implementing a root file system, isolating RAM, and controlling access to system devices in kernel through management structures. The primary restriction imposed by OS level virtualisation is that only a single OS family, that of the host OS, can be used as a guest OSs. Modern containerisation platforms such as Docker (Petazzoni and LeClaire, 2014) provide root file systems for containers that can encapsulate different distributions of the Linux OS while sharing the same kernel.

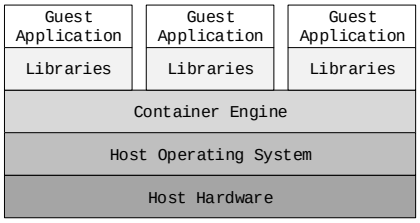


Figure 2.9: OS Level Virtualisation Architecture, After Bernstein (2014)

The FreeBSD OS introduced OS Level Virtualisation in the form of "Jails" in FreeBSD 4.0-RELEASE (Kamp and Watson, 2000) in March 2000. Jails were developed as a method to host multiple clients from a single machine while isolating disk access, network interfaces, and processes between clients (Ohrhallinger, 2010). Jails formed the base management and virtualisation system for Virtual Private Servers and enabled hosting providers to increase the number of clients hosted per physical machine. The introduction of `VIMAGE` (Zec, 2003), a subsystem to virtualise the FreeBSD network stack, allowed Jails to have multiple network interfaces independent from the host OS. Jails were followed by Solaris Zones (Price and Tucker, 2004). Solaris Zones extended resource isolation for containers by providing pools of resources that could be shared by multiple Zones.

Linux namespaces (Biederman, 2006), a resource isolation system and the base mechanism for OS Level Virtualisation in the Linux kernel, was introduced with the release of version 2.4.19 of the Linux kernel in August 2002. Mount, the first Linux namespace,

enabled applications to have isolated access to the file system, ensuring that an application cannot modify files outside of its restrictions. The success of the Mount namespace lead to the design of nine additional namespaces. Of the original ten namespaces only six has been implemented (Rosen, 2013). A seventh namespace, the `cgroups`[12] namespace (Menage *et al.*, 2008), was released with Linux 4.6. By combining the available namespaces into groups through the use of `croups` (Rosen, 2013), a lightweight VM can be instantiated. These lightweight VMs form the base technology used by Container-Based Network Emulators (CBNEs) to construct experimental networks (Chapter 3).

The low overhead required for containers lead to quick adoption within the "Cloud Computing" space. Container management engines such as Kubernetes (Crall, 2014), analogous to Type II VMMs, are used as resource management engines for "Function-as-a-Service" offerings by cloud computing providers, a cloud computing technology that allows individual software functions to be deployed as standalone units. CoreOS[13] is a container management engine that shares architectural similarities with Type I VMMs.

2.2.4 Application Programming Interface Virtualisation

Application Programming Interface (API) level virtualisation enables binaries from one OS (the native OS) to be executed on another OS (the foreign or host OS), provided that the same hardware architecture is used. API level virtualisation systems such as Wine Is Not an Emulator (WINE) (Amstadt and Johnson, 1994) and Microsoft Corporation's Windows Subsystem for Linux (WSL) (Hammons, 2016; Microsoft Corporation, 2016) are built to execute non-native binaries without the need for a virtualisation system. Such systems enable non-native binaries to interact with local system resources, providing a near seamless user experience. In the first phase of executing non-native binaries, API level virtualisation systems provide facilities that enable the host OS to interpret the binary structure of another OS and load required (foreign) libraries into the address space of the binary. The second phase of executing non-native binaries involves the API level virtualisation system intercepting system calls made by the binary. The intercepted system calls are translated to the system calls of the host OS and re-issued. The intercept and re-issue system enables non-native binaries to interact with resources of the host OS.

WINE[14] is a OS level virtualisation system that enables Microsoft Windows[15] binaries

[12] `http://man7.org/linux/man-pages/man7/cgroups.7.html`
[13] `https://coreos.com`
[14] `https://www.winehq.org`
[15] Hereafter referred to just as Windows

to be executed on Linux. WINE re-implements the functions in the Microsoft Windows SDK. When a Windows binary is executed on Linux, WINE intercepts the execution of the Windows Portable Executable (PE) binary and translates system calls for the Windows environments to Linux system calls. Similarly, WSL is a ISA level virtualisation system that intercepts system calls made by Linux binaries and translates these to Windows specific system calls.

2.2.5 Application Virtualisation

Application level virtualisation systems, called Application Virtual Machines, enable source code to be compiled to binaries that are OS and architecture agnostic. AVMs such as the Oracle Java Virtual Machine (JVM) (Lindholm and Yellin, 1997) and the Microsoft .NET Common Language Runtime (CLR) (Box and Sells, 2002) act as a middle layer between executables and the host OS. The JVM and the CLR are each complemented by a set of libraries that provide a consistent programming environment for developers across host OSs. These binaries are compiled to an intermediary form called bytecode. The AVM interprets the bytecode and, through just-in-time compiling, emits machine code of the host hardware. Applications programmed for an AVM interact with the host OS through native interface bindings provided by the AVM. The AVM is compiled for the specific host hardware and host OS and takes care of the intricacies of providing a consistent execution environment.

In Figure 2.10 an interpretation of the JVM architecture as described in Lindholm and Yellin (1997) shows the relationship between bytecode (Class Loader), the JVM memory, the execution engine and the interfaces on the host OS. The JVM *Execution Engine* translates bytecode to native machine instructions through a Just-in-Time (JIT) compiler. The operation of an AVM is similar to that of ISA level virtualisation in that it translates or interprets an ISA made for a different architecture to that of the host architecture. AVMs do not provide a virtual platform, instead AVMs expose the native platform through software bindings known as *Native Method Interfaces*.

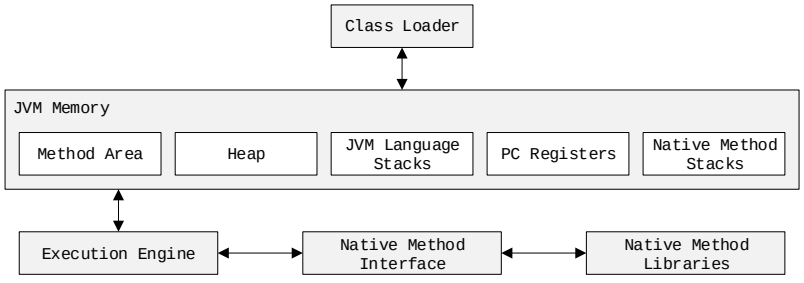


Figure 2.10: Java™ Virtual Machine Architecture[16], After Lindholm and Yellin (1997)

2.2.6 Behavioural Abstraction

At the highest level of abstraction a computer system is presented as a set of algorithms that model the *behaviour* of computer systems. Network simulation systems such as `ns3` (Bonada *et al.*, 2008) and Riverbed Modeler [17] (previously called `OPNET Modeler`, Cohen, 1986) use deterministic and stochastic models to replicate the interaction of a host in a simulated environment with network traffic. Implementations of network protocols within simulated hosts are deterministic and follow the Request for Comments (RFCs) specifications. Packet transmission metrics such as jitter and loss and bit errors in packets are modelled using stochastic processes, and model fitting parameters are fine-tuned based on observed network traffic.

Simulation assists in the development of new technologies. During development the behaviour of new technologies can be studied without full implementations. By studying the behaviour of modelled versions of new technologies, reliability can be increased and maintenance of the technology can be reduced (Rampfl, 2013). In addition, the performance and flexibility of the new technology can be assessed during simulation (Austin *et al.*, 2002). If a new technology is introduced into a computer network, simulation assists in capacity planning and assessment of the impact of the new technology in the current network (Heidemann *et al.*, 2001; Heilmann and Fohler, 2018). Simulation systems can also be implemented to provide high speed alternatives for specific functions, such as routing exclusively (Herbert and Irwin, 2013).

The primary weakness of simulation is uncertainty. Each model used in a simulation introduces a level of uncertainty in the results of a simulated network (Floyd and Paxson, 2001; Pujeri and Palanisamy, 2014) and might be a source of incorrect behaviour (Rampfl, 2013; Guo and Lee, 2018; Mazur, 2018).

2.2.7 Building Blocks for Network Experimentation Platforms

Each of the virtualisation mechanisms discussed in this section can serve as building blocks for NEPs. The following examples illustrate how NEPs utilise these mechanisms.

ISA level virtualisation can be used by NEPs where a network component or computer system needs to be used that has a different ISA to that of the host system. As an

[16] Adapted from image by Michelle Ridomi [CC BY-SA 3.0], from Wikimedia Commons

[17] `https://www.riverbed.com/za/products/steelcentral/steelcentral-riverbed-modeler.html`

example: Graphical Network Simulator 3 (GNS3)[18] uses DynaMIPS (Section 2.2.1) to include certain Cisco IOS devices in experimental networks.

VMMs are used by NEPs where multiple OS families are required for an experimental network. The limitation of using VMMs is that all OSs have to be for the same architecture (ISA). Common Open Research Emulator (CORE), a CBNE discussed in Section 3.3.4, utilises Xen (Section 2.2.2) to incorporate non-Linux OSs into experimental networks.

Container-Based Network Emulators (CBNEs) utilise OS level virtualisation (Section 2.2.3) as the primary virtualisation mechanism for nodes in an experimental network. OS level virtualisation is used where different user-space applications are executed in a network and the OS and architecture is the same as the host system. CORE can incorporate simulation tools such as GNS3 (Section 2.2.6) into experimental networks.

Behavioural level abstraction is used in network simulation systems such as Riverbed Modeller, where a desktop analysis of large-scale network deployments or changes to production networks are required. Simulation systems enable network engineers to model the impact that changes to a network will have on traffic volume and QoS metrics before rolling out the intended changes. The GNS3 network simulator can incorporate CBNEs such as Mininet (Section 3.3.1) to provide realistic hosts for network simulations[19].

In Section 2.3, NEPs that focus on utilising a single abstraction mechanism are explored, illustrating the application of abstraction technologies in building NEPs.

2.3 Network Experimentation Platform Types

Network Experimentation Platforms (NEPs) are platforms composed of various computer and networking systems that enable researchers, implementers, and operators to test interaction between computer systems in a networked environment. By exploiting the layered virtualisation model of computer systems (Section 2.2), each layer can be used to create an abstract machine. These abstract machines replicate enough of the behaviour of the original machine to be usable within environments that do not require all the functionality of the original machine. By combining abstracted and non-abstracted machines into a networked environment, experiments can be conducted using hundreds or thousands of computers at a fraction of the cost of recreating the environment using physical machines

[18] https://www.gns3.com
[19] https://docs.gns3.com/appliances/mininet.html

alone. NEPs enable researchers to execute repeatable experiments, ensuring consistent results (Fall, 1999; Handigol *et al.*, 2012; Heller, 2013). These platforms can be constructed using various techniques (Davis and Magrath, 2013). This section presents an overview of four broad technology classes that can be employed to create network experimentation platforms.

2.3.1 Network Testbeds

Network testbeds are deployments of computer and networking hardware that aims to replicate computer networks and the conditions in which network protocol and software applications will be utilised. A key goal of a testbed is to recreate the expected conditions of a network at the highest possible level of realism. An advantage of testbeds is low to no abstraction of the components used to construct experimental networks. The use of minimal abstraction aides in ensuring that results are reproducible (Nussbaum, 2017). Depending on size, the hardware required to build testbeds can be prohibitively expensive and can require large amounts of space. Testbeds present a low-cost alternative to dedicated laboratories for education (Riga *et al.*, 2015) and have become viable platform for conducting Cyber Security[20] experiments on Internet of Things (IoT) networks (Gunduz and Das, 2018).

EmuLab[21] (White *et al.*, 2002) is a distributed testbed that supports bare-metal machines and virtualisation technologies such as VMs. EmuLab was built for network protocol and application experiments. EmuLab is used as the base framework for DeterLab and Global Environment for Network Innovations (GENI). Most recently EmuLab has added support for Docker as a management system (Johnson *et al.*, 2018). EmuLab focuses on ensuring that traffic flow within experiments is as realistic as possible (Syed, 2014; Syed and Ricci, 2015).

PlanetLab[22] (Peterson *et al.*, 2003) is an overlay network that supports the use of containerisation and VMs. PlanetLab pioneered the "sliceability" concept for testbeds, a mechanism used to partition a testbed into multiple smaller testbeds to allow multiple experiment to run concurrently. Measurement Lab (Dovrolis *et al.*, 2010), dedicated to accurately measure internet performance, and VICCI (Peterson *et al.*, 2011), a programmable cloud-computing research testbed, originated from work done within PlanetLab.

[20] The practice of protecting computer systems from unlawful access.
[21] `https://www.emulab.net/`
[22] `https://www.planet-lab.org/`

DeterLab[23] (Mirkovic *et al.*, 2010), a testbed built on EmuLab, was built for furthering cyber defence research and for education (Mirkovic and Benzel, 2012). Current and ongoing research regarding the DeterLab system involves enhancing the repeatability of experiments (Sharma *et al.*, 2017), enabling Software Defined Networking (SDN) experiments (Sivaramakrishnan *et al.*, 2017), and developing standardised methods for instantiating distributed experiments (Mirkovic *et al.*, 2018).

GENI[24] (Berman *et al.*, 2014; McGeer *et al.*, 2016) is a testbed for large scale research into network and distributed systems with a deep focus on instrumentation and measurement tools. It provides the ability to implement and experiment with custom Layer 2 protocols. GENI is federated with various other testbeds including EmuLab and PlanetLab. Users of the GENI testbed can utilise resources from federated testbeds. Repeatability of experiments (Edwards *et al.*, 2015) and management of experimental data (Nussbaum, 2018) are key concepts of the GENI testbed.

EdgeNet[25] (Cappos *et al.*, 2018) is a next-generation software-only testbed built on cloud technologies (Mercan, 2018). EdgeNet has its origins in PlanetLab and GENI (Bavier *et al.*, 2018) and aims to enable "Testbed-as-a-Service" functionality.

2.3.2 Virtualisation

The cost of deploying a testbed can be reduced through the use of virtualisation. The functional fidelity of virtualised computer hardware is near perfect, complementing the repeatability of experiments. By replacing costly end user hardware with virtualised instances, the total hardware required is reduced. Additional advantages of virtualisation are reductions in physical space and maintenance requirements.

Virtualisation is used as an alternative to hardware-based laboratories for education (Bullers *et al.*, 2006; Schmidt *et al.*, 2018) and can remove the risks involved with the assessment of students in information security training (Willems and Meinel, 2012). By combining network virtualisation technology with VM technology, experimental networks consisting of several machines can be instantiated while isolating risks presented (Xu *et al.*, 2014). The cost and time requirements of educational and training experiments involving IoT can be minimised using virtualisation (Liu *et al.*, 2018).

[23] `https://www.isi.deterlab.net/`
[24] `https://www.geni.net/`
[25] `https://edge-net.org/`

Conducting network and information security experiments on production Supervisory Control and Data Acquisition (SCADA) and Industrial Control System (ICS) systems is impractical due to the unknown effects experiments might have on these systems (Queiroz *et al.*, 2009). SCADA and ICS sandboxes use a hybrid approach to create experimental environments. A mixture of simulation and physical hardware is used to model the effects on physical components, while virtualisation is used to model the software aspects of these environments. Using hybrid environments can reduce the reconfiguration time of the experimental environment and result in repeatability of experiments (Lemay *et al.*, 2013; Urdaneta *et al.*, 2018).

The use of virtualisation to replicate segments of a computer network provides capabilities that testbeds cannot provide. The ability to capture the current state of virtualised nodes, reset nodes to a previous state, and remove nodes at will is a major benefit of using virtualisation (van Heerden *et al.*, 2013). These capabilities, combined with the ability to integrate any TCP/IP based device into the experimental network (Browne *et al.*, 2018) makes virtualisation an attractive environment for security experiments on computer networks.

2.3.3 Containerisation

The introduction of containerisation and link emulation tools in Linux, FreeBSD and Solaris introduced the possibility of creating experimental networks using OS components. Containers have little overhead and provide node and network isolation methods to assist in the creation of experimental networks. NEPs based on containerisation technology exploit these components to provide the user with a lightweight and flexible environment to create experimental network topologies. These systems are referred to as Container-Based Network Emulators (CBNEs), and are discussed in detail in Chapter 3.

The reproducibility of network experiments is a key focus area of all NEPs. CBNEs enable the publication of network research environments similarly to how research results are published (Handigol *et al.*, 2012). By controlling resource utilisation of nodes and applying consistent network metrics to links, experiments can be published as configuration files and the results of the experiment can be reproduced independently (Heller, 2013). CBNEs can be extended to integrate with non-real time network simulation systems by contracting or dilating the timing mechanisms of containers (Lamps *et al.*, 2018).

CBNEs are used as platforms for information security experiments in research and education. The lightweight nature of containers (Section 3.2) allows experimental networks to

be constructed with tens to hundreds of "user" machines that can replicate the expected behaviour of users (de Berlaere, 2018). This allows the quality of service that the end users experience to be monitored during live attacks. The configurable and distributable nature of CBNEs allows for the distribution of personalised scenarios during evaluation of information security related skills (Thompson and Irvine, 2018).

Outside of CBNEs, containers are used as testing platforms for new network protocols (Qu, 2018), monitoring network metrics in multimedia rich environments (Cinar *et al.*, 2016), and for generating realistic user traffic for emergent technologies (Gries *et al.*, 2018; Muelas *et al.*, 2018).

2.3.4 Simulation

Simulation is used as an alternative to physical testing. Simulating the effects that an attack has on a network enables better preparation and response procedures without incurring loss in a production network. Simulation of attacks on computer networks is particularly useful in education and training in information security concepts, where it is used as a low cost alternative to physical testing (Saunders, 2001; Pastor *et al.*, 2010).

In industrial applications, simulating computer networks as a component of the industrial system is a complex task. Choosing the appropriate aspects to simulate as well as a simulation platform that can respond to the requirements of industrial networks is not trivial (Anton *et al.*, 2018). In many cases simulating a cyber physical system will involve more than one simulation platform. These simulations are broken down into simulators for the physical systems, simulators for the communications network, and simulators for management and control systems (Hammad *et al.*, 2019).

Simulation of the information security ecosystem within the military context ranges from simulation of the effect of attacks and responses on networks (DeLooze *et al.*, 2004) to simulating the behavioural and cognitive patterns of users (Veksler *et al.*, 2018) during an attack. In between these extremes the management and control of military operations during a cyber attack can be simulated by adapting classic Command and Control methodologies (Grant *et al.*, 2007; Grant, 2009).

2.4 Abstraction, Realism and Scalability

There's no one universal way to scientifically describe the level of realism achieved in a given abstraction of a host. The term fidelity is frequently used to describe the level at which a specific component of an abstracted machine performs. During the development of a NEP (SELENA) Pediaditakis *et al.* (2014) defined three metrics against which the developed platform should be measured: fidelity, scalability, and reproducibility. The main goal of the proposed platform was to enable reproducibility of network experiments. Within this context, the ability of the platform to accurately model network traffic metrics was used as a primary measure of fidelity. A second measure of fidelity was the ability to accurately represent the topology of a network. These measures were well suited within the goals of the project, but do not represent the measures that might be appropriate for other types of systems.

When comparing fidelity measures across different types of NEPs, the impact that an abstraction technology has on realism or fidelity is entirely dependent on the context of both the platform and the types of experiments that the platform supports. In Handigol *et al.* (2012) the MiniNet project was enhanced to support reproducible experiments. The realism measures defined and used within this project (Table 2.1) were based on the ability to execute binaries (*Functional Realism*), the time keeping mechanisms of the platform (*Timing Realism*), and the ability of the platform to interact with real network traffic (*Traffic Realism*).

The works of Pediaditakis *et al.* (2014) and Handigol *et al.* (2012) have similar goals (reproducibility) and similar measures of realism, though within the contexts of these platforms the way that fidelity is measured differs. Handigol *et al.* regard simulators as not having *Functional Realism*, while Pediaditakis *et al.* regard simulation as having high node fidelity. These measures are similar in concept, but are measured differently within the respective experimental contexts.

Table 2.1: Platform Characteristics for Reproducible Network Experiments, After Handigol *et al.* (2012).

		Testbeds		
	Simulators	Shared	Custom	Emulators
Functional Realism		✓	✓	✓
Timing Realism	✓	✓	✓	?
Traffic Realism		✓	✓	✓
Topology Flexibility	✓	limited		✓
Easy Replication	✓	✓		✓
Low Cost	✓			✓

In Xu *et al.* (2014) VMMs were explored as a technology to enable isolated yet easy to use information security experiments. The work of Xu *et al.* found both *Physical Labs* and *Multi-VM & Multi-Network Labs* (Table 2.2) to have *High Fidelity*. These configurations are analogous to testbeds. In contrast, Pediaditakis *et al.* (2014) found testbeds to have medium node fidelity and low link speed fidelity. In Pediaditakis *et al.* (2014) the context was based on experiments involving network traffic, while in Xu *et al.* (2014) the context was based on experiments involving information security experiments, such as conducting Man-in-the-Middle attacks.

Table 2.2: Virtual Laboratory Feature Comparison, Partial extract from Xu *et al.* (2014)

Lab Type	Virtualisation Type	Fidelity
Physical Lab	None	High
Simulation Lab	Application Based	Low
Virtual Application Lab	Application Based	Low
Shared Host Lab	Session Based	Low
Single VM Lab	Single VM	Medium
Multi VM Lab	Multi VM	Medium
Multi VM Lab & Multi-Network Lab	Dedicated Multi VM & Virtual Networks	High

Abstraction techniques do not only influence fidelity. Node density/scalability - the number of abstracted nodes per physical machine - is influenced by abstraction as well. An example of how abstraction influences scalability is shown in Table 2.3. Within the context of the DeterLab project (Mirkovic *et al.*, 2010), the abstraction technology used shows an inverse relation between fidelity and scalability - as abstraction is increased, node density is increased and fidelity is decreased.

Table 2.3: DETERlab Testbed Node Densities[a]

Container Type	Fidelity	Scalability
Physical Machine	Complete fidelity	1 per physical machine
Qemu virtual Machine	Virtual hardware	10s of containers per physical machine
Openvz container	Partitioned resources in one Linux kernel	100s of containers per physical machine
ViewOS process	Process with isolated network stack	1000s of containers per physical machine

[a] Obtained from `https://containers.deterlab.net/`

In Rimondini (2007) a collection of NEPs was classified into a taxonomy based on scale (Scalability) and emulation type (Abstraction). In Table 2.4 the original data is summarised to illustrate how abstraction influences scale within the context of the original work. A *Small* network is defined as "very few instances of virtual machines", whereas *Large* is defined as possibly being a distributed cluster. The scales reported by Rimondini (2007) are confined to evaluated NEPs.

Table 2.4: Influence of Abstraction on Network Scale, After Rimondini (2007)

Abstraction Layer	Small	Medium	Large
Instruction Set Architecture	•		
Hardware Abstraction Layer	•	•	•
Operating System		•	•
Application Programming Interface	•	•	
Application		None	
Behavioural		•	
Hybrid			•

EmuLab (White *et al.*, 2002), the technology on which DeterLab is built, is a hybrid testbed that utilises simulation, virtualisation, emulation, and real devices to construct experimental networks. EmuLab aims to balance the advantages and disadvantages of different abstraction technologies to enable researchers and experimenters to construct an experimental network by using suitable technologies for each node. This balanced approach is shown in Table 2.5.

Table 2.5: Characteristics of Experimental Platforms, Extract from White *et al.* (2002)

Metric	Simulation	Emulation	Live Network	Emulab
Ease of Use	✓	ModelNet?[a]		✓
Performance		✓	✓	✓
Repeatability	✓	✓		✓
Packet-Level Control	✓			✓
Coarse-Grain Control	✓	✓		✓
Scalability	varies	w/ModelNet	varies	✓
Parameter Space Exploration	✓	ModelNet?		✓
Reuse of Models	✓	ModelNet?		✓
Real Links			✓	✓
Real Router			✓	✓
Real Hosts		✓	✓	✓
Real Applications		✓	✓	✓
Real Users			✓	✓

[a] Vahdat *et al.* (2002)

2.5 Summary

In Section **2.1** the architectural choices made during the design of computer systems, and how these choices enable abstraction, were investigated beginning with the primitive constructs that defines a computer system as envisioned by John von Neumann during the creation of the EDVAC (von Neumann, 1945). These primitive constructs have remained the basis on which modern computer systems are built. Separating the control

that software, and thus the user, has over the underlying hardware has been a major focus of microprocessor engineering. The privilege ring model created for the Multics OS is a mainstay of contemporary processors to enable OSs to separate kernel and user process. Incorporating an ISA that fulfils the Popek and Goldberg requirements, a computer system can run multiple VMs efficiently and securely, while ensuring that the VMs are functionally equivalent to the host machine. By combining these two models, modern OSs enable a range of additional abstractions at various layers.

Opportunities for abstracting a computer system at the hardware layer (Section 2.1) enables abstraction systems such as VMs and ISA emulators. In Section **2.2** a layered model of virtualisation (abstraction) that encompasses the abstraction techniques presented by both hardware and software was described. The layered model created by Hwang *et al.* (2013) was extended to include behavioural abstraction (simulation) as a representation of the full abstraction of computer systems. Simulation abstracts a computer system to the extent that only the required component(s) of the system are presented. Each layer of abstraction presents the end user with a choice in compromises that have to be made.

The mechanisms used to abstract computer systems are used by NEPs. In Section **2.3**, four types of NEPs were investigated. The typical (and documented) use of these platforms were presented to the reader within the context of information security experiments. Though these platforms were presented as "pure" representations, it is common for NEPs to utilise multiple types of abstractions to enable an end user to construct a "fit-for-purpose" testing environment. NEPs that use multiple abstraction technologies combined with real hardware are referred to as hybrid platforms.

Section **2.4** detailed how abstraction influences the realism or fidelity of computer systems within a NEP. Within the context of NEPs, the terms realism and fidelity do not have an exact and encompassing definition. Instead, these terms are defined and measured based on the context within which the platform is used. In research and experimentation where network traffic metrics are the focus, fidelity is used in the sense of being able to replicate measurable network metrics such as latency, throughput, and jitter. In research and experimentation, where the interaction between computer systems through the use of network protocols is of primary concern, the ability of an abstracted computer to accurately interpret network protocols is used as a measure of fidelity.

Chapter 3

Container-Based Network Emulators

It's not wise to violate rules
until you know how to observe them.

T. S. Eliot

Chapter 2 introduced Network Experimentation Platforms (NEPs) - systems designed to replicate computer networks - as research and experimentation platforms for network protocols and networked applications. This chapter delves deeper into a specific type of NEP referred to as Container-Based Network Emulators (CBNEs) (Section 2.3.3). CBNEs are software systems designed to abstract the complexity of creating computer networks constructed using Operating System (OS) level virtualisation technologies.

CBNEs combine containers (Section 2.2.3) and in-kernel network virtualisation technologies such as Linux bridges (Böhme and Buytenhenk, 2001) to create networks of lightweight Virtual Machines (VMs). The low resource requirements of these technologies, and thus networks created using these technologies, allow end users to create complex research and experimentation networks on commodity hardware such laptops (Bhatia *et al.*, 2008; Lantz *et al.*, 2010). CBNEs provide a low cost and portable alternative to other forms of NEPs such as network testbeds. These systems can be deployed as sand-boxed platforms for research and education in fields that require computer networks.

Within the context of this study, a CBNE is defined as a purpose-made suite of utilities and applications that abstracts the complexity of creating networked containers and enables end users to define and instantiate a set of networked containers, with the ability to define and control configuration values for each deployed component, through a single User Interface (UI).

This chapter begins with an introduction to CBNEs (Section **3.1**). CBNEs originated a method to create small networks for research and experimentation on commodity hardware. A brief overview is given of how CBNEs came into being and what the typical applications of these systems are.

In Section **3.2** the abstraction mechanism used in CBNEs - Linux namespaces - is discussed. Understanding the technology used to create CBNEs is key to understanding how realism, from the perspective of a remote attacker, will differ between the host machine and virtualised hosts within an experimental network.

A selection of CBNEs built for the Linux OS is introduced in Section **3.3**. The initial goals for the creation of each CBNE are discussed and the applications and use cases for each CBNE is detailed.

Section **3.4** explores the architecture of each of the selected CBNEs. The architecture of a CBNE is driven by use cases and applications of the CBNE. Architectural choices such the HMI and remote control capabilities of each CBNE as well as the organisation of the core components, such as library design and choice of virtualisation technologies, is discussed.

A wide selection of technologies that can be used to construct CBNEs exists within the Linux ecosystem. In Section **3.5**, the technologies and components that CBNEs use to create the components of experimental networks are catalogued. The options for creating computers through containerisation, network devices through virtual network software, and manipulating link metrics such as jitter and packet loss through network emulation systems are detailed. The chapter ends with a summary in Section **3.6**.

3.1 Background

The origins of CBNEs can be traced back to systems designed for rapid network protocol development. Predecessors to CBNEs such as ENTRAPID (Huang *et al.*, 1999) and the work of Wang and Kung (1999) utilised the Berkeley Software Distribution (BSD) network stack to pass packets between applications to simulate networked computers. Alpine (Ely *et al.*, 2001) improved on the design by implementing virtual network devices and interconnecting applications. In these designs, applications were interconnected to simulate computer networks and shared a single network stack. The Integrated Multiprotocol Network Emulator/Simulator (IMUNES) (Zec and Mikuc, 2004) extended the concept by

implementing a cloneable network stack (Zec, 2003) that interconnected FreeBSD Jails (Riondato, 2020), an OS level virtualisation technology, to create networks of lightweight virtual machines (Section 2.2.3). These systems specialised in the rapid development of network protocols and testing applications in real-world networking conditions.

User Mode Linux (UML) (Dike, 2006), an alternative technology to Linux namespaces was utilised by Marionnet (Loddo and Saiu, 2007, 2008) to create a lightweight NEP. The UML project ported the Linux kernel to a user-space application. UML VMs require more resources, such as dedicated block of Random Access Memory (RAM), than Linux containers, but less dedicated resources overall than VMs. Marionnet utilised UML VMs to create networks of computers for education and student evaluation that could run on commodity hardware.

The Common Open Research Emulator (CORE) CBNE (Ahrenholz *et al.*, 2008) extended IMUNES with support for Linux namespaces and redesigned the architecture to support distributed emulation. CORE enhanced the functionality by including simulation components for wireless communications with a plug-in called Extendable Mobile Ad-hoc Network Emulator (EMANE) (Ahrenholz *et al.*, 2011). EMANE integrated the simulation of mobile wireless links into networked containers to explore the effects of wireless networks on network protocols and applications.

The applications of CBNEs goes beyond protocol development and application testing. The lightweight nature and isolation mechanisms of Linux containers enable CBNEs to be used as platforms to study the effects of network attacks (Salopek *et al.*, 2017), and they can even be used as high interaction honeypots (Kuman *et al.*, 2017).

In the rest of this chapter the architecture and technologies used by six open source CBNEs are explored. Table 3.1 lists the evaluated CBNEs, along with their current version, the evaluated version, and initial and latest release dates.

Table 3.1: Container-Based Emulator Implementations

Implementation	Current Version	Evaluated Version	Initial Public Release	Latest Release
MiniNet	2.3.0d6	2.2.2	2009-09-19	2019-06-12
Marionnet	0.94.0	0.90.6	2005-04	2018-01-31
IMUNES	2.3.0	2.3.0	2003-06-13	2019-05-09
CORE	5.4.0	5.3.1	2008-11-13	2019-09-24
VNX	2.0b 6604	2.0	2012-05-24	2019-08-28
Kathará	0.36.1	0.35.3	2017-10-31	2019-06-30

3.2 Linux Namespaces

Linux namespaces[1] implement OS level virtualisation (Section 2.2.3) for the Linux kernel. The implementation of the first Linux namespace, the Mount namespace, borrowed ideas from the namespace implementations in Bell Labs' Plan 9 OS (Pike *et al.*, 1992). The success of the Mount namespace lead to discussions regarding the inclusion of additional namespaces to increase the versatility of the Linux kernel. The original plan was to implement ten namespaces (Biederman, 2006), though only 6 of these have been implemented. Resource allocation and management of sets of namespaces are done through `cgroups` (Menage *et al.*, 2008). A seventh namespace, the `cgroups` namespace, was added to the 4.6 release of the Linux kernel. Table 3.2 lists the 7 implemented namespaces and the first release of the Linux kernel in which the namespace was available. Continuous efforts went into refining the management models and resource requirements of namespaces (Rosen, 2013). These improvements enabled containers to perform better in horizontal scaling and request handling tests when compared to hypervisors (Joy, 2015). The low resource requirements and performance capabilities of containers makes them an ideal base for the creation of network experiment platforms. A brief overview of each of the seven current namespaces is given below.

Table 3.2: Linux Namespace Availability According to Kernel Version[a]

Shorthand	Namespace	Kernel Version	Release Date
`MNT`	Mount	2.4.19	2002-08-03
`UTS`	UTS	2.6.19	2006-11-29
`IPC`	IPC	2.6.19	2006-11-29
`PID`	Process ID	2.6.24	2008-01-25
`NET`	Network	2.6.29	2009-03-24
`USER`	User ID	3.8	2013-02-19
`CGROUP`	cgroup	4.6	2016-05-16

[a] `http://containerz.info/`

`MNT` The mount namespace[2] isolates the filesystem mount points that can be seen by a process. Any filesystem action taken within the namespace can thus not be seen by any process not residing within the namespace. The mount namespace is used to create the root filesystem for a container.

`UTS` The UTS namespace (Hallyn, 2006) enables each namespace to have a unique hostname.

[1] `http://man7.org/linux/man-pages/man7/namespaces.7.html`
[2] `http://man7.org/linux/man-pages/man7/mount_namespaces.7.html`

`IPC` The IPC namespace isolates message queues used for Inter-Process Communication (IPC). The IPC namespace enables all processes within the same namespace to utilise System V IPC objects and POSIX message queues, while isolating these message queues from any process not within the namespace.

`PID` The process ID namespace[3] is used to create a child process tree associated with the parent process that created the namespace. Processes within the namespace can call functions like `fork(2)`[4] to create new processes without affecting the host OS. The first process for a namespace that used the `PID` namespace is the `init` process. The process ID namespace is used to create a distinct process tree for each container instantiated.

`NET` The network namespace[5] enables each namespace to have an independently functioning network stack and network interfaces. The network namespace allows each namespace to set its own routing tables. Network namespaces and `veth(4)`[6] pairs are used to form the network links between containers, and are the primary namespace used to construct CBNEs.

`USER` The user namespace[7] enables a namespace to have a set of user and group IDs distinct to that of the parent process. The user namespace enables a process within that namespace to have a privilege level other than that of the owner of the parent process. A process within such a namespace can be owned by the user with ID 0 (privileged) and have all allowed capabilities of that user within the namespace, while from the parent process' view have no elevated privileges.

`CGROUP` `cgroup` namespace extends Linux `cgroups` to present a hierarchical view of resource control for each namespace. With the `cgroup` namespace extension, groups of namespaces can share a set of resource limitations. The `cgroup` namespace ensures that each namespace can view only the `/proc/self/cgroup` that controls its resource limitations.

The process of creating a container from namespaces is illustrated in Section 3.2.1.

[3] `http://man7.org/linux/man-pages/man7/pid_namespaces.7.html`
[4] `http://man7.org/linux/man-pages/man2/fork.2.html`
[5] `http://man7.org/linux/man-pages/man7/network_namespaces.7.html`
[6] `http://man7.org/linux/man-pages/man4/veth.4.html`
[7] `http://man7.org/linux/man-pages/man7/user_namespaces.7.html`

3.2.1 Constructing a Container

The process of using `cgroups` and Linux namespaces to create a container is shown in Figure 3.1. For simplicity the illustrated process utilises standard `cgroups` and not the `cgroups` namespace and the IPC namespace is excluded. Setting up a root filesystem and gathering the requirements for resource restrictions, needed kernel capabilities, and which syscalls should be used is not discussed. The `PID` and `NET` namespaces are implicitly enabled in this example during the *Set Namespaces* step and does not have additional configuration requirements.

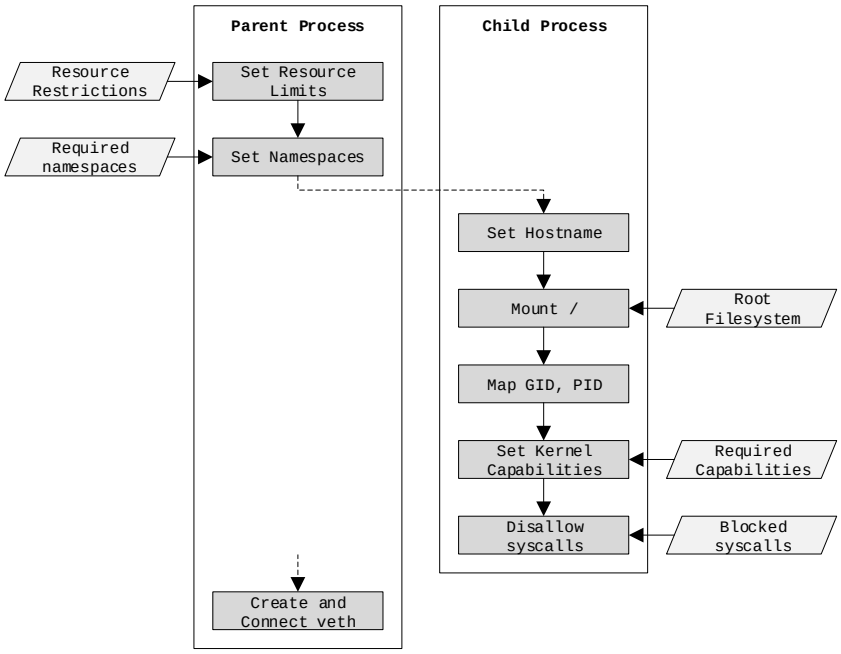


Figure 3.1: Container Creation Process

The process for creating a container starts off with setting resource utilisation limits using `cgroups`. In multi-user and Cloud service environments resource limits have to be applied to running containers to ensure that a single container does not starve the system of resources. `cgroups` can be used to set the following limitations for a container:

- Central Processing Unit (CPU) time used
- The amount of RAM used
- Bandwidth utilisation of block devices (harddrives etc.)

- The number of processes that can be spawned in the container
- Creation of and access to devices
- Usage of Remote Direct Memory Access (RDMA) resources

Once limits for a container have been configured, the namespaces for the new container can be configured. A container can be created using all of the available namespace, or a minimal set that is sufficient. The choice of namespaces used depends on the security and trust environment that container will be used in.

As part of the function that creates a new container, a function is passed that will configure the individual containers. The first task of the *child* process is to execute the function. The simplified steps for creating a container from namespaces are as follows:

1. The function that the *child* process executes starts with setting the hostname for the new container using the `UTS` namespace.
2. The root filesystem for the new container is mounted. A directory that contains the necessary file structure is mounted as the root (`/`) filesystem of the new container using the `MNT` namespace.
3. A mapping of user and group IDs (UID, GID) are created for the container using the `USER` namespace. User and group IDs within the container will map to the user ID that owns the parent process.
4. The new container will start with a nearly full set of kernel capabilities[8]. Any kernel capabilities that are not required or are deemed unsafe are disabled.
5. Any system calls that are deemed unsafe or unnecessary are blocked, preventing the container from causing harm to the host OS.

The final step in setting up a container is connecting it to the outside world. The parent process (or user with sufficient privileges) creates a pair of `veth`[9] devices and attaches one to the container and the other to a network interface or Linux bridge on the host machine. In this scenario, the final step will be for the container to configure the network interface.

[8] `http://man7.org/linux/man-pages/man7/capabilities.7.html`
[9] `http://man7.org/linux/man-pages/man4/veth.4.html`

CBNEs that utilise custom Linux namespaces will repeat this process for each node in the experimental network and the `veth` pairs will be used to connect the containers into the required topology. The tools used by CBNEs to construct network topologies are discussed in Section 3.5.

3.3 Implementations

In this section a set of open-source CBNEs are reviewed and a brief overview of each CBNE is given with regards to the reason for its creation, the history of the CBNE, and how the CBNEs has been utilised since its inception. The driving requirement for the creation of each CBNE and how it is utilised influences its development path and technological choices. Changing operational requirements and a fast-moving technological landscape influences how CBNEs evolve over time. This section attempt to capture this evolution. For CBNE families, a single overview is given for the *family*, and not for individual projects within the family.

3.3.1 Mininet

MiniNet (Lantz *et al.*, 2010) started out as a project to enable large scale OpenFlow (McKeown *et al.*, 2008) experimentation on commodity hardware. The MiniNet project was then expanded to increase functional realism of network simulations, which resulted in MiniNet-HiFi (Handigol *et al.*, 2012; Heller, 2013). The MiniNet Graphical User Interface (GUI) (Figure 3.2) exposes a minimal set of components to construct a network topology. The base components: a host, an OpenFlow switch and controller, a basic switch and

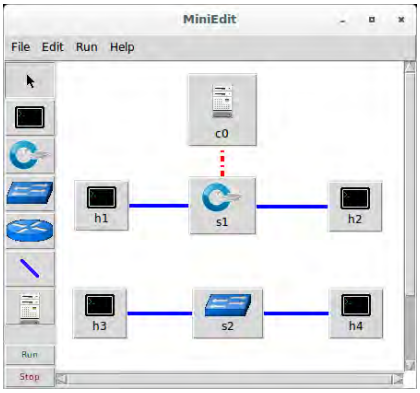


Figure 3.2: Mininet MiniEdit Editor

basic router is provided. The GUI offers options to provide minimal configuration of each component. Constructing a network topology using the command line tools and configuration files allows the user to exert greater control over network topology and component configuration. MiniNet is used in experiments that study the effects of network metrics such as delay and jitter (Qu, 2018). MiniNet is used in education to teach students about the reproducibility of experiments (Yan and McKeown, 2017).

