

**Frankfurt University
of Applied Sciences**

Faculty 2: Computer Science and Engineering

**Deployment of Android, Haiku, and Google Fuchsia
in Containers and Virtual Machines with Web-UI
Access via Apache Guacamole**

– Master Thesis –

Submitted in order to obtain the academic degree

High Integrity Systems (M.Sc.)

submitted on August 11, 2025 by

Jatinkumar Nakrani

Matr. Nr.: 1386383

First Supervisor : Prof. Dr. Christian Baun
Second Supervisor : Prof. Dr. Thomas Gabel

Declaration

I thus certify that the work included in this thesis, which is named Deployment of Android, Haiku, and Google Fuchsia in Containers and Virtual Machines with Web-UI Access via Apache Guacamole, is the outcome of my own independent investigation, which was conducted under the guidance of Professors Prof. Dr. Christian Baun and Prof. Dr. Thomas Gabel.

I further declare that:

- No other academic degree or certification at this or any other institution has received this thesis, in whole or in part.
- Every source and piece of information used to prepare this thesis has been appropriately cited and acknowledged.
- Any help obtained during the study process has been specifically mentioned.
- All ethical guidelines have been closely followed, and ethical permission has been acquired whenever appropriate.

This statement demonstrates my dedication to academic integrity and is made in compliance with Frankfurt University of Applied Sciences policies.

Frankfurt, August 11, 2025



Jatinkumar Nakrani

“Education is the best friend. An educated person is respected everywhere. Education beats the beauty and the youth.”

— Chanakya

“We are kept from our goal not by obstacles but by a clear path to a lesser goal.”

— Bhagavad Gita

Acknowledgements

It has been a difficult but worthwhile journey to finish this thesis, and I am incredibly appreciative of the several people and institutions that helped make it happen.

First and foremost, I want to express my sincere gratitude to Prof. Dr. Christian Baun, my first supervisor, for his important advice, knowledge, and unwavering support during this project. His encouraging words and perceptive criticism were very helpful in determining the course of this thesis. My second supervisor, Prof. Dr. Thomas Gabel, has my sincere gratitude for his insightful criticism and scholarly viewpoint, which significantly improved the caliber of my work.

Additionally, I would like to thank Frankfurt University of Applied Sciences' Faculty of Computer Science and Engineering for providing the tools and a first-rate research environment for this study. I also want to express my gratitude to my peers and colleagues for their encouraging conversations, encouragement, and companionship along this trip.

I would also like to express my gratitude to my current employment, LensWare International GmbH, for their kind assistance in giving the laptop for configurations, research materials, and flexibility that allowed me to manage my work and thesis obligations. Their support and comprehension were crucial to this project's success.

I would especially like to express my gratitude to my family and friends for their continuous support, encouragement, and patience during my academic career. In times of uncertainty, their emotional support served as my compass.

Last but not least, I would like to express my gratitude to the open-source communities that created Android, Haiku OS, Google Fuchsia, and Apache Guacamole for their crucial contributions that enabled this study.

Abstract

Highly evolving operating systems, both stable and production-ready platforms, as well as experimental and research-oriented kernels, require an urgent possibility to unify and reproduce the test environments. The given project responds to that necessity by planning and realizing a multi-operating system virtualization framework based on QEMU/KVM on a Linux-based host, and Android-x86, Haiku OS, and Google Fuchsia as the example operating systems to test drive. The primary focus of the project was to provide a secure and fully browser-based and clientless remote access solution using Apache Guacamole, and thus eliminate the necessity of specialized client software programs, and provide cross platform support.

The project was developed under Ubuntu 24.04 LTS utilizing hardware-enforced virtualization so as to be as effective as possible. The one that took care of virtualization was QEMU/KVM, and vm lifecycle was taken care through virt-manager and libvirt. Android-x86 and Haiku OS were deployed, configured, and able to be integrated into the Apache Guacamole platform, which streamed its graphical user interface over VNC to any modern web browser. Using this methodology, high-quality access on a consistent basis to remote areas could be provided independent of host-specific drivers or applications. Test of such systems was based on analyzing the usage of CPU, memory speed, boot time, and GUI latency. The findings based on repeated tests helped to establish stable and predictable results and affirm that the framework was performant in relation to supporting both operating systems under similar conditions.

The situation with Google Fuchsia was more complicated. The project bootstrapped the source code of OS using fx build system and booted them in FEMU the Fuchsia emulator. Although the OS booted and was usable via a serial shell, graphical output was not possible because of unfinished virtual GPU support and framebuffer initialization state under the QEMU/KVM. These limitations are in line with the hardware known limits of compatibility posted in the Fuchsia developer community.

All virtual machines acquired VNC endpoints restricted to the localhost interface only, meaning that no direct and unauthenticated connection was possible using the outside sites. Apache Guacamole was the secure Web gateway. Although in this project Docker was not used in the deployment process, Apache Guacamole could also be containerized to allow additional process isolation and ease maintenance. This would enable the users to log in to the system using a single security-controlled point, after which they would operate the virtual machines. Snapshots were also used extensively in configuration and testing, and allowed instant software rollback in case of an error, and extensive downtimes were saved in the process. This testbed was specifically designed to be modular, in such a way that operating system images, network configurations and resource allocations could be changed without affecting the rest of the system.