3.3.2 Marionnet

The Marionnet project (Loddo and Saiu, 2007, 2008) was developed by Jean-Vincent Loddo as a teaching aid for his course in networking at the Université Paris 13[10]. Network components in Marionnet are organised into virtual computers and virtual network devices. The virtual computer components emulate networked machines on the emulated network and virtual network devices emulates hubs, switches, routers and links in the emulated network. A virtual external socket component is provided to link physical Ethernet ports on the host machine to the emulated network. Marionnet provides a desktop application with a GUI (Figure 3.3) that allows the end user to configure each component in detail.

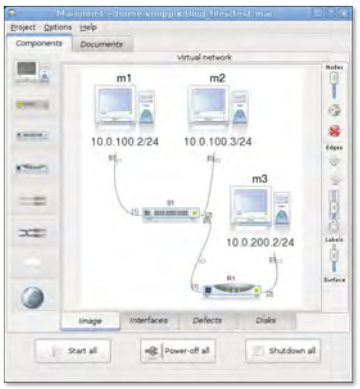


Figure 3.3: Marionnet User Interface

3.3.3 IMUNES

The IMUNES project (Zec and Mikuc, 2004) started out aiming to expand network emulation on the FreeBSD OS in order to incorporate user space applications. The focus on providing application compatibility and multiple network interfaces per Jail, required

[10]Now called Université Sorbonne Paris Nord

extensive modification to the FreeBSD kernel, in particular modifications to the network stack (Zec, 2002, 2003). IMUNES utilises FreeBSD kernels compiled with VIMAGE support to create Jails with multiple network interfaces. By joining these Jails using the netgraph system, complex emulated networks can be created. IMUNES has been used as the base platform for studying various types of network attacks (Salopek *et al.*, 2017) and has been enhanced to replicate Industrial Control System (ICS) systems in a high interactivity honeynet (Kuman *et al.*, 2017). IMUNES has been ported to the Linux kernel and utilises Docker containers and Open vSwitch to create experimental networks on Linux.

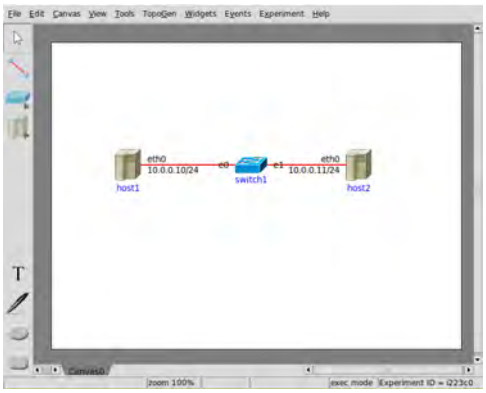


Figure 3.4: IMUNES User Interface

3.3.4 CORE

The CORE (Ahrenholz *et al.*, 2008; Ahrenholz, 2010) started off as a fork of IMUNES by the United States Naval Research Laboratory (NRL) and Boeing, and is maintained by the Networks and Communication Systems Branch of the NRL. The CORE project

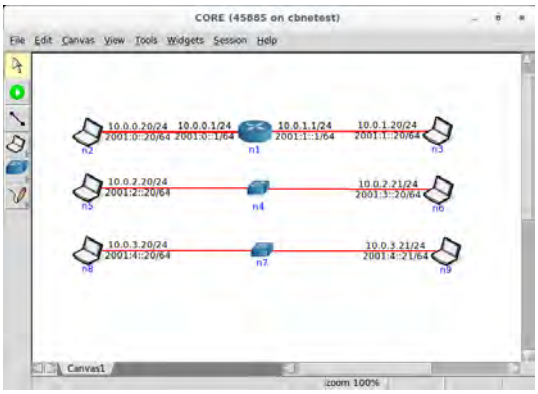


Figure 3.5: CORE User Interface

extended IMUNES with the ability to execute on Linux, a Remote Procedure Call (RPC) Application Programming Interface (API), a Python library and various UI enhancements. Additional goals of the CORE project are to allow wireless network experiments through the EMANE (Ahrenholz *et al.*, 2011), and the ability to distribute network emulation across multiple hosts. Acosta *et al.* (2017) combined the OS level virtualisation features of CORE with VirtualBox VMs to create an environment to capture network traffic that shows the "inside-view" of Red and Blue team operations. The CORE GUI is shown in Figure 3.5.

3.3.5 VNX and VNUML

Virtual Networks over Linux (VNX) (Fernández *et al.*, 2011) is a continuation of the Virtual Network User Mode Linux (VNUML) project (Galan *et al.*, 2004). VNUML started as an emulation platform to study the address assignment model in IPv6 (Fernandez *et al.*, 2004). The emulation platform used for the study was developed into VNUML to support research projects related to computer networks. Development of the VNUML platform was halted in 2009 and has been replaced by VNX. The goals of VNX is to include virtualisation tools to support operating systems other than the host platform in network experiments. It incorporates libvirt and DynaMIPS to achieve these goals. VNX does not have a graphical user interface, however it can produce a graphical map of the current emulation (Figure 3.6). VNX has been used as an interactive honeynet (Fan *et al.*, 2015) and as an experimental environment to test Software Defined Networking (SDN) monitoring solutions (Martínez-Casanueva, 2018).

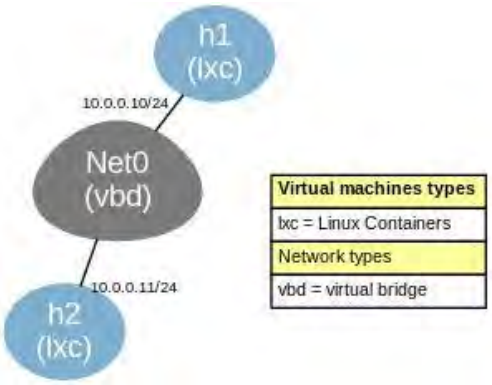


Figure 3.6: VNX Emulation Output

3.3.6 Kathará and Netkit

NetKit (Rimondini, 2007; Pizzonia and Rimondini, 2008) is a project by the Computer Networks Laboratory of the Roma Tre University to enable network experiments to be executed on commodity hardware. NetKit-NG (Iguchi-Cartigny, 2014) is a fork of Netkit, aiming to update the operating system version used. NetKit and NetKit-NG does not provide a GUI, however 3rd party tools such as Visual Netkit (Fazio and Minasi, 2009) are available. Netkit has been superseded by Kathará (Bonofiglio *et al.*, 2018). Kathará extended Netkit with support for Docker as a node emulation technology. The NetKit Lab Generator (Figure 3.7), a web based experimental network topology configuration tool, is now maintained by the Kathará developer. Kathará is used as a teaching aid for computer network technologies at Rome Tre University.

Figure 3.7: Netkit Lab Generator Interface

3.4 Architecture

The architectural choices made during the implementation of each CBNE are analysed and compared to better understand the current state of CBNEs as frameworks for network experimentation. Each CBNE is analysed to assess choices regarding the human-machine interface, how it exposes backend functionality, the design of the backend and the choice of virtualisation technologies. An additional comparison that is included is the capability of a CBNE to distribute an emulation across multiple host machines. In Figure 3.8, a preliminary model is shown that will be used to analyse each CBNE.

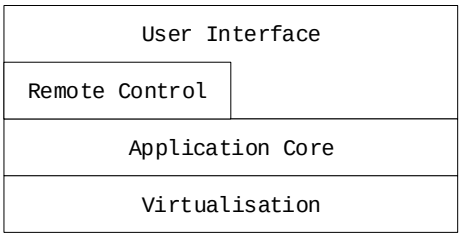


Figure 3.8: CBNE Architecture Comparison Framework

3.4.1 User Interface

The choice of Human-Machine Interface (HMI), such as GUI or Command Line Interface (CLI), that CBNEs expose to end users is a balancing act between ease of use and control. CBNEs that expose GUIs focus on ease of use and rapid design of experimental networks, whereas CBNEs that expose CLIs focus on fine grained control and automation. The interfaces that CBNEs expose are listed in Table 3.3. Marionnet, IMUNES, and CORE are the only CBNEs that natively expose GUIs. It is important to note that the distinction between graphical and command-line based CBNEs is not clear cut: CORE provides both graphical and command line interfaces, while VNX provides a CLI for configuration and initiation of networks and a GUI for interacting with experimental networks.

Table 3.3: CBNE User Interface Architecture

	Human-Machine Interface	
CBNE	Graphical	Command-Line
MiniNet	Limited	•
Marionnet	•	•
IMUNES	•	×
CORE	•	•
VNX	×	•
Kathará	○	•

• Built-in
○ Configuration only
× Run-time only

MiniNet (Section 3.3.1) provides a GUI with a limited set of configuration options through MiniEdit. MiniEdit allows for the creation of network topologies utilising standard switching and routing, as well as for network topologies making use of NFV through OpenFlow. Advanced configuration of nodes can be done by manually editing configuration files through the standard CLI.

Marionnet (Section 3.3.2) provides a primary UI through an integrated UI. The Marionnet UI can be used to create network topologies and configure nodes within an experimental network. A secondary UI is provided in the form of a CLI. The CLI can be used to start, stop, and monitor running experiments. The Marionnet GUI supports an exam mode that is used during the evaluation of students.

IMUNES (Section 3.3.3) provides a GUI that allows end users to create network topologies and configure nodes. Additional CLI tools are provided that enable experimenters and researches to execute commands in nodes, copy files between emulated nodes and the host, and to modify links settings during runtime.

CORE (Section 3.3.4) provides both an extensible GUI and a CLI. Additional applications that run on emulated nodes can be integrated into CORE by extending the routines that initialise nodes. Both the CORE GUI and CLI provide end users with extensive configuration and customisation option for both the network topology and individual nodes. The GUI and CLI acts as client-side utilities that interface with a remote daemon.

VNX (Section 3.3.5) provides the end user with a suite of CLI-based utilities to create and stop network experiments, as well as a suite of utilities to interact with a running experiment. XML configuration files are used to define nodes and network topologies for VNX. The topology of a scenario can be visualised by rendering a raster image.

Kathará (Section 3.3.6) was designed as an upgrade of NetKit and provides a CLI to the end user that maintains compatibility with the original NetKit CLI. Maintaining compatibility with the NetKit CLI enables end users to utilise Kathará with third party GUI utilities such as VisualNetkit (Fazio and Minasi, 2009) and NetKit Lab Generator.

3.4.2 Application Core and Remote Control

CBNEs can be used to create complex experimental networks on commodity hardware, though situations arise where a single computer is starved of resources due to the number of nodes instantiated in an experimental network. The design choices made during the implementation of a CBNE determines whether an experimental network can be distributed across a larger number of computers. In this section, an analysis of CBNEs focuses on the design of the software and how remote control is achieved for distributing experimental networks across multiple hosts.

The evaluated CBNEs can be group into modular and monolithic systems. Table 3.4 shows the application architecture groupings of the evaluated CBNEs. The modular CBNEs are

designed as sets of standalone libraries and executables. Each of these components handle specific tasks during the design and instantiation of experimental networks. The modular CBNEs are designed as layered libraries to create a unified Application Programming Interface (API) that can be used to instantiate multiple types of nodes in an experimental network. The unified API can then be used to define and configure nodes regardless of the virtualisation mechanisms used. Monolithic CBNEs are built as a single application that incorporates all aspects of network emulation. The only CBNEs that is monolithic is IMUNES.

Table 3.4: CBNE Application Architecture

CBNE	Modular	Monolithic
MiniNet	•	
Marionnet	•	
IMUNES		•
CORE	•	
VNX	•	
Kathará	•	

The ability of a CBNE to be remotely controlled and to create distributed network experiments limits the applications of a CBNE. In situations where an experimental network has to execute on fixed infrastructure and has to be controlled from a portable device, the ability to remotely control an experiment is required. Remote control of a CBNE does not imply that network experiments can be distributed over multiple sets of hardware. For distributed emulation each CBNE is evaluated for its ability to fragment and instantiate an experiment network topology on multiple disparate hardware platforms. A summary of the analysis is shown in Table 3.5.

Table 3.5: CBNE Remote Control Architecture

CBNE	Remote Control	Distributed Emulation
MiniNet		×
Marionnet		
IMUNES	○	×
CORE	•	•
VNX	×	×
Kathará	○	

× 3rd party solution
○ Implicit remote control

CORE is the only CBNE designed for remote control and distributed emulation. CORE utilises a RPC control interface that issues instructions to a daemon that handles the

lifecycle of nodes in an experimental network.

Kathará and IMUNES (on Linux) utilise Docker to instantiate nodes for an experimental network (Section 3.4.3) and thus have implicit remote control of the instantiated network. For these two CBNEs the hardware that executes network experiments is disjointed from the hardware used to control an experiment. On FreeBSD, IMUNES has been extended to enable distributed emulation (Puljiz and Mikuc, 2006).

VNX has no native remote control or distributed emulation capabilities, though third-party solutions exist to enable these features. EDIV, a solution for distributed emulation in NetKit, has been updated to support VNX (Fernández *et al.*, 2011). Similar to VNX, distributed emulation support for MiniNet v2.2.1 is supplied by a third-party solution (Wette *et al.*, 2014).

An alternative to distributed emulation is multi-instancing. In multi-instancing, the configuration of an experimental network is broken up into more than one configuration, and multiple instances of a CBNE are used to execute the experiment. Synchronisation of the lifecycle of a running experiment is done manually. Multi-instancing can theoretically be accomplished on any CBNE that supports incorporating physical devices.

3.4.3 Virtualisation

The following comparison of CBNE virtualisation techniques assesses the type of technology used to instantiate nodes within the network topology. Each CBNE implementation can either use containerisation on its own or use containerisation in combination with virtualisation. The only CBNE that still maintains support for technologies other than containerisation is VNX. VNX currently supports instantiating nodes in experimental networks using containerisation, Hardware Abstraction Layer (HAL) level virtualisation, and Instruction Set Architecture (ISA) level virtualisation through Dynamips. CORE dropped support for Xen based nodes in v5.1. The current status of virtualisation mechanisms supported by the evaluated CBNEs is shown in Table 3.6.

Table 3.6: CBNE Virtualisation Architecture

CBNE	Containerisation	Virtualisation	Other
MiniNet	•		
Marionnet	•		
IMUNES	•		
CORE	•		
VNX	•	•	•
Kathará	•		

3.5 Technology

In this section, technologies used to implement the main features of a CBNE are enumerated. A CBNE must address a minimum of two aspects - nodes and topology - the base components for a computer network. A third aspect of a computer network, link metrics, is addressed by some CBNEs. Built-in capability to generate background traffic in the emulated network is not addressed. The three technological aspects of CBNEs that were assessed are:

- Node Emulation - the technologies used to instantiate devices such as computers
- Network Device Emulation - the technologies used to create networking devices such as switches and routers
- Link Emulation - the technologies used to apply network traffic metrics such as jitter and packet loss

This comparison framework is shown in Figure 3.9.

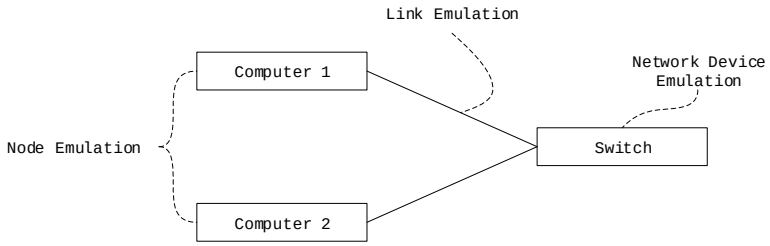


Figure 3.9: CBNE Technology Comparison Framework

3.5.1 Node Emulation

The first component that a CBNE needs to create is that of virtualised nodes within an emulated network topology. Each CBNE is analysed with respect to the different technologies that it can use to instantiate nodes. The technologies used to instantiate nodes can range from existing, well-known systems such as UML (Dike, 2006), to custom containerisation implementations that address needs specific to the CBNE. The technologies used by the evaluated CBNEs to instantiate nodes are shown in Table 3.7 and discussed below.

Table 3.7: CBNE Node Emulation

CBNE	Virtualisation Subsystem(s)
MiniNet	Linux namespaces
Marionnet	UML
IMUNES	Docker
CORE	Linux namespaces, Xen[a,b]
VNX	Dynamips[c], Linux Containers (LXC), UML[d], VirtualBox[a]
Kathará	Docker

[a] HAL level virtualisation (Section 2.2.2)
[b] Support for virtualisation discontinued in v5.1
[c] ISA level virtualisation (Section 2.2.1)
[d] Support for UML is no longer maintained

UML is a project to port the Linux kernel to a user space process. UML allows a user to start a full Linux OS as an executable. UML is not dependent on the host kernel and can be used to instantiate a VM using a different kernel to that of the host OS. The UML project was merged into the Linux kernel in January of 2002 (Boissiere, 2002).

LXC[11] is a Linux containerisation project started in 2008. LXC is built on Linux namespaces and provides a suite of utilities to create, manage, and maintain containers.

Docker (Petazzoni and LeClaire, 2014) is a container management and orchestration system developed by a team at dotCloud (now defunct), a PaaS company, to provide lightweight virtual machines (containers) to customers. The initial version of Docker used LXC to handle the creation of virtual machines.

Dynamips (Fillot, 2005) is an ISA-level virtualisation system for the Microprocessor without Interlocked Pipelined Stages (MIPS) architecture. It was created to provide an

[11] `https://linuxcontainers.org/`

emulation environment that can boot the Cisco Internetwork Operating System (IOS) for training and experimentation using IOS.

Xen (Barham *et al.*, 2003) is an open-source Type-I hypervisor (Section 2.2.2) developed by the Linux Foundation.

VirtualBox is an open source Type-II hypervisor (Section 2.2.2) developed by Oracle Corporation.

Certain CBNEs such as MiniNet and CORE do not rely on third-party virtualisation systems. These projects implement virtualisation through Linux namespaces (containerisation) by directly interacting with the Linux kernel, and also provide custom container management utilities.

3.5.2 Network Device Emulation

The second component that a CBNE needs to address is the creation of a network topology that links instantiated nodes. CBNEs can create network devices using instantiated nodes, such as routers, using Linux and a routing package[12]. For switching networks, L2 switches can be created using Linux bridges. The technologies used to create network devices by the evaluated CBNEs are shown in Table 3.8.

Table 3.8: CBNE Network Device Emulation

CBNE	Network Emulation Subsystem(s)
Mininet	Open vSwitch, Indigo Virtual Switch, OpenFlow reference implementation
Marionnet	VDE switch, Linux bridges, TUN/TAP
IMUNES	Open vSwitch
CORE	Linux bridges
VNX	UML virtual switch, Open vSwitch, Linux bridges
Kathará	Open vSwitch, BMv2

Linux bridges (Böhme and Buytenhenk, 2001) were created to bridge physical network interfaces into a single network according to the IEEE 802.1D-2004 (2004) standard. Linux bridges support the Spanning Tree Protocol (STP) as defined in the same standard.

TUN/TAP devices and the UML virtual switch were both created as part of the UML project (Dike, 2000, 2006). TUN/TAP devices emulate packets arriving from an external

[12]A common package used for routing on Linux is `quagga` (`https://www.quagga.net/`)

source by injecting packets into the network stack of the host OS. TAP devices carry ethernet (L2) frames and TUN devices carry Internet Protocol (IP) packets. The UML virtual switch was created to connect UML VMs into a Local Area Network (LAN).

Open vSwitch (Pfaff *et al.*, 2009, 2015) is an open-source software network switch. Open vSwitch can function as a distributed switch, enabling hypervisors (Virtual Machine Monitors (VMMs)) to create a single switch across multiple hardware instances. Open VSwitch supports the OpenFlow (McKeown *et al.*, 2008) specification.

Indigo Virtual Switch[13] is a software switch that provides support for the Indigo Framework and the OpenFlow protocol. The Indigo Virtual Switch was designed for use with Kernel-Based Virtual Machine (KVM) VMs.

OpenFlow reference implementation is the original reference software switch created during the creation of OpenFlow(McKeown *et al.*, 2008). An archive of the original source code can be found on GitHub[14].

Virtual Distributed Ethernet (VDE) switch (Davoli, 2005) is an open-source distributed software switch. VDE switch was primarily developed to interconnect geographically distributed hardware into a single overlay network.

BMv2[15] (behavioural model, version 2) is the second iteration of the P4 language (Bosshart *et al.*, 2014) software switch reference implementation. BMv2 supports programmable switching using the P4 language.

3.5.3 Link Emulation

The third component of implementation comparison is link emulation. Link emulation addresses the need to control the characteristics of network traffic flowing in the instantiated topology. The ability to control link metrics, such as throughput and packet loss, increases the realism (fidelity) of the emulated network, allowing replication of real-world conditions for network experiments. Table 3.9 lists the link emulation technologies used by the evaluated CBNEs. A short description of each link emulation technology is given below.

[13] http://www.projectfloodlight.org/indigo-virtual-switch/
[14] https://github.com/cl4u2/openflow-reference-implementation
[15] https://github.com/p4lang/behavioral-model/

Table 3.9: CBNE Link Emulation

CBNE	Link Emulation Subsystem(s)
Mininet	tc, netem
Marionnet	wirefilter
IMUNES	tc
CORE	tc
VNX	tc
Kathará	

The Linux **tc**[16] (Traffic Control) utility allows various network traffic control policies to be enforced on a Linux OS. The `tc` utility allows a user to control shaping, scheduling, policing, and dropping policies. Shaping controls the transmission rate of network traffic on egress, scheduling controls the transmission priority of packets, policing allows policies to be set on arriving traffic, and dropping allows packet to be dropped if bandwidth limits are exceeded.

The Linux **netem**[17] (Hemminger, 2005) utility extends the `tc` utility with the ability to add network metrics such as delay, packet loss, duplication and corrupt network traffic. The `netem` (Network Emulator) utility was created to expose networked applications to real world conditions during stress testing.

VDE provides the **wirefilter**[18] utility to emulate packet loss, delays, and duplication between VDE components. VDE wirefilter can also impose bandwidth restrictions and introduce errors into network packets. The VDE wirefilter utility can utilise user generated Markov chains to vary chosen metrics over time.

3.6 Summary

Available network experimentation platforms give the end user a choice of fidelity level that is most suited to experiments that are to be done. CBNEs as an alternative network experimentation platform presents a middle ground in terms of node density and fidelity. CBNEs are able to have hundreds of nodes in an experimental network while still having access to an operating system kernel capable of executing real-world applications. This allows for experimentation that requires interaction with real-world applications at a large

[16] `http://man7.org/linux/man-pages/man8/tc.8.html`
[17] `http://man7.org/linux/man-pages/man8/tc-netem.8.html`
[18] `https://manpages.debian.org/stretch/vde2/wirefilter.1.en.html`

scale. The open-source CBNEs reviewed vary in architecture and implementation. Variations in the architecture and implementation specifics of the different CBNEs allows the end user to select the most appropriate system based on his or her requirements. For education and training environments, the availability of a graphical user interface supersedes the ability to programmatically control the experimental network. By contrast, experimentation with large-scale networks that span multiple computers will benefit from the ability to exercise remote programmatic control over the experimental network. CBNEs present a viable, low cost alternative for network administration, education, and security specialists. The specific requirements of an experimental setup will lead the end user to select a Container-Based Emulator (CBE) that can function within constraints of the environment that the experiment will be executed.

Chapter 4

Remote Fidelity of Abstracted Hosts

One sees qualities at a distance
and defects at close range.

VICTOR HUGO

In Chapter 3 Container-Based Network Emulators (CBNEs), were investigated to determine if any components used during the construction of these systems would be able to alter the fingerprint of an abstracted host. As was shown, the technologies used in CBNE systems have the ability to alter network traffic, confirming the hypothesis that from the viewpoint of an attacker the fingerprints of hosts abstracted in CBNEs could differ from the fingerprint of the host Operating System (OS) on which the CBNEs is executed.

OS fingerprinting (Trowbridge, 2003) is the process of identifying a remote OS by extracting and analysing specific features from network traffic between a scanning entity and a target entity. This extraction and analysis is based on the knowledge that each OS will introduce artefacts into network traffic protocols based on assumptions and interpretations made during the implementation of standards governing these protocols. Classical examples of the processes to build this body of knowledge can be found in Fyodor (1998) and Lyon (2009).

This chapter starts off by exploring the techniques and technologies used in identifying the OS and software of a remote computer using network traffic. The methods used to construct fingerprints of remote hosts from network traffic are investigated. The artefacts collected during fingerprint generation are investigated for their relation to both the abstracted and host OSs. These artefacts are evaluated as to how an abstracted machine's fingerprint may differ from a real machine.

In Section **4.1** the primary techniques of OS fingerprinting were investigated to assess the use of feature extraction and how this would relate to abstracted hosts on a network. During the fingerprinting process artefacts are collected that relate to various components of a computer system. Features extracted from collected artefacts mainly relate to the network and OS, and these features have the most impact when identifying an OS from a generated fingerprint.

Next, theoretical models that attempt to explain the patterns and practices of a remote attacker were reviewed to assess the applicability of fingerprinting during such activities. By analysing these attack patterns, the utilisation and importance of fingerprinting networked hosts during attacks on remote systems can be gauged. In Section **4.2** four classes of network attack models were investigated to understand the processes that a hacker will likely take to exploit and gain access to a networked computer. The models that do rely on OS fingerprinting during target selection characterize the expected behaviour of attackers targeting a specific machine. Models that have no or little reliance on OS fingerprinting characterize the behaviour of attackers attempting to gain access to any exploitable machine within a network.

Lastly, remote fingerprinting technology is used as a basis for the construction of a model for measuring the realism of an abstracted host. Fingerprinting of remote systems utilises concepts similar to the concepts commonly used in Naval Sound Navigation and Ranging (SONAR) systems in the physical world. This similarity is utilised to construct a model to be used for the measurement of realism of the abstracted hosts. In Section **4.3** the conceptual similarities between OS fingerprinting and SONAR are used as a basis to construct a model for measuring the remote fidelity of an OS. If a fingerprint generated from a host machine matches the fingerprint generated from an abstracted machine, the abstracted machine is assumed to have perfect fidelity for the purposes of this study.

4.1 Operating System Fingerprinting

Contemporary OS fingerprinting utilities can be split into two main methods of operation: active and passive (Spangler, 2003). Active fingerprinting utilities are designed to send probes to a targeted entity that are crafted to solicit responses that contain features that will enable the utility to determine a probable match for the OS and services. Passive fingerprinting utilities (Lippmann *et al.*, 2003) rely solely on the observation and analysis of "legitimate" communications between two networked entities, using the assumption

that a sufficient number of features will be found in these communications to enable the utility to find a probable match for the host OS or services. Albanese *et al.* (2016) created a taxonomy of fingerprinting utilities (Figure 4.1), further refining active fingerprinting utilities. The first refinement of active utilities was to split these utilities into those extracting features from application traffic and those extracting features from network protocols. In the original text, each bottom most layer of the taxonomy is associated to a particular utility that serves as an example of the particular type of fingerprinting utility. It is worthy to note that `nmap` is the only utility listed in Albanese *et al.* that falls into more than one classification in the taxonomy in the original text.

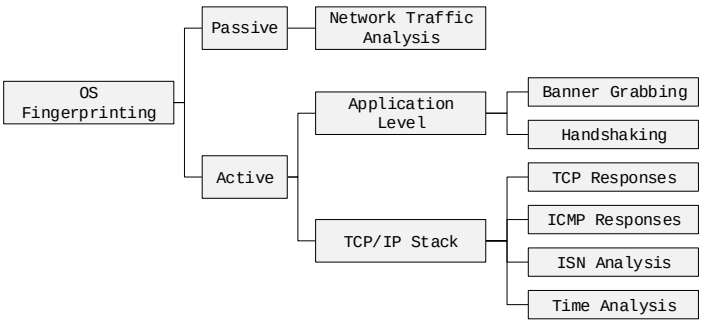


Figure 4.1: Operating System Fingerprinting Taxonomy, After Albanese *et al.* (2016)

Identifying the OS and enumerating services on a remote computer is crucial to an attacker attempting to penetrate a remote network. Successfully identifying the OS or services on a remote computer enables an attacker to assess the vulnerability of the system, and the hacker(s) may then proceed to use known exploits or create custom exploits to gain access to the remote system. Without the knowledge of the OS or services on the remote system, an attacker will most likely not be able to enumerate weaknesses in the system that can be exploited.

The simplistic feature extraction methods of first-generation fingerprinting utilities have evolved into techniques that exploit identifiable characteristics of protocol implementations. By soliciting responses that would leak tell-tale features, positive identification of OSs can be performed. Similarly, banner grabbing has evolved, from simplistic string matching to application data content analysis, to identify the services running on the remote machine. Table 4.1 lists the first and current releases of a selection of OS fingerprinting utilities.

`Queso` (Hack Story, 2009), an active fingerprinting utility released in March of 1997, had a major influence on the direction that active fingerprinting research would take. `Queso`

Table 4.1: Operating System Fingerprinting Utilities

Name	Current Version	Initial Release	Last Update
Queso	980922	1997-04	1998-09-22
nmap	7.7	1997-09	2018-03-21
Siphon		2000-05-04	
p0f	3.09b	2000-08-01	2016-04-18
ettercap	0.8.2-Ferri	2001-01-25	2015-04-14
X	0.0.1	2001-07-13	
Ring	1.1	2002-04	
xprobe2	0.3	2003-10-13	2005-07-27
SinFP	2.10	2005-06-20	2015-02-15
SinFP3	1.24	2012-09-22	2018-07-21

pioneered having a fingerprint database in a separate file. Prior to Queso, fingerprint databases where compiled into the executable of fingerprinting utilities. Queso was soon followed by the release of nmap (Lyon, 2009) in September of 1997. The feature set and capabilities that nmap introduced quickly made it the de-facto active fingerprinting utility. Queso and nmap represent the first of the active fingerprinting utilities.

In 2000, two passive scanning utilities were released. Siphon (bind, 2000) and p0f (Zalewski, 2000). Both these utilities created fingerprints for OSs by extracting a specific, but limited, set of features from packets of the Transmission Control Protocol (TCP) handshake. Siphon extracted three features from SYN and ACK packets, whereas p0f extracted eight features from the SYN packet (Lippmann *et al.*, 2003). Ettercap[1] (Ornaghi and Valleri, 2019), another passive fingerprinting utility, was released in 2001. Ettercap expanded the number of features extracted to ten, and incorporated features from SYN-ACK packets, thus utilising both the client and server packets to generate fingerprints. Recent use of p0f has shifted from passive fingerprinting to conducting MitM and Address Resolution Protocol (ARP) spoofing attacks (Salame, 2019). This shift in focus could explain the lack of updates to the p0f passive fingerprint database (Section 5.2.3, Table 5.4).

Both x (also named xprobe) and xprobe2[2] (Arkin and Yarochkin, 2002; Yarochkin *et al.*, 2009) introduced fuzzy algorithms to match features sets to OSs. Through the use of fuzzy matching, xprobe2 can match an OS even if some tests failed to produce results. RING (Veysset *et al.*, 2002), released in 2002, created fingerprints for OSs without violating any RFCs, effectively preventing fingerprinting activity from being detected by Intrusion Detection and Prevention Systems (IDPSs).

SinFP (Auffret, 2008) and SinFP3[3] (Auffret, 2010) was designed to positively identify

[1] http://www.ettercap-project.org/
[2] https://sourceforge.net/projects/xprobe/
[3] https://metacpan.org/release/Net-SinFP3

OSs in worst case scenarios. In active fingerprinting mode, `SinFP` aims to identify an OS using three packets. In passive fingerprinting mode, `SinFP` can intercept live traffic or conduct analysis on static captures.

4.1.1 Active Fingerprinting

Active fingerprinting utilities such as `xprobe2` (Arkin and Yarochkin, 2002), `nmap` (Lyon, 2009) and `SinFP3` (Auffret, 2010) are so called as they actively interrogate a remote host to identify the OS and services available. This interrogation is conducted by sending probes to a remote host and analysing the responses, or lack thereof, for features that are typical to a class of OS. Interrogation of the remote system may optionally conduct a "port scan", an attempt at enumerating the open port and protocol pairings of the remote machine.

During simplistic port scanning, TCP and User Datagram Protocol (UDP) connections are attempted to a selection of ports on a remote host. Services that have well-known protocol and port combinations (Cotton *et al.*, 2011) can be identified through a simple port scan, an example of this is 53/tcp or 53/udp that can likely be matched to a DNS service. Service discovery can be conducted by inspecting return traffic from a protocol and port combination to enumerate services remapped to uncommon ports, for example Hypertext Transfer Protocol Secure (HTTPS) remapped from 443/tcp to 8443/tcp or SSH remapped from 22/tcp to 8022/tcp. Aggressive service discovery can be conducted by attempting to establish service specific connections to uncommon ports.

The probes that an active OS fingerprinting utility sends are aimed at exploiting known characteristics such as ambiguities in Request for Comments (RFCs) and implementations of the TCP/IP stack within the remote host, thereby soliciting responses that will confirm or refute a match against a known OS or service. The algorithms of contemporary active OS fingerprinting utilities such as `nmap` are enhanced to only solicit responses that will result in more information. Probes that will not solicit new information are not utilised (Lyon, 2009, Chapter 7). Barnett and Irwin (2008) investigated the types of scans performed by fingerprinting (scanning) utilities and created a taxonomy of the types of scans that can be expected. OS identification and service enumeration is most commonly associated with Layer 3 scanning. The taxonomy of Barnett and Irwin (Figure 4.2) illustrates the types of Layer 3 scans that can be used to solicit responses that will contain features that can identify a particular OS.

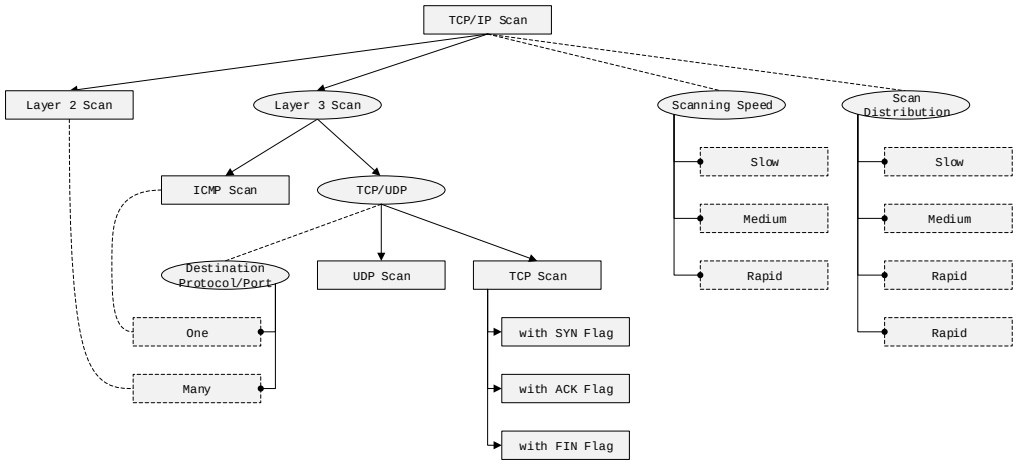


Figure 4.2: Active Scanning Taxonomy, After Barnett and Irwin (2008)

During an assessment of machine learning applied to automated fingerprinting, Richardson *et al.* (2010) investigated the relationship between features that can be extracted from TCP/Internet Protocol (IP) traffic and how these features relate to OSs. Table 4.2 summarises their findings. Extracted features that relate to the OS source code definitely (indicated by $\times$) influence the fingerprint of a target machine and extracted features that relate to the network between the attacking and target machine probably (indicated by $\bullet$) influences the fingerprints of the target machine. In Table 4.2 *non-determinism* is interpreted as being any action that a network device or OS can take with regards to network traffic that is not deterministic, such as packet re-ordering, randomly generated sequence numbers, or packet loss. *Hidden state* expresses the complexity of the machines being modeled, specifically the ability of complex systems to change behaviour over time.

When an active OS fingerprinting utility has positive confirmation of a specific OS family (Linux; Windows; BSD Family), the algorithms of the utility will be directed at crafting probes that are capable of soliciting responses from the remote system that will enable the utility to refine the matched OS (e.g. Linux Kernel 2.x, Linux Kernel 3.x or Linux Kernel 4.x[4]). The algorithms of active OS fingerprinting utilities include soliciting responses that may result in information leakage, typically known as banner grabbing. Many services are known to leak information regarding the host OS or the version of the service, a default Apache HTTP Server and PHP (Listing 4.1) stack will leak software versions and details of the host OS. This information enables the active OS fingerprinting utility to create a positive match to a particular OS.

[4]The 4.x naming scheme was an administrative decision - Torvals (2015)

Table 4.2: Observable Behavioural Differences for IP and TCP Network Traffic, After Richardson *et al.* (2010)

	non-determinism	hidden state	network	hardware	applications	system configuration	OS source code
IP Field							
version			•		•	•	×
hdrlen			•		•	•	×
tos			•		•	•	×
len			•		•	•	×
id	×	×	•				×
flags			•				×
frag			•				×
ttl	×		×		×	×	×
proto			•				×
chksum	×	×	×	×	×	×	×
TCP Field							
seq	×	×	•				×
ack			•				×
dataofs			•		•	•	×
reserved			•				×
flags			•		•	•	×
win size			•	•	•	•	×
chksum	×	×	×	×	×	×	×
urgptr			•		•	•	×
op order			•		•	•	×
op MSS			•		×	×	×
opt wscale			•	×	×	×	×
opt tsval	×	×	•		×	×	×

× Influences fingerprint
• Might influence fingerprint

Listing 4.1: Apache & PHP Information Leakage

```
HTTP/1.1 400 Bad Request
Date: Fri, 30 Mar 2007 09:59:37 GMT
Server: Apache/2.0.54 (Debian GNU/Linux) PHP/4.3.10-18
Content-Length: 337
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

Active OS fingerprinting utilities are considered to be "noisy". The volume of probes sent to a remote system can be identified by modern Unified Threat Management (UTM) (Sophos, 2018) and Next Generation Firewall (NGFW) (Cisco Systems, 2018) systems as well as by host-based scan detection systems such as `psad`[5] (Rash, 2001), alerting the operators of the remote system or network that scanning activity is under way. Active OS fingerprinting utilities such as `nmap` can be instructed to apply evasion techniques (Lyon, 2009, Section 4.3.5) in order to minimise the chance of being detected.

[5] `https://cipherdyne.org/psad/`

4.1.2 Passive Fingerprinting

Passive fingerprinting utilities such as `p0f` (Zalewski, 2000) and `ettercap` (Ornaghi and Valleri, 2019) are used in situations where an attacker prefers to avoid detection, or to analyse historical network traffic captures. In contrast to active fingerprinting, passive fingerprinting does not send any probes, but merely relies on "legitimate" communications to generate a probable match for a remote system. Passive fingerprinting extracts features from network traffic between the attacker and target that can be used to generate a fingerprint for the remote system. The initial Time To Live (TTL) of an IP packet and the TCP window size for a particular OS family will differ significantly from other OSs and can be used to determine, at minimum, the family of the target machine's OS (Hjelmvik, 2011; Chen *et al.*, 2014). In Table 4.3, a sample of such pairing are shown, illustrating the differences between a small selection of OSs.

Table 4.3: Typical Initial IP TTL Values and TCP Window Sizes of Common OSs, After Hjelmvik (2011)

OS	IP Initial TTL	TCP Window Size
Linux (kernel 2.4 and 2.6)	64	5840
Google's customised Linux	64	5720
FreeBSD	64	65535
Windows XP	128	65535
Windows 7, Vista and Server 2008	128	8192
Cisco Router (IOS 12.4)	255	4128
Ubuntu 18.04 (Linux 4.15)	64	87380
MacOS High Sierra (10.13.6)	64	131072

Older generation passive fingerprinting utilities such as `p0f` and `ettercap` (Spitzner, 2000) extract features of specific protocols only, relying on the limited set of features that these protocols are able to provide. Features are extracted from the first connection made by the attacker to the host and are then matched to a database of known fingerprints. For protocols such as Hypertext Transfer Protocol (HTTP) and FTP, the initial connection may contain a banner - a disclaimer indicating the version of the service running. This banner is used by passive fingerprinting utilities to supplement the OS fingerprint and enhance the quality of the result returned. When a user connects to a web service the web browser in use typically sends a user agent string to the remote web server. Extracting the user agent string can enable the web service to fingerprint the browser and host OS of the user (Eckersley, 2010; Tanabe *et al.*, 2019). In Table 4.4, a sample of platform tokens and the corresponding OS is shown. If web traffic is sent unencrypted (HTTP and not HTTPS), the user agent string can be intercepted and used to enhance the fingerprint for a device (Hjelmvik, 2011). A comprehensive database of user agent strings can be found at `https://developers.whatismybrowser.com/useragents/`.

Table 4.4: Web Browser User Agent String to OS Match, After Hjelmvik (2011)

Platform Token	Description
Windows NT 6.0	Windows Vista
Windows NT 6.1	Windows 7
Windows NT 10.0	Windows 10
Linux x86_64	Linux 64bit
Intel Mac OS X 10_13_6	macOS High Sierra (Safari)
Intel Mac OS X 10.13	macOS High Sierra (Firefox)
Intel Mac OS X 10_14	macOS Mojave (Safari)
Intel Mac OS X 10_14_1	macOS Mojave (Chrome)
Apple-iPhone7C2	iPhone 6 CDMA
Apple-iPad2C4	iPad 2 Wifi Only 16GB

Passive fingerprinting has various uses outside of hacking. These techniques are employed in UTM and NGFW devices to identify device types and apply filtering rules based on corporate policies (CDW Corporation, 2018). The application of filtering rules based on device type is of particular interest and is used in "BYOD" environments, where employee devices are segmented off from the corporate network, minimising the risk of data loss due to devices infected with malware. Passive fingerprinting can be applied to historical data; in Irwin (2012, 2013) and Hunter *et al.* (2012) passive fingerprinting was applied to historical network telescope data to identify the distribution of OSs that fell victim to the `Conficker` worm.

4.1.3 Current Research

Current research into passive fingerprinting is exploring multi-session fingerprinting (Anderson and McGrew, 2017), protocol agnostic, and machine learning (Aksoy *et al.*, 2017) techniques to enhance the accuracy of OS matches. Multi-session fingerprinting uses all network activity by a specific host to build up a living fingerprint of the device. As connections are made, the fingerprint of a specific device will be augmented with any new information, and the accuracy and granularity of the matched OS improved over time. These multi-session fingerprints are analysed using Machine Learning (ML) techniques, a subset of Artificial Intelligence (AI), such as genetic algorithms and expert systems, enabling modern passive OS fingerprinting utilities to respond to changes in the behaviour of devices, and maintain accuracy of the matched OS across vendor updates to the base OS. In Hunter *et al.* (2012) and Hunter (2017), multiple fingerprinting techniques are combined through data fusion to track network entities, creating temporal situational awareness. In Davis (2019), BigData techniques are explored to analyse and categorise passively collected data. By utilising BigData tools and techniques, passively collected data can be classified according to source entity type such as research institution, residential or reflected data.

4.2 Network Attack Models

An attacker's view of a remote machine is limited to what can be sent and received over a network. One way to verify that a host is exploitable is by fingerprinting the OS. The components used to build CBNEs can influence the fingerprint of emulated hosts. Fan *et al.* (2015) used Virtual Networks over Linux (VNX) to create high-interaction hosts in a honeynet deployment called Honeybrid. Detectability was defined as one aspect of measurement between various honeynet deployments. Boyd (2000) found that User Mode Linux (UML) based honeynets could be detected by remote attackers. Chapman *et al.* (2017) evaluated NetKit and Virtual Network User Mode Linux (VNUML) as systems to create low overhead virtual machines that can be used to create realistic environments for network security training. The work of Chapman *et al.* (2017) focused on the ability to recreate the expected application in a network. The use of CBNEs passed all application level validation tests. In this section, models describing attacks on networks are reviewed to assess the applicability of fingerprinting during activities expected to be performed on CBNEs.

In *A Formalised Ontology for Network Attack Classification*, van Heerden (2014) investigated models describing the behaviour of attacks against networks. These attack models can be divided into three broad classes: generalised models, models focused on penetrating a specific host, and complex and specialised models. The Cyber Kill Chain (Hutchins *et al.*, 2011, a registered trademark of the Lockheed Martin Corporation) is included as a representative model of defensive network security operations. The Cyber Kill Chain is an example of a militarised attack model. In this section, a brief overview of each of these classes of model is given.

4.2.1 Generalised Attack Models

During an analysis of the network attack processes described in Cheswick (1992); Boyd (2000); Schultze (2002); McClure *et al.* (2012); Teumim (2010) and Knapp and Langill (2014), van Heerden (2014, Section 3.2) modelled the common overarching aspects of the described processes. The resultant model is shown in Figure 4.3. The analysed models attempted to describe the most general way that an attacker may go about selecting, exploiting, and gaining access to a target host using commonly available tools and utilities.

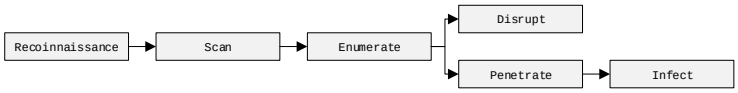


Figure 4.3: Generalised Network Attack Model, After van Heerden (2014)

The *Scan* and *Enumerate* phases of the generalised network attack model focuses on gaining information regarding a target host operating system and remotely accessible services. Terminology regarding when OS fingerprinting and when service enumeration occurs is not consistent. Schultze (2002) and McClure *et al.* (2012) model a *Footprinting* phase to contain both OS fingerprinting and service enumeration, whereas Boyd (2000) uses the *Reconnaissance* phase to describe both. In contrast, the generalised attack model of van Heerden (2014) separates these phases. In the work of van Heerden, the *Scan* phase is used to describe actions taken by the attacker where scanning of a remote computer occurs, and the *Enumerate* phase is used to describe gathering of information from services.

The generalised attack model of van Heerden is primarily informed by the works of Teumim (2010) and Knapp and Langill (2014), who analysed the design and construction of industrial networks. A major part of the analyses focused on security considerations when designing industrial networks. The theoretical attack models used in both these instances modelled an attacker looking for Industrial Control System (ICS) or Supervisory Control and Data Acquisition (SCADA) systems to penetrate. An attacker that wishes to specifically target systems such as ICS or SCADA systems would require positive confirmation that the selected target is such a system. The use of OS fingerprinting technologies is indispensable during such an operation.

The generalised network attack model of van Heerden (2014) offers a succinct overview of the general expected behaviour of attackers. The combination of the *Scan* and *Enumerate* phases describes the actions of a conceptual attacker seeking out specific systems.

4.2.2 Models Focusing on Penetration of Remote Hosts

Van Heerden (2014) analysed five models focused on gaining access to a specific computer system. The models of Nachenberg (2002); Tutănescu and Sofron (2003); Hansman (2003); Gadge and Patil (2008), and Sharan (2010) vary in objectives for creating the model and the motivation of the conceptual attacker, though all models expand on particular details

of the generalised network attack model in Section 4.2.1. A common aspect of the five models above is an increased focus on the motives and objectives of the final phases of an attack on a particular system.

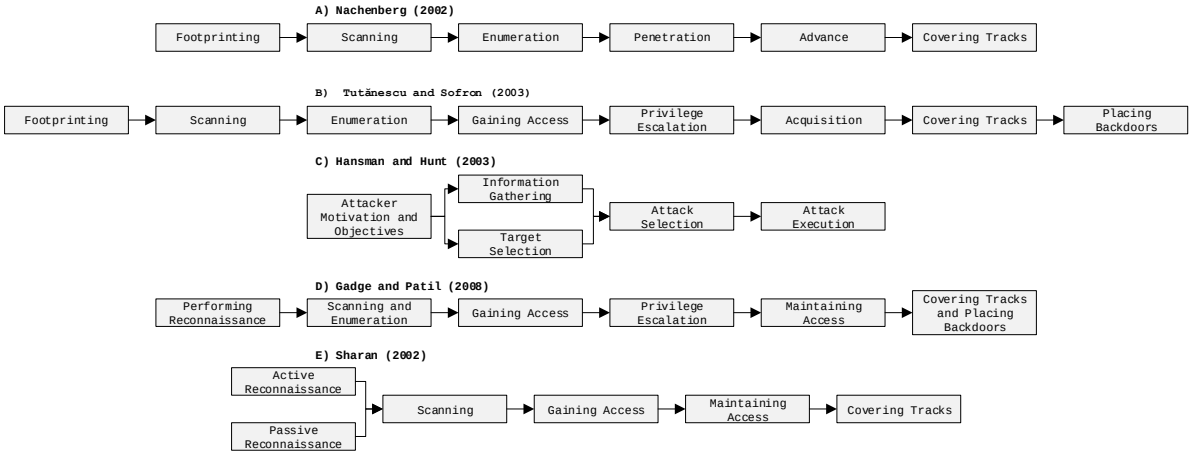


Figure 4.4: Network Attack Models Focused on Penetrating Networks, After van Heerden (2014)

The models presented by Nachenberg; Tutănescu and Sofron (2003); Gadge and Patil (2008) and Sharan modelled scenarios where the attacker wishes to maintain access to a particular system, and extended the generalised model to include the *Covering Tracks* phase. These models were created to assist in attack detection. The model of Tutănescu and Sofron (2003) (Figure 4.4 B) starts with the same phases as the generalised model, but expanded the *Penetrate - Infect* branch of the generalised model into smaller "sub" phases that would assist in attack detection. The research of Gadge and Patil (2008) (Figure 4.4 D) focused on the development of port scan detection methodologies. The *Performing Reconnaissance* phase of Gadge and Patil (2008) models *Footprinting* (*Reconnaissance* in the generalised model) and *Scanning* as sub-phases where port scanning occurs. The model of Sharan (Figure 4.4 E) follows a similar methodology to that of Gadge and Patil (2008) but explicitly differentiates between *Active Reconnaissance* and *Passive Reconnaissance*. The model of Nachenberg (Figure 4.4 A) is a condensed version of the model found in McClure *et al.* (2012).

The model presented by Hansman (2003) (Figure 4.4 C) aimed not to model the execution phases of an attack, but rather the higher level processes that the attacker would follow, with the aim of creating a taxonomy that would accurately describe a wide variety of attacks (including electromagnetic attacks). The taxonomy proposed by Hansman (2003) uses four distinct dimensions (with the possibility to include more) to classify an attack.

The actual process followed by an attacker is not modelled, as the taxonomy classifies an attack that has happened.

During the development of a game theoretic model of Computer Network Exploitation (CNE) campaigns, Mitchell and Healy based the actions of an adversary on the multistage model shown in Figure 4.5. The model of Mitchell and Healy is modelled on an attacker selecting a target (*Survey*) and implementing custom tooling and exploits (*Tool*) to infect (*Implant*) a selected target. The model then differentiates from the models previously mentioned, model allowing for an attacker to abort the campaign at any stage and for an attacker to decide (*Picot*) which malicious activity to perform at a late phase in the campaign.

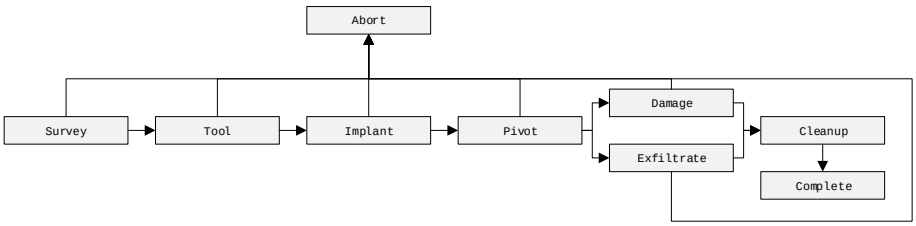


Figure 4.5: Multistage Computer Network Attack Model, After Mitchell and Healy (2018)

The models focused on penetrating a remote host place high value on identifying a target that can be exploited. Regardless of the naming convention used, each of the five models is based on the assumption that an attacker would attempt to gather as much information about a remote host as is feasible, and all five models incorporate the fingerprint as a primary source during the "gathering of information".

4.2.3 Extended Attack Models

Grant *et al.* (2007) and Janczewski and Colarik (2008) model in detail the required or expected inputs and the expected outputs of the process that an attacker could take. The common goal of these models is the incorporation of expected loss during an attack.

The model presented by Grant *et al.* (2007) furthers the research originally conducted in Grant and Kooter (2005), and extends the well-known Observe-Orient-Decide-Act (OODA) loop (Boyd, 1987) by incorporating various other operational process models attempting to describe modern warfare, applying these process to cyber terrorism. The

resultant model (Figure 4.6a) details the inputs and outputs of each phase, and incorporates propagation of the attack and dissemination of information regarding the attack in online and other media.

The model of Janczewski and Colarik (2008) (Figure 4.6b) expands on a basic attack model by modelling the required inputs and expected outputs of each phase. The model presented caters for both disrupting and maintaining a foothold once a target is compromised. Similar to the model of Grant *et al.* (2007), propagation of the attack from a compromised target is included.

Wood (2018) applies the Cyber Kill Chain to attacks focused on stealing sensitive information from corporate environments. The model of Wood (Figure 4.6c) disregards disruption or persistent backdoors and focuses exclusively on the exfiltration of sensitive information.

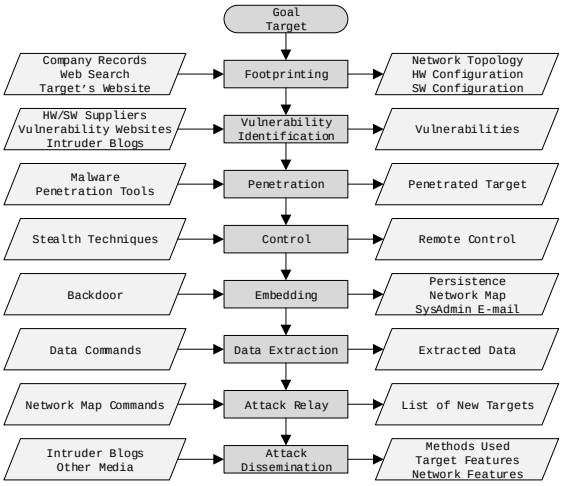
The models described in this section focus on modelling the *modus operandi* of a cyber terrorist. Focus is placed on the motivation for and outcomes of the attack. Information gathering operations within these models place higher value on resources detailing the software systems used and lesser value on techniques used in OS fingerprinting.

4.2.4 Militarised Attack Models

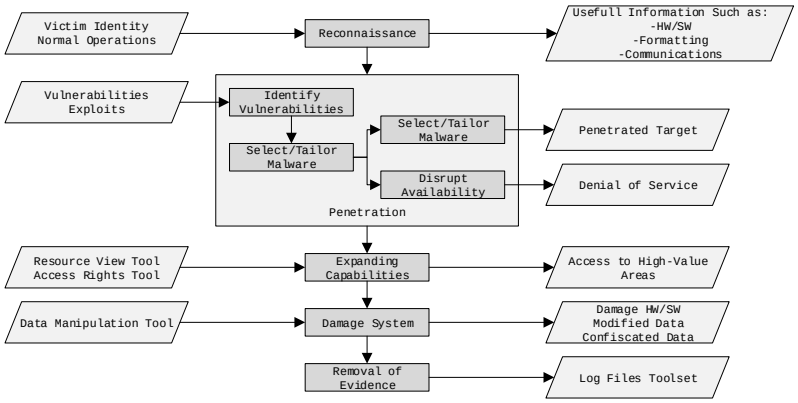
As a response to the nature and *modus operandi* of Advanced Persistent Threats, Hutchins *et al.* (2011) created a model of APT behaviour to assist in the development of defensive strategies that can deal with the sophisticated nature of APTs. The resulting model, the Cyber Kill Chain (shown in Figure 4.7), aims to better enable defensive technologies to detect the presence of APTs during any phase of an attack.

Figure 4.7: The Cyber Kill Chain, After Hutchins *et al.* (2011)

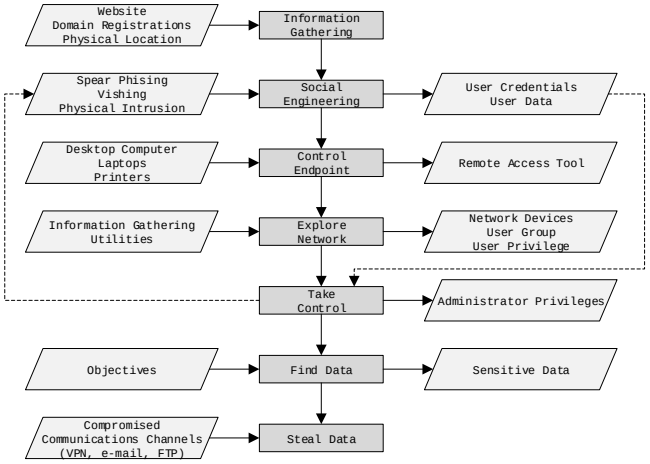
The Cyber Kill Chain, developed by Lockheed-Martin, deviates from the generalised and penetration focused models by expanding the *Delivery* phase to include alternative means of delivery of malicious payloads such as email. The *Reconnaissance* phase of the Cyber Kill Chain focuses mainly on finding exploitable wetware where the previous models focus on the exploitability of hardware or software. In the Cyber Kill Chain, the *Reconnaissance*



(a) Network attack model of Grant *et al.* (2007)



(b) Network attack model of Janczewski and Colarik (2008)



(c) Network attack model of Wood (2018)

Figure 4.6: Extended Attack Models

phase makes use of leaked Personally Identifiable Information (PII) (Swart, 2015) to identify human beings that could be exploited to install malicious software. First contact with computer systems occurs during the *Delivery* phase and relies on the targeted human opening a document that contains a malicious payload. The abstraction level of the computer system that the human uses will influence the success of the attack. The requirement that a human interacts with a computer places the model outside of the scope of this thesis. The Cyber Kill Chain highlights the changes in tactics of adversaries due to enhanced perimeter security of computer networks. Defensive mechanisms in enterprise networks are discussed in 5.1. These defensive mechanisms make the exploitation of hosts behind a Demilitarised Zone (DMZ) from an external location an increasingly difficult task. The delivery of APTs from inside such a network, whether by self-replication or by an insider, could typically utilise one of the previous models.

4.2.5 Applicability of Fingerprinting in Network Attacks

From the reviewed models, the use of fingerprinting by remote attackers is applicable to situations where specific networks or machines are targeted. The general and penetration focused models attempt to describe such situations where attacks are focused on specific machines, whereas the comprehensive models attempt to describe a "get in by any means necessary" approach to gaining a foothold in the targeted network. Deploying educational or experimental networks in CBNEs is associated with training exercises, where a Red Team will attempt to break into infrastructure controlled by a Blue Team, a scenario that matches the models where fingerprinting is used (Vykopal *et al.*, 2017). This testing methodology aligns with the expected use and application of Network Experimentation Platforms (NEPs) within the context of this study.

4.3 Remote Fidelity of Networked Hosts

With the primary goal of assessing the fidelity of hosts within CBNE emulations from the viewpoint of a remote attacker, it is necessary to establish a definition for fidelity, and the techniques and technologies available to measure such fidelity. From the viewpoint of an attacker, the only information available with regard to the target host OS is contained within network traffic - either traffic broadcast by the target or responses solicited by the attacker.

4.3.1 Network Traffic Modification

Apart from deliberate modification of network traffic by using the technologies listed in Section 3.5.3, additional opportunities exist for CBNEs to modify network traffic.

The choice of namespaces used to construct a container and the capabilities and limitations applied to the container (Section 3.2.1) can have an impact on how network traffic is handled. If a container has certain capabilities, the way that network packets are handled by the Linux kernel can be influenced. The nature of containers implies that the kernel of the OS remains the same. As was shown by Richardson *et al.* (2010) the primary component in a computer system that can influence fingerprints is the source code of the OS (Table 4.2).

The primary technology used in CBNEs that could influence the features extracted by fingerprinting utilities is the components used to emulate network devices (Section 3.5.2). These devices live outside of the bounds of the kernel and can apply various optimisations to traffic, and thus influence fingerprints generated from network traffic.

4.3.2 Active Hunting

In active SONAR, a "ping" signal is generated at a specific frequency and an array of detectors listens for the ping signal's return. The conceptual design of an active SONAR system is shown in Figure 4.8. The returned signal is analysed for time delays and frequency distributions for probable matches to a "contact". In some cases, a signature can be created from analysis of the returned signal.

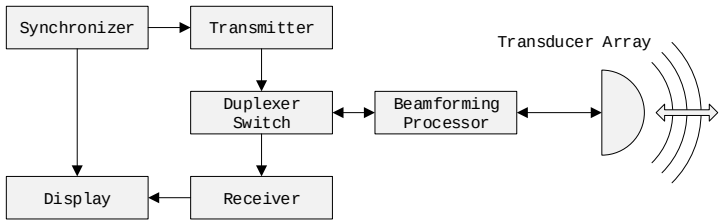


Figure 4.8: Active SONAR Block Diagram, After FAS Military Analysis Network (1998)

Analogous to active SONAR, in active fingerprinting an Internet Control Message Protocol (ICMP) Ping packet (ICMP Type 8, Postel (1981)) is sent to a target address to establish a "contact", a host that is alive on the network. The active fingerprinting utility will then

proceed to send crafted probes to generate a "signature", a fingerprint in OS fingerprinting parlance. In Figure 4.9, the block diagram of `xprobe2` (Yarochkin *et al.*, 2009) is shown. Active fingerprinting utilities can select what modules to activate and depending on the modules selected, port scanning can be used to enhance the fingerprint of the target device.

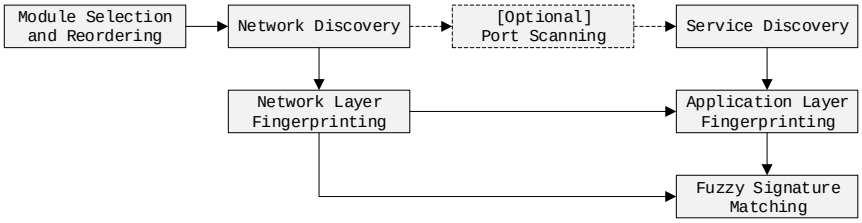


Figure 4.9: `xprobe2` Active Fingerprinting Block Diagram, After Yarochkin *et al.* (2009)

4.3.3 Passive Stalking

In passive SONAR an array of sensors is deployed to capture and analyse ambient "noise" surrounding the array. The captured noise is then plotted on a time versus frequency graph. The passive SONAR array, shown in Figure 4.10, can be configured to listen to a specific bearing using direction finding arrays, enabling the SONAR signal processing systems and operators to clean up the received signal. A cleaned signal can then be analysed and matched to a set of known fingerprints for seafaring vessels.

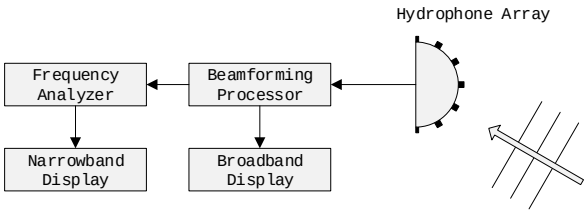


Figure 4.10: Passive SONAR Block Diagram, After FAS Military Analysis Network (1998)

Passive OS fingerprinting utilises similar methods to create a fingerprint of networked hosts. By intercepting legitimate communications to a target, a passive fingerprinting utility can extract relevant features from the network traffic to generate a fingerprint and create a possible match. Figure 4.11 shows the architecture used by Medeiros *et al.* (2010) to enhance the matched OS from a fingerprint generated using a passive fingerprint utility by adding a database of known fingerprints that can be queried.

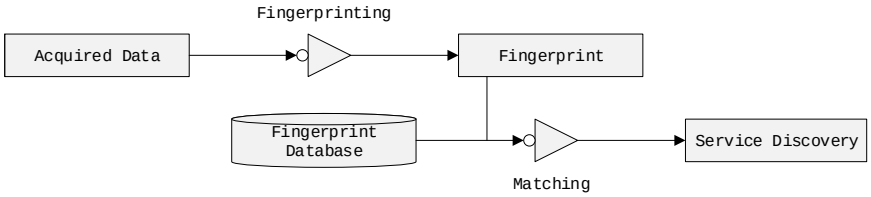


Figure 4.11: Passive Fingerprinting Block Diagram, After Medeiros *et al.* (2010)

4.3.4 Fingerprint Databases

The operation and functioning of SONAR is governed by the laws of physics. Any physical objects must respond to acoustic stimuli according to the laws of physics and the acoustic signature of a ship is most definitely different to that of a whale. These known responses enable SONAR systems to create a database that assists in fishing out metal objects in water.

In OS fingerprinting, the implementation of network stacks and networked services is governed by the "Laws of the Internet". Each implementation of a protocol RFC may differ due to either ambiguity in the description of a protocol or interpretation of semantics (Song *et al.*, 2014). These interpretations of protocol RFCs (Paxson *et al.*, 1999) are used to create databases of known artefacts that can then be used to identify specific operating system versions or families of operating systems.

SONAR systems build up a database of known acoustic signatures for seafaring vessels to assist SONAR operators in creating a match for a contact. Similarly, OS fingerprinting utilities builds up databases of features unique to specific OS families and OS versions. These databases enable an attacker to identify a remote operating system.

4.3.5 Abstracted Host Fidelity

To define the fidelity of an abstracted host, whether virtualised, containerised or simulated, we can utilise the methodologies of SONAR. In SONAR, a class of vessel will have an acoustic signature and any vessel in that class will match the signature with accuracy even though each vessel will have minor deviations. For OS fingerprinting, we can state that any family (class) of OS will have a set of features in common, and each implementation or distribution will have some form of variance on this signature. A high fidelity abstracted host will replicate the functioning of the host to such a degree that the

base host and the abstracted host are almost indistinguishable (for a given measurement technique), while a low fidelity host will implement a partial set of features that replicates the behaviour of the base host.

From this analogy we can define Remote Fidelity of an Abstracted Networked Host as follows. The remote fidelity of an abstracted networked host is the level of accuracy at which the abstracted host is able to replicate the behaviour and characteristic deviations of network traffic generated by the real host.

4.4 Summary

In Chapter 3 it was established that CBNEs are constructed using technologies that could influence fingerprints, confirming one part of the research hypothesis.

In Section **4.1** studies of fingerprinting remote hosts were analysed to identify techniques used and how a fingerprint would be generated for a particular OS. It was discovered that fingerprinting utilities have the ability to extract features from network traffic that correlate to different types of OSs, and features that relate to networking environments.

In Section **4.2** four classes of network attack model were reviewed to gain a better understanding of how a "typical" hacker would operate. OS fingerprinting is predominantly used in scenarios where an attacker targets a specific system; in scenarios where gaining any form of access is desired OS fingerprinting is used to a lesser extent.

In Section **4.3** the similarities between SONAR and OS fingerprinting were used to create a model for measuring the fidelity of an abstracted machine as seen from the perspective of a remote attacker.

Knowing that CBNEs can influence fingerprints and that fingerprints are sensitive to changes in the networking environment and OS subsystems, a model was created to assist in the measurement of the fidelity of an abstracted host as seen from the point of view of a remote attacker. In Chapter 5 this model is used as the basis for constructing an experimental methodology to test the hypothesis. By applying this model to OS fingerprinting of abstracted hosts, the extent to which such hosts will be able to "fool" a remote attacker can be assessed.

Chapter 5

Experimentation and Results

The single biggest problem in communication
is the illusion that it has taken place.

George Bernard Shaw

The primary goal of this study was to evaluate if Container-Based Network Emulators (CBNEs) (Chapter 3) can be used to construct experimental networks for information security research, education, and training. From the perspective of a remote attacker conducting Operating System (OS) fingerprinting and service enumeration (Section 4.2) during target selection these fingerprints are crucial. The components used by CBNEs to construct emulated networks can modify network traffic (Section 4.3.1) and influence the features extracted by OS fingerprinting utilities. In this chapter fingerprints are generated and analysed for a selection of Linux-based CBNEs to establish the fidelity of emulated hosts according to the definition provided in Section 4.3.

This chapter starts off with an overview of components in a computer network that can influence OS fingerprints generated by utilities such as `nmap`. Section **5.1** presents the network and host-based components that can influence OS fingerprints and discusses how these components can manipulate network traffic and thus cause fingerprinting utilities to report false results.

Section **5.2** details the software environment, test network design, and process used for testing. The CBNEs subjected to OS fingerprinting, and the design of the test network are defined. The list of standard Linux utilities, and the suite of OS fingerprinting utilities used for testing are discussed.

The first set of results, the Linux kernel versions reported, is presented in Section **5.3**. OS fingerprinting utilities often report human readable results. In this section the human readable OSs reported are compared to the known OS version of the host to assess the accuracy to which OS fingerprinting utilities can detect OSs. An attacker might rely on the human readable OS reported and use this information during the target selection process. Incorrect results could lead to failed attempts at penetrating a remote host.

Section **5.4** presents incidental findings on the effects that CBNE components had on `ping` Round Trip Times (RTTs). Each of the combinations of node and network link emulation technologies can influence processing time on network packets. This section presents a cursory statistical analysis of the effects that combinations of components used by CBNEs have on latency of packets in an emulated network.

The following two sections present an analysis of raw fingerprints generated during testing. Section **5.5** presents the results obtained from passive fingerprinting and Section **5.6** presents the results obtained from active fingerprinting. By analysing raw fingerprints, the different influences that CBNEs have on features extracted from network traffic can be assessed and the fidelity of hosts in an emulated network can be determined.

In Section **5.7** the cause of a fingerprint deviation reported in Sections 5.5 and 5.6 is investigated. The active and passive fingerprints generated for one CBNE deviated from the fingerprints generated for the host and all other CBNEs tested and the cause of the deviation was corrected and the CBNEs was subjected to an additional round of tests. This chapter concludes with a summary of the findings, presented in Section **5.8**.

5.1 Network and Host Influences on Fingerprinting

Fingerprinting utilities are sensitive to changes in the L2 and L3 headers of network traffic. These sections are used to extract features that are used by fingerprinting utilities to identify the OS of targeted systems. Modifications to L2 and L3 headers can be introduced by networking devices such as NAT devices, firewalls, and routing equipment. Within the context of this research, the design of the experimental network and the configuration of the emulated hosts must take these influences into account. The experimental design must therefore exclude devices that could bring unwanted influences in network, traffic as the focus of the study is the measurement of the fidelity of emulated hosts themselves and not how such devices can influence fidelity.

5.1.1 Network Based Influences

As a network packet traverses the network from the attacker to the target system, the packet will pass through many network devices that can and will modify the packet. Each device in a network can lead to loss of information during the packet's traversal of the network. The simplest example is that of a routed network, where a router will strip away its own Media Access Control (MAC) address and replace it with the MAC address of the next "hop" and decrease the Time To Live (TTL) of certain protocols by one. In Figure 5.1, an example routed network is shown. As a packet traverses the router the source and destination MAC addresses are changed to those of the next segment in the network and the TTL is decreased by one. Listing 5.1a shows the contents of a Transmission Control Protocol (TCP) SYN packet before being routed, and Listing 5.1b shows the same packet after being routed. The changes in the packet are highlighted as follows: red indicates the destination MAC, the first bold section indicates the source MAC, the second bold section indicates the TTL, and the last bolded section indicates the checksum of the packet. In a routed network, the first level of loss of information affects vendor-specific details regarding the Network Interface Controller (NIC) of the target host, as this information will not reach the attacker and is network local only. Listing 5.2 shows a TCP/SYN packet exiting a node in a local switched network and entering the target node. As is shown, there are no differences in these packets. Local switched networks apply no modifications to switched traffic.

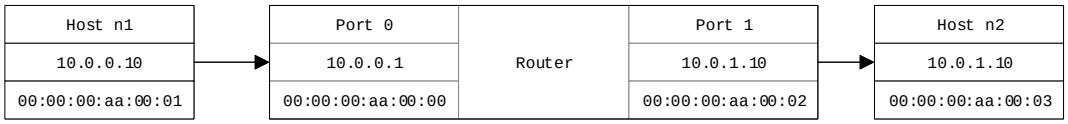


Figure 5.1: Basic Routed Network

Routing devices are not the only components in a computer network that will modify packets as they traverse a network. A corporate network being scanned may utilise a Demilitarised Zone (DMZ) that incorporates perimeter security devices such as Next

Listing 5.1: SYN Packet Pre & Post Router

(a) Pre

```
00 00 00 aa 00 00 00 00 00 aa 00 01 08 00 45 00
00 3c d0 16 40 00 40 06 55 92 0a 00 00 0a 0a 00
01 0a aa ec 1f 40 34 b4 d2 16 00 00 00 00 a0 02
fa f0 30 6a 00 00 02 04 05 b4 04 02 08 0a 38 5e
fe 3b 00 00 00 00 01 03 03 07
```

(b) Post

```
00 00 00 aa 00 03 00 00 00 aa 00 02 08 00 45 00
00 3c d0 16 40 00 3f 06 56 92 0a 00 00 0a 0a 00
01 0a aa ec 1f 40 34 b4 d2 16 00 00 00 00 a0 02
fa f0 30 6a 00 00 02 04 05 b4 04 02 08 0a 38 5e
fe 3b 00 00 00 00 01 03 03 07
```

Listing 5.2: SYN Packet Pre & Post Switch

(a) Pre

```
00 00 00 aa 00 01 00 00 00 aa 00 00 08 00 45 00
00 3c 05 8b 40 00 40 06 21 1d 0a 00 00 0a 0a 00
00 0b 93 ce 1f 40 e7 8d 34 99 00 00 00 00 a0 02
72 10 e0 4f 00 00 02 04 05 b4 04 02 08 0a e9 3d
29 18 00 00 00 00 01 03 03 07
```

(b) Post

```
00 00 00 aa 00 01 00 00 00 aa 00 00 08 00 45 00
00 3c 05 8b 40 00 40 06 21 1d 0a 00 00 0a 0a 00
00 0b 93 ce 1f 40 e7 8d 34 99 00 00 00 00 a0 02
72 10 e0 4f 00 00 02 04 05 b4 04 02 08 0a e9 3d
29 18 00 00 00 00 01 03 03 07
```

Generation Firewalls (NGFWs), Intrusion Detection and Prevention Systems (IDPSs), and Web Application Firewalls. Each of these devices can be configured to detect, analyse and respond to OS fingerprinting utilities to a varying degree. These devices may simply terminate connections or reroute OS fingerprinting probes to honeypot systems to lure the attacker into a false sense of accomplishment. In the latter, an attacker will expose themselves and gather information on systems designed to track and report on attacker activity. Certain honeypots are designed to give responses to known scanning probes that replicate the features of known vulnerable OSs and services. A conceptual DMZ and the paths that network traffic can take through such a configuration, are shown in Figure 5.2.

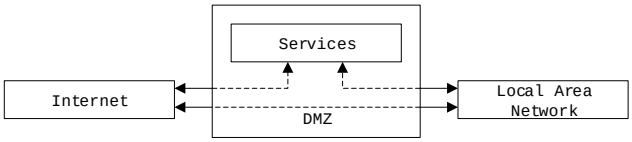


Figure 5.2: Corporate Network with a DMZ

When attempting to fingerprint the OS and enumerate services of web hosts in a DMZ (*Internet* to *Services* path) OS fingerprinting utilities may return strange results. The host OS could be reported as FreeBSD, while the services on the host may be reported as being Microsoft IIS. In these situations, one possible conclusion that can be made is that the host is acting as a load balancer or reverse proxy. Fingerprinting targeted at any of the services will return results that contain features of both the load balancer or proxy (L2 and L3) and of the service itself (L7).

Thus for the purposes of this research, the experimental network must exclude any devices that are known to manipulate network traffic. Any modifications to network traffic by such devices can influence the measurement of the fidelity of emulated hosts.

5.1.2 Host Based Influences

Once a connection is established to a remote host that host has several opportunities to manipulate the headers of response packets. Figure 5.3 shows a simplified sequence diagram of a connection being established to a web server that serves active content. As individual packets that form part of the connection makes their way back to the requester each component in the stack can manipulate response packets. In addition to manipulating outgoing packets each component can terminate the connection.

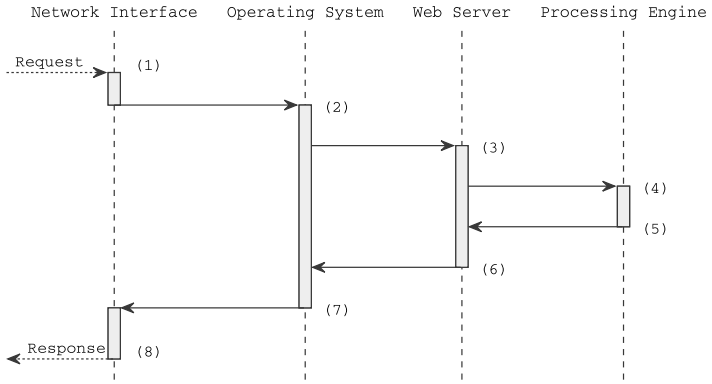


Figure 5.3: Packet Entering A Computer System

Each of the eight steps shown in Figure 5.3 represent unique opportunities to manipulate responses or to terminate connections. These steps are divided into two groups: incoming (1-4) and outgoing (5-8). The opportunities for each step to reject or manipulate packets are discussed below:

1. Incoming - *Network Interface* (L2)
   Each packet arriving at a host has to be validated by verifying its checksum. Certain NICs can offload L2 and L3 processing and verification. If an OS fingerprinting utility uses probes that violate Request for Comments (RFCs), these NICs can terminate the connection before any packets are passed to the OS.

2. Incoming - *Operating System* (L3)
   Before the payload of a connection is passed to the intended application, the network stack of the OS can apply check and filters. Packets intended for a different host, suspicious protocol versions (such as packets having an IP version of 5), or connection attempts to closed ports can be rejected. The rejection strategy of an OS can be to send a TCP reset packet or to drop the connection with no response.

3. Incoming - *Web Server* (L7)
   Once a successful connection is established the OS passes the payload of the connection to the Web Server. Common ways in which a Web Server can manipulate connections include rewriting URIs and proxying connection to a next host. The Web Server can scan payloads for malicious content and reject connections if such content is found.

4. Incoming - *Processing Engine* (Payload)
   Once the payload of a connection researches the Processing Engine the request is interpreted and a response is generated. A Processing Engine could apply additional validation on the request and reject it if validation fails.

5. Outgoing - *Processing Engine* (Payload)
   The return path to the originator of a request starts with the Processing Engine packaging a response. Responses from a Processing Engine might include details, such as the version of the engine used. The Web Server can also rewrite sections of the response prepared by the Processing Engine.

6. Outgoing - *Web Server* (L7)
   The response from the Processing Engine is packaged into L7 traffic by the Web Server. L7 (HTTP(S)) data includes various headers that help the requester process responses and maintain the connection. The Web Server could leak information such as the host OS, the Web Server version, and the version of the Processing Engine.

7. Outgoing - *Operating System* (L3)
   The primary opportunity for an OS to manipulate packets is when a response is prepared for transmission back to the requester. The default configuration and optimisation of the network stack of the Operating System can be manipulated to appear similar to a different OS.

8. Outgoing - *Network Interface* (L2)
   Before transmitting a packet over physical or virtual infrastructure, the checksum for the packet is calculated and inserted into the appropriate header. A miscalculation of the checksum could result in networking equipment rejecting the packet.

The example above illustrates a very simplified view of network traffic entering a host for processing. The *Operating System* can assert control over both L2 and L3 headers in packets and can rewrite any content in the packet. The Linux kernel can apply any modification to network traffic through the use of the hooks provided by `iptables`. Figure

5.4 shows the paths that a packet can take through the *iptables* system. In the example, each component has an opportunity to misinterpret a request or introduce anomalies into network traffic, be it by design or coding error. Each of these components can influence the remote fidelity of the networked host, compared to that of a real host. Many of these anomalies are regarded as indicative of a particular OS or service, though each component may introduce anomalies that will reduce the accuracy of the signature generated by OS fingerprinting.

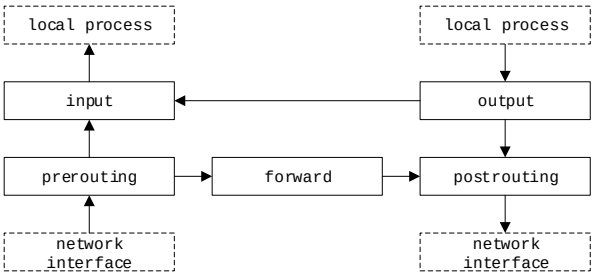


Figure 5.4: Hook points for `iptables`, Adapted from Purdy (2004)

The L3 (*Operating System*) and L7 (*Web Server*) components in the example have additional opportunities for intentional interference with network traffic. At L3, systems such as `iptables`, `UFW`, and `fail2ban` can modify (honeypot), disallow (Firewall) or re-route (port forward) network traffic based on pre-defined rule sets. These utilities utilise the hooks shown in Figure 5.4 and use the abilities of each of these components to intercept and manipulate network traffic. At OSI Layer 7 (L7), the service may impose additional scrutiny on traffic through the use of plugins; Apache in particular utilises `mod_evasive` and `mod_security2` to defeat brute force attacks and reject suspicious network traffic such as application enumeration. Common applications of OSI Layer 3 (L3) and L7 traffic manipulation techniques are found in load balancers applying Destination Network Address Translation (DNAT) at L3 and reverse proxies rerouting request at L7.

Generating a fingerprint of a remote host can be complicated by utilities designed to defeat OS fingerprinting. Utilities such as `ip personalities` (Roualland and Saffroy, 2002), `morph` (Wang, 2004), and SNOS (Huber, 2011) are designed to manipulate the features in network traffic that OS fingerprinting utilities rely on to generate a fingerprint. Stopforth (2007) and Kaur (2009) investigated techniques to counter the fingerprinting techniques as part of their studies.

The nodes instantiated in the experimental networks will be subjected to fingerprinting

to assess remote fidelity. To gather responses from protocols such as TCP and User Datagram Protocol (UDP) that are free of unwanted manipulation, minimalistic services will be required on well-known ports.

5.2 Testing Environment

The objective of the evaluation methodology was to subject nodes in emulated networks to active and passive fingerprinting. The fingerprints generated would then be used to establish the remote fidelity of nodes in such networks according to the definition given in Section 4.3. The testing procedure was designed to emulate the "reconnaissance" phase of an attack targeting a specific host (Section 4.2).

5.2.1 Test Platforms

The host platform for testing was Ubuntu Linux 19.04 AMD64 (Disco Dingo) with the current supported version of the Linux kernel (Torvalds, 2019, v5.0.0) installed. A laptop running the same version of Ubuntu Linux with the same kernel version was used as the attacker node during fingerprinting of the host. Of the six CBNE families evaluated in Section 3.3, only four were evaluated. The versions of the evaluated CBNEs are listed in Table 5.1. Kathará (Section 3.3.6) was excluded from testing as it uses Docker and Open vSwitch, the same components as used by IMUNES (Section 3.3.3). Marionnet (Section 3.3.2) was excluded from testing as it uses User Mode Linux (UML), a hardware abstraction layer virtualisation system.

Table 5.1: Systems Under Test

Name	Version
CORE	5.3.1
IMUNES	2.3.0
MiniNet	2.2.0
VNX	2.0
LXC	3.0.3

The CBNEs included in the evaluation use a mix of components for node and network emulation, and are shown in Table 5.2. The evaluated CBNEs (CORE, MiniNet, IMUNES, and VNX) make use of Linux namespaces, Docker, and Linux Containers (LXC) for node

emulation (Section 3.5.1). Linux bridges and Open vSwitch are used for network emulation (Section 3.5.2). LXC was included as it uses Linux bridges for networking. The evaluated CBNEs were chosen for the mix between node and network emulation technologies.

Table 5.2: System Under Test Emulation Components

Name	Node Emulation	Network Device Emulation
CORE	Linux namespaces	Linux Bridges
IMUNES	Docker	Open vSwitch
MiniNet	Linux namespaces	Open vSwitch
VNX	LXC	Open vSwitch
LXC	LXC	Linux Bridges

5.2.2 Test Network

The design of the experimental network should consider the influences that network components have on network traffic features used by OS fingerprinting utilities. Any component added has an influence on the fingerprint generated by fingerprinting tools and care should be taken to avoid components that will have an adverse effect on the generated fingerprint.

The most simplistic design for a computer network is two computers connected using a "crossover" cable - this has the benefit of excluding any and all influences that a network device may have on generated fingerprints. Due to the construction of CBNEs, this design is not feasible and at least one network component has to be included for the purposes of this research. The most minimal feasible network design for evaluating the remote fidelity of CBNEs is two emulated hosts connected through a given CBNE's implementation of an L2 switch. This design is illustrated in Figure 5.5. The only network component that can have an influence on the generated fingerprints is the CBNE switch implementation. As was shown in Listing 5.2 in Section 5.1.1, an L2 switch is not expected to apply any modifications to network packets, and is thus expected to have no influence on generated fingerprints. The design of the test network is in contrast to the expected network architecture between a remote attacker and a target. This design was chosen to eliminate any influence outside of the CBNE systems evaluated as the focus of the study is not on the influence of devices external to CBNE systems.

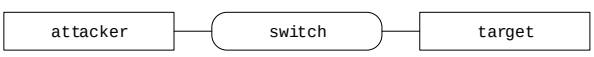


Figure 5.5: Experimental Network

5.2.3 Test Suite

The test suite for the experiments conducted included standard Unix utilities such as `uname` and `ping` as well as OS fingerprinting utilities. Listing 5.3 shows an example of `uname` output. The `uname` utility is used to display the version and architecture of the running kernel.

Listing 5.3: Example `uname` Command Output

```
$ uname -s -r -v
Linux 4.4.0-21-generic #37-Ubuntu SMP Mon Apr 18 18:33:37 UTC 2016
```

The `ping` utility was used to assess the differences in packet RTTs between the different CBNE implementations. An example of the output of the `ping` utility is shown in Listing 5.4. The reported RTT of each echo request is shown at the end of each output line. The reported statistics for all echo requests are shown at the bottom of the listing.

Listing 5.4: Round Trip Time and Statistics for the `ping` Utility

```
1  $ ping -4 -c 4 scanme.nmap.org
2  PING scanme.nmap.org (45.33.32.156) 56(84) bytes of data.
3  64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=1 ttl=41 time=305 ms
4  64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=2 ttl=41 time=304 ms
5  64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=3 ttl=41 time=304 ms
6  64 bytes from scanme.nmap.org (45.33.32.156): icmp_seq=4 ttl=41 time=304 ms
7
8  --- scanme.nmap.org ping statistics ---
9  4 packets transmitted, 4 received, 0% packet loss, time 3001ms
10 rtt min/avg/max/mdev = 304.281/304.710/305.096/0.625 ms
```

The active fingerprinting utilities (Section 4.1.1) used included `nmap`, `xprobe2`, and `SinFP3`. The passive fingerprinting utilities (Section 4.1.2) used included `p0f`, `ettercap`, and `SinFP3`. Table 5.3 list the version of the utilities used and the origin of the binary used. The latest version of `nmap` was not available in the Ubuntu package repositories at time of testing and was compiled from source. The `SinFP3` utility is not available through the Ubuntu package repositories and was compiled directly from the CPAN.

Table 5.3: Fingerprint Utility Versions

Utility	Type	Binary Origin	Version
nmap	active	source	7.80
xprobe2	active	repository	0.3
p0f	passive	repository	3.09b
ettercap	passive	repository	0.82
SinFP3	active/passive	source	1.24

OS fingerprinting utilities require databases to match the fingerprints generated for a target to a known OS. Table 5.4 shows the version of the fingerprinting utilities used and

the latest database for each utility. The databases for `xprobe2`, `p0f`, and `ettercap` are very old. A possible cause is that these tools are no longer used for fingerprinting and are currently being used for the non-fingerprinting capabilities they possess (Section 4.1).

Table 5.4: Fingerprinting Utility Database Dates

Tool	Version	Date
nmap	7.80	2018-09-27
xprobe2	0.3	2005-07-11
p0f	3.09b	2016-04-16
ettercap	0.82	2015-04-14
SinFP3	1.24	2018-07-21

5.2.4 Test Procedure and Scoring

The testing procedure for each of the CBNEs was as follows:

1. Each CBNE was installed on the host along with an accompanying filesystem for emulated nodes that contained the suite of test utilities.
2. From a clean boot, the test network was instantiated on a CBNE.
3. The first series of tests conducted were the ping latency tests. Static Address Resolution Protocol (ARP) entries were inserted for each of the emulated nodes within the network. This was done to prevent additional latency in ping RTTs due to an ARP request. An explanation of how an ARP request influences the RTT of a ping request is shown in Appendix A.
4. The second series of tests conducted were the active fingerprinting tests. The `ncat` utility was used to simulate TCP and UDP services running the target node and the active fingerprinting utilities were run from the attacker node. Active fingerprinting utilities were informed of the ports used by the simulated services to ensure that complete fingerprints can be generated.
5. The last series of tests conducted were the passive fingerprinting tests. The `ncat` utility was used to establish TCP and UDP connections from the attacker node to the target node. The `tcpdump` utility was used to capture the traffic for offline analysis.

6. Scoring the remote fidelity of emulated nodes was done by comparing the raw fingerprints generated by passive and active fingerprinting utilities to raw fingerprints generated for the host. For each component of a fingerprint for an emulated node where the fingerprint differed from the host, one was subtracted from the total possible score for that particular fingerprinting utility.

The commands used for active fingerprinting are shown in Table 5.5. The `TESTNAME` for each CBNE was the name of the CBNE and the `TARGETADDRESS` was the IPv4 address of the target node.

Table 5.5: Active Fingerprinting Commands

Tool	Command

nmap	`nmap -sS -sU -A -T4 -n -vv -dd \`
	`--version-all --osscan-guess --top-ports 200\`
	`-oA TESTNAME TARGETADDRESS`

xprobe2	`xprobe2 -v -F TARGETADDRESS > TESTNAME`

xprobe2 PortSpec	`xprobe2 -v -F \`
	`-p tcp:8000:open -p tcp:8001:closed \`
	`-p udp:11487:open -p udp:8001:closed \`
	`TARGETADDRESS > TESTNAME`

SinFP3	`sinfp3.pl -input-ipport -port 8000 -output-console \` `-active-3 -target TARGETADDRESS > TESTNAME`

For passive fingerprinting, the same `TESTNAME` and `TARGETADDRESS` were the same as those used for active fingerprinting. Both `p0f` and `SinFP3` only process TCP connections. The traffic captures produced for each CBNE had all traffic except a successful TCP connection filtered out. The traffic capture detailed in this example[1] was fed to each of the passive fingerprinting utilities using the commands shown in Table 5.6. For `ettercap`, the Graphical User Interface (GUI) version was used.

Table 5.6: Passive Fingerprinting Commands

Tool	Command
p0f	`p0f -r TESTNAME.conntest.filtered.pcap`
sinfp3.pl	`sinfp3.pl -mode-passive -search-passive -input-pcap -output-console -target TARGETADDRESS -pcap-file TESTNAME.conntest.filtered.pcap`

[1] `TESTNAME.conntest.filtered.pcap`

5.3 Reported Kernel Versions

Kernel version analysis for this research was done to establish the accuracy which each of the fingerprinting tools reported the kernel version of emulated hosts. For each CBNE, the `uname` command was run on the target emulated host to retrieve the kernel version. This was done to confirm that the kernel version reported by CBNE emulated hosts were the same as the host machine. The kernel version of each CBNE was then compared to the kernel version reported by each of the passive and active fingerprinting utilities to assess the ability of fingerprinting utilities to detect kernel versions. Table 5.7 lists the kernel versions as reported by the `uname` utility. As expected, due to the design of containers, all containerised hosts reported the same kernel version as the host OS.

Table 5.7: CBNE Kernel Version

CBNE	Linux Kernel Version
CBNE Host	5.0.0-29-generic
CORE	5.0.0-29-generic
IMUNES	5.0.0-29-generic
MiniNet	5.0.0-29-generic
VNX	5.0.0-29-generic
LXC	5.0.0-29-generic

As discussed in Section 4.1, the fingerprinting of emulated hosts was conducted using two different methodologies. Passive fingerprinting was conducted offline using traffic captured during testing. Passive fingerprinting was conducted using `p0f` v3.09b, `ettercap` v0.82, and `SinFP` v1.24. Active fingerprinting was conducted on emulated hosts using `nmap` v7.80, `xprobe2` v0.3, and `SinFP` v.1.24.

5.3.1 Passive Kernel Version Results

The passive fingerprinting tests conducted delivered poor results. The `p0f` and `ettercap` utilities did not resolve any information regarding the kernel of the target node. The fingerprint databases of these two utilities are very old (Table 5.4) and were not expected to contain signatures for the latest releases of the Linux kernel. The `SinFP` utility resolved the OS of the target node as an unknown version of the Android mobile OS (Table 5.8) for all CBNEs except for MiniNet, for which no match could be found. The `SinFP` passive fingerprint database contains only 49 entries. The latest version of the Linux kernel for which passive fingerprints have been generated is the Linux 3.2.0 kernel.

Table 5.8: SinFP3 Passive Reported Operating Systems

Platform	Reported OS Version	Score
Host	Android (Unknown Version)	94%
CORE	Android (Unknown Version)	94%
IMUNES	Android (Unknown Version)	94%
MiniNet	Unknown	—
VNX	Android (Unknown Version)	94%
LXC	Android (Unknown Version)	94%

5.3.2 Active Kernel Version Results

Active fingerprinting against CBNE emulated nodes was conducted using `nmap`, `xprobe2`, and SinFP. The commands used for these utilities are shown in Table 5.5. The `nmap` utility delivered consistent results across all CBNEs (Table 5.9) but identified the target nodes as being part of the Linux 2.6 or Linux 3.2 - 4.9 series kernels with equal probability.

Table 5.9: `nmap` Reported Operating Systems

Platform	Reported OS Version	
Host	Linux 2.6.32 (96%)	Linux 3.2 - 4.9 (96%)
CORE	Linux 2.6.32 (96%)	Linux 3.2 - 4.9 (96%)
IMUNES	Linux 2.6.32 (96%)	Linux 3.2 - 4.9 (96%)
MiniNet	Linux 2.6.32 (96%)	Linux 3.2 - 4.9 (96%)
VNX	Linux 2.6.32 (96%)	Linux 3.2 - 4.9 (96%)
LXC	Linux 2.6.32 (96%)	Linux 3.2 - 4.9 (96%)

The `xprobe2` utility was run in two different configurations. The first configuration (Table 5.5, `xprobe2`) was run without specifying known ports, whereas the second configuration (Table 5.5, `xprobe2 PortSpec`) was run with known port statuses for both open and closed states for both TCP and UDP. For both the configurations OS versions were reported with varying degrees of confidence. The readable output from the tool was indecipherable as it appeared to print out raw binary data to the command line. A sample of this output is shown in Figure 5.6. In preliminary testing conducted on an earlier version of Ubuntu (17.04 AMD64), the `xprobe2` utility did show readable output for the OS guess.

```
[+] Signature looks like:
[+]  gV
       J (94%)
```

Figure 5.6: `xprobe2` Output on Ubuntu 19.04 AMD64

Active fingerprinting conducted using the `SinFP` utility matched all emulated nodes against older versions of the Linux kernel. Emulated nodes for all the CBNEs except

MiniNet were reported as running the Linux 2.6.22 kernel with a confidence score of 79. `SinFP` matched the target node for MiniNet against Linux kernel versions 2.4.18 through 3.2.0 with equal scores of 73(†). The latest version of the Linux kernel in the active fingerprint database for `SinFP` is the Linux 3.2.0 kernel.

Table 5.10: SinFP3 Active Reported Operating Systems

Platform	Reported OS Version	Score
Host	Linux: 2.6.x (2.6.22)	79
CORE	Linux: 2.6.x (2.6.22)	79
IMUNES	Linux: 2.6.x (2.6.22)	79
MiniNet	Linux: 3.2.x (3.2.0)	73†
VNX	Linux: 2.6.x (2.6.22)	79
LXC	Linux: 2.6.x (2.6.22)	79

5.3.3 Kernel Version Findings

In the kernel version tests, none of the fingerprinting tools were able to report the exact kernel version of the tested platforms. Active fingerprinting tools fared better at identifying the OS family than passive fingerprinting tools. In Table 5.4 (repeated below), the last update date for each of the fingerprinting tool's databases are shown. From this it can be seen that the entries in the database of a fingerprinting tool are of crucial value, regardless of the accuracy of a fingerprint. If the database that is used by a fingerprinting utility is outdated, the reported kernel version cannot reflect the actual kernel version for an OS released after the database date. The Linux kernel used in testing (v5.0.0) was released in March of 2019 and the databases of all the fingerprinting utilities used pre-dates the release of this kernel. The most recent version of the Linux kernel reported by the fingerprinting utilities is v4.9, released December 2016 (Torvals, 2016), which predates the databases of both `nmap` and SinFP. There is no reasonable expectation that the fingerprinting utilities will include fingerprints for the kernel version used. This anomaly will require investigation into patches applied[2] to the original source for changes that could affect printing of OS results to a terminal.

5.4 Ping Latency Results

To assess the influence that the different components used in the construction of CBNEs have on packet latency the `ping` utility was used to test the timings of Internet

[2] `https://launchpad.net/ubuntu/+source/xprobe/+changelog`

Control Message Protocol (ICMP) packets between emulated hosts. The `ping` utility reports statistics such as minimum RTT (min), average RTT (avg), maximum RTT (max), and standard deviation (mdev) for all ping requests sent (ICMP Type 8) and responses received (ICMP Type 0) during one session. Listing 5.5 show an example of the data extracted from the output of the `ping` utility. The statistics reported by the ping utility was extracted and used for initial assessment of the timings for CBNEs in Section 5.4.1. The timing of each response were extracted and used for comparisons between the CBNEs and the host in Section 5.4.2.

Listing 5.5: Round Trip Time and Statistics for the `ping` Utility - Extended

```
1  $ ping -c 100 10.0.1.10
2  PING 10.0.1.10 (10.0.1.10) 56(84) bytes of data.
3  64 bytes from 10.0.1.10: icmp_seq=1 ttl=63 time=0.115 ms
4  64 bytes from 10.0.1.10: icmp_seq=2 ttl=63 time=0.093 ms
5  64 bytes from 10.0.1.10: icmp_seq=3 ttl=63 time=0.098 ms
6  64 bytes from 10.0.1.10: icmp_seq=4 ttl=63 time=0.083 ms
7  .
8  .
9  .
10 .
11 64 bytes from 10.0.1.10: icmp_seq=97 ttl=63 time=0.072 ms
12 64 bytes from 10.0.1.10: icmp_seq=98 ttl=63 time=0.044 ms
13 64 bytes from 10.0.1.10: icmp_seq=99 ttl=63 time=0.098 ms
14 64 bytes from 10.0.1.10: icmp_seq=100 ttl=63 time=0.103 ms
15
16 --- 10.0.1.10 ping statistics ---
17 100 packets transmitted, 100 received, 0% packet loss, time 433ms
18 rtt min/avg/max/mdev = 0.044/0.085/0.127/0.016 ms
```

5.4.1 Ping Statistics

Immediately after emulated nodes booted for each CBNEs, two pings with a count 4 were run in quick succession to gather data on the ARP resolution timings for each CBNE. The statistics reported by the `ping` utility for these tests are shown in Tables 5.11a and 5.11b. As can be seen from the max column of the two tables, the maximum ping RTT was significantly higher in the first run for all CBNEs. The first time that a node establishes a network connection in a switched Local Area Network (LAN) the node has to request the MAC addresses that corresponds to an IP address of the remote node. The effect that an ARP resolution has on ping timings is discussed in Appendix A. Directly after the second 4 count ping test, static ARP entries were created for both the attacker and target nodes in the emulated networks to prevent further ARP queries from influencing the next series of ping tests, where a ping count of 100 was used.

Table 5.11: Ping Statistics, Initial

(a) Run 1

	Time (ms)			
Platform	min	avg	max	mdev
Host	0.360	0.421	0.572	0.090
CORE	0.070	0.097	0.162	0.039
IMUNES	0.063	0.284	0.943	0.380
MiniNet	0.067	0.298	0.988	0.398
VNX	0.055	0.283	0.950	0.385
LXC	0.070	0.085	0.122	0.023

Ping count = 4

(b) Run 2

	Time (ms)			
Platform	min	avg	max	mdev
Host	0.267	0.299	0.360	0.042
CORE	0.067	0.073	0.089	0.013
IMUNES	0.065	0.189	0.528	0.195
MiniNet	0.031	0.180	0.562	0.221
VNX	0.054	0.062	0.074	0.010
LXC	0.070	0.073	0.077	0.008

Ping count = 4

With static ARP entries created for each of the emulated nodes, an additional set of ping tests were conducted, with a total count of 100 pings. These tests were conducted to confirm that the ARP entries had the desired effect. The results from these tests are shown in Tables 5.12a and 5.12b. The min and max results for all CBNEs for both runs were within close range of the results obtained for the second 4 count ping test.

Table 5.12: Ping Statistics, Confirmation

(a) Run 1

	Time (ms)			
Platform	min	avg	max	mdev
Host	0.245	0.350	0.398	0.041
CORE	0.050	0.067	0.090	0.011
IMUNES	0.032	0.072	0.535	0.048
MiniNet	0.025	0.062	0.593	0.057
VNX	0.039	0.070	0.633	0.058
LXC	0.028	0.072	0.094	0.012

Ping count = 100

(b) Run 2

	Time (ms)			
Platform	min	avg	max	mdev
Host	0.238	0.352	0.397	0.031
CORE	0.028	0.065	0.089	0.013
IMUNES	0.029	0.069	0.547	0.049
MiniNet	0.050	0.061	0.472	0.043
VNX	0.029	0.071	0.585	0.053
LXC	0.027	0.059	0.084	0.011

Ping count = 100

The max result for Integrated Multiprotocol Network Emulator/Simulator (IMUNES), MiniNet, and Virtual Networks over Linux (VNX) remained significantly higher than the avg result for these CBNEs. Initially the cause was thought to be incorrect or failed static ARP entries and that an ARP resolution caused the delays. A traffic capture from one of the tests showed that no ARP resolution occurred during the `N=100` tests. No additional investigations were conducted into the cause of these anomalies, though it was noted that all three CBNEs utilise Open vSwitch (Table 5.2) to construct emulated network components. The CBNE that utilises Linux bridges, Common Open Research Emulator (CORE), and LXC itself had *max* results much lower than those of the CBNEs that use Open vSwitch.

5.4.2 Ping Latency Distribution

As an extension to the 100 count ping tests, extended ping tests with a ping count of 1000 were conducted. The 1000 count was arbitrarily chosen to ensure a sufficient sample size for exploratory statistical data analysis (Turkey, 1977) of ping RTTs. Using the same methodology as the tests conducted in Section 5.4.1, the `ping` utility was used to send 1000 pings from the target to the attacker node in two rounds. The output from the commands was parsed to extract the reported statistics and the RTTs of the individual pings. The statistics reported by the `ping` utility for the two runs are shown in Tables 5.13a and 5.13b. The min and max RTTs for each CBNE for both runs corresponded to the statistics reported for the 100 count ping tests.

Table 5.13: Ping Statistics

(a) Run 1

	Time (ms)			
Platform	min	avg	max	mdev
Host	0.235	0.347	0.405	0.040
CORE	0.025	0.068	0.101	0.010
IMUNES	0.027	0.060	0.506	0.015
MiniNet	0.020	0.060	0.506	0.015
VNX	0.029	0.056	0.549	0.017
LXC	0.018	0.057	0.096	0.007

Ping count = 1000

(b) Run 2

	Time (ms)			
Platform	min	avg	max	mdev
Host	0.228	0.346	0.405	0.037
CORE	0.025	0.067	0.116	0.013
IMUNES	0.022	0.060	0.586	0.018
MiniNet	0.025	0.066	0.530	0.017
VNX	0.016	0.057	0.502	0.018
LXC	0.018	0.057	0.096	0.007

Ping count = 1000

The statistics shown in Tables 5.13a and 5.13b detail the expected behaviour of the tested systems within switched networks with no other traffic. The reported statistics do not enable one to establish if similarities exist between the latency of the CBNEs. The individual timings extracted from the output of the `ping` utility were then analysed for the distribution of the packets. The quartiles for the two runs can be seen in Tables 5.14a and 5.14a. For all the tested CBNEs, the first and second quartiles differ marginally and in some cases these values are the same. The same can be seen for the second and third quartiles.

For the rest of this section only graphs for the first run will be shown for readability. The complete set of graphs for the first and second runs can be found in Appendix B. The closeness of the quartiles for the CBNEs indicates that half of all RTTs will be placed within these incredibly tight bounds. Figure 5.7 visualises the tight clustering of RTTs as box plots, with outliers omitted. The box plot for the host is based on the left-hand y-axis and the box plot for the CBNEs are based on the right-hand y-axis. The box plot illustrates that though there is a very tight clustering of the RTTs around the median (second quartile), there are large tails at the lower and higher ends for all the CBNEs.

Table 5.14: Ping Quartiles

(a) Run 1

Platform	Min	Quartile 1st	Quartile 2nd	Quartile 3rd	Max
Host	0.235	0.322	0.361	0.369	0.405
CORE	0.025	0.068	0.070	0.071	0.101
IMUNES	0.027	0.057	0.058	0.061	0.506
MiniNet	0.02	0.065	0.066	0.068	0.294
VNX	0.029	0.054	0.054	0.055	0.549
LXC	0.018	0.055	0.055	0.057	0.096

Ping count = 1000, time in ms

(b) Run 2

Platform	Min	Quartile 1st	Quartile 2nd	Quartile 3rd	Max
Host	0.228	0.349	0.358	0.364	0.405
CORE	0.025	0.068	0.069	0.071	0.116
IMUNES	0.022	0.057	0.058	0.061	0.586
MiniNet	0.025	0.065	0.067	0.068	0.53
VNX	0.016	0.054	0.054	0.056	0.502
LXC	0.018	0.054	0.055	0.057	0.096

Ping count = 1000, time in ms

To illustrate this distribution of data across the full range of RTTs, histograms, were used. Like the box plots, the histograms (Figure 5.8) show that RTTs are concentrated at the median for all CBNEs. An immediate observation that can be made from the histograms and the box plots is that the spread of RTTs of the host significantly differs to those of the CBNEs. The same difference can be observed for the second run (Appendix B).

The aim of this study was to compare CBNE emulated nodes to the host. To test for similarities in the distribution of RTTs between the host and the CBNEs Q-Q plots (Wilk and Gnanadesikan, 1968) can be used.

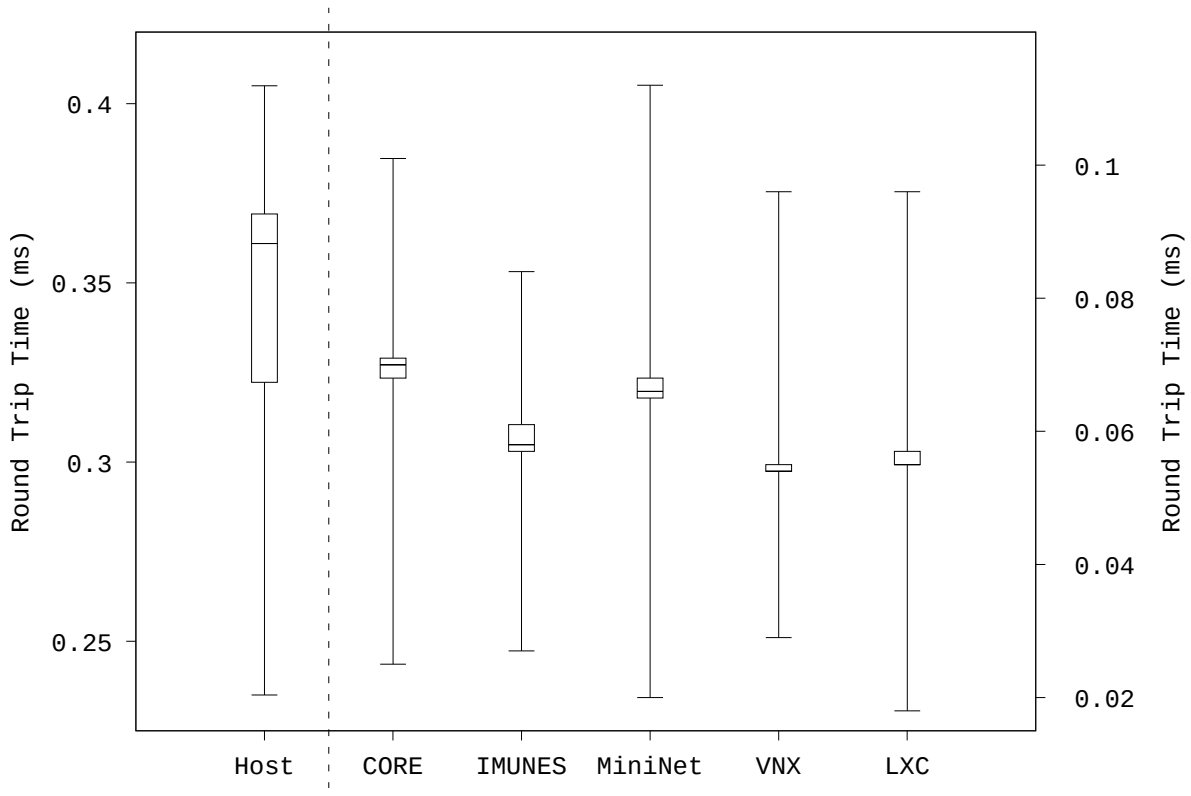
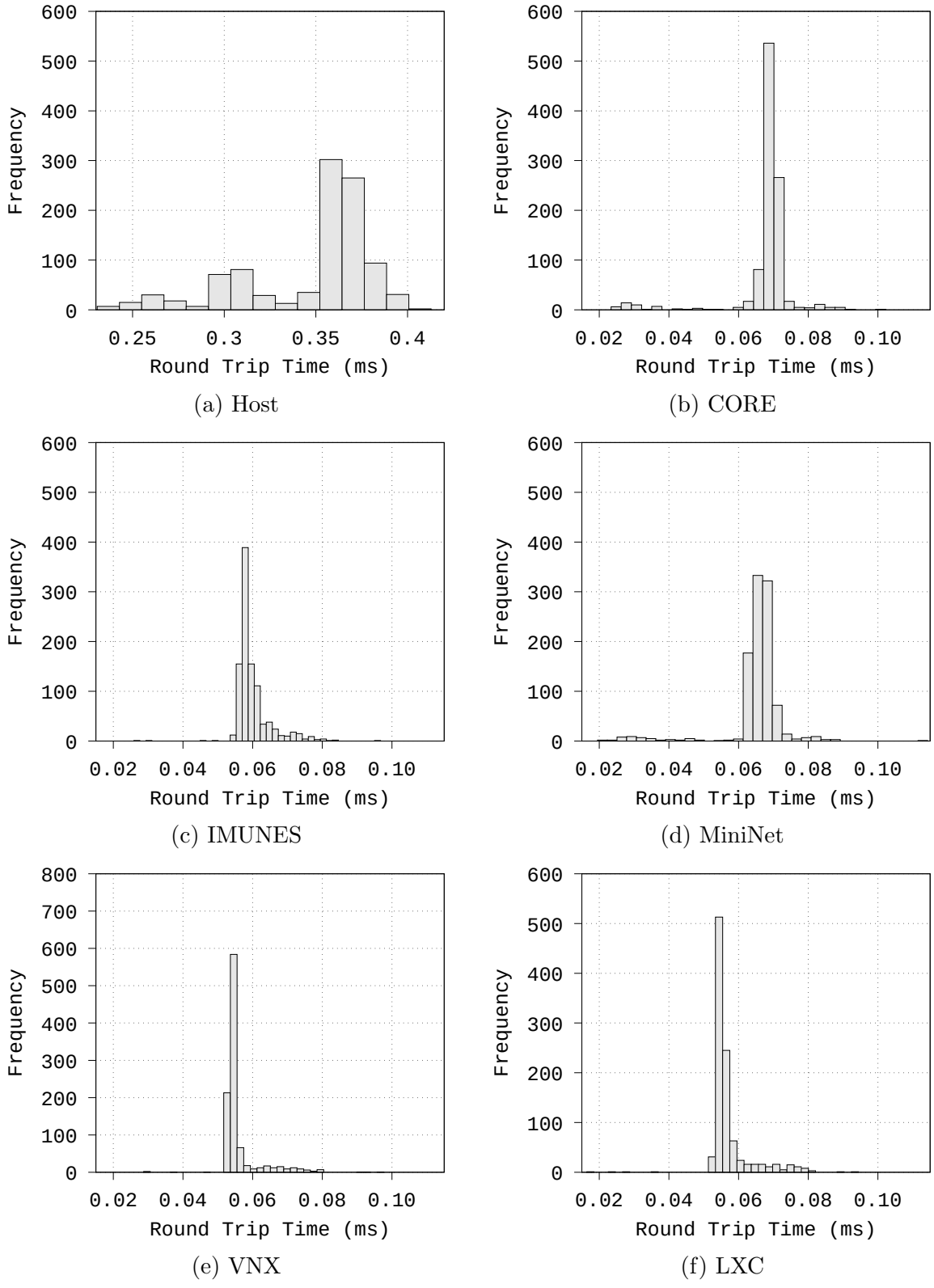


Figure 5.7: Ping Latency Distribution - Run 1



(a) Host

(b) CORE

(c) IMUNES

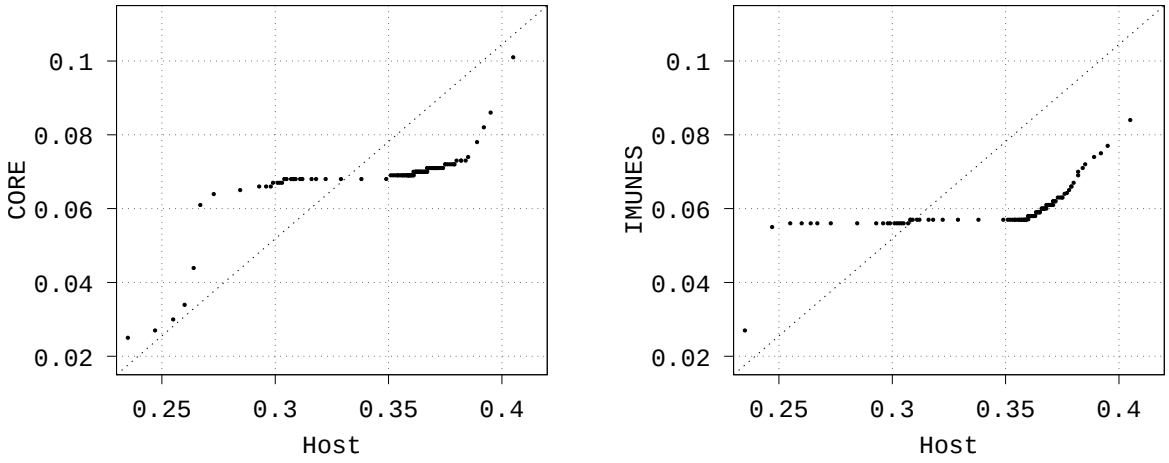
(d) MiniNet

(e) VNX

(f) LXC

Figure 5.8: Ping Distribution Histograms - Run 1

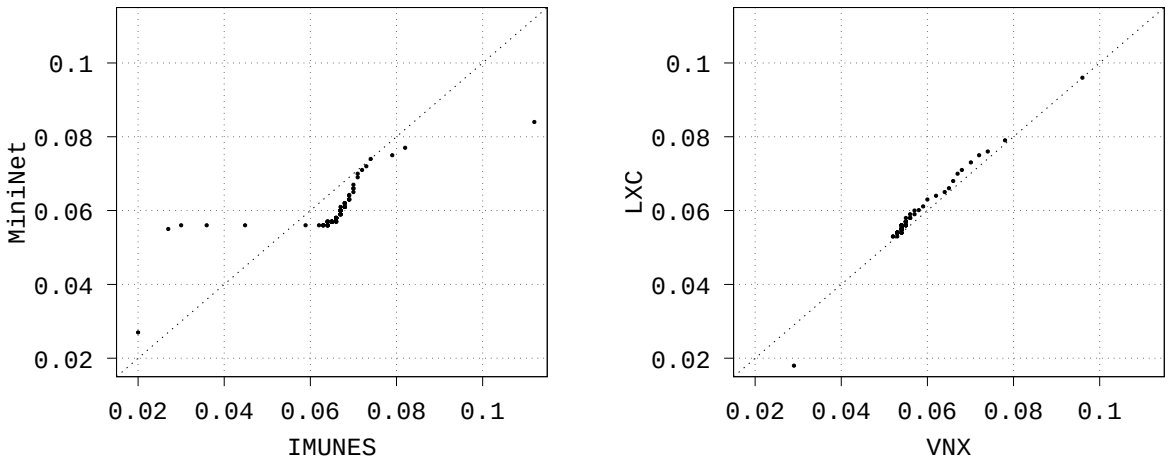
Q-Q plots (quantile-quantile plots) are used to compare the distributions of datasets. For the RTT datasets gathered, this technique enables an indirect comparison of the "shape" of the returned data. Specific, P-P plots (Q-Q plots using percentiles) were used to assess whether or not any form of correlation exists between the RTTs for different CBNEs. To generate a P-P plot the first to ninety-ninth percentiles are calculated for each set of RTTs. These percentiles can then be represented as a scatter plot between pairings of CBNEs. The premise of P-P plots is that if any similarity exists between two datasets, plotting the percentiles against one another will result in a straight line on the diagonal. The P-P plots for each CBNE-host pairing visually (Figures B.4 and B.5 in Appendix B) confirms that no correlation exists between the RTTs for CBNEs and the host. An extract of the P-P plots for two CBNE-host pairings is shown in Figure 5.9. As is shown in the two plots, the plotted data significantly deviates from the diagonal and neither of the plots show a linear relationship. It can therefore be concluded that no correlation exists between the RTTs of the CBNEs and the host.



(a) Host - CORE (b) Host - IMUNES

Figure 5.9: Ping P-P Plot Host Comparison Sample

P-P plots for CBNE-CBNE pairings delivered interesting results, and can be split into two groups. The first group of pairings shows no correlation, while the second group of pairings shows definitive correlation. An extract of the plots is shown in Figure 5.10. Figure 5.10a shows a similar pattern to that of the CBNE-host plots, whereas Figure 5.10b shows a remarkably linear correlation between the RTTs for the VNX CBNE and LXC. The complete list of P-P plots for both runs and all test platform combinations can be found in Appendix B, Figures B.4 through B.9. Additional findings are discussed in Section 5.4.3 and a summary of the findings based on the P-P plots is shown in Table 5.15.



(a) IMUNES - MiniNet (b) VNX - LXC

Figure 5.10: Ping P-P Plot CBNEs Comparison Sample

5.4.3 Ping Latency Findings

The statistics reported (Section 5.4.1) by the `ping` utility indicated that CBNEs using Open vSwitch to construct emulated switches have higher ping RTTs for the first ping on networks with low traffic. The increased RTT is not related to ARP resolution.

The exploratory statistical data analysis conducted on the 1000 count ping tests in Section 5.4.2 indicated that no similarity exists between the distribution of RTT of CBNEs and the host. Comparisons between the RTT distributions of pairings of CBNEs indicated that the technologies used for node emulation have an influence on packet timings. CBNEs that use managed containers like Docker and LXC have very strong similarities in the distribution of RTT. CBNEs that use custom Linux namespace implementations to instantiate emulated nodes also share a very strong similarity in the distribution of ping RTTs. A summary of the correlations between tested platforms is shown in Table 5.15.

Table 5.15: Ping Distribution Visual Correlations

	CORE	IMUNES	MiniNet	VNX	LXC
Host	×	×	×	×	×
CORE		×	•	×	×
IMUNES			×	•	•
MiniNet				×	×
VNX					•

• Linear correlation
× No correlation

5.5 Passive Fingerprinting Results

The reported kernel version results of Section 5.3 gave some hints, such as a single CBNEs reported as running a different OS, that there might be a difference in the fidelity of the MiniNet CBNE. The reported kernel version results are, however, not sufficient to make any deductions on the remote fidelity of CBNEs. In this section a detailed analysis of the fingerprints generated by passive OS fingerprinting utilities is presented.

Guides on how to interpret fingerprints for the passive OS fingerprinting utilities are available in Appendix C. Section C.1 details the fingerprints generated by `p0f`, Section C.2 details the fingerprints generated by `ettercap`, and Section C.4 details the fingerprints generated by `SinFP3` passive fingerprinting.

5.5.1 `p0f` Fingerprint Analysis

Fingerprints generated by the `p0f` utility (discussed in Sections 4.1.2 and C.1) are based on features extracted from the TCP handshake exclusively. The passive fingerprints generated from the CBNE packet captures by the `p0f` utility are shown in Table 5.16.

Table 5.16: p0f v3.09b Fingerprints

Platform	Fingerprint
Host	`4:64+0:0:1460:mss*45,7:mss,sok,ts,nop,ws:df:0`
CORE	`4:64+0:0:1460:mss*45,7:mss,sok,ts,nop,ws:df:0`
IMUNES	`4:64+0:0:1460:mss*45,7:mss,sok,ts,nop,ws:df:0`
MiniNet	`4:64+0:0:1460:mss*`**`30,9`**`:mss,sok,ts,nop,ws:df:0`
VNX	`4:64+0:0:1460:mss*45,7:mss,sok,ts,nop,ws:df:0`
LXC	`4:64+0:0:1460:mss*45,7:mss,sok,ts,nop,ws:df:0`

Passive fingerprint generation for all CBNEs returned the same results except for MiniNet. The TCP window size and TCP window scaling factor for MiniNet differed from the other CBNEs, as highlighted in Table 5.16

5.5.2 `ettercap` Fingerprint Analysis

The `ettercap` passive fingerprinting utility, similar to `p0f`, utilises features extracted from the TCP handshake. The functioning of `ettercap` is discussed in Section 4.1.2

and a guide on how to interpret fingerprints generated by `ettercap` is available in Section C.2. The fingerprints generated by `ettercap` (Table 5.17) from packet captures created during testing returned results similar to those of `p0f`. The fingerprints generated indicated that MiniNet uses a different TCP windows size and TCP window scaling factor. These differences are highlighted in the table below.

Table 5.17: ettercap v0.82 Fingerprints

Platform	Fingerprint
Host	FE88:05B4:40:07:1:1:1:1:A:3C
CORE	FE88:05B4:40:07:1:1:1:1:A:3C
IMUNES	FE88:05B4:40:07:1:1:1:1:A:3C
MiniNet	**A9B0**:05B4:40:**09**:1:1:1:1:A:3C
VNX	FE88:05B4:40:07:1:1:1:1:A:3C
LXC	FE88:05B4:40:07:1:1:1:1:A:3C

5.5.3 SinFP3 Passive Fingerprint Analysis

The `SinFP3` fingerprinting utility can generate both passive and active fingerprints (Sections 4.1.1 and 4.1.2). As with `ettercap` and `p0f`, `SinFP3` utilises features extracted from the TCP protocol. The passive fingerprints generated by the `SinFP3` utility are shown in Table 5.18.

Table 5.18: SinFP3 Passive Fingerprints

Platform	TF	TWS	TO	MSS	TWSF	TOL
Host	F0x02	W64240	O0204ffff...03ff†	M1460	S7	L20
CORE	F0x02	W64240	O0204ffff...03ff†	M1460	S7	L20
IMUNES	F0x02	W64240	O0204ffff...03ff†	M1460	S7	L20
MiniNet	F0x02	**W42340**	O0204ffff...03ff†	M1460	**S9**	L20
VNX	F0x02	W64240	O0204ffff...03ff†	M1460	S7	L20
LXC	F0x02	W64240	O0204ffff...03ff†	M1460	S7	L20

† O0204ffff0402080a........00000000010303ff

As with the other passive fingerprinting utilities, `SinFP3` reported that MiniNet utilised a TCP windows size and TCP windows scaling factor different to the other CBNEs. These differences are highlighted in Table 5.18.

5.5.4 Passive Fingerprinting Findings

All the passive fingerprinting utilities reported that MiniNet used a different TCP window size and TCP window scaling factor. All three passive fingerprinting utilities calculated the TCP window size used by MiniNet differently, `p0f` calculated it as 43800, `ettercap` calculated it as 43440, and `SinFP3` calculate it as 42340. All passive fingerprinting utilities did record the TCP window scaling factor used by MiniNet as 9. All other tested systems used a TCP window scaling factor of 7. Table 5.19 shows the passive remote fidelity of the tested systems based on the scoring system defined in Section 5.2.4. All CBNEs scored perfectly except for MiniNet, which scored 27/33. The reduced passive fidelity score of the MiniNet CBNE is a direct result of the TCP window size and window scale option used. Based on the scoring system, MiniNet had the lowest passive remote fidelity of all the tested systems.

Table 5.19: Passive Fidelity Scores

Platform	p0f	ettercap	SinFP3	Score
CORE	0	0	0	33/33
IMUNES	0	0	0	33/33
MiniNet	-2	-2	-2	27/33
VNX	0	0	0	33/33
LXC	0	0	0	33/33

5.6 Active Fingerprinting Results

Passive fingerprinting makes use of "legitimate" TCP connections to a target host. In the passive fingerprinting tests, all utilities reported the same deviations for MiniNet - that a different TCP window size and TCP window scale factor were used. Active fingerprinting utilities extend the protocol usage beyond the TCP protocol. `xprobe2` utilises TCP, ICMP, and SNMP probes to generate fingerprints. In the test network configuration target nodes did not have SNMP services enabled. `SinFP3` utilises only the TCP protocol and generates fingerprints based on three TCP probes. `nmap` uses TCP, UDP, and ICMP probes to generate fingerprints, however `nmap` uses multiple probes per protocol to enumerate the possible combinations of protocol options that a target node might use. In this section the results obtained from active fingerprinting against emulated target nodes is presented.

Guides on how to interpret fingerprints for the active OS fingerprinting utilities are available in Appendix C. Section C.3 details the fingerprints generated by xprobe2, Section C.4 details the fingerprints generated by SinFP3 fingerprinting. The fingerprints generated by nmap are the largest of all the fingerprinting utilities used. A detailed overview of the fingerprinting methods used by nmap and how an nmap fingerprint is structured can be found in Lyon (2009, Chapter 8 - Remote OS Detection), specifically the sections titled *TCP/IP Fingerprinting Methods Supported by Nmap* and *Understanding an Nmap Fingerprint*.

5.6.1 xprobe2 Fingerprint Analysis

The xprobe2 utility was used in two configurations. The first configuration relied on xprobe2 to discover and fingerprint open ports on target nodes. For the second configuration, the command used to execute xprobe2 included information regarding TCP and UDP in both the open and closed states. The second configuration is referred to as xprobe2 PortSpec within the text.

Table 5.20 presents a condensed version of the fingerprints generated using the first configuration of xprobe2. Tests that returned results returned the same values for all CBNEs; a full set of results can be found in Appendix D in Table D.1. Two test systems, the Host and VNX, failed to return results for the *tcp_rst* test, indicating that the default list of ports scanned by xprobe2 did not find a closed TCP port on these systems. The *icmp_echo* failed to return results for VNX. Due to the guesswork required by the xprobe2 utility to generate fingerprints, the results of the first configuration were excluded from the remote fidelity score.

Table 5.20: xprobe2 Results - Condensed

Test	Host	CORE	IMUNES	MiniNet	VNX	LXC
icmp_echo	•	•	•	•	×	•
icmp_timestamp_reply	•	•	•	•	•	•
icmp_addrmask_reply	•	•	•	•	•	•
icmp_info_reply	×	×	×	×	×	×
icmp_unreach	•	•	•	•	•	•
icmp_unreach_echoed	•	•	•	•	•	•
tcp_syn_ack (1)	×	×	×	×	×	×
tcp_syn_ack (2)	×	×	×	×	×	×
tcp_rst	×	•	•	•	×	•

• Test returned results - same as host
× Test did not return results

For the xprobe2 PortSpec tests, the ports of the simulated services were included as part of the command used to execute xprobe2 (Table 5.5). The results of the PortSpec test

are shown in condensed form in Table 5.21, and the full results are shown in Appendix D, Table D.2. All CBNEs returned results for all tests except for the icmp_info_reply test, where no CBNE returned results. For the PortSpec tests all test systems returned the same results expect for MiniNet.

Table 5.21: `xprobe2` PortSpec Results - Condensed

Test	Host	CORE	IMUNES	MiniNet	VNX	LXC
icmp_echo	•	•	•	•	•	•
icmp_timestamp_reply	•	•	•	•	•	•
icmp_addrmask_reply	•	•	•	•	•	•
icmp_info_reply	×	×	×	×	×	×
icmp_unreach	•	•	•	•	•	•
icmp_unreach_echoed	•	•	•	•	•	•
tcp_syn_ack (1)	•	•	•	•	•	•
tcp_syn_ack (2)	•	•	•	○	•	•
tcp_rst	•	•	•	•	•	•

• Test returned results - same as host
○ Test returned results - differs from host
× Test did not return results

An extract of the full results where the test systems returned different test values is shown in Table 5.22. For the *tcp_syn_ack_window_size* and *tcp_syn_ack_wscale* tests MiniNet returned a result inconsistent with the rest of the systems. Similar to the results from the passive fingerprinting tests, `xprobe2` indicated that MiniNet uses a TCP windows size of 43440 and a TCP window scale option of 9.

Table 5.22: `xprobe2` PortSpec Results - Extract

Test	Host	CORE	IMUNES	MiniNet	VNX	LXC
tcp_syn_ack_ack	1	1	1	1	1	1
tcp_syn_ack_window_size	65160	65160	65160	**43440**	65160	65160
tcp_syn_ack_options_order	MSS SACK TIMESTAMP NOP WSCALE					
tcp_syn_ack_wscale	7	7	7	**9**	7	7
tcp_syn_ack_tsval	!0	!0	!0	!0	!0	!0
tcp_syn_ack_tsecr	!0	!0	!0	!0	!0	!0

5.6.2 SinFP3 Fingerprint Analysis

`SinFP3` uses three TCP packets to generate a fingerprint of a host. Each of the three packets are designed to solicit responses that contain features that can be used to fingerprint a remote host. The fingerprints of `SinFP3` are detailed in Section C.4. Like the test results for `xprobe2`, all test systems except for MiniNet returned the same fingerprints. Table 5.23 lists the results of the `SinFP3` test probes. For the S1 probe MiniNet was reported as having a TCP window size of 42340 and for the S2 probe MiniNet was

reported as having a TCP window size of 43440 and a TCP windows scale factor of 9. The S3 probe did not report any differences between the test systems.

Table 5.23: SinFP3 Active Fingerprints

Test	Platform	BF	TF	TWS	TO	MSS	TWSF	TOL
S1	Host	B10113	F0x12	W64240	O0204ffff	M1460	S0	L4
	CORE	B10113	F0x12	W64240	O0204ffff	M1460	S0	L4
	IMUNES	B10113	F0x12	W64240	O0204ffff	M1460	S0	L4
	MiniNet	B10113	F0x12	**W42340**	O0204ffff	M1460	S0	L4
	VNX	B10113	F0x12	W64240	O0204ffff	M1460	S0	L4
	LXC	B10113	F0x12	W64240	O0204ffff	M1460	S0	L4
S2	Host	B10113	F0x12	W65160	O0204ffff...03ff†	M1460	S7	L20
	CORE	B10113	F0x12	W65160	O0204ffff...03ff†	M1460	S7	L20
	IMUNES	B10113	F0x12	W65160	O0204ffff...03ff†	M1460	S7	L20
	MiniNet	B10113	F0x12	**W43440**	O0204ffff...03ff†	M1460	**S9**	L20
	VNX	B10113	F0x12	W65160	O0204ffff...03ff†	M1460	S7	L20
	LXC	B10113	F0x12	W65160	O0204ffff...03ff†	M1460	S7	L20
S3	Host	B10120	F0x04	W0	O0	M0	S0	L0
	CORE	B10120	F0x04	W0	O0	M0	S0	L0
	IMUNES	B10120	F0x04	W0	O0	M0	S0	L0
	MiniNet	B10120	F0x04	W0	O0	M0	S0	L0
	VNX	B10120	F0x04	W0	O0	M0	S0	L0
	LXC	B10120	F0x04	W0	O0	M0	S0	L0

† O0204ffff0402080affffffff44454144010303ff

5.6.3 nmap Fingerprint Analysis

The nmap active fingerprinting utility (Section 4.1.1) uses the widest range of tests (Lyon, 2009, Chapter 8) of all the fingerprinting utilities used throughout testing, and has the largest number of components that make up a fingerprint. The structure of an nmap fingerprint and a discussion on how to decode a fingerprint can be found in Lyon (2009, Chapter 8). In the interest of saving space, only the fingerprint components that pointed to differences between the tested systems and the host are discussed. The results for components that returned the same values for all tests systems can be found in Appendix D.2.

The command to fingerprint the test systems using nmap can be found in Table 5.5. In Table 5.24, the execution times for fingerprinting the test systems using nmap is shown. Fingerprinting took roughly 8 minutes for the host and 11 minutes for the other test systems except for IMUNES. The test execution time for IMUNES was 88 seconds in both runs, significantly less than any of the other test systems.

Table 5.24: nmap Scan Execution Time

	Time (ms)	
Platform	Run 1	Run 2
Host	496.31	565.37
CORE	653.05	645.33
IMUNES	88.02	88.14
MiniNet	662.77	654.08
VNX	652.00	640.31
LXC	664.62	656.60

Table 5.25 lists the total number of UDP ports reported as Open, Open|Filtered or Closed by nmap for the tested systems. UDP ports are reported as Open if a response is received to a probe, Open|Filtered if no response is received, and Closed if an ICMP Type 3 Code 3 (Destination port unreachable) is returned. UDP ports reported as Open or Open|Filtered are subjected to 48 service discovery probes. The image used by IMUNES for node emulation explicitly responded with ICMP Destination port unreachable when probed by nmap on unused ports and was subjected to fewer service discovery tests, explaining the difference in total test time.

Table 5.25: nmap UDP Ports Detected Summary

	Port State		
Platform	Open	Open|Filtered	Closed
Host	1	27	172
CORE	1	43	156
IMUNES	2	0	198
MiniNet	1	44	155
VNX	1	43	156
LXC	1	44	155

The nmap sequence generation tests attempt to fingerprint the Internet Protocol (IP) ID sequence generation algorithms used by operating systems. This process can be complicated by OSs that send the IP ID field in host byte order and not in network byte order. The results of the sequence generation tests are shown in Table 5.26. The TCP ISN sequence predictability index (*SP*) and TCP ISN counter rate (*ISR*) tests showed different results for all tested platforms, except for the *ISR* of VNX being the same as the host. The *SP* and *ISR* values are encoded as ranges for fingerprints in the nmap database. At the time of writing, the nmap fingerprint database did not contain a fingerprint for the Linux 5.0.0 kernel, and thus no *SP* and *ISR* ranges were available to use for comparing the tested systems.

Table 5.26: nmap Sequence Generation Test Results

Platform	SP	GCD	ISR	TI	CI	II	TS
Host	F9	1	109	Z	Z	I	A
Core	FE	1	10D	Z	Z	I	A
IMUNES	101	1	10B	Z	Z	I	A
MiniNet	100	1	10F	Z	Z	I	A
VNX	107	1	109	Z	Z	I	A
LXC	103	1	10A	Z	Z	I	A

The nmap TCP Options, TCP Window Size, and Explicit Congestion Notification tests (Tables 5.27a through 5.27c) showed the same deviations from the host fingerprint for MiniNet as the other fingerprinting utilities. The nmap tests showed that MiniNet uses a TCP window scaling factor of 9 and a TCP window size different to the other tested systems.

Table 5.27: nmap Fingerprint Results

(a) TCP Options Test Results

Platform	O1	O2	O3	O4	O5	O6
Host	M5B4ST11NW7	M5B4ST11NW7	M5B4NNT11NW7	M5B4ST11NW7	M5B4ST11NW7	M5B4ST11
Core	M5B4ST11NW7	M5B4ST11NW7	M5B4NNT11NW7	M5B4ST11NW7	M5B4ST11NW7	M5B4ST11
IMUNES	M5B4ST11NW7	M5B4ST11NW7	M5B4NNT11NW7	M5B4ST11NW7	M5B4ST11NW7	M5B4ST11
MiniNet	M5B4ST11N**W9**	M5B4ST11N**W9**	M5B4NNT11N**W9**	M5B4ST11N**W9**	M5B4ST11N**W9**	M5B4ST11
VNX	M5B4ST11NW7	M5B4ST11NW7	M5B4NNT11NW7	M5B4ST11NW7	M5B4ST11NW7	M5B4ST11
LXC	M5B4ST11NW7	M5B4ST11NW7	M5B4NNT11NW7	M5B4ST11NW7	M5B4ST11NW7	M5B4ST11

(b) TCP Window Size Test Results

Platform	W1	W2	W3	W4	W5	W6
Host	FE88	FE88	FE88	FE88	FE88	FE88
CORE	FE88	FE88	FE88	FE88	FE88	FE88
IMUNES	FE88	FE88	FE88	FE88	FE88	FE88
MiniNet	**A9B0**	**A9B0**	**A9B0**	**A9B0**	**A9B0**	**A9B0**
VNX	FE88	FE88	FE88	FE88	FE88	FE88
LXC	FE88	FE88	FE88	FE88	FE88	FE88

(c) Explicit Congestion Notification Test Results

Platform	R	DF	T	W	O	CC	Q
Host	Y	Y	40	FAF0	M5B4NNSNW7	Y	—
CORE	Y	Y	40	FAF0	M5B4NNSNW7	Y	—
IMUNES	Y	Y	40	FAF0	M5B4NNSNW7	Y	—
MiniNet	Y	Y	40	**0564**	M5B4NNSN**W9**	Y	—
VNX	Y	Y	40	FAF0	M5B4NNSNW7	Y	—
LXC	Y	Y	40	FAF0	M5B4NNSNW7	Y	—

5.6.4 Active Fingerprinting Findings

Active fingerprinting tests delivered results similar to those of the passive fingerprinting tests. Active fingerprinting on all tested CBNEs generated the same fingerprints as the host except for MiniNet. As with the passive fingerprinting tests, MiniNet used a TCP window size and TCP window scaling factor different to all other tested systems.

The active remote fidelity scores for the tested systems are shown in Table 5.28. Remote fidelity scoring for `nmap` excluded the sequence generation (SEQ) tests. The ranges for the *SP* (F9 to 107) and *ISR* (109 to 10F) values are within similar bounds to the ranges of the `nmap` database entry for *Linux 4.2.5-1-ARCH* (*SP=FE-108*, *ISR=100-10A*) and were not regarded as indicating differences between the tested systems.

Table 5.28: Active Fidelity Scores

Platform	`xprobe2` PortSpec	SinFP3	`nmap`	Score
CORE	0	0	0	153/153
IMUNES	0	0	0	153/153
MiniNet	-2	-3	-13	135/153
VNX	0	0	0	153/153
LXC	0	0	0	153/153

5.7 MiniNet Modification and Re-run

The active and passive fingerprinting tests all showed that MiniNet utilises a TCP windows size and TCP window scaling factor different to all other systems tested, including the host. Table 5.29 shows the `sysctl` options for the host before (Host) and after (MiniNet (Default)) initialising an experimental network using MiniNet.

Table 5.29: `sysctl` Configuration Results

`sysctl` Option	Host	MiniNet (Default)
net.core.netdev_max_backlog	1000	5000
net.core.rmem_max	212992	16777216
net.core.wmem_max	212992	16777216
net.ipv4.neigh.default.gc_thresh1	128	4096
net.ipv4.neigh.default.gc_thresh2	512	8192
net.ipv4.neigh.default.gc_thresh3	1024	16384
net.ipv4.tcp_rmem	4096 131072 6291456	10240 87380 16777216
net.ipv4.tcp_wmem	4096 16384 4194304	10240 87380 16777216

The source of the changes to the `sysctl` options of the host and MiniNet's Window Scaling Factor of 9 was traced to a configuration applied before a simulation is started. In the MiniNet source file `mininet/mininet/util.py` (Listing 5.6), the maximum send socket memory (wmem_max) and maximum receive socket memory (rmem_max) of the network stack is increased when an experiment is initialised.

Listing 5.6: MiniNet `sysctl` Configuration Changes

```
1  def fixLimits():
2      "Fix ridiculously small resource limits."
3      debug( "*** Setting resource limits\n" )
4      try:
5          rlimitTestAndSet( RLIMIT_NPROC, 8192 )
6          rlimitTestAndSet( RLIMIT_NOFILE, 16384 )
7          #Increase open file limit
8          sysctlTestAndSet( 'fs.file-max', 10000 )
9          #Increase network buffer space
10         sysctlTestAndSet( 'net.core.wmem_max', 16777216 )
11         sysctlTestAndSet( 'net.core.rmem_max', 16777216 )
12         sysctlTestAndSet( 'net.ipv4.tcp_rmem', '10240 87380 16777216' )
13         sysctlTestAndSet( 'net.ipv4.tcp_wmem', '10240 87380 16777216' )
14         sysctlTestAndSet( 'net.core.netdev_max_backlog', 5000 )
15         #Increase arp cache size
16         sysctlTestAndSet( 'net.ipv4.neigh.default.gc_thresh1', 4096 )
17         sysctlTestAndSet( 'net.ipv4.neigh.default.gc_thresh2', 8192 )
18         sysctlTestAndSet( 'net.ipv4.neigh.default.gc_thresh3', 16384 )
19         #Increase routing table size
20         sysctlTestAndSet( 'net.ipv4.route.max_size', 32768 )
21         #Increase number of PTYs for nodes
22         sysctlTestAndSet( 'kernel.pty.max', 20000 )
```

The `mininet/mininet/util.py` source was modified to reflect the `sysctl` options of the host before experimental networks were instantiated on MiniNet. Listing 5.30 shows the modified version of the source file. The modifications made to the functions that set the net.core.wmem_max, net.core.rmem_max, net.ipv4.tcp_rmem, and net.ipv4.tcp_wmem `sysctl` options are highlighted in lines 10 to 13.

Listing 5.7: MiniNet `sysctl` Configuration Changes Modified

```
1  def fixLimits():
2      "Fix ridiculously small resource limits."
3      debug( "*** Setting resource limits\n" )
4      try:
5          rlimitTestAndSet( RLIMIT_NPROC, 8192 )
6          rlimitTestAndSet( RLIMIT_NOFILE, 16384 )
7          #Increase open file limit
8          sysctlTestAndSet( 'fs.file-max', 10000 )
9          #Increase network buffer space
10         sysctlTestAndSet( 'net.core.wmem_max', 212992 )
11         sysctlTestAndSet( 'net.core.rmem_max', 212992 )
12         sysctlTestAndSet( 'net.ipv4.tcp_rmem', '4096 131072 6291456' )
13         sysctlTestAndSet( 'net.ipv4.tcp_wmem', '4096 16384 4194304' )
14         sysctlTestAndSet( 'net.core.netdev_max_backlog', 1000 )
15         #Increase arp cache size
16         sysctlTestAndSet( 'net.ipv4.neigh.default.gc_thresh1', 128 )
17         sysctlTestAndSet( 'net.ipv4.neigh.default.gc_thresh2', 512 )
18         sysctlTestAndSet( 'net.ipv4.neigh.default.gc_thresh3', 1024 )
19         #Increase routing table size
20         sysctlTestAndSet( 'net.ipv4.route.max_size', 32768 )
21         #Increase number of PTYs for nodes
22         sysctlTestAndSet( 'kernel.pty.max', 20000 )
```

Post modification of the MiniNet source, when initialising an experimental network, the `sysctl` options of the host remained the same as before initialising a network. Table

5.30 shows the `sysctl` options for the host before (Host) and after (MiniNet (Modified)) initialising an experimental network using the modified version of MiniNet.

Table 5.30: `sysctl` Configuration Results After Modification

`sysctl` Option	Host	MiniNet (Modified)
net.core.netdev_max_backlog	1000	1000
net.core.rmem_max	212992	212992
net.core.wmem_max	212992	212992
net.ipv4.neigh.default.gc_thresh1	128	128
net.ipv4.neigh.default.gc_thresh2	512	512
net.ipv4.neigh.default.gc_thresh3	1024	1024
net.ipv4.tcp_rmem	4096 131072 6291456	4096 131072 6291456
net.ipv4.tcp_wmem	4096 16384 4194304	4096 16384 4194304

5.7.1 Summary

The active and passive fingerprinting tests were re-run on the modified version of MiniNet, and all test results showed that post modification MiniNet generated the same fingerprints as the other test systems. The results of fingerprinting an emulated node instantiated by the modified version of MiniNet are shown in Appendix D, Section D.3. The active and passive remote fidelity scores for MiniNet were updated based on the post modification fingerprinting results. The updated remote fidelity scores are shown in Table 5.31, showing that the modified MiniNet achieved the same perfect remote fidelity score as the other tested systems.

Table 5.31: Remote Fidelity for Modified MiniNet

(a) Passive

Platform	p0f	ettercap	SinFP3	Score
CORE	0	0	0	33/33
IMUNES	0	0	0	33/33
MiniNet	-2	-2	-2	27/33
VNX	0	0	0	33/33
LXC	0	0	0	33/33
MiniNet†	**0**	**0**	**0**	**33/33**

† Modified version of MiniNet

(b) Active

Platform	`xprobe2` PortSpec	SinFP3	`nmap`	Score
CORE	0	0	0	153/153
IMUNES	0	0	0	153/153
MiniNet	-2	-3	-13	133/153
VNX	0	0	0	153/153
LXC	0	0	0	153/153
MiniNet†	**0**	**0**	**0**	**153/153**

† Modified version of MiniNet

The differences between the fingerprints of the default MiniNet and the host was not related to the components used by MiniNet to instantiate emulated nodes and network components. The differences were related to `sysctl` changes applied to the kernel of the host.

5.8 Summary

Section **5.1** presented an overview of the components in a computer network that can influence the ability of active and passive fingerprinting utilities to generate accurate fingerprints of a remote host. Network devices such load balancers and firewalls can strip away or modify L2 and L3 headers and prevent fingerprinting utilities from generating accurate fingerprints and server applications such as web servers can implement protection mechanisms that prevent fingerprinting. The test network used for fingerprinting was a basic switched network, and the target node did not have any applications or utilities installed capable of interfering with the fingerprinting process. Section **5.2** detailed the fingerprinting utilities, the CBNEs selected for testing and the process used to test and evaluate the remote fidelity of CBNEs.

The OS families and versions reported during active and passive fingerprinting were presented in Section **5.3**. The passive fingerprinting utilities used did not return OS family and version results that reflected the known OS used. In part, the results were due to the age of the fingerprint databases used by these utilities. The active fingerprinting utilities used returned a mixed set of results. The `xprobe2` active fingerprinting utility did not return readable results. The OS version printed appeared to be a random set of ASCII characters. The `nmap` and `SinFP3` utilities accurately identified the OS family of the target nodes' OSs but could not identify the OS version. This can be attributed to the age of the fingerprinting databases used. Though the latest version of the fingerprinting databases for these tools were used, they pre-dated the release of the kernel version used during testing, and as such did not contain fingerprints for the kernel and could not identify the version.

Section **5.4** presented the findings of the ping (ICMP Type 8 - echo request) RTT latency tests. Through the use of exploratory data analysis techniques it was established that the distribution of RTTs for each CBNE was related to the networking subsystems used. P-P plots were used to test for similarities in the distributions of RTTs between pairings of test systems. None of the ping RTT distributions for none of the CBNEs could be correlated to the host, however clear correlations could be established between CBNEs that used the same networking subsystem.

The results for active and passive fingerprint tests were presented in Sections **5.5** and **5.6**, respectively. All the fingerprinting utilities used showed that emulated hosts in experimental networks instantiated by MiniNet used a TCP window size and TCP window scaling factor different to the other systems tested. Both fingerprinting methodologies

showed that all tested CBNEs except MiniNet had perfect remote fidelity as defined in Section 4.3 and scored according to the method in Section 5.2.

The results of an investigation into the cause of the fingerprint differences for nodes emulated using the MiniNet CBNE were presented in Section **5.7**. The origin of the fingerprint deviations was traced to performance optimisations applied to the host OS by MiniNet before an experimental network was instantiated. MiniNet increases the read and write buffers of the network stack of the host to increase the packets per second that the kernel can handle during experimentation. The source code for MiniNet was modified to reflect the `sysctl` options of the host OS. When instantiating an experimental network with the modified version of MiniNet, no changes were applied to the host OS. The test procedure was re-run on the modified version of MiniNet. Emulated nodes in the modified version of MiniNet returned fingerprinting results that were the same as all the other tested systems and had a perfect remote fidelity score.

5.8.1 Major Findings

The major findings of the experimental component of the research conducted are presented below.

The hypothesis that components used to construct CBNEs can influence the features extracted by OS fingerprinting utilities such as `nmap` when scanning emulated nodes was shown to be false. In one exceptional case, that of MiniNet, deviations in the fingerprints of emulated nodes where caused by the CBNE applying network performance optimisations to the network stack of the host OS. After disabling the optimisations applied to the host OS by MiniNet, fingerprints generated for MiniNet emulated nodes returned the same results as all other tested CBNEs.

The behaviour of emulated nodes during fingerprinting can possibly assist in identifying the type of CBNE used. For one specific CBNE a behavioural difference, when compared to all other tested systems, was detected. The base image used for instantiating nodes using the IMUNES CBNE was configured to send explicit ICMP Type 3 Code 3 (Destination port unreachable) to `nmap` UDP port scans. The result of this behavioural difference was a significantly reduced scan time and zero reported open UDP ports. A report of zero open UDP ports is an unexpected result. Such subtle behavioural differences can by used to identify the CBNE used to construct research and experimentation networks.

Through the use of exploratory statistical analysis of the round trip times (RTTs) of ping packets (ICMP Type 8) between nodes emulated in a CBNE, it was discovered that these independent data sets could be used to identify CBNEs that use the same networking sub-system. A comparison of the distribution of ping packet RTTs of the tested CBNEs was conducted using P-P plots (percentile-percentile plots). This pair-wise comparison of the ping RTT of CBNEs revealed that CBNEs using the same networking sub-system had strong correlation, and that ping RTTs can be used to identify these systems.

Chapter 6

Conclusion

The world is still a weird place,
despite my efforts to make clear and perfect sense of it.

Hunter S. Thompson

The research presented in this document investigated the viability of Container-Based Network Emulators (CBNEs), a type of Network Experimentation Platform (NEP), as platforms for information security research, experimentation, and training. The abstraction mechanisms used by NEPs were investigated to assess the impact that abstraction techniques could have on the realism of experimental networks. CBNEs were investigated in detail. A selection of open-source CBNE implementations were analysed with respect to the architectural choices made during development and the technologies used to create experimental networks.

Remote attackers utilise fingerprinting utilities such as `nmap` to discover a target's Operating System (OS) and to enumerate remotely accessible services. The techniques used to fingerprint a remote OS and how fingerprinting would conceptually be used by attackers were investigated. Based on the investigation, a model for measuring the remote fidelity of an emulated host was created and used to measure the remote fidelity of emulated nodes for a selection of CBNEs.

The design of a computer system as a system of abstractions was introduced in Chapter 2. These abstractions can be used to implement fully functional computer systems, and serve as building blocks for different types of virtualisation systems. The types of network experimentation platforms constructed using different kinds of virtualisation technologies, and how these systems are currently used, where discussed. This chapter showed how the

definition of fidelity for emulated computer systems varied depending on the context of how the systems are used as well as the experiments being performed.

Chapter 3 provided an overview of the origins of CBNEs and how these systems utilise Linux containers to create networks of lightweight virtualised computers for research, experimentation, and training. The chapter gave an overview of namespaces in the Linux kernel and how namespaces are used to construct lightweight virtual machines called containers. A selection of open-source CBNEs for the Linux OS was introduced, and the history and typical uses of each system was discussed. This chapter also detailed the architecture of each CBNE and the technologies used by each system to construct emulated networks.

The concepts and techniques used by active and passive OS fingerprinting utilities were reviewed in Chapter 4. Models that describe the various processes used by remote attackers attempting to penetrate a remote host were investigated to assess the applicability of fingerprinting during attacks on remote systems. The concept of extracting features from network traffic that can be used identify OSs was used to construct a model to remotely measure the fidelity of a host emulated by CBNEs.

Chapter 5 began with an overview of the components in a computer network and software systems on a computer that can influence the features extracted by active and passive OS fingerprinting utilities to generate fingerprints. The test network for experimentation was designed to exclude any components that can interfere with the fingerprinting process. Details of the fingerprinting utilities and CBNEs used during testing were listed. Analysis of the results obtained from testing started off with an analysis of the kernel versions reported by active and passive fingerprinting utilities. An additional analysis was done on the distribution of Round Trip Times (RTTs) of ping requests to assess whether or not the components used by CBNEs influence latency within emulated networks. The fingerprints generated by active and passive OS fingerprinting utilities indicated that emulated nodes for one of the CBNEs differed from the host. An investigation into the cause of the differences revealed that an optimisation to the network stack of the host OS was the cause.

The objectives of the research conducted are listed in Section 1.2. The first objective was to review the technologies used by CBNEs to create emulated networks as well as to review the techniques used to remotely fingerprint computer systems. Open-source CBNEs for the Linux OS use a wide array of technologies to construct experimental networks (Section 3.5). Though Linux namespaces form the base technology for constructing nodes in a network, a mix of pre-existing tools and custom techniques are used to construct containers

from namespaces. To discover what OS a remote computer is using OS fingerprinting utilities such as `nmap` can be used (Section 4.1). OS fingerprinting can be conducted through direct interaction using active fingerprinting, or through indirect means using passive fingerprinting. Both active and passive fingerprinting utilities extract artefacts from network traffic that can be used to identify the family and version of a remote computer.

The second objective was to create a model to measure the remote fidelity of emulated nodes in an experimental network. The artefacts extracted from network traffic by fingerprinting utilities can be modified by networking equipment and by OSs. CBNEs instantiate emulated nodes in experimental networks using abstraction mechanisms such as containerisation. Through this process of abstraction the fidelity of emulated nodes could be decreased. A model for testing how well an emulated node replicates the base OS was created in Section 4.3.

The third objective was to use the model to measure the remote fidelity of emulated nodes for a selection of open-source CBNEs within the context of information security research, experimentation, and training. Passive (Section 5.5) and active (Section 5.6) fingerprinting conducted on emulated nodes showed that nodes emulated by MiniNet had fingerprints that differed from all other test systems. Once the source of the deviation was corrected and the test suite was re-run, emulated nodes returned the same fingerprints as the other test systems (Section 5.7). The test results indicated that the emulated nodes have perfect fidelity when measured according to the model created in Section 4.3.

The significance of this study is that it has shown that Container-Based Network Emulators (CBNEs) can be used to create small scale information security research, experimentation, and training networks using commodity hardware. Analyses of the experimental data collected has shown that nodes instantiated in emulated networks do present valid and viable targets for information security experimentation, research, and education. When measured according to the definition of remote fidelity for abstracted hosts, nodes instantiated in emulated networks present targets that cannot be distinguished from the host system.

Computers are complex systems made up of multiple systems and sub-systems. Each of these systems can leak identifying information to remote attackers in unexpected ways. Data such as the time taken for a ping packet to traverse a network appears not to give away information that can be used to identify a system. As was shown, if a large enough sample of data is collected, even seemingly insignificant and irrelevant information can be used to gain deeper insights into the components of complex systems.

6.1 Future Work

This section lists additional research avenues that fell outside of the scope of this document, as well as future research avenues to be explored.

6.1.1 Extended Experimentation

The experimental component of this research covered a limited set of CBNEs. Only open-source CBNEs for the Linux OS were considered and experimental networks were limited to L2 networking equipment. Extensive testing of CBNEs for both the Linux and FreeBSD OSs, inclusive of L3 networking equipment, will quantify the effects that CBNEs could have on network traffic and the artefacts that sub-systems of CBNEs may introduce into traffic generated on experimental networks.

6.1.2 Ping RTT Distibution Correlation on Physical Networks

During the analysis of the ping round trip time data, it was discovered that the distribution of the round trip times can be used to identify CBNEs that utilise the same network sub-systems. Testing should be expanded to include physical networking devices to assess whether the same phenomena are present within the physical space, or if these phenomena are localised to virtualised networking systems. The informal correlation techniques used, that of P-P plots, should be developed into formal techniques. Formalising the techniques used to identify networking systems from round trip time distribution data will assist in generating a new class of device fingerprint.

References

Acosta, J. C., McKee, J., Fielder, A., and Salamah, S. A platform for evaluator-centric cybersecurity training and data acquisition. In *2017 IEEE Military Communications Conference*, pages 394–399. Oct 2017. doi:10.1109/MILCOM.2017.8170768.

Adams, K. and Agesen, O. A Comparison of Software and Hardware Techniques for x86 Virtualization. *ACM SIGOPS Operating Systems Review*, 40(5):2–13, October 2006. ISSN 0163-5980. doi:10.1145/1168917.1168860.

Ahrenholz, J. Comparison of CORE network emulation platforms. In *2010 IEEE Military Communications Conference*, pages 166–171. Oct 2010. ISSN 2155-7578. doi:10.1109/MILCOM.2010.5680218.

Ahrenholz, J., Danilov, C., Henderson, T. R., and Kim, J. H. CORE: A real-time network emulator. In *2008 IEEE Military Communications Conference*, pages 1–7. Nov 2008. ISSN 2155-7578. doi:10.1109/MILCOM.2008.4753614.

Ahrenholz, J., Goff, T., and Adamson, B. Integration of the CORE and EMANE Network Emulators. In *2011 IEEE Military Communications Conference*, pages 1870–1875. Nov 2011. ISSN 2155-7578. doi:10.1109/MILCOM.2011.6127585.

Aiken, H. H. and Hopper, G. M. The Automatic Sequence Controlled Calculator — I. *Electrical Engineering*, 65(8-9):384–391, Aug 1946a. ISSN 0095-9197. doi:10.1109/EE.1946.6434251.

Aiken, H. H. and Hopper, G. M. The Automatic Sequence Controlled Calculator — II. *Electrical Engineering*, 65(10):449–454, Oct 1946b. ISSN 0095-9197. doi:10.1109/EE.1946.6439869.

Aiken, H. H. and Hopper, G. M. The Automatic Sequence Controlled Calculator — III. *Electrical Engineering*, 65(11):522–528, Nov 1946c. ISSN 0095-9197. doi:10.1109/EE.1946.6439921.

Aksoy, A., Louis, S., and Gunes, M. H. Operating system fingerprinting via automated network traffic analysis. In *2017 IEEE Congress on Evolutionary Computation*, pages 2502–2509. June 2017. doi:10.1109/CEC.2017.7969609.

Albanese, M., Battista, E., and Jajodia, S. Cyber Deception: Building the Scientific Foundation, chapter Deceiving Attackers by Creating a Virtual Attack Surface, pages 167–199. Springer, 2016. ISBN 978-3-319-32699-3. doi:10.1007/978-3-319-32699-3_8.

Alcosser, E., Phillips, J. P., and Wolk, A. M. How to Build a Working Digital Computer. Hayden Book Company, Inc., New York, NY, USA, June 1967. ISBN 978-0-810407-480. Last accessed: 2019-03-18.
URL `https://archive.org/details/howtobuildaworkingdigitalcomputer_jun67`

Amstadt, B. and Johnson, M. K. Wine. *Linux Journal*, 1994(4), August 1994. ISSN 1075-3583.

Anderson, B. and McGrew, D. OS Fingerprinting: New Techniques and a Study of Information Gain and Obfuscation. In *2017 IEEE Conference on Communications and Network Security*, pages 1–9. Las Vegas, NV, USA, October 2017. doi:10.1109/CNS.2017.8228647.

Anton, S. D., Fraunholz, D., Krummacker, D., Fischer, C., Karrenbauer, M., and Schotten, H. D. The Dos and Don'ts of Industrial Network Simulation: A Field Report. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control*, ISCSIC '18, pages 6:1–6:8. ACM, Stockholm, Sweden, 2018. ISBN 978-1-4503-6628-1. doi:10.1145/3284557.3284716.

Arkin, O. and Yarochkin, F. Xprobe v2.0 A "Fuzzy" Approach to Remote Active Operating System Fingerprinting. August 2002. Last accessed: 2019-03-18.
URL `http://ouah.org/Xprobe2.pdf`

Asada, T. Implements BIOS Emulation Support for BHyVe: A BSD Hypervisor. In *AsiaBSDCon 2013 Proceedings*, pages 63–73. Tokyo, Japan, March 2013.

Auffret, P. SinFP, unification de la prise d'empreinte active et passive des systèmes d'exploitation. In *Symposium sur la sécurité des technologies de l'information et des communications*. June 2008. Last accessed: 2019-03-24.
URL `https://www.sstic.org/2008/presentation/SinFP_unification_de_la_prise_d_empreinte_active_et_passive_des_systemes_d_exploitation/`

Auffret, P. SinFP, Unification of Active and Passive Operating System Fingerprinting. *Journal in Computer Virology*, 6(3):197–205, August 2010. ISSN 1772-9890. doi:10.1007/s11416-008-0107-z.

Austin, T., Larson, E., and Ernst, D. SimpleScalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb 2002. ISSN 0018-9162. doi:10.1109/2.982917.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177. ACM, Bolton Landing, NY, USA, 2003. ISBN 1-58113-757-5. doi:10.1145/945445.945462.

Barnett, R. J. and Irwin, B. Towards a Taxonomy of Network Scanning Techniques. In *Proceedings of the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries: Riding the Wave of Technology*, SAICSIT '08, pages 1–7. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-286-3. doi:10.1145/1456659.1456660.

Bavier, A., Berman, M., Brinn, M., McGeer, R., Peterson, L., and Ricart, G. Realizing the Global Edge Cloud. *IEEE Communications Magazine*, 56(5):170–176, May 2018. ISSN 0163-6804. doi:10.1109/MCOM.2018.1700131.

Bellard, F. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*, pages 41–46. USENIX Association, April 2005. Last accessed: 2019-03-18.

Berman, M., Chase, J. S., Landweber, L., Nakao, A., Ott, M., Raychaudhuri, D., Ricci, R., and Seskar, I. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 61:5 – 23, 2014. ISSN 1389-1286. doi:10.1016/j.bjp.2013.12.037. Special issue on Future Internet Testbeds – Part I.

Bernstein, D. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, September 2014. ISSN 2325-6095. doi:10.1109/MCC.2014.51.

Bhatia, S., Motiwala, M., Muhlbauer, W., Mundada, Y., Valancius, V., Bavier, A., Feamster, N., Peterson, L., and Rexford, J. Trellis: A Platform for Building Flexible, Fast Virtual Networks on Commodity Hardware. In *Proceedings of the 2008 ACM CoNEXT Conference*, pages 72:1–72:6. ACM, Madrid, Spain, 2008. ISBN 978-1-60558-210-8. doi:10.1145/1544012.1544084.

Biederman, E. W. Multiple Instances of the Global Linux Namespaces. In *Proceedings of the Linux Symposium*, volume 1, pages 101–112. Ottawa, ON, Canada, July 2006.

bind. Passive Network Mapping. Online, May 2000. Last accessed 2019-10-23.
URL `http://lwn.net/2000/0511/a/siphon.html`

Böhme, U. and Buytenhenk, L. Linux BRIDGE-STP-HOWTO, January 2001. Last accessed: 2019-09-15.
URL `http://www.losurs.org/docs/LDP/HOWTO/pdf/BRIDGE-STP-HOWTO.pdf`

Boissiere, G. STATUS 2.5. Online, January 2002. Last accessed 2019-10-23.
URL `http://lkml.iu.edu/hypermail/linux/kernel/0201.2/0331.html`

Bonada, E., Cavic, D., and Sala, D. Implementation of a Layer 2 Bridge in Ns-3. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 49:1–49:1. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST), ICST, Brussels, Belgium, Belgium, 2008. ISBN 978-963-9799-20-2.

Bonofiglio, G., Iovinella, V., Lospoto, G., and Battista, G. D. Kathará: A container-based framework for implementing network function virtualization and software defined networks. In *2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. April 2018. ISSN 2374-9709. doi:10.1109/NOMS.2018.8406267.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. ISSN 0146-4833. doi:10.1145/2656877.2656890.

Box, D. and Sells, C. Essential .NET, Volume I: The Common Language Runtime. Microsoft .NET Development Series. Addison-Wesley Professional, 2002. ISBN 9780201734119.

Boyd, I. M. The Fundamentals of Computer Hacking. Online, December 2000. Last accessed: 2019-03-24.
URL `http://www.sans.org/reading_room/whitepapers/hackers/fundamentals-computer-hacking_956`

Boyd, J. R. Organic Design for Command and Control. *A Discourse on Winning and Losing*, 1987. Unpublished lecture notes.

Browne, A. F., Watson, S., and Williams, W. B. Development of an Architecture for a Cyber-Physical Emulation Test Range for Network Security Testing. *IEEE Access*, 6:73273–73279, 2018. ISSN 2169-3536. doi:10.1109/ACCESS.2018.2882410.

Bullers, W. I., Jr., Burd, S., and Seazzu, A. F. Virtual Machines - an Idea Whose Time Has Returned: Application to Network, Security, and Database Courses. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '06, pages 102–106. ACM, New York, NY, USA, 2006. ISBN 1-59593-259-3. doi:10.1145/1121341.1121375.

Cappos, J., Hemmings, M., McGeer, R., Rafetseder, A., and Ricart, G. EdgeNet: A Global Cloud That Spreads by Local Action. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 359–360. Oct 2018. doi:10.1109/SEC.2018.00045.

CDW Corporation. Securing BYOD, March 2018. Last accessed: 2018-10-29.
URL `https://webobjects.cdw.com/webobjects/media/pdf/solutions/security/BYOD-Security-G.pdf`

Chapman, S., Smith, R., Maglaras, L., and Janicke, H. Can a Network Attack Be Simulated in an Emulated Environment for Network Security Training? *Journal of Sensor and Actuator Networks*, 6(3):16, Aug 2017. ISSN 2224-2708. doi:10.3390/jsan6030016.

Chen, I., King, C., Chen, Y., and Lu, J. Full System Emulation of Embedded Heterogeneous Multicores Based on QEMU. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 771–778. Dec 2018. ISSN 1521-9097. doi:10.1109/PADSW.2018.8645045.

Chen, Y.-C., Liao, Y., Baldi, M., Lee, S.-J., and Qiu, L. OS Fingerprinting and Tethering Detection in Mobile Networks. In *Proceedings of the 2014 Internet Measurement Conference*, IMC '14, pages 173–180. ACM, Vancouver, BC, Canada, 2014. ISBN 978-1-4503-3213-2. doi:10.1145/2663716.2663745.

Cheswick, B. An Evening with Berferd in Which a Cracker is Lured, Endured, and Studied. In *Proceedings of the Winter 1992 USENIX Conference*, pages 163–174. USENIX Association, San Francisco, CA, USA, January 1992.
URL `http://www.cheswick.com/ches/papers/berferd.pdf`

Cinar, Y., Melvin, H., Pocta, P., and Alahmadi, M. Containerisation in Multimedia Research Test Beds. In *Proceedings of the 5th ISCA/DEGA Workshop on Perceptual Quality of Systems (PQS 2016)*, pages 142–145. 2016. doi:10.21437/PQS.2016-30.

Cisco Systems. Firepower Management Center Configuration Guide. Cisco, 170 West Tasman Drive San Jose, CA 95134-1706 USA, June 2018. Last accessed: 2018-10-29.
URL `https://www.cisco.com/c/en/us/td/docs/security/firepower/620/configuration/guide/fpmc-config-guide-v62.pdf`

Cohen, A. J. Simulating Virtual Circuits in Mobile Packet Radio Networks. Bachelor's thesis, Massachusetts Institute of Technology, June 1986.
URL `http://hdl.handle.net/1721.1/14904`

Corbató, F. J. and Vyssotsky, V. A. Introduction and Overview of the Multics System. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, pages 185–196. ACM, November 1965. doi:10.1145/1463891.1463912.

Cotton, M., Eggert, L., Touch, J., Westerlund, M., and Cheshire, S. Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. RFC 6335 (Best Current Practice), August 2011. doi:10.17487/RFC6335.

Crall, C. Configuration in a World of Containers. In *2014 USENIX Release Engineering Summit West*. USENIX Association, Seattle, WA, USA, 2014. Last accessed: 2019-03-18.
URL `https://www.usenix.org/conference/ures14west/summit-program/presentation/configuration-world-containers`

Davis, J. and Magrath, S. A Survey of Cyber Ranges and Testbeds. Technical Report ADA594524, Defence Science and Technology Organisation, Cyber and Electronic Warfare Division, Edinburgh, Australia, October 2013. Last accessed: 2019-03-23.
URL `http://www.dtic.mil/dtic/tr/fulltext/u2/a594524.pdf`

Davis, M. R. Categorising Network Telescope Data Using Bigdata Enrichment Techniques. Master's thesis, Rhodes University, Grahamstown, South Africa, January 2019.

Davoli, R. VDE: Virtual Distributed Ethernet. In *First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 213–220. Feb 2005. doi:10.1109/TRIDNT.2005.38.

de Berlaere, T. V. G. Containerised Cybersecurity Lab for Rapid and Secure Evaluation of Threat Mitigation Tactics. Master's thesis, Ghent University, 2018.

DeLooze, L. L., McKean, P., Mostow, J. R., and Graig, C. Incorporating simulation into the computer security classroom. In *34th Annual Frontiers in Education, 2004. FIE 2004.*, pages S1F/13–S1F/18 Vol. 3. Oct 2004. ISSN 0190-5848. doi:10.1109/FIE.2004.1408699.

Dennis, J. B. Segmentation and the Design of Multiprogrammed Computer Systems. *Journal of the ACM*, 12(4):589–602, October 1965. doi:10.1145/321296.321310.

Dike, J. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference*. The USENIX Association, The USENIX Association, Atlanta, Georgia, USA, October 2000.
URL `https://www.usenix.org/legacy/publications/library/proceedings/als00/2000papers/papers/full_papers/dike/dike_html/`

Dike, J. User Mode Linux. Prentice Hall, 1st edition, April 2006. ISBN 0-13-186505-6.
URL `http://www.informit.com/store/user-mode-linux-9780131865051`

Dovrolis, C., Gummadi, K., Kuzmanovic, A., and Meinrath, S. D. Measurement Lab: Overview and an Invitation to the Research Community. *SIGCOMM Computer Communication Review*, 40(3):53–56, June 2010. ISSN 0146-4833. doi:10.1145/1823844.1823853.

Eckersley, P. How Unique Is Your Web Browser? In *Privacy Enhancing Technologies*, pages 1–18. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14527-8.

Edwards, S., Liu, X., and Riga, N. Creating Repeatable Computer Science and Networking Experiments on Shared, Public Testbeds. *SIGOPS Operating System Review*, 49(1):90–99, January 2015. ISSN 0163-5980. doi:10.1145/2723872.2723884.

Ely, D., Savage, S., and Wetherall, D. Alpine: A User-Level Infrastructure for Network Protocol Development. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*. USENIX, San Francisco, CA, United Stated of America, March 2001.

Falkoff, A. D. The IBM family of APL systems. *IBM Systems Journal*, 30(4):416–432, 1991. ISSN 0018-8670. doi:10.1147/sj.304.0416.

Fall, K. Network Emulation in the VINT/NS Simulator. In *Proceedings IEEE International Symposium on Computers and Communications (Cat. No.PR00250)*, pages 244–250. July 1999. doi:10.1109/ISCC.1999.780820.

Fan, W., Fernández, D., and Du, Z. Adaptive and Flexible Virtual Honeynet. In **Boumerdassi, S., Bouzefrane, S., and Renault, É.**, editors, *Mobile, Secure, and Programmable Networking*, pages 1–17. Springer International Publishing, Cham, 2015. ISBN 978-3-319-25744-0.

FAS Military Analysis Network. ES310 Introduction to Naval Weapons Engineering: SONAR Systems. Online, January 1998. Last accessed: 2018-10-29.
URL `https://fas.org/man/dod-101/navy/docs/es310/asw_sys/asw_sys.htm`

Fazio, A. D. and Minasi, P. VisualNetkit. Online, March 2009. Last accessed: 2018-03-27.
URL `https://code.google.com/archive/p/visual-netkit/`

Fernandez, D., de Miguel, T., and Galan, F. Study and emulation of IPv6 Internet-exchange-based addressing models. *IEEE Communications Magazine*, 42(1):105–112, Jan 2004. ISSN 0163-6804. doi:10.1109/MCOM.2004.1262169.

Fernández, D., Cordero, A., Somavilla, J., Rodriguez, J., Corchero, A., Tarrafeta, L., and Galán, F. Distributed virtual scenarios over multi-host Linux environments. In *2011 5th International DMTF Academic Alliance Workshop on Systems and Virtualization Management: Standards and the Cloud (SVM)*, pages 1–8. Oct 2011. doi:10.1109/SVM.2011.6096467.

Fillot, C. Cisco 7200 Simulator. Online, 2005. Last Accessed: 2019-03-18.
URL `https://web.archive.org/web/20130430014653/http://www.ipflow.utc.fr/blog/`

Floyd, S. and Paxson, V. Difficulties in Simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, August 2001. ISSN 1063-6692. doi:10.1109/90.944338.

Fyodor. Remote OS detection via TCP/IP Stack FingerPrinting. *Phrack Magazine*, Volume 8(54), December 1998. Last accessed: 2018-10-29.
URL `http://phrack.org/issues/54/9.html`

Gadge, J. and Patil, A. A. Port Scan Detection. In *16th IEEE International Conference on Networks*, pages 1–6. Dec 2008. ISSN 1531-2216. doi:10.1109/ICON.2008.4772622.

Galan, F., Fernandez, D., Ruiz, J., Walid, O., and de Miguel, T. Use of virtualization tools in computer network laboratories. In *Information Technology Based Proceedings of the FIfth International Conference onHigher Education and Training, 2004. ITHET 2004.*, pages 209–214. May 2004. doi:10.1109/ITHET.2004.1358165.

Gluck, S. E. The electronic discrete variable computer. *Electrical Engineering*, 72, February 1953. doi:10.1109/ee.1953.6438502.

Goldberg, R. P. Architecture of Virtual Machines. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 74–112. ACM, New York, NY, USA, July 1973. doi:10.1145/800122.803950.

Goldberg, R. P. Survey of Virtual Machine Research. *Computer*, 7(6):34–45, June 1974. ISSN 0018-9162. doi:10.1109/MC.1974.6323581.

Goldstine, H. H. and Goldstine, A. The Electronic Numerical Integrator and Computer (ENIAC). *Mathematical Tables and Other Aids to Computation*, 2(15):97–110, July 1946. doi:10.2307/2002620.

Graham, R. M. Protection in an Information Processing Utility. *Communications of the ACM*, 11(5):365–369, May 1968. ISSN 0001-0782. doi:10.1145/363095.363146.

Grant, T. Modelling Network-Enabled C2 using Multiple Agents and Social Networks. In *Proceedings of the Social Networks and Multi-Agent Systems Symposium (SNAMAS-09)*, pages 13–18. The Society for the Study of Artificial Intelligence and the Simulation of Behaviour, Edinburgh, Scotland, April 2009.

Grant, T. and Kooter, B. Comparing OODA & Other Models as Operational View C2 Architecture. *International Command and Control Research and Technology Symposium*, June 2005.

Grant, T. J., Venter, H. S., and Eloff, J. H. P. Simulating Adversarial Interactions Between Intruders and System Administrators Using OODA-RR. In *Proceedings of the 2007 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, SAICSIT '07, pages 46–55. ACM, Port Elizabeth, South Africa, 2007. ISBN 978-1-59593-775-9. doi:10.1145/1292491.1292497.

Gries, S., Meyer, O., Ollesch, J., Wessling, F., Hesenius, M., and Gruhn, V. Developing a Convenient and Fast to Deploy Simulation Environment for Cyber-Physical Systems. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1551–1552. July 2018. ISSN 2575-8411. doi:10.1109/ICDCS.2018.00166.

Gunduz, M. Z. and Das, R. A comparison of cyber-security oriented testbeds for IoT-based smart grids. In *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–6. March 2018. doi:10.1109/ISDFS.2018.8355329.

Guo, L. and Lee, J. Y. B. On TCP Simulation Fidelity in Ns-2. In *Proceedings of the 14th ACM International Symposium on QoS and Security for Wireless and Mobile Networks*, Q2SWinet'18, pages 55–62. ACM, New York, NY, USA, 2018. ISBN 978-1-4503-5963-4. doi:10.1145/3267129.3267132.

Hack Story. Savage. Online, August 2009. Last accessed 2019-10-23.
URL `https://hackstory.net/Savage`

Hallyn, S. E. uts namespaces: Introduction. Online, April 2006. Last accessed 2019-10-23.
URL `https://lwn.net/Articles/179345/`

Hammad, E., Ezeme, M., and Farraj, A. Implementation and development of an offline co-simulation testbed for studies of power systems cyber security and control verification. *International Journal of Electrical Power & Energy Systems*, 104:817 – 826, 2019. ISSN 0142-0615. doi:10.1016/j.ijepes.2018.07.058.

Hammons, J. Windows Subsystem for Linux Overview. Online, April 2016. Last accessed 2019-02-25.
URL `https://blogs.msdn.microsoft.com/wsl/2016/04/22/windows-subsystem-for-linux-overview/`

Handigol, N., Heller, B., Jeyakumar, V., Lantz, B., and McKeown, N. Reproducible Network Experiments Using Container-based Emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 253–264. ACM, Nice, France, 2012. ISBN 978-1-4503-1775-7. doi:10.1145/2413176.2413206.

Hansman, S. A Taxonomy of Network and Computer Attack Methodologies. Master's thesis, University of Canterbury, Christchurch, New Zealand, November 2003.

Heidemann, J., Mills, K., and Kumar, S. Expanding confidence in network simulations. *IEEE Network*, 15(5):58–63, Sep 2001. ISSN 0890-8044. doi:10.1109/65.953234.

Heilmann, F. and Fohler, G. Impact of Time-triggered Transmission Window Placement on Rate-constrained Traffic in TTEthernet Networks. *SIGBED Rev.*, 15(3):7–12, August 2018. ISSN 1551-3688. doi:10.1145/3267419.3267420.

Heller, B. Reproducible Network Research with High-Fidelity Emulation. Ph.D. thesis, Stanford University, 2013.

Hemminger, S. Network Emulation with NetEm. In *Linux Conference Australia*. Canberra, Australia, April 2005. Last accessed: 2019-09-15.
URL `https://www.rationali.st/blog/files/20151126-jittertrap/netem-shemminger.pdf`

Herbert, A. and Irwin, B. A kernel-driven framework for high performance internet routing simulation. In *2013 Information Security for South Africa*, pages 1–6. Aug 2013. ISSN 2330-9881. doi:10.1109/ISSA.2013.6641048.

Hjelmvik, E. Passive OS Fingerprinting. Online, November 2011. Last accessed: 2018-10-29.
URL `https://www.netresec.com/index.ashx?page=Blog&month=2011-11&post=Passive-OS-Fingerprinting`

Huang, X. W., Sharma, R., and Keshav, S. The ENTRAPID Protocol Development Environment. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, volume 3, pages 1107–1115. IEEE, New York, NY, United States of America, March 1999. doi:10.1109/INFCOM.1999.751666.

Huber, K. E. Host-Based Systemic Network Obfuscation System for Windows. Master's thesis, Air Force Intitute of Technology, Wright-Patterson Air Force Base, Ohio, June 2011.
URL `https://apps.dtic.mil/docs/citations/ADA545766`

Hunter, S. O. A Framework for Malicious Host Fingerprinting Using Distributed Network Sensors. Master's thesis, Rhodes University, Grahamstown, South Africa, December 2017.

Hunter, S. O., Stalmans, E., Irwin, B., and Richter, J. Remote Fingerprinting and Multisensor Data Fusion. In *2012 Information Security for South Africa*, pages 1–8. IEEE, Aug 2012. ISSN 2330-9881. doi:10.1109/ISSA.2012.6320449.

Hutchins, E. M., Cloppert, M. J., and Amin, R. M. Leading Issues in Information Warfare and Security Research, volume 1, chapter Intelligence-driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains, pages 78–104. Academic Publishing International, 2011. ISBN 978-1-908272-08-9.

Hwang, K., Fox, G. C., and Dongarra, J. J. Distributed and Cloud Computing: From Parallel Processing to the Internet of Things. Morgan Kaufmann, 2013. ISBN 978-0-12-385880-1.

IBM. IBM Virtual Machine Facility/370: Introduction. Internation Business Machines Corporation, 1st edition, July 1972a.

IBM. IBM Virtual Machine Facility/370: Planning Guide. Internation Business Machines Corporation, 1st edition, August 1972b.

IEEE 802.1D-2004. Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges. June 2004. doi:10.1109/IEEESTD.2004.94569.

Iguchi-Cartigny, J. Netkit-NG. Online, 2014.
URL `https://netkit-ng.github.io/`

Intel Corporation. Intel® 64 and IA-32 Architecutures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide. Online, January 2019. Last accessed: 2019-03-18.
URL `https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html`

Irwin, B. A Network Telescope Perspective of the Conficker Outbreak. In *2012 Information Security for South Africa*, pages 1–8. IEEE, August 2012. ISSN 2330-9881. doi:10.1109/ISSA.2012.6320455.

Irwin, B. A Source Analysis of the Conficker Outbreak from a Network Telescope. *SAIEE Africa Research Journal*, 104(2):38–53, June 2013. ISSN 1991-1696. doi:10.23919/SAIEE.2013.8531865.

Janczewski, L. J. and Colarik, A. M. Cyber Warfare and Cyber Terrorism, chapter Introduction to Cyber Warfare and Cyber Terrorism, pages xiii–xxx. IGI Global, 2008. ISBN 978-1-59140-991-5.

Johnson, D., Grubb, E., and Eide, E. Supporting Docker in Emulab-Based Network Testbeds. In *11th USENIX Workshop on Cyber Security Experimentation and Test (CSET 18)*. USENIX Association, Baltimore, MD, 2018.
URL `https://www.usenix.org/conference/cset18/presentation/johnson`

Jones, M. T. Platform emulation with Bochs. Technical report, IBM developerWorks, January 2011. Last accessed: 2019-03-10.
URL `https://www.ibm.com/developerworks/library/l-bochs/l-bochs-pdf.pdf`

Joy, A. M. Performance Comparison Between Linux Containers and Virtual Machines. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 342–346. March 2015. doi:10.1109/ICACEA.2015.7164727.

Kamp, P. and Watson, R. N. M. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*. Maastricht, The Netherlands, May 2000.

Kaur, R. Hardening Linux Operating System to Mask Against Fingerprinting. Master's thesis, Thapar University, July 2009.

King, S. T., Dunlap, G. W., and Chen, P. M. Operating System Support for Virtual Machines. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference*, pages 71–84. USENIX Association, USENIX Association, San Antonio, TX, USA, June 2003. Last accessed: 2019-03-18.
URL `https://www.usenix.org/legacy/events/usenix03/tech/king.html`

Knapp, E. D. and Langill, J. T. Industrial Network Security: Securing Critical Infrastructure Networks for Smart Grid, SCADA, and Other Industrial Control Systems. Syngress, August 2014. ISBN 978-1597496452.

Koons, F. and Lubkin, S. Conversion of Numbers from Decimal to Binary Form in the EDVAC. *Mathematical Tables and Other Aids to Computation*, 3(26):427–431, 1949. ISSN 08916837.

Kuman, S., Groš, S., and Mikuc, M. An experiment in using IMUNES and Conpot to emulate honeypot control networks. In *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1262–1268. May 2017. doi:10.23919/MIPRO.2017.7973617.

Lamps, J., Babu, V., Nicol, D. M., Adam, V., and Kumar, R. Temporal Integration of Emulation and Network Simulators on Linux Multiprocessors. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 28(1):1:1–1:25, January 2018. ISSN 1049-3301. doi:10.1145/3154386.

Lantz, B., Heller, B., and McKeown, N. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 19:1–19:6. ACM, Monterey, California, 2010. ISBN 978-1-4503-0409-2. doi:10.1145/1868447.1868466.

Lawton, K. P. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 1996(29), September 1996. ISSN 1075-3583. Last accessed: 2019-03-18.
URL `https://www.linuxjournal.com/article/1310`

Lemay, A., Fernandez, J., and Knight, S. An Isolated Virtual Cluster for SCADA Network Security Research. In *Proceedings of the 1st International Symposium on ICS & SCADA Cyber Security Research 2013*, ICS-CSR 2013, pages 88–96. BCS, UK, 2013. ISBN 978-1-780172-32-3.

Lindholm, T. and Yellin, F. The Java™ Virtual Machine Specification. Addison-Wesley, 1997. ISBN 0-201-63452-X.

Lippmann, R., Fried, D., Piwowarski, K., and Streilein, W. Passive Operating System Identification from TCP/IP Packet Headers. In *ICDM Workshop on Data Mining for Computer Security*, pages 40–49. 2003.
URL `https://www.biostat.wisc.edu/~page/ICDM_Workshops/dmsec03-workshop.pdf`

Liu, K., Liu, P., Wang, C., and Fu, T. Remote Virtual Experiment and Simulation Platform for IoT Related Courses. In *2018 13th International Conference on Computer Science Education (ICCSE)*, pages 1–6. Aug 2018. ISSN 2473-9464. doi:10.1109/ICCSE.2018.8468820.

Loddo, J.-V. and Saiu, L. Status Report: Marionnet or "How to Implement a Virtual Network Laboratory in Six Months and Be Happy". In *Proceedings of the 2007 Workshop on Workshop on ML*, ML '07, pages 59–70. ACM, Freiburg, Germany, 2007. ISBN 978-1-59593-676-9. doi:10.1145/1292535.1292545.

Loddo, J.-V. and Saiu, L. Marionnet: A Virtual Network Laboratory and Simulation Tool. In *Industry Track to The First International Conference on Simulation Tools and*

Techniques for Communications, Networks and Systems. ACM, 5 2008. doi:10.4108/ICST.SIMUTOOLS2008.3102.

Lyon, G. F. Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning. Insecure, USA, 2009. ISBN 9780979958717.

Marks, P. Dot-dash-diss: The gentleman hacker's 1903 lulz. *New Scientist*, 212(2844/2845):48–49, December 2011. ISSN 70989-30690.

Martínez-Casanueva, I. D. Development, Deployment and Analysis of a Software Defined Networking Test Environment for Network Traffic Monitoring. Master's thesis, Universidad Politécnica de Madrid, February 2018.
URL `http://oa.upm.es/49587/`

Mazur, D. A Validation Method of a Real Network Device Model in the Riverbed Modeler Simulator. In *Computer Networks*, pages 26–39. Springer International Publishing, 2018. ISBN 978-3-319-92459-5.

McClure, S., Scambray, J., and Kurtz, G. Hacking Exposed: Network Security Secrets and Solutions. McGraw-Hill Education, 7th edition, August 2012. ISBN 978-0071780285.

McGeer, R., Berman, M., Elliott, C., and Ricci, R., editors. The GENI Book. Springer International Publishing, 2016. ISBN 978-3-319-33767-8. doi:10.1007/978-3-319-33769-2.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, March 2008. ISSN 0146-4833. doi:10.1145/1355734.1355746.

Medeiros, J. P. S., Júnior, A. M. B., and Pires, P. M. Using Intelligent Techniques to Extend the Applicability of Operating System Fingerprint Databases. *Journal of Information Assurance and Security*, 5:554–560, 01 2010. ISSN 1554-1010.

Menage, P., Jackson, P., and Lameter, C. cgroups. Online, October 2008. Last accessed: 2019-03-25.
URL `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`

Mercan, S. Cloud Assisted Overlay Routing. In *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 410–414. July 2018. ISSN 2165-8536. doi:10.1109/ICUFN.2018.8436951.

Microsoft Corporation. Windows Subsystem for Linux Documentation. Online, November 2016. Last accessed 2019-10-23.
URL `https://docs.microsoft.com/en-us/windows/wsl/about`

Mirkovic, J., Bartlett, G., and Blythe, J. DEW: Distributed Experiment Workflows. In *11th USENIX Workshop on Cyber Security Experimentation and Test (CSET 18)*. USENIX Association, Baltimore, MD, 2018.
URL `https://www.usenix.org/conference/cset18/presentation/mirkovic`

Mirkovic, J. and Benzel, T. Teaching Cybersecurity with DeterLab. *IEEE Security Privacy*, 10(1):73–76, Jan 2012. ISSN 1540-7993. doi:10.1109/MSP.2012.23.

Mirkovic, J., Benzel, T. V., Faber, T., Braden, R., Wroclawski, J. T., and Schwab, S. The DETER project: Advancing the science of cyber security experimentation and test. In *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, pages 1–7. Nov 2010. doi:10.1109/THS.2010.5655108.

Mitchell, R. and Healy, B. A game theoretic model of computer network exploitation campaigns. In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 431–438. Jan 2018. doi:10.1109/CCWC.2018.8301630.

Muelas, D., Ramos, J., and de Vergara, J. E. L. Software-Driven Definition of Virtual Testbeds to Validate Emergent Network Technologies. *Information*, 9(2):45, February 2018. ISSN 2078-2489. doi:10.3390/info9020045.

Nachenberg, C. Hacking. Online, October 2002. Last accessed: 2019-01-14.
URL `https://www.encyclopedia.com/science-and-technology/computers-and-electrical-engineering/computers-and-computing/hacking`

Nussbaum, L. Testbeds Support for Reproducible Research. In *Proceedings of the Reproducibility Workshop*, Reproducibility '17, pages 24–26. ACM, New York, NY, USA, 2017. ISBN 978-1-4503-5060-0. doi:10.1145/3097766.3097773.

Nussbaum, L. Experiment Data Management. In *Global Experimentation for Future Internet*. Tokyo, Japan, October 2018. Last accessed: 2019-03-23.
URL `https://hal.inria.fr/hal-01944472`

Ohrhallinger, K. P. Virtual Private Systems for FreeBSD. In *Proceedings of EuroBSDCon 2010*. Karlsruhe, Germany, October 2010. Last accessed: 2019-09-12.
URL https://2010.eurobsdcon.org/fileadmin/fe_user/klaus/37R5uB.pdf

Ornaghi, A. and Valleri, M. Ettercap Home Page. 2019. Last accessed 2019-01-16.
URL http://www.ettercap-project.org/about.html

Pastor, V., Díaz, G., and Castro, M. State-of-the-art simulation systems for information security education, training and awareness. In *IEEE Global Engineering Education Conference*, pages 1907–1916. April 2010. ISSN 2165-9559. doi: 10.1109/EDUCON.2010.5492435.

Paxson, V., Allman, M., Dawson, S., Fenner, W., Griner, J., Heavens, I., Lahey, K., Semke, J., and Volz, B. Known TCP Implementation Problems. RFC 2525 (Informational), March 1999. doi:10.17487/RFC2525.

Pediaditakis, D., Rotsos, C., and Moore, A. W. Faithful Reproduction of Network Experiments. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 41–52. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-2839-5. doi:10.1145/2658260.2658274.

Penneman, N., Kudinskas, D., Rawsthorne, A., De Sutter, B., and De Bosschere, K. Formal Virtualization Requirements for the ARM Architecture. *Journal of Systems Architecture: the EUROMICRO Journal*, 59(3):144–154, March 2013. ISSN 1383-7621. doi:10.1016/j.sysarc.2013.02.003.
URL http://dx.doi.org/10.1016/j.sysarc.2013.02.003

Petazzoni, J. and LeClaire, N. Introduction to Docker. November 2014.
URL https://www.usenix.org/conference/lisa14/conference-program/presentation/turnbull

Peterson, L., Anderson, T., Culler, D., and Roscoe, T. A Blueprint for Introducing Disruptive Technology into the Internet. *SIGCOMM Computer Communication Review*, 33(1):59–64, January 2003. ISSN 0146-4833. doi:10.1145/774763.774772.

Peterson, L., Bavier, A., and Bhatia, S. VICCI: A Programmable Cloud-Computing Research Testbed. Techreport TR-912-11, Princeton University, September 2011. Last accessed: 2019-03-23.
URL http://www.cs.princeton.edu/research/techreps/TR-912-11

Pfaff, B., Pettit, J., Koponen, T., Amidon, K., Casado, M., and Shenker, S. Extending Networking into the Virtualization Layer. In *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*. New York, NY, United States of America, October 2009.

Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., and Casado, M. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130. USENIX Association, Oakland, CA, 2015. ISBN 978-1-931971-218.
URL `https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff`

Pike, R., Presotto, D., Thompson, K., Trickey, H., and Winterbottom, P. The Use of Name Spaces in Plan 9. In *Proceedings of the 5th ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring*. ACM, Mont Saint-Michel, France, 1992. doi:10.1145/506378.506413.

Pizzonia, M. and Rimondini, M. Netkit: Easy Emulation of Complex Networks on Inexpensive Hardware. In *Proceedings of the 4th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, TridentCom '08, pages 7:1–7:10. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, ICST, Brussels, Belgium, Belgium, 2008. ISBN 978-963-9799-24-0.

Plummer, D. An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826 (Internet Standard), November 1982. doi:10.17487/RFC0826. Updated by RFCs 5227, 5494.

Popek, G. J. and Goldberg, R. P. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7):412–421, July 1974. ISSN 0001-0782. doi:10.1145/361011.361073.

Postel, J. Internet Control Message Protocol. RFC 792 (Internet Standard), September 1981. doi:10.17487/RFC0792. Updated by RFCs 950, 4884, 6633, 6918.

Price, D. and Tucker, A. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proceedings of the 18th USENIX Large Installation System*

Administration Conference, pages 241–254. USENIX Association, Atlanta, GA, USA, November 2004. Last accessed: 2019-03-18.

Pujeri, U. and Palanisamy, V. Survey of Various Open Source Network Simulators. *International Journal of Science and Research (IJSR)*, 3(12):2064–2079, December 2014. ISSN 2319-7064.

Puljiz, Z. and Mikuc, M. IMUNES Based Distributed Network Emulator. In *2006 International Conference on Software in Telecommunications and Computer Networks*, pages 198–203. Sept 2006. doi:10.1109/SOFTCOM.2006.329743.

Purdy, G. N. Linux iptables Pocket Reference. O'Reilly, September 2004. ISBN 978-0596005696.

Qu, W. Congestion Control Tuning of the QUIC Transport Layer Protocol. Bachelor's thesis, Universitat Politecnica de Catalunya, April 2018.

Queiroz, C., Mahmood, A., Hu, J., Tari, Z., and Yu, X. Building a SCADA Security Testbed. In *2009 Third International Conference on Network and System Security*, pages 357–364. Oct 2009. doi:10.1109/NSS.2009.82.

Rampfl, S. Network Simulation and its Limitations. In **Carle, G., Pahl, M.-O., Raumer, D., Schwaighofer, L., Baumgarten, U., and Sollner, C.**, editors, *Proceedings of the Seminars Future Internet (FI), Innovative Internet Technologies and Mobile Communications (IITM), and Autonomous Communication Networks (ACN)*, pages 57–64. Chair for Network Architectures and Services Department of Computer Science, Technische Universität München, Munich, Germany, August 2013.
URL `https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2013-08-1.pdf`

Rash, M. Detecting Suspect Traffic. *Linux Journal*, 2001(91):22–25, November 2001. ISSN 1075-3583.

Richardson, D. W., Gribble, S. D., and Kohno, T. The Limits of Automatic OS Fingerprint Generation. In *Proceedings of the 3rd ACM Workshop on Artificial Intelligence and Security*, AISec '10, pages 24–34. ACM, Chicago, IL, USA, 2010. ISBN 978-1-4503-0088-9. doi:10.1145/1866423.1866430.

Riga, N., Thomas, V., Maglaris, V., Grammatikou, M., and Anifantis, E. Virtual Laboratories. In *Proceedings of the 7th International Conference on Computer Supported Education - Volume 1*, CSEDU 2015, pages 516–521. SCITEPRESS - Science

and Technology Publications, Lda, Portugal, 2015. ISBN 978-989-758-107-6. doi:10.5220/0005496105160521.

Rimondini, M. Emulation of Computer Networks with Netkit. Technical Report RT-DIA-113-2007, Dipartimento di Informatica e Automazione, Università degli Studi Roma Tre, Via della Vasca Navale, 79 – 00146 Roma, Italy, January 2007.

Riondato, M. FreeBSD Handbook, chapter 14: Jails. 2020. Last accessed: 2020-01-03. URL `https://www.freebsd.org/doc/handbook/`

Rosen, R. Resource Management: Linux Kernel namespaces and cgroups. Online, May 2013. Last accessed: 2019-03-18.
URL `www.haifux.org/lectures/299/netLec7.pdf`

Roualland, G. and Saffroy, J.-M. IP Personality. 2002.
URL `http://ippersonality.sourceforge.net`

Salame, W. How to use Ettercap. Online, April 2019. Last accessed 2019-10-23.
URL `https://www.kalitut.com/2019/04/how-to-use-ettercap.html`

Salopek, D., Vasić, V., and Mikuc, M. Security research and learning environment based on scalable network emulation. *Tehnički vjesnik: znanstveno-stručni časopis tehničkih fakulteta Sveučilišta u Osijeku*, 24(Supplement 2):535, 2017.

Saunders, J. H. The Case for Modeling and Simulation of Information Security. Online, October 2001. Last accessed: 2019-03-23.
URL `http://www.johnsaunders.com/papers/securitysimulation.htm`

Schmidt, M., Stovkmayer, A., Heimgaertner, F., and Menth, M. A Semi-Virtualized Testbed Cluster with a Centralized Server for Networking Education. In *2018 30th International Teletraffic Congress (ITC 30)*, volume 01, pages 200–208. Sep. 2018. doi:10.1109/ITC30.2018.00038.

Schroeder, M. D. and Saltzer, J. H. A Hardware Architecture for Implementing Protection Rings. *Communications of the ACM*, 15(3):157–170, March 1972. ISSN 0001-0782. doi:10.1145/361268.361275.

Schultze, E. Thinking Like a Hacker. Online, March 2002. Last accessed: 2019-03-24.
URL `http://pdf.textfiles.com/security/thinkhacker.pdf`

Sharan, R. The Five Stages of Ethical Hacking. Online, December 2010. Last accessed: 2019-03-24.

URL http://hack-o-crack.blogspot.com/2010/12/five-stages-of-ethical-hacking.html

Sharif, M. and Sadeghi-Niaraki, A. Ubiquitous Sensor Network Simulation and Emulation Environments: A Survey. *Journal of Network and Computer Applications*, 93:150–181, 2017. ISSN 1084-8045. doi:10.1016/j.jnca.2017.05.009.
URL https://www.sciencedirect.com/science/article/pii/S1084804517302059

Sharma, S., Hussain, A., and Saran, H. Towards repeatability & verifiability in networking experiments: A stochastic framework. *Journal of Network and Computer Applications*, 81:12 – 23, 2017. ISSN 1084-8045. doi:10.1016/j.jnca.2016.07.001.

Shuja, J., Gani, A., Bilal, K., Khan, A. U. R., Madani, S. A., Khan, S. U., and Zomaya, A. Y. A Survey of Mobile Device Virtualization: Taxonomy and State of the Art. *ACM Computing Surveys*, 49(1):1:1–1:36, April 2016. ISSN 0360-0300. doi:10.1145/2897164.

Sivaramakrishnan, S. R., Mikovic, J., Kannan, P. G., Choon, C. M., and Sklower, K. Enabling SDN Experimentation in Network Testbeds. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks #38; Network Function Virtualization*, SDN-NFVSec '17, pages 7–12. ACM, New York, NY, USA, 2017. ISBN 978-1-4503-4908-6. doi:10.1145/3040992.3040996.

Smith, J. E. and Nair, R. The Architecture of Virtual Machines. *Computer*, 38(5):32–38, May 2005. ISSN 0018-9162. doi:10.1109/MC.2005.173.

Song, J., Cadar, C., and Pietzuch, P. SYMBEXNET: Testing Network Protocol Implementations with Symbolic Execution and Rule-Based Specifications. *IEEE Transactions on Software Engineering*, 40(7):695–709, July 2014. ISSN 0098-5589. doi:10.1109/TSE.2014.2323977.

Sophos. Sophos UTM: Understanding Portscan Detection. Online, May 2018. Last accessed: 2018-10-29.
URL https://community.sophos.com/kb/en-us/115153

Spangler, R. Analysis of Remote Active Operating System Fingerprinting Tools. Technical report, University of Wisconsin - Whitewater, May 2003. Last accessed: 2019-03-24.
URL https://www.cs.bgu.ac.il/~denat/documents/analysis_operating_system.pdf

Spitzner, L. Passive Fingerprinting. Online, May 2000. Last accessed: 2019-03-24. URL https://www.symantec.com/connect/articles/passive-fingerprinting

Stopforth, R. Techniques and Countermeasures of TCP/IP OS Fingerprinting on Linux Systems. Master's thesis, University of KwaZulu-Natal, December 2007.

Swart, I. P. Pro-active Visualization of Cyber Security on a National Level: A South African Case Study. Ph.D. thesis, Rhodes University, Grahamstown, South Africa, January 2015. URL http://research.ict.ru.ac.za/SNRG/Theses/Swart%202015%20Phd.pdf

Syed, A. Realistic Traffic Shaping in the Dummynet Link Emulator. Master's thesis, University of Utah, 2014.

Syed, A. and Ricci, R. Realistic Packet Reordering for Network Emulation and Simulation. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 25:1–25:7. ACM, New York, NY, USA, 2015. ISBN 978-1-4503-3412-9. doi:10.1145/2716281.2836110.

Tanabe, K., Hosoya, R., and Saito, T. Combining Features in Browser Fingerprinting. In *Advances on Broadband and Wireless Computing, Communication and Applications*, pages 671–681. Springer International Publishing, 2019. ISBN 978-3-030-02613-4.

Teumim, D. J. Industrial Network Security. International Society of Automation, 2nd edition, January 2010. ISBN 978-1-936-00707-3.

Thompson, M. F. and Irvine, C. E. Individualizing Cybersecurity Lab Exercises with Labtainers. *IEEE Security & Privacy*, 16(2):91–95, March 2018. ISSN 1540-7993. doi:10.1109/MSP.2018.1870862.

Torvalds, L. Linux 5.0. Online, March 2019. Last accessed 2019-10-23. URL https://lkml.org/lkml/2019/3/3/236

Torvals, L. Linux 4.0-rc1 out.. Online, February 2015. Last accessed 2019-10-23. URL http://lkml.iu.edu/hypermail/linux/kernel/1502.2/04059.html

Torvals, L. Linux 4.9. Online, December 2016. Last accessed 2019-10-23. URL https://lkml.org/lkml/2016/12/11/102

Trowbridge, C. An Overview of Remote Operating System Fingerprinting. Online, July 2003. Last accessed: 2019-03-24.
URL `https://www.sans.org/reading-room/whitepapers/testing/overview-remote-operating-system-fingerprinting-1231`

Turkey, J. W. Exploratory Data Analysis. Addison-Wesley Publishing Company, 1977. ISBN 978-0201076165.

Türpe, S. and Eichler, J. Testing Production Systems Safely: Common Precautions in Penetration Testing. In *2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, pages 205–209. September 2009. doi:10.1109/TAICPART.2009.17.

Tutănescu, I. and Sofron, E. Anatomy and Types of Attacks Against Computer Networks. In *Proceedings of the Second RoEduNet International Conference*, pages 264–270. June 2003. Last accessed: 2019-03-24.
URL `http://193.226.6.174/roedunet2003/html/pgid66_site_EN.html`

Urdaneta, M., Lemay, A., Saunier, N., and Fernandez, J. A Cyber-Physical Testbed for Measuring the Impacts of Cyber Attacks on Urban Road Networks. In **Staggs, J. and Shenoi, S.**, editors, *Critical Infrastructure Protection XII*, pages 177–196. Springer International Publishing, 2018. ISBN 978-3-030-04537-1.

Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostić, D., Chase, J., and Becker, D. Scalability and Accuracy in a Large-scale Network Emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, December 2002. ISSN 0163-5980. doi:10.1145/844128.844154.
URL `http://doi.acm.org/10.1145/844128.844154`

van Heerden, R., Pieterse, H., Burke, I., and Irwin, B. Developing a Virtualised Testbed Environment in Preparation for testing of Network Based Attacks. In *2013 International Conference on Adaptive Science and Technology*, pages 1–8. Nov 2013. ISSN 2326-9448. doi:10.1109/ICASTech.2013.6707509.

van Heerden, R. P. A Formalised Ontology for Network Attack Classification. Ph.D. thesis, Rhodes University, Grahamstown, South Africa, 2014.

Vaughan-Nichols, S. J. New Approach to Virtualization Is a Lightweight. *Computer*, 39(11):12–14, Nov 2006. ISSN 0018-9162. doi:10.1109/MC.2006.393.

Veksler, V. D., Buchler, N., Hoffman, B. E., Cassenti, D. N., Sample, C., and Sugrim, S. Simulations in Cyber-Security: A Review of Cognitive Modeling of Network Attackers, Defenders, and Users. *Frontiers in Psychology*, 9(691), 2018. ISSN 1664-1078. doi:10.3389/fpsyg.2018.00691.

Veysset, F., Courtay, O., Heen, O., Team, I. *et al.* New Tool and Technique for Remote Operating System Fingerprinting. *Intranode Software Technologies*, 4, 2002.

von Neumann, J. First Draft of a Report on the EDVAC. Technical Report, University of Pennsylvania, June 1945.

Vykopal, J., Vizvary, M., Oslejsek, R., Celeda, P., and Tovarnak, D. Lessons Learned from Complex Hands-on Defence Exercises in a Cyber Range. In *2017 IEEE Frontiers in Education Conference*, pages 1–8. Oct 2017. doi:10.1109/FIE.2017.8190713.

Wang, K. Frustrating OS Fingerprinting with Morph, 2004. DEFCON 12. URL `https://www.defcon.org/images/defcon-12/dc-12-presentations/Wang/dc-12-wang.pdf`

Wang, S. Y. and Kung, H. T. A Simple Methodology for Constructing Extensible and High-Fidelity TCP/IP Network Simulators. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, volume 3, pages 1134–1143. IEEE, New York, NY, United States of America, March 1999. doi:10.1109/INFCOM.1999.751669.

Wette, P., Dräxler, M., and Schwabe, A. MaxiNet: Distributed emulation of software-defined networks. In *2014 IFIP Networking Conference*, pages 1–9. June 2014. doi:10.1109/IFIPNetworking.2014.6857078.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An Integrated Experimental Environment for Distributed Systems and Networks. *SIGOPS Operating System Review*, 36(SI):255–270, December 2002. ISSN 0163-5980. doi:10.1145/844128.844152.

Wilk, M. B. and Gnanadesikan, R. Probability Plotting Methods for the Analysis of Data. *Biometrika*, 55(1):1 – 17, March 1968. doi:10.1093/biomet/55.1.1.

Willems, C. and Meinel, C. Online Assessment for Hands-On Cyber Security Training in a Virtual Lab. In *Proceedings of the 2012 IEEE Global Engineering Education Conference (EDUCON)*, pages 1–10. April 2012. ISSN 2165-9567. doi:10.1109/EDUCON.2012.6201149.

Wood, P. Trends in Cyber Attack Vectors. *ITNOW*, 60(2):40–41, May 2018. ISSN 1746-5702. doi:https://doi.org/10.1093/itnow/bwy049.

Xu, L., Huang, D., and Tsai, W. Cloud-Based Virtual Laboratory for Network Security Education. *IEEE Transactions on Education*, 57(3):145–150, Aug 2014. ISSN 0018-9359. doi:10.1109/TE.2013.2282285.

Yan, L. and McKeown, N. Learning Networking by Reproducing Research Results. *SIGCOMM Comput. Commun. Rev.*, 47(2):19–26, May 2017. ISSN 0146-4833. doi:10.1145/3089262.3089266.

Yarochkin, F. V., Arkin, O., Kydyraliev, M., Dai, S. Y., Huang, Y., and Kuo, S. Y. Xprobe2++: Low Volume Remote Network Information Gathering Tool. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 205–210. June 2009. ISSN 1530-0889. doi:10.1109/DSN.2009.5270338.

Zalewski, M. p0f - Passive OS Fingerprinting Tool. Online, Jun 2000. Last accessed: 2019-01-16.
URL `https://seclists.org/bugtraq/2000/Jun/141`

Zec, M. BSD Network stack virtualization. In *BSDCon Europe*, volume 2. Amsterdam, The Netherlands, Nov 2002.

Zec, M. Implementing a Clonable Network Stack in the FreeBSD Kernel. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference*, pages 137–150. USENIX Association, June 2003. Last accessed: 2019-03-18.
URL `https://www.usenix.org/legacy/events/usenix03/tech/freenix03/zec.html`

Zec, M. and Mikuc, M. Operating System Support for Integrated Network Emulation in IMUNES. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS)*. Boston, MA, USA, October 2004.

Appendix A

ARP Delays in Ping Timings

The Address Resolution Protocol (ARP) (Plummer, 1982) was designed to help systems translate between various network addressing methods available at the time to a "wire" address, or more commonly the Media Access Control (MAC) address of the network interface. In switched Local Area Networks (LANs), machines can be addressed directly by the address of the network interface. The Address Resolution Protocol (ARP) assists with translating Internet Protocol (IP) addresses to MAC addresses.

Listing A.1 shows an example ping between two machines on a switched network. The first ping takes considerably more time to resolve than the remaining three. When the first ping request (ICMP Type 8) is sent from the local machine, no translation exists between the targeted IP address (`10.0.0.11`) and the MAC address of the machine's Network Interface Controller (NIC) (or simply stated, network card). If the local machine was on a routed network, the request would have been sent to a default gateway, and the gateway would have been responsible for forwarding the packet to the next hop in the chain. A packet-by-packet account of the first 159ms, the time taken for the first ping to resolve, is shown in Figure A.1.

Listing A.1: ARP Delay in `ping` RTT

```
1  $ ping -4 -c 4 10.0.0.11
2  PING 10.0.0.11 (10.0.0.11) 56(84) bytes of data.
3  64 bytes from 10.0.0.11: icmp_seq=1 ttl=128 time=159 ms
4  64 bytes from 10.0.0.11: icmp_seq=2 ttl=128 time=7.12 ms
5  64 bytes from 10.0.0.11: icmp_seq=3 ttl=128 time=7.63 ms
6  64 bytes from 10.0.0.11: icmp_seq=4 ttl=128 time=10.0 ms
7
8  --- 10.0.0.11 ping statistics ---
9  4 packets transmitted, 4 received, 0% packet loss, time 3004ms
10 rtt min/avg/max/mdev = 7.121/46.046/159.401/65.454 ms
```

Before the ping request can be sent, the local machine first has to resolve the MAC address of the target host. Once the MAC address of the target has been resolved the actual packet can be transmitted. Once the target machine has received the ping request, a response to the request has to be sent. At this point in time, the target host knows the IP address of the local machine, but not the MAC address. The remote machine resolves the MAC address of the local machine using ARP. Once the MAC of the local machine is known, the response (ICMP Type 0) is sent. At the end of the first 0.159ms, both the local and remote machines have cached the IP to MAC translations. Subsequent pings thus resolve significantly faster.

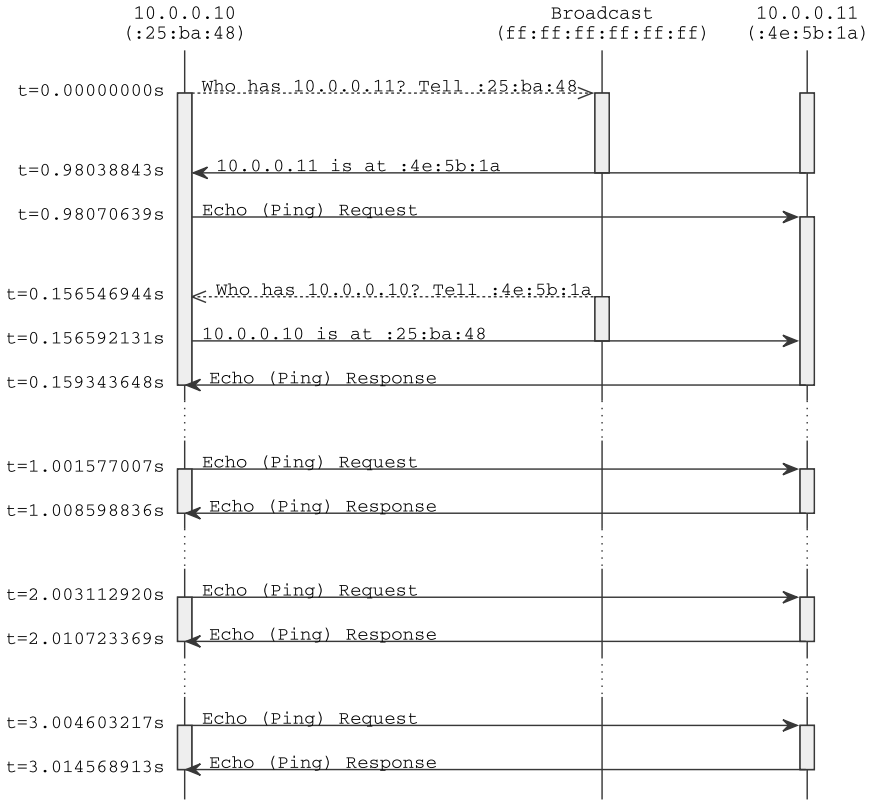


Figure A.1: Ping ARP Delay Message Sequence Diagram

The initial delay caused by ARP resolutions can have a negative impact on the metrics for set of pings as was used with the ping latency tests (5.4.1). The long delay on the first packet negatively influences the maximum and average statistics of ping runs. In order to avoid these negative influences, for the ping latency tests static ARP entries were used in emulated nodes.

Appendix B

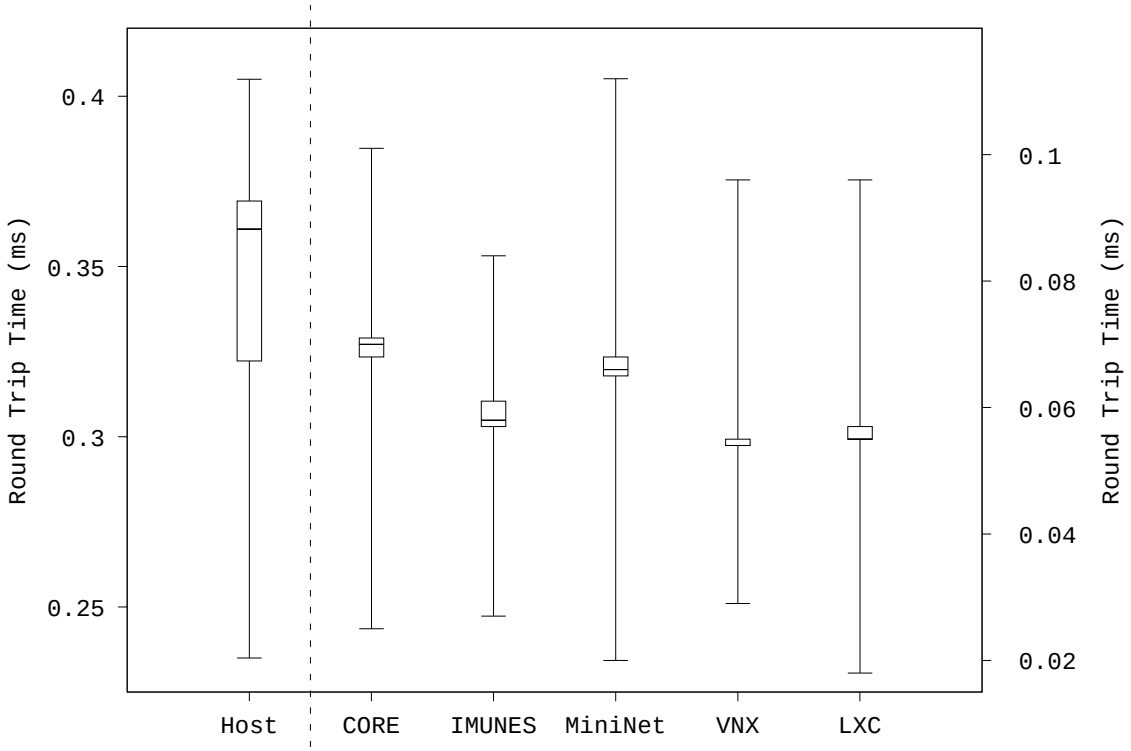
Ping Latency Distribution Results

Section 5.4.2 showed a partial set of figures generated during exploratory statistical data analysis of the ping latency distribution results. The full complement of figures that were generated are presented in this appendix and a brief overview of the significance of each set of figures is given.

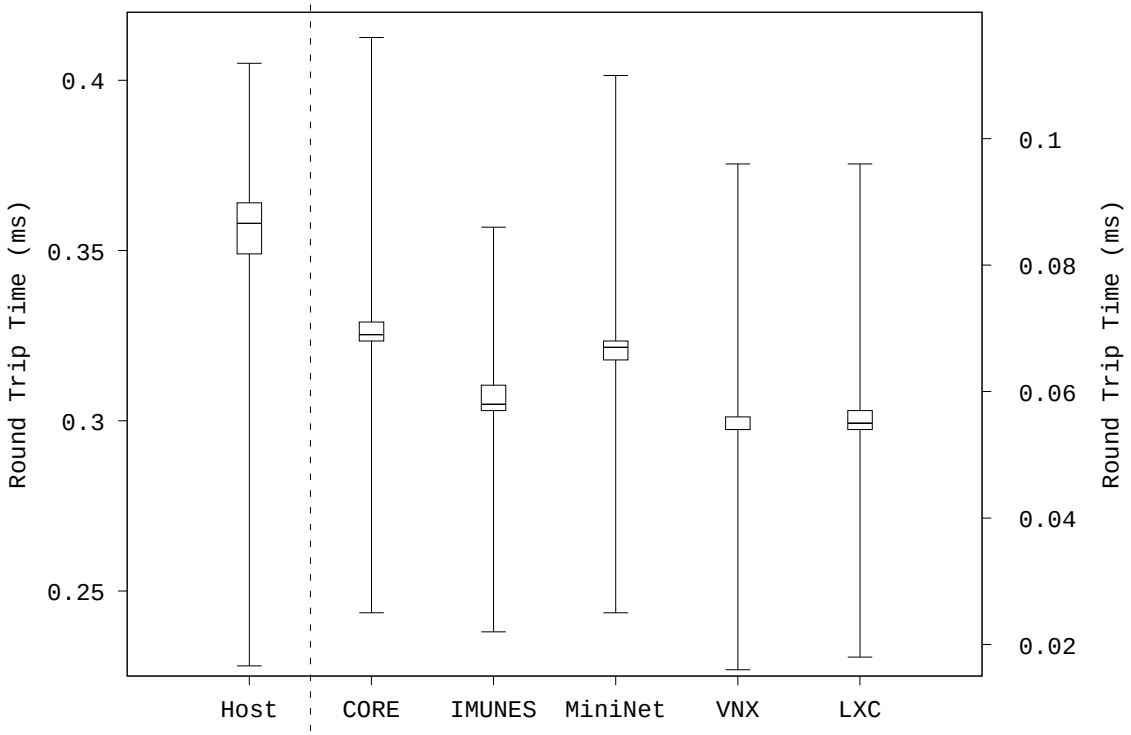
Figure B.1 shows boxplots generated for both runs of the ping latency distribution tests. These boxplots show that the middle 50% of timing distributions (P25 to P50) remain consistent between runs.

Complementary to the boxplots are histograms. The histograms for the ping latency distribution tests (Figures B.2 and B.3) presents an estimate to the Probability Distribution Function (PDF) for the data collected. The histograms were used to compare distribution across runs, and shows that the distributions of ping latency timings remain consistent between successive runs.

The final exploratory statistical data analysis tool used was Percentile-Percentile plots (P-P Plots). These plots were used to test for correlation between the host and Container-Based Network Emulators (CBNEs), and between pairings of CBNEs. If a P-P Plot approaches a straight line, there is cause to assume a correlation between the data sets. Figures B.4 and B.5 respectively show the P-P Plots for the Host-CBNE pairings for the two runs. Both runs returned comparable plots and both showed no correlation. Figures B.6 and B.7 shows the results for the two consecutive runs for CBNEs pairings that are not correlated. Figures B.8 and B.9 shows the results for the two runs where the P-P Plots approach a straight line.

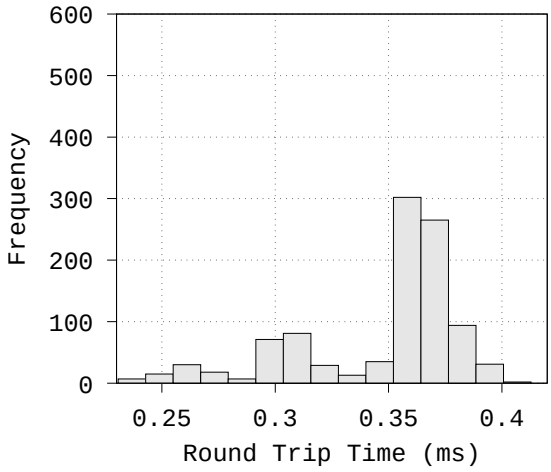


(a) Ping Latency Distribution Box Plots - Run 1

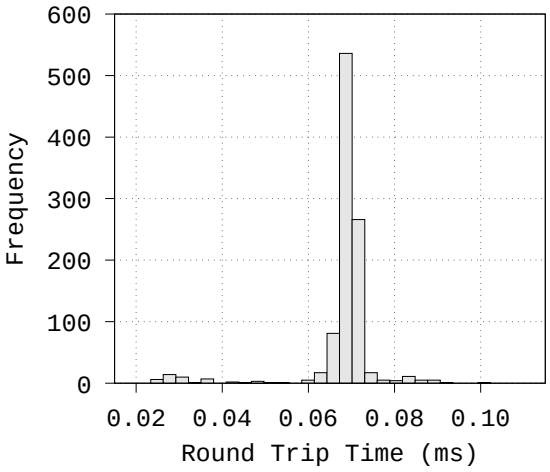


(b) Ping Latency Distribution Box Plots - Run 2

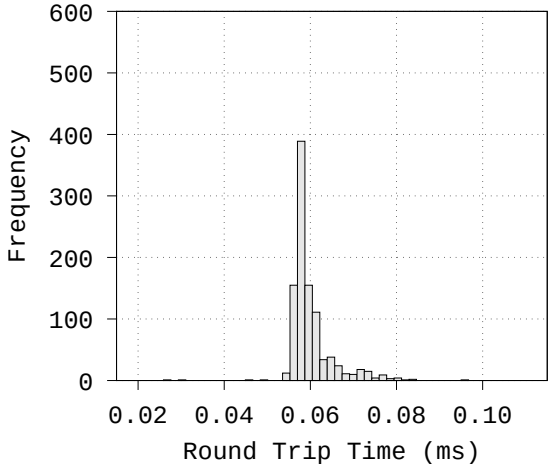
Figure B.1: Ping Latency Distribution



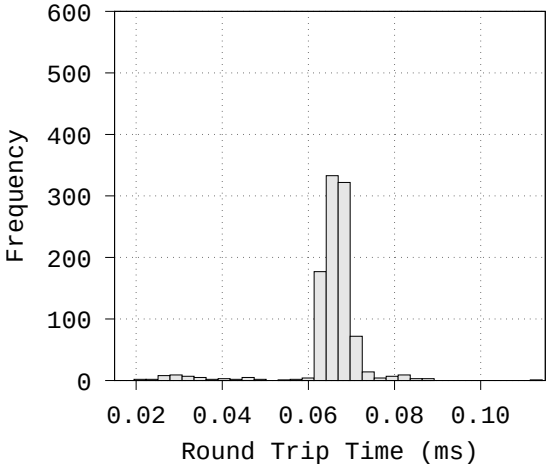
(a) Host



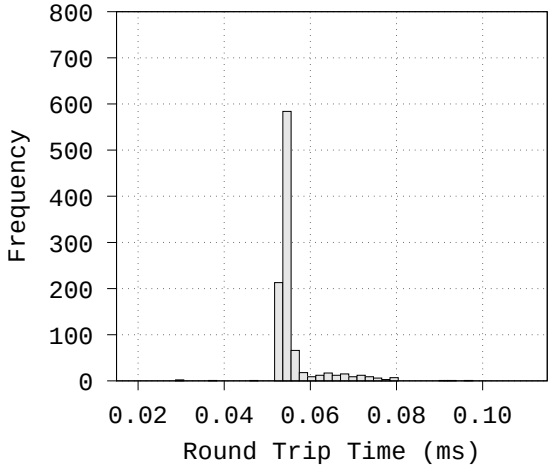
(b) CORE



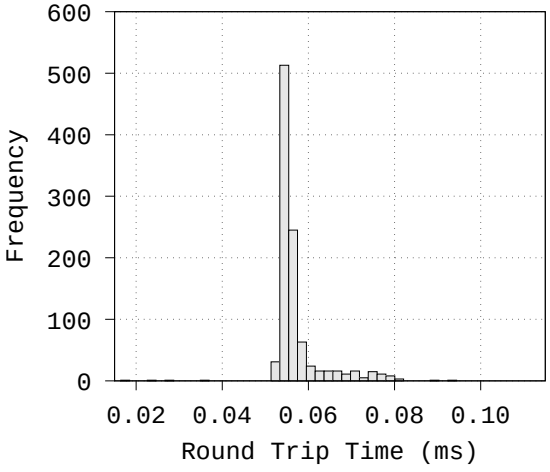
(c) IMUNES



(d) MiniNet

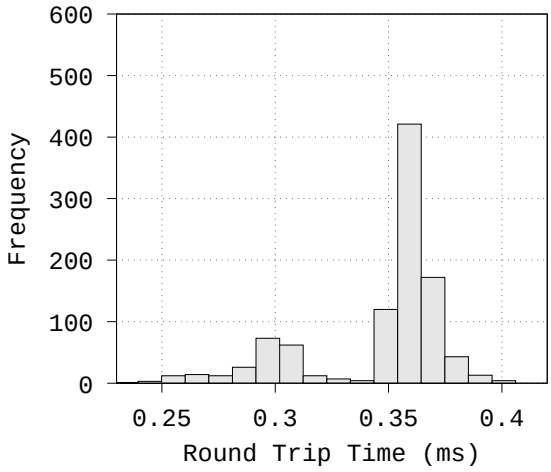


(e) VNX

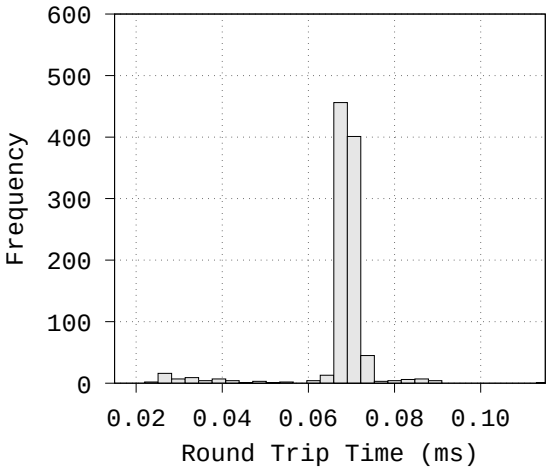


(f) LXC

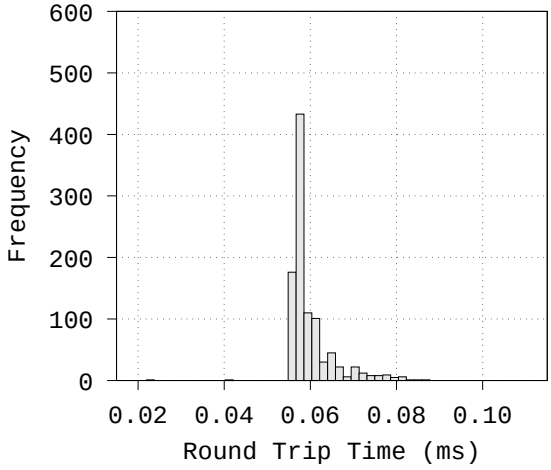
Figure B.2: Ping Distribution Histograms - Run 1



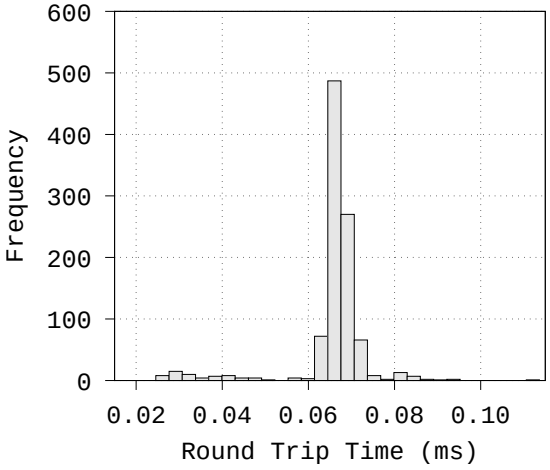
(a) Host



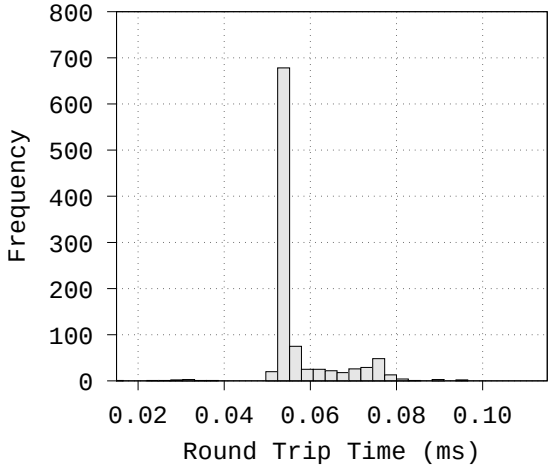
(b) CORE



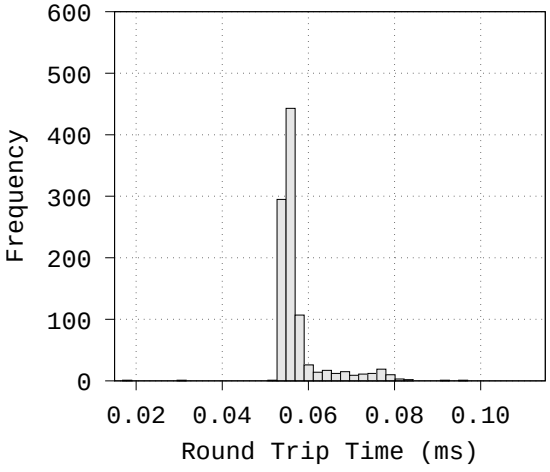
(c) IMUNES



(d) MiniNet

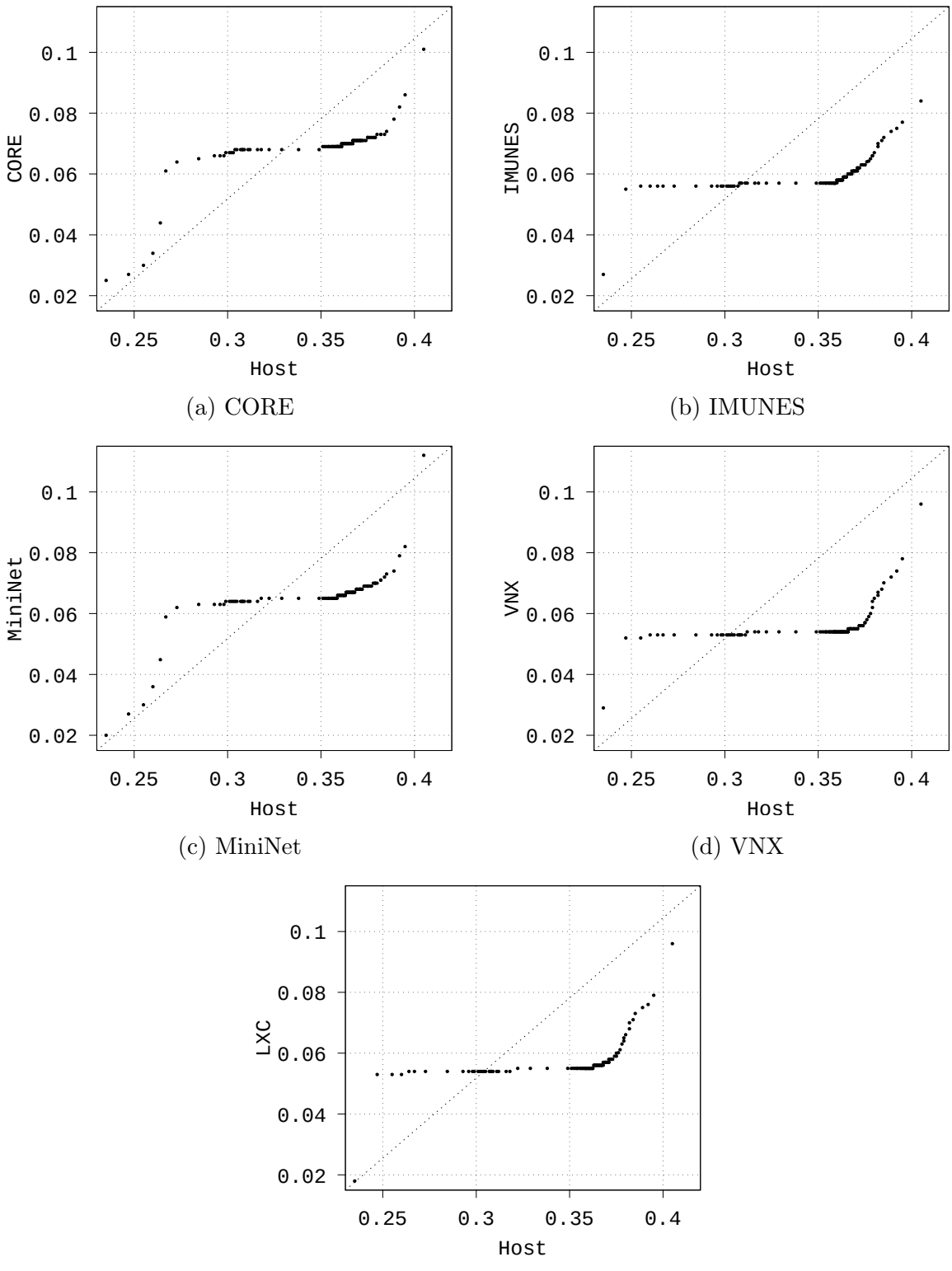


(e) VNX



(f) LXC

Figure B.3: Ping Distribution Histograms - Run 2



(a) CORE

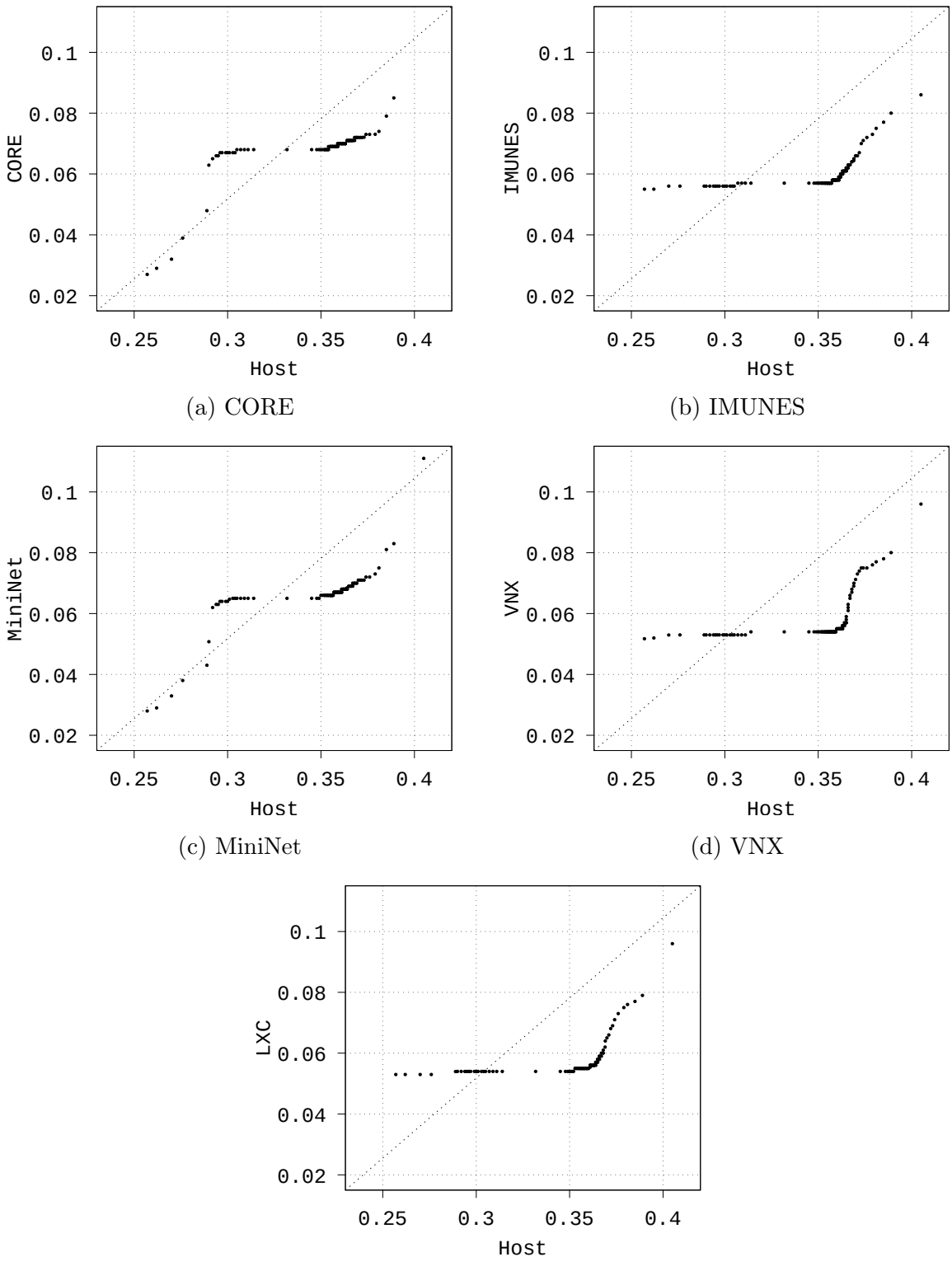
(b) IMUNES

(c) MiniNet

(d) VNX

(e) LXC

Figure B.4: Ping Distribution P-P Plots - CBNE to Host - Run 1



(a) CORE

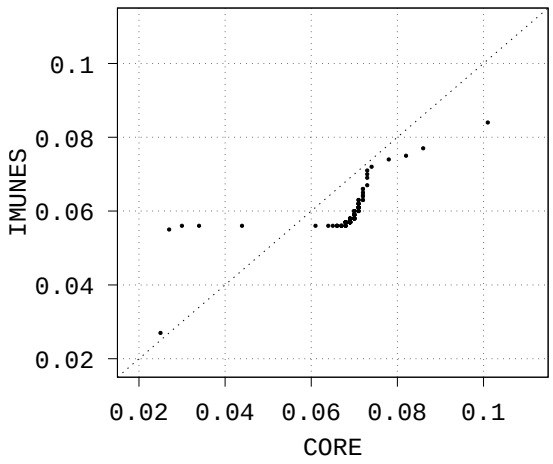
(b) IMUNES

(c) MiniNet

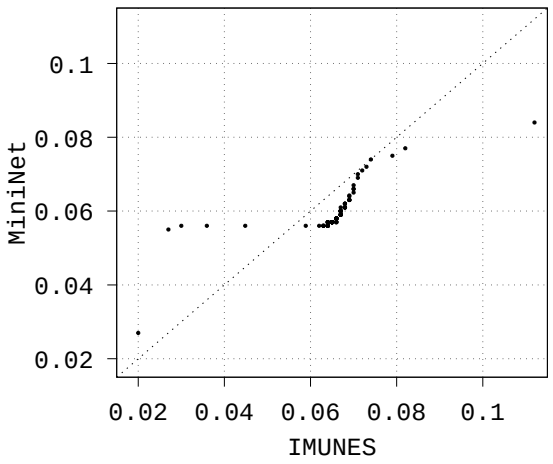
(d) VNX

(e) LXC

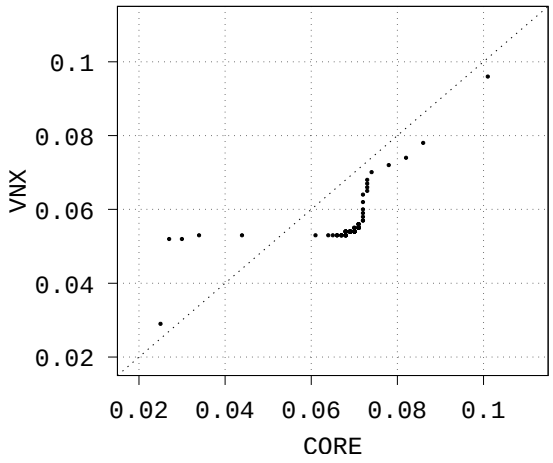
Figure B.5: Ping Distribution P-P Plots - CBNE to Host - Run 2



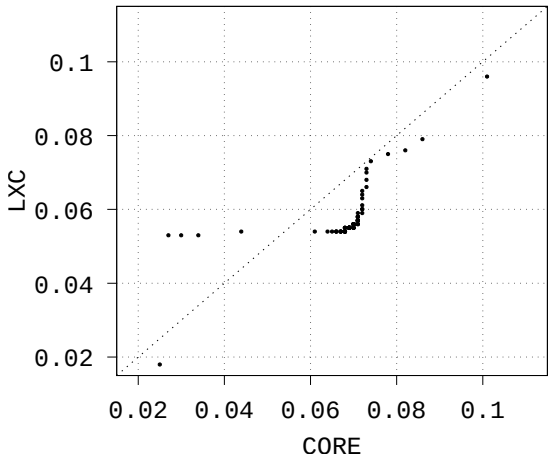
(a) CORE - IMUNES



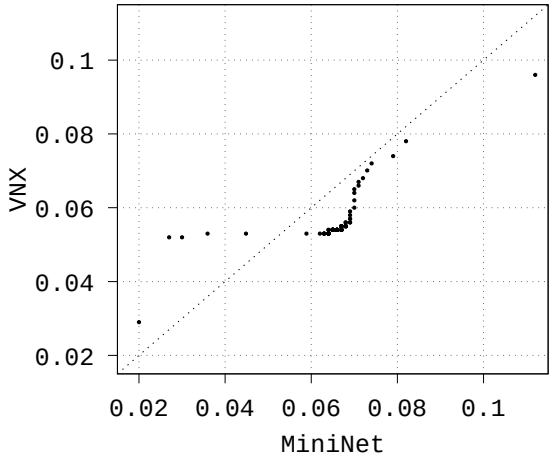
(b) IMUNES - MiniNet



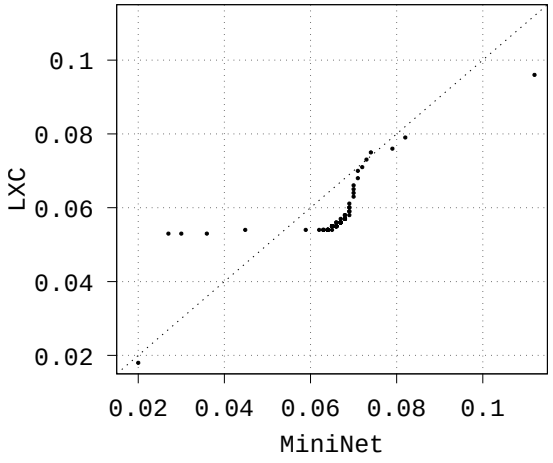
(c) CORE - VNX



(d) CORE - LXC

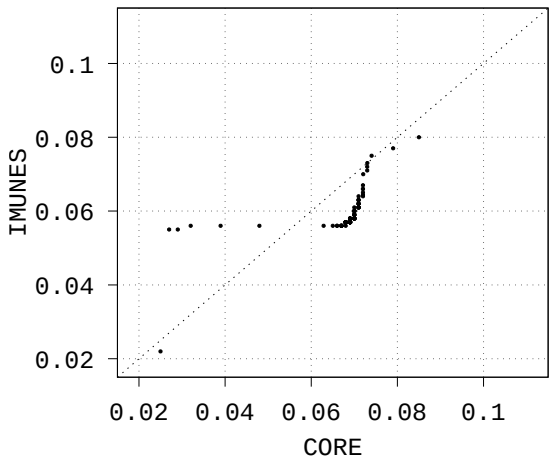


(e) MiniNet - VNX

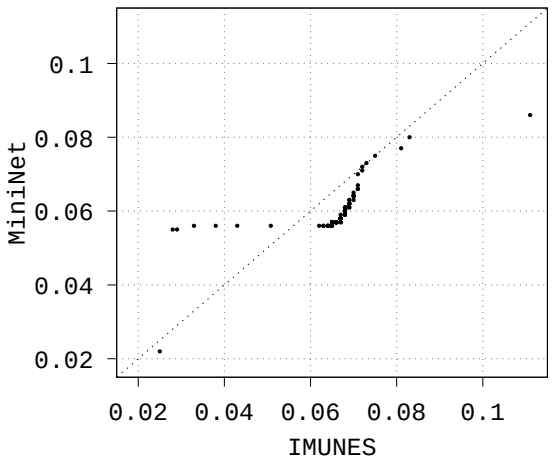


(f) MiniNet - LXC

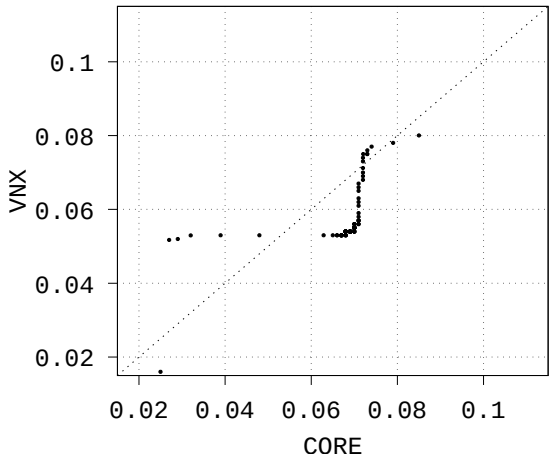
Figure B.6: Ping Distribution P-P Plots - Uncorrelated CBNEs - Run 1



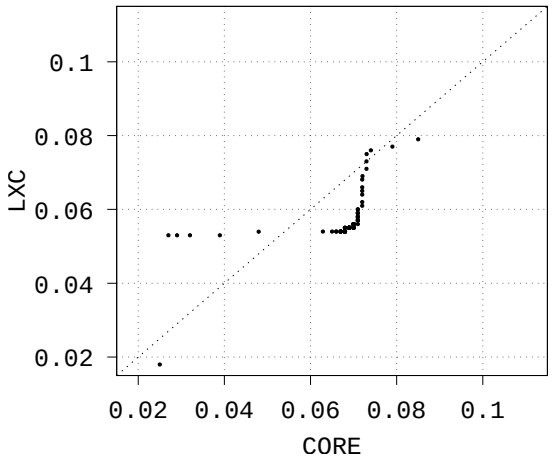
(a) CORE - IMUNES



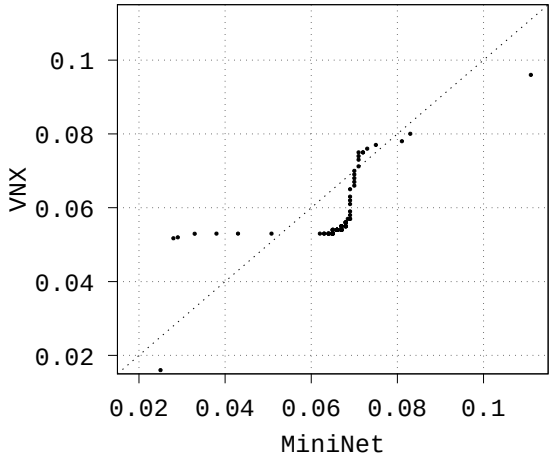
(b) IMUNES - MiniNet



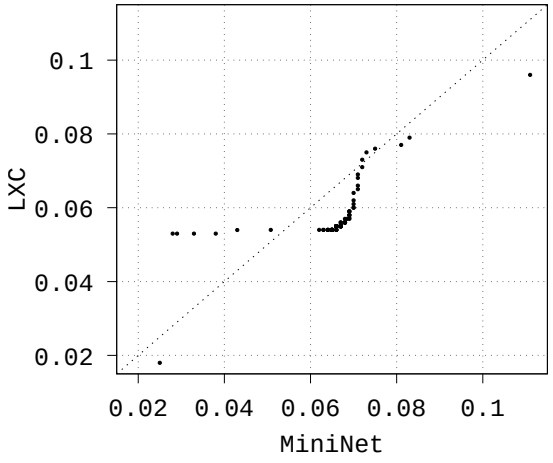
(c) CORE - VNX



(d) CORE - LXC

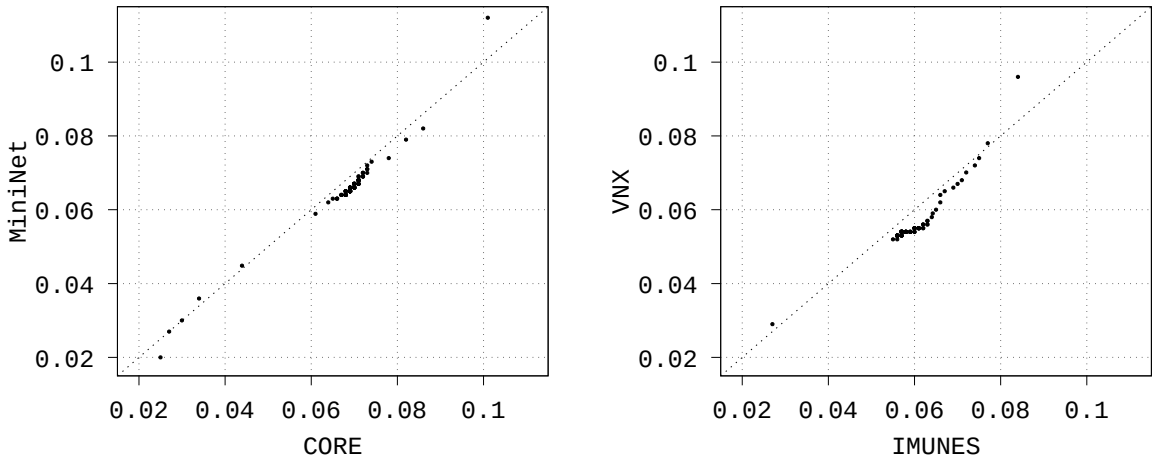


(e) MiniNet - VNX



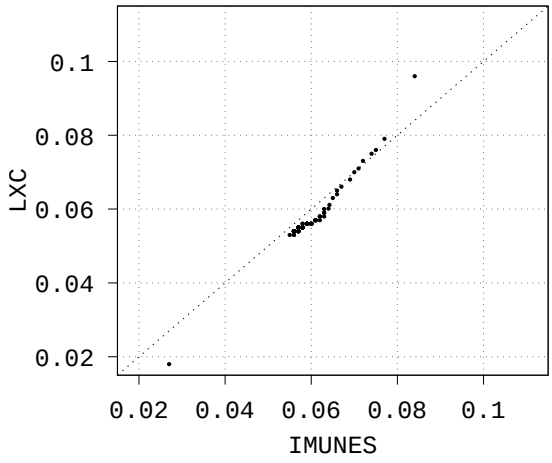
(f) MiniNet - LXC

Figure B.7: Ping Distribution P-P Plots - Uncorrelated CBNEs - Run 2

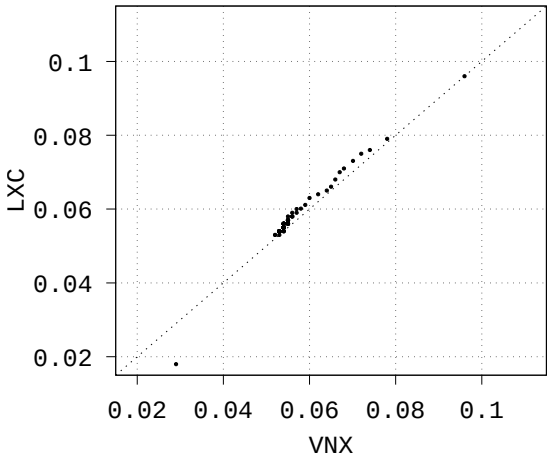


(a) CORE - MiniNet

(b) IMUNES - VNX

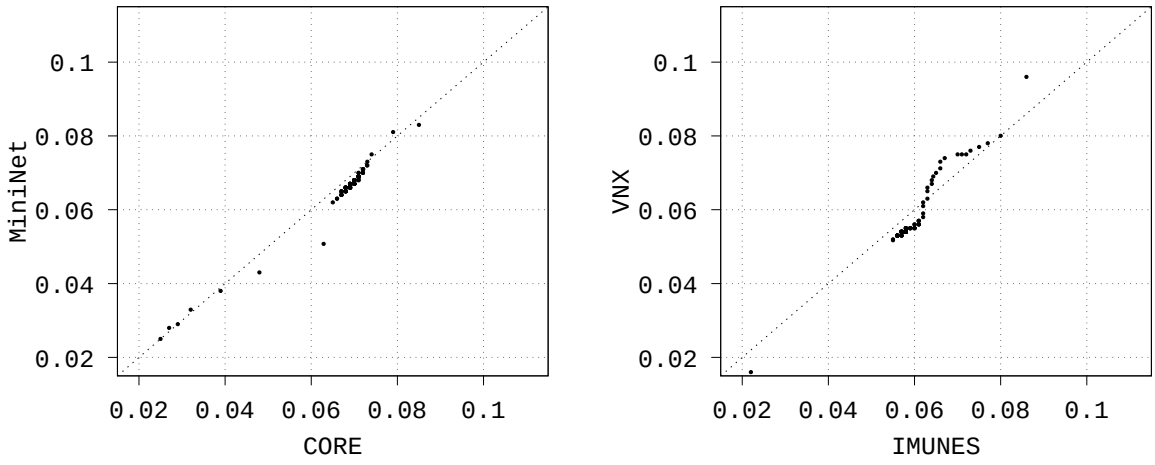


(c) IMUNES - LXC



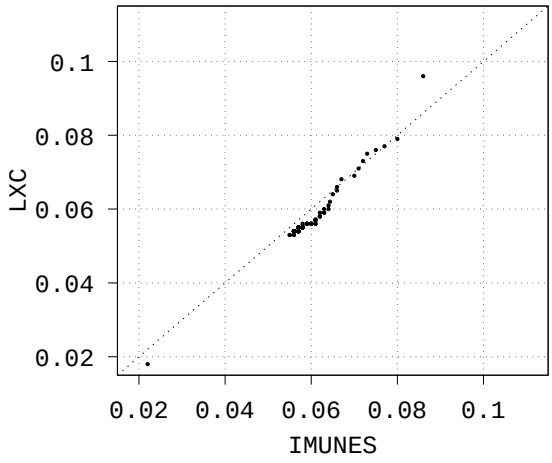
(d) VNX - LXC

Figure B.8: Ping Distribution P-P Plots - Correlated CBNEs - Run 1

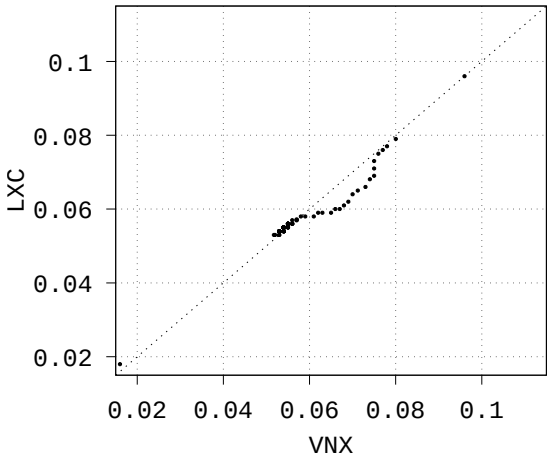


(a) CORE - MiniNet

(b) IMUNES - VNX



(c) IMUNES - LXC



(d) VNX - LXC

Figure B.9: Ping Distribution P-P Plots - Correlated CBNEs - Run 2

Appendix C

Interpreting Fingerprints

This appendix serves as a guide to understanding how fingerprints are used to store features extracted by Operating System (OS) fingerprinting utilities. Fingerprint "decoding" guides are provided for `p0f`, `ettercap`, `xprobe2`, and `SinFP3`. No guide is provided for `nmap`. The fingerprinting methods and structure of an `nmap` fingerprint is covered extensively in Lyon (2009, Chapter 8 - Remote OS Detection).

C.1 `p0f` Fingerprint Details

Fingerprints for `p0f` v3.09b are stored in human readable form and are contained in the `p0f.fp` file. Fingerprint entries are specified by a label indicating the OS and a signature containing the fingerprint. A description of each field in the fingerprint is provided in Table C.1. Signatures in the database have the following structure:

```
label = type:class:name:flavor
sig   = ver:ittl:olen:mss:wsize,scale:olayout:quirks:pclass
```

The database entry for Mac OS X is shown in Listing C.1. In the example entry there are seven different signatures for Mac OS X, illustrating the how an OS fingerprint can vary based on connection parameters for different protocols.

Table C.1: p0f v3.09b Passive Fingerprint Structure[a]

Field	Description
type	The type of signature. s for specific, g for generic
class	The family of the OS
name	The common or human readable name of the OS
flavor	The version of the OS
ver	The Internet Protocol (IP) version for the fingerprint, * denotes both IPv4 and IPv6
ittl	The IP Time To Live (TTL)
olen	Length in bytes for Transmission Control Protocol (TCP) options
mss	The TCP Maximum Segment Size (MSS) TCP option, * denotes variation dependant on sender network
wsize	The TCP Window Size
scale	The Windows Scale TCP option
olayout	Ordering of TCP options
quirks	A list of inconsistencies ("quirks") found in intercepted packets
pclass	The class of packet payload

[a] Adapted from http://lcamtuf.coredump.cx/p0f3/README

Listing C.1: p0f v3.09b Fingerprint Entry for Mac OS X

```
1  ; --------
2  ; Mac OS X
3  ; --------
4
5  label = s:unix:Mac OS X:10.x
6  sig   = *:64:0:*:65535,0:mss,nop,ws:df,id+:0
7  sig   = *:64:0:*:65535,0:mss,sok,eol+1:df,id+:0
8  sig   = *:64:0:*:65535,0:mss,nop,nop,ts:df,id+:0
9  sig   = *:64:0:*:65535,0:mss,nop,ws,sok,eol+1:df,id+:0
10 sig   = *:64:0:*:65535,0:mss,nop,ws,nop,nop,ts:df,id+:0
11 sig   = *:64:0:*:65535,0:mss,nop,nop,ts,sok,eol+1:df,id+:0
12 sig   = *:64:0:*:65535,0:mss,nop,ws,nop,nop,ts,sok,eol+1:df,id+:0
```

As an example of how to interpret a signature, the last signature entry for Mac OS X will be used. The label for the entry (`s:unix:Mac OS X:10.x`) indicates that it is a specific (`s`) signature and that the class of OS is Unix-like (`unix`). The common name of the OS is Mac OS X and cover all versions (`10.x`). The last signature for the Mac OS X entry is unpacked in Table C.2.

Table C.2: p0f v3.09b Passive Fingerprint Decoding for Mac OS X

Field	Value	Description
ver	*	The signature is for both IPv4 and IPv6
ittl	64	The initial TTL is 64 hops
olen	0	IPv4 options length is 0
mss	*	Multiple MSS sizes are expected
wsize	65535	The windows size for connections is 65535 bytes
scale	0	The window scale is 0
olayout	mss,nop,ws,nop,nop,ts,sok,eol+1	(discussed below)
quirks	df,id+	The don't fragment flag is set but the IPID is non-zero
pclass	0	The length of the payload is 0

The olayout field in the example contains a comma separated list of values. The ordering of these values indicates the ordering of the TCP options in the analysed packets. For the example signature the TCP options are shown in Table C.3.

Table C.3: p0f v3.09b TCP Options Decoding for Mac OS X

TCP Option	Description
mss	Maximum Segment Size (MSS)
nop	No-option (padding)
ws	Windows Scale
nop	No-option (padding)
nop	No-option (padding)
ts	Timestamp
sok	Selective Acknowledgement permitted
eol+1	Explicit end of option plus one No-option

For p0f v3.09b the olayout section of the fingerprint is considered the most imported aspect of signatures. The options present as well as the ordering of the options is one of the most reliable way to identify an OS or family of OSs. A full list of TCP options and quirks that p0f can extract and identify can be found in section 5 of the p0f README[1].

C.2 ettercap Fingerprint Details

Similar to p0f, ettercap (Ornaghi and Valleri, 2019) stores passive fingerprint information in a human readable form. Each entry in the ettercap database[2] consists of eleven fields structured as WWWW:MSS:TTL:WS:S:N:D:T:F:LEN:OS. The description of each field is shown in Table C.4.

Table C.4: ettercap v0.8.3 Passive Fingerprint Structure[a]

Field	Description
WWWW	The TCP Window Size
MSS	The TCP MSS TCP option or _MSS if it is unkown or omitted
TTL	The IP TTL
WS	The Windows Scale TCP option or WS if it is unkown or omitted
S	1 if the TCP Selective Acknowledgement (SACK) option is permitted, 0 otherwise
N	1 if the TCP options contain a No Option (NOP) byte, 0 otherwise
D	1 if the TCP Don't Fragment (DF) flag is set, 0 otherwise
T	1 if the TCP timestamp is set, 0 otherwise
F	S if the packet is a SYN packet, A if the packet is a SYN/ACK packet
LEN	The length of the packet of LT if irrelevant or unknown
OS	The name of the OS

[a] Adapted from the share/etter.finger.os[2] source file

[1] http://lcamtuf.coredump.cx/p0f3/README

[2] https://github.com/Ettercap/ettercap/blob/master/share/etter.finger.os

As an example of how to "decode" a fingerprint, database entries for the Mac OS 9 operating system will be decoded. The Mac OS 9 operating system has two entries, one for a SYN packet and one for a SYN/ACK packet. These entries are as follows:

```
FFFF:05B4:FF:01:0:1:1:0:A:30:Mac OS 9
FFFF:05B4:FF:01:0:1:1:0:S:30:Mac OS 9
```

For both entries all fields are the same except for the `F` field, which indicates the two types of packets: `A` for a SYN/ACK packet and `S` for a SYN packet. Descriptions for the fields that are the same for both packets is shown in Table C.5.

Table C.5: `ettercap` Passive Fingerprint Decoding for Mac OS 9

Field	Value	Description
WWWW	FFFF	The window size for TCP connections is 65535 bytes
MSS	05B4	The MSS for packets is 1460 bytes
TTL	FF	The initial TTL for packets is 255 hops
WS	01	The window scaling factor for TCP connections is set to 1
S	0	Selective Acknowledgement (SACK) is not permitted
N	1	TCP SYN and SYN/ACK packets contain No Option (NOP) bytes
D	1	Packets intended for the system must not be fragmented
T	0	No timestamp is present in TCP packets
LEN	30	The length for both the SYN and SYN/ACK packets is 48 bytes
OS	Mac OS 9	The name of the OS is Mac OS 9

C.3 `xprobe2` Fingerprint Details

Fingerprints for `xprobe2` (Yarochkin *et al.*, 2009) are split into separate sections for each module that produces the specific section. As with `p0f` and `ettercap`, the fingerprint database for `xprobe2` is human readable and stored in the `etc/xprobe2.conf` text file. `xprobe2` produces a complete fingerprint using eight modules and primarily relies on the Internet Control Message Protocol (ICMP) protocol for OS identification. Descriptions for the eight modules used to generate a fingerprint are shown in Table C.6:

Table C.6: `xprobe2` v0.3 Probe Modules[a]

Module	Description
Module A	The ICMP Echo test extracts specific features from an ICMP Echo reply
Module B, C, & D	These modules test whether or not a response is received for ICMP timestamp, ICMP address mask, and ICMP information reply probes
Module E	The ICMP port unreachable test attempts to solicit an ICMP Type 3 Code 3 response by sending a UDP request to a closed UDP port and tests if the response matches the request
Module F	The TCP SYN/ACK response test extract features from a response on an open TCP port
Module G	The TCP RST/ACK response test extract features from a response on an closed TCP port

[a] Adapted from the `xprobe2 docs/new-fingerprints-howto.txt` source file

A sample fingerprint from the xprobe2 database for Mac OS 10.4.1 is shown in Listing C.2. An example "decoding" of the fingerprint is shown in Table C.7. The results for *Module B* and *Module C* are shown in the fingerprint though both indicated that no response was received. For these modules xprobe2 inserts default values into the fingerprint.

C.4 SinFP3 Fingerprint Details

SinFP3 (Auffret, 2010) is the only fingerprinting utility utilised in testing that can generate both passive and active fingerprints. SinFP3 generates a single fingerprint for passive fingerprinting and three fingerprints (based on three probes) for active fingerprinting. A single fingerprint structure is used for all fingerprints and is shown in Table C.8.

Table C.8: SinFP3 Fingerprint Structure

Field	Description
BF[a]	Comparison of binary flags between request and response packets
TF	TCP flags of response packet
TWS	TCP windows size of connection
TO	TCP options and ordering of response packet
MSS	MSS for the connection
TWSF	TCP window scaling factor
TOL	Length of TCP options of the response packet[b]

[a] Active fingerprints only
[b] For passive fingerprinting, this is for the intercepted packet

Fingerprint decoding examples for SinFP3 will be done for both passive and active fingerprints. An example fingerprint for Mac OS X 10 extracted from the SinFP3 fingerprint database is shown below:

```
F0x02 W65535 O0204ffff0103....04020000[3] M1460 S1 L24
```

For active fingerprint decoding all three generated fingerprints will be used. Below is an example fingerprint for Mac OS X 10.5 is shown below:

```
S1: B11113 F0x12 W65535 O0204ffff M1460 S0 L4
S2: F0x12 W65535 O0204ffff0103414404020000[4] M1460 S3 L24
S3: B11020 F0x04 W0 O0 M0 S0 L0
```

[3]Full TCP Options: O0204ffff010303ff0101080a................04020000
[4]Full TCP Options: O0204ffff010303ff0101080affffffff4445414404020000

Listing C.2: `xprobe2` v0.3 Fingerprint Entry for Mac OS X 10.4.1

```
fingerprint {
    OS_ID = "Apple Mac OS X 10.4.1"
    #Entry inserted to the database by: Ofir Arkin (ofir@sys-security.com)
    #Entry contributed by: Ofir Arkin (ofir@sys-security.com)
    #Date: 6 June 2005
    #Modified:

    #Module A [ICMP ECHO Probe]
    icmp_echo_reply = y
    icmp_echo_code = !0
    icmp_echo_ip_id = !0
    icmp_echo_tos_bits = !0
    icmp_echo_df_bit = 1
    icmp_echo_reply_ttl = <64

    #Module B [ICMP Timestamp Probe]
    icmp_timestamp_reply = n
    icmp_timestamp_reply_ttl = <64
    icmp_timestamp_reply_ip_id = !0

    #Module C [ICMP Address Mask Request Probe]
    icmp_addrmask_reply = n
    icmp_addrmask_reply_ttl = <255
    icmp_addrmask_reply_ip_id = !0

    #Module E [UDP -> ICMP Unreachable probe]
    #IP_Header_of_the_UDP_Port_Unreachable_error_message
    icmp_unreach_reply = y
    icmp_unreach_echoed_dtsize = 8
    icmp_unreach_reply_ttl = <64
    icmp_unreach_precedence_bits = 0
    icmp_unreach_df_bit = 1
    icmp_unreach_ip_id = !0

    #Original_data_echoed_with_the_UDP_Port_Unreachable_error_message
    icmp_unreach_echoed_udp_cksum = 0
    icmp_unreach_echoed_ip_cksum = OK
    icmp_unreach_echoed_ip_id = OK
    icmp_unreach_echoed_total_len = OK
    icmp_unreach_echoed_3bit_flags = OK

    #Module F [TCP SYN | ACK Module]
    #IP header of the TCP SYN ACK
    tcp_syn_ack_tos = 0
    tcp_syn_ack_df = 1
    tcp_syn_ack_ip_id = !0
    tcp_syn_ack_ttl = <64

    #Information from the TCP header
    tcp_syn_ack_ack = 1
    tcp_syn_ack_window_size = 65535
    tcp_syn_ack_options_order = MSS NOP WSCALE NOP NOP TIMESTAMP
    tcp_syn_ack_wscale = 0
    tcp_syn_ack_tsecr = !0
    tcp_syn_ack_tsval = !0

    #Module G
    tcp_rst_reply = y
    tcp_rst_df = 1
    tcp_rst_ip_id_1 = !0
    tcp_rst_ip_id_2 = !0
    tcp_rst_ip_id_strategy = I
    tcp_rst_ttl = <64

    snmp_sysdescr = Darwin Kernel Version
}
```

Table C.7: `xprobe2` v0.3 Fingerprint Decoding

Field	Value	Description
Module A - ICMP Echo Probe		
icmp_echo_reply	y	A response to an ICMP Echo request was received
icmp_echo_code	!0	The echo code was non-zero
icmp_echo_ip_id	!0	The IP ID field of the response was non-zero
icmp_echo_tos_bits	!0	The type of service bit was non-zero
icmp_echo_df_bit	1	The explicit do not fragment bit was set
icmp_echo_reply_ttl	<64	The upper bound for the time to live for the response packet was 64 hops
Module B - ICMP Timestamp Probe		
icmp_timestamp_reply	n	No response was received for the ICMP timestamp probe
icmp_timestamp_reply_ttl	<64	[Default value as no response received]
icmp_timestamp_reply_ip_id	!0	[Default value as no response received]
Module C - ICMP Address Mask Request Probe		
icmp_addrmask_reply	n	No response was received for the ICMP address mask probe
icmp_addrmask_reply_ttl	<64	[Default value as no response received]
icmp_addrmask_reply_ip_id	!0	[Default value as no response received]
Module D - ICMP Information Request Probe		
Module D did not run during the test		
Module E - ICMP Unreachable Request Probe		
icmp_unreach_reply	y	A response was received to a probe packet sent to a closed UDP port
icmp_unreach_echoed_dtsize	8	The size of the response packet was 8
icmp_unreach_reply_ttl	<64	The upper bound for the time to live for the response packet was 64 hops
icmp_unreach_precedence_bits	0	The precedence bits of the IP ID header was set to 0
icmp_unreach_df_bit	1	The explicit do not fragment bit was set
icmp_unreach_ip_id	!0	The IP ID field of the response was non-zero
icmp_unreach_echoed_udp_cksum	0	The response had no UDP checksum
icmp_unreach_echoed_ip_cksum	OK	The checksum for the IP component of the response was good
icmp_unreach_echoed_ip_id	OK	The response packet returned the same value for the don't fragment bit as the request packet
icmp_unreach_echoed_total_len	OK	The length of the response packet was 20
icmp_unreach_echoed_3bit_flags	OK	The 3bit flags of the response packet echoed correctly
Module F - Open TCP Port Probe		
tcp_syn_ack_tos	0	The Type of Service bits were all set to 0
tcp_syn_ack_df	1	The explicit do not fragment bit was set
tcp_syn_ack_ip_id	!0	The IP ID field of the response was non-zero
tcp_syn_ack_ttl	<64	The upper bound for the time to live for the response packet was 64 hops
tcp_syn_ack_ack	1	Expected to be 1 but can be any value
tcp_syn_ack_window_size	65534	The initial window size of the connection
tcp_syn_ack_options_order		The ordering of TCP options for the connection was `MSS NOP WSCALE NOP NOP TIMESTAMP`
tcp_syn_ack_wscale	0	The TCP window scale factor for the connection was 0
tcp_syn_ack_tsecr	!0	The TCP timestamp echo reply was non-zero
tcp_syn_ack_tsval	!0	The TCP timestamp value was non-zero
Module G - Closed TCP Port Probe		
tcp_rst_reply	y	A reply was received to a TCP reset request
tcp_rst_df	1	The explicit do not fragment bit was set
tcp_rst_ip_id_1	!0	The IP ID field of the first response packet was non-zero
tcp_rst_ip_id_2	!0	The IP ID field of the second response packet was non-zero
tcp_rst_ip_id_strategy	I	The identifier for the IP packet increases per packet, not randomly
tcp_rst_ttl	<64	The upper bound for the time to live for the response packet was 64 hops

Table C.9 illustrates how to decode a passive fingerprint generated by SinFP3 and Table C.10 illustrates how to decode the three components of an active SinFP3 fingerprint. The *BF* components of the fingerprints are not decoded as the author of SinFP3 makes no explicit mention of how these values are computed.

Table C.9: SinFP3 Passive Fingerprint Decoding for Mac OS X 10

Field	Value	Description
TF	F0x02	The packet is a SYN packet
TWS	W65535	The initial window size of the packet is 65535
TO		The ordering of the TCP options for the packet is as follows: MSS NOP WS NOP NOP TS SACK
MSS	M1460	The maximum segment size for the connection is 1460bytes
TWSF	S1	The window scaling factor for the connection is 1
TOL	L24	The total length of the options for the intercepted packet is 24bytes

Table C.10: SinFP3 Active Fingerprint Decoding for Mac OS X 10.5

Field	Value	Description
Probe S1 Fingerprint		
BF	B11113[a]	
TF	F0x12	The packet is a SYN/ACK packet
TWS	W65535	The initial window size of the packet is 65535
TO		The ordering of the TCP options for the packet is as follows: MSS
MSS	M1460	The maximum segment size for the connection is 1460bytes
TWSF	S1	The window scaling factor for the connection is 0
TOL	L4	The total length of the options for the intercepted packet is 24bytes
Probe S2 Fingerprint		
BF	B11113[a]	
TF	F0x12	The packet is a SYN/ACK packet
TWS	W65535	The initial window size of the packet is 65535
TO		The ordering of the TCP options for the packet is as follows: MSS NOP WS NOP NOP TS SACK
MSS	M1460	The maximum segment size for the connection is 1460bytes
TWSF	S3	The window scaling factor for the connection is 3
TOL	L24	The total length of the options for the intercepted packet is 24bytes
Probe S3 Fingerprint		
BF	B11020[a]	
TF	F0x04	The packet is a RST packet
TWS	W0	Not applicable
TO	O0	Not applicable
MSS	M0	Not applicable
TWSF	S0	Not applicable
TOL	L0	Not applicable

[a] No explicit mention is made in any SinFP3 documentation regarding the computation of the binary flags, thus no interpretation is made

Appendix D

Fingerprint Results Tables

This appendix lists all results tables deemed too cumbersome to be included in the main text. Brief descriptions of the tables listed in this appendix and the reason for being excluded from the main text are given below.

The results for the `xprobe2` tests where shown in an abbreviated from in Section 5.6.1 with specific extracts shown that illustrated the deviations found for the MiniNet Container-Based Network Emulator (CBNE). The results for the two `xprobe2` tests are shown in Tables D.1 (no port specification) and D.2 (with port specification).

Section 5.6.3 presented results for the `nmap` tests where deviations were found. Table D.3 shows the test for which no deviations were found. The UDP Probes (Table D.3a), TCP Probe (Table D.3b), and ICMP Echo Tests (Table D.3c) returned the same results for all tested systems. For the TCP Probe test results, Table D.3b lists the TCP probe packets (`T1` through `T7`) against the features extracted as all test systems returned the same results.

Section 5.7 presented the main findings of the MiniNet modification tests. In Section D.3 extracts from the modification tests are shown where MiniNet deviations were corrected. Section D.3.1 presents the results for passive scanning utilities and Section D.3.2 presents the results for active scanning utilities. Each of the tables includes test results for the host, MiniNet before modification (*Default*), and MiniNet after modification (*Modified*). Deviations between results for the host and the default MiniNet are highlighted in bold.

D.1 `xprobe2` ICMP Results

Table D.1: Fingerprint Results for `xprobe2`

Test	Host	CORE	IMUNES	MiniNet	VNX	LXC
icmp_echo_reply	y	y	y	y		y
icmp_echo_code	!0	!0	!0	!0		!0
icmp_echo_ip_id	!0	!0	!0	!0		!0
icmp_echo_tos_bits	!0	!0	!0	!0		!0
icmp_echo_df_bit	0	0	0	0		0
icmp_echo_reply_ttl	<64	<64	<64	<64		<
icmp_timestamp_reply	y	n	y	y	y	y
icmp_timestamp_reply_ttl	<64	<64	<64	<64	<64	<64
icmp_timestamp_reply_ip_id	!0	!0	!0	!0	!0	!0
icmp_addrmask_reply	n	n	n	n	n	n
icmp_addrmask_reply_ttl	<255	<255	<255	<255	<255	<255
icmp_addrmask_reply_ip_id	!0	!0	!0	!0	!0	!0
icmp_info_reply						
icmp_info_reply_ttl						
icmp_info_reply_ip_id						
icmp_unreach_reply	y	y	y	y	y	y
icmp_unreach_echoed_dtsize	>64	>64	>64	>64	>64	>64
icmp_unreach_reply_ttl	<64	<64	<64	<64	<64	<64
icmp_unreach_precedence_bits	0xc0	0xc0	0xc0	0xc0	0xc0	0xc0
icmp_unreach_df_bit	0	0	0	0	0	0
icmp_unreach_ip_id	!0	!0	!0	!0	!0	!0
icmp_unreach_echoed_udp_cksum	OK	OK	OK	OK	OK	OK
icmp_unreach_echoed_ip_cksum						
icmp_unreach_echoed_ip_id	OK	OK	OK	OK	OK	OK
icmp_unreach_echoed_total_len	OK	OK	OK	OK	OK	OK
icmp_unreach_echoed_3bit_flags	OK	OK	OK	OK	OK	OK
tcp_syn_ack_tos						
tcp_syn_ack_df						
tcp_syn_ack_ip_id						
tcp_syn_ack_ttl						
tcp_syn_ack_ack						
tcp_syn_ack_window_size						
tcp_syn_ack_options_order						
tcp_syn_ack_wscale						
tcp_syn_ack_tsval						
tcp_syn_ack_tsecr						
tcp_rst_reply		y	y	y		y
tcp_rst_df		1	1	1		1
tcp_rst_ip_id_1		0	0	0		0
tcp_rst_ip_id_2		0	0	0		0
tcp_rst_ip_id_strategy		0	0	0		0
tcp_rst_ttl		<64	<64	<64		<64

Table D.2: Fingerprint Results for `xprobe2` with Port Specification

Test	Host	CORE	IMUNES	MiniNet	VNX	LXC
icmp_echo_reply	y	y	y	y	y	y
icmp_echo_code	!0	!0	!0	!0	!0	!0
icmp_echo_ip_id	!0	!0	!0	!0	!0	!0
icmp_echo_tos_bits	!0	!0	!0	!0	!0	!0
icmp_echo_df_bit	0	0	0	0	0	0
icmp_echo_reply_ttl	<64	<64	<64	<64	<64	<64
icmp_timestamp_reply	y	y	y	y	y	y
icmp_timestamp_reply_ttl	<64	<64	<64	<64	<64	<64
icmp_timestamp_reply_ip_id	!0	!0	!0	!0	!0	!0
icmp_addrmask_reply	n	n	n	n	n	n
icmp_addrmask_reply_ttl	<255	<255	<255	<255	<255	<255
icmp_addrmask_reply_ip_id	!0	!0	!0	!0	!0	!0
icmp_info_reply						
icmp_info_reply_ttl						
icmp_info_reply_ip_id						
icmp_unreach_reply	y	y	y	y	y	y
icmp_unreach_echoed_dtsize	>64	>64	>64	>64	>64	>64
icmp_unreach_reply_ttl	<64	<64	<64	<64	<64	<64
icmp_unreach_precedence_bits	0xc0	0xc0	0xc0	0xc0	0xc0	0xc0
icmp_unreach_df_bit	0	0	0	0	0	0
icmp_unreach_ip_id	!0	!0	!0	!0	!0	!0
icmp_unreach_echoed_udp_cksum	OK	OK	OK	OK	OK	OK
icmp_unreach_echoed_ip_cksum						
icmp_unreach_echoed_ip_id	OK	OK	OK	OK	OK	OK
icmp_unreach_echoed_total_len	OK	OK	OK	OK	OK	OK
icmp_unreach_echoed_3bit_flags	OK	OK	OK	OK	OK	OK
tcp_syn_ack_tos	0	0	0	0	0	0
tcp_syn_ack_df	1	1	1	1	1	1
tcp_syn_ack_ip_id	0	0	0	0	0	0
tcp_syn_ack_ttl	<64	<64	<64	<64	<64	<64
tcp_syn_ack_ack	1	1	1	1	1	!
tcp_syn_ack_window_size	65160	65160	65160	**43440**	65160	65160
tcp_syn_ack_options_order		MSS SACK TIMESTAMP NOP WSCALE				
tcp_syn_ack_wscale	7	7	7	**9**	7	7
tcp_syn_ack_tsval	!0	!0	!0	!0	!0	!0
tcp_syn_ack_tsecr	!0	!0	!0	!0	!0	!0
tcp_rst_reply	y	y	y	y	y	y
tcp_rst_df	1	1	1	1	1	1
tcp_rst_ip_id_1	0	0	0	0	0	0
tcp_rst_ip_id_2	0	0	0	0	0	0
tcp_rst_ip_id_strategy	0	0	0	0	0	0
tcp_rst_ttl	<64	<64	<64	<64	<64	<64

D.2 nmap Results

Table D.3: nmap Fingerprint Results - No Deviations

(a) nmap UDP Probe Packet Test Results

Platform	R	DF	T	IPL	UN	RIPL	RID	RIPCK	RUCK	RUD
Host	Y	N	40	164	0	G	G	G	G	G
CORE	Y	N	40	164	0	G	G	G	G	G
IMUNES	Y	N	40	164	0	G	G	G	G	G
MiniNet	Y	N	40	164	0	G	G	G	G	G
VNX	Y	N	40	164	0	G	G	G	G	G
LXC	Y	N	40	164	0	G	G	G	G	G

(b) nmap TCP Probe Packet Test Results

Test	R	DF	T	W	S	A	F	O	RD	Q
T1	Y	Y	40		O	S+	AS		0	
T2	N									
T3	N									
T4	Y	Y	40	0	A	Z	R		0	
T5	Y	Y	40	0	Z	S+	AR		0	
T6	Y	Y	40	0	A	Z	R		0	
T7	Y	Y	40	0	Z	S+	AR		0	

(c) nmap ICMP Echo Test Results

Platform	R	DFI	T	CD
Host	Y	N	40	S
CORE	Y	N	40	S
IMUNES	Y	N	40	S
MiniNet	Y	N	40	S
VNX	Y	N	40	S
LXC	Y	N	40	S

D.3 Fingerprint Results Post Modification

D.3.1 Passive Fingerprinting Results After Modification

Table D.4: p0f v3.09b Fingerprints - Pre & Post Modification

Platform	Fingerprint
Host	4:64+0:0:1460:mss*45,7:mss,sok,ts,nop,ws:df:0
MiniNet (Default)	4:64+0:0:1460:mss***30**,**9**:mss,sok,ts,nop,ws:df:0
MiniNet (Modified)	4:64+0:0:1460:mss*45,7:mss,sok,ts,nop,ws:df:0

Table D.5: ettercap v0.82 Fingerprints - Pre & Post Modification

Platform	Fingerprint
Host	FE88:05B4:40:07:1:1:1:1:A:3C
MiniNet (Default)	**A9B0**:05B4:40:**09**:1:1:1:1:A:3C
MiniNet (Modified)	FE88:05B4:40:07:1:1:1:1:A:3C

Table D.6: SinFP3 Passive Fingerprints - Pre & Post Modification

Platform	TF	TWS	TO	MSS	TWSF	TOL
Host	F0x02	W64240	O0204ffff...03ff†	M1460	S7	L20
MiniNet (Default)	F0x02	**W42340**	O0204ffff...03ff†	M1460	**S9**	L20
MiniNet (Modified)	F0x02	W64240	O0204ffff...03ff†	M1460	S7	L20

† O0204ffff0402080a........00000000010303ff

D.3.2 Active Fingerprinting Results After Modification

Table D.7: `xprobe2` PortSpec Test Results - Pre & Post Modification

Test	Host	MiniNet (Default)	MiniNet (Modified)
tcp_syn_ack_ack	1	1	1
tcp_syn_ack_window_size	65160	**43440**	65160
tcp_syn_ack_options_order	MSS SACK TIMESTAMP NOP WSCALE		
tcp_syn_ack_wscale	7	**9**	7
tcp_syn_ack_tsval	!0	!0	!0
tcp_syn_ack_tsecr	!0	!0	!0

Table D.8: SinFP3 Active Fingerprints - Pre & Post Modification

Test	Platform	BF	TF	TWS	TO	MSS	TWSF	TOL
S1	Host	B10113	F0x12	W64240	O0204ffff	M1460	S0	L4
	MiniNet (Default)	B10113	F0x12	**W42340**	O0204ffff	M1460	S0	L4
	VNX (Modified)	B10113	F0x12	W64240	O0204ffff	M1460	S0	L4
S2	Host	B10113	F0x12	W65160	O0204ffff...03ff†	M1460	S7	L20
	MiniNet (Default)	B10113	F0x12	**W43440**	O0204ffff...03ff†	M1460	**S9**	L20
	MiniNet (Modified)	B10113	F0x12	W65160	O0204ffff...03ff†	M1460	S7	L20

† O0204ffff0402080affffffff44454144010303ff

Table D.9: `nmap` Fingerprint Results - Pre & Post Modification

(a) Sequence Generation Test Results

Platform	SP	GCD	ISR	TI	CI	II	TS
Host	F9	1	109	Z	Z	I	A
MiniNet (Default)	100	1	10F	Z	Z	I	A
MiniNet (Modified)	100	1	10C	Z	Z	I	A

(b) Transmission Control Protocol (TCP) Options Test Results

Test	Host	MiniNet (Default)	MiniNet (Modified)
O1	M5B4ST11NW7	M5B4ST11N**W9**	M5B4ST11NW7
O2	M5B4ST11NW7	M5B4ST11N**W9**	M5B4ST11NW7
O3	M5B4NNT11NW7	M5B4NNT11N**W9**	M5B4NNT11NW7
O4	M5B4ST11NW7	M5B4ST11N**W9**	M5B4ST11NW7
O5	M5B4ST11NW7	M5B4ST11N**W9**	M5B4ST11NW7
O6	M5B4ST11	M5B4ST11	M5B4ST11

(c) TCP Windows Size Test Results

Platform	W1	W2	W3	W4	W5	W6
Host	FE88	FE88	FE88	FE88	FE88	FE88
MiniNet (Default)	**A9B0**	**A9B0**	**A9B0**	**A9B0**	**A9B0**	**A9B0**
MiniNet (Modified)	FE88	FE88	FE88	FE88	FE88	FE88

(d) Explicit Congestion Notification Test Results

Platform	R	DF	T	W	O	CC	Q
Host	Y	Y	40	FAF0	M5B4NNSNW7	Y	—
MiniNet (Default)	Y	Y	40	**0564**	M5B4NNSN**W9**	Y	—
Host (Modified)	Y	Y	40	FAF0	M5B4NNSNW7	Y	—