As demonstrated by the project, the integration of QEMU/KVM and Apache Guacamole is a flexible, lightweight and scalable platform able to offer multiple operating systems in an isolated but single standalone environment remotely. The solution is most appropriate in academic laboratories, OS development workshops, thin-client deployments and comparative studies where the emphasis is on rapid provisioning and high availability. Maturity of the guest operating system, availability of compatible drivers, and hardware graphics feature enhancements are also pointers in the success of such deployments, highlighted by the results. The graphical constraints in using Google Fuchsia emphasize the need to insert the GPU passthrough or a more developed virtual GPU driver in future releases of the testbed.

In the future, Adding more operating systems into the mix would be more comprehensive; for example, a distribution of BSDs, or server-oriented distributions with containers as a focus. Comparison of other remote display protocols like SPICE or WebRTC could show latency or performance gains on a specific workload. Using orchestration tools to automatically deploy and monitor would also increase scalability and decrease manual configuration overhead.

Keywords — Virtualization, QEMU/KVM, Apache Guacamole, Android-x86, Haiku OS, Google Fuchsia, Remote Desktop, VNC, FEMU, GPU Passthrough, Operating System Evaluation.

Contents

Acknowledgements	I
Abstract	II
1 Introduction	1
1.1 Background	1
1.2 Motivation	2
1.3 Problem Statement	2
1.4 Objectives	3
1.5 Scope and Limitations	4
1.6 Methodology Overview	4
1.7 Structure of the Project	5
2 State Of Art	7
2.1 Android OS Architecture and Deployment	7
2.2 Haiku OS Overview and Virtualization Support	8
2.3 Virtualization Support for Haiku	9
2.4 Fuchsia OS: Microkernel Architecture and Development Status	10
2.5 QEMU/KVM Virtualization	12
2.6 VNC and RDP Protocols	13
2.7 Apache Guacamole and noVNC	15
3 System Design	16
3.1 Host system setup	16
3.2 OS Deployment Strategy	18
3.3 Remote Access Architecture	23
3.4 Security Consideration	27
3.5 System Diagram	29
4 Implementation	35
4.1 Android Setup in QEMU with Web Access	35
4.2 Haiku Setup in QEMU with Web Access	41
4.3 Fuchsia Setup Attempt and Shell Access	49
4.4 Apache Guacamole Configuration	57
4.5 Network and Access Configuration	63
5 Evaluation and Testing	66
5.1 Functional and GUI Testing Overview	66

5.2	System Performance and Resource Utilization	68
5.3	Fuchsia-Specific Boot and CLI Behaviour	69
5.4	Comparative Analysis of OS Behavior	72
5.5	Troubleshooting Summary and Observations	75
6	Conclusion	77
6.1	Summary of Work	77
6.2	Justification of Objectives	78
6.3	Limitations	81
6.4	Recommendations and Future Research Scope	82
	List of Abbreviations	83
	Appendices	84
	Bibliography	85

List of Figures

1	Screenshot of <code>uname -a</code> showing Ubuntu 24.04.1 LTS host system	16
2	Screenshot of RAM and Swap Memory Statistics via <code>free -h</code>	16
3	Terminal Output of <code>lscpu</code> Confirming Virtualisation Support	17
4	Screenshot of Virtual Machine Manager (virt-manager) Interface	18
5	Fuchsia Emulator Boot Screen Showing ASCII Logo on Serial Console	20
6	Fuchsia OS Serial Console Shell Running via <code>fx</code> on Ubuntu 24.04 Host	20
7	QEMU/KVM Storage Pool Configuration Showing <code>qcow2</code> Disk Images for Android and Haiku Virtual Machines	22
8	Remote Access Architecture Diagram	24
9	Apache Guacamole Web Interface	25
10	Remote Access Flowchart	25
11	Host Fuchsia OS Running in Serial Shell Mode on Ubuntu Host	26
12	Discord Query Posted by Jatinkumar on Fuchsia Support Channel	27
13	Remote Access Flow- Browser-Based GUI Access	30
14	Storage Layout and Network Configuration for Virtual Machines	31
15	Product bundle error	32
16	Bundle Setup Failure Screenshot	32
17	Build Success but Runtime Failure	33
18	Diagram of Architecture of the virtualisation-based testbed used to deploy and manage three guest operating systems.	34
19	Virtual Machine Manager interface launched to begin Android VM creation.	35
20	Initial VM creation step with local install media option selected.	36
21	File browser opened to locate the Android ISO file.	36
22	Android-x86 ISO successfully selected for installation.	37
23	DroidVNC-NG password configuration screen inside Android-x86	37
24	Manual selection of Android-x86 9.0 as the guest operating system.	38
25	Allocation of 3GB RAM and 2 CPU cores for optimal performance.	39
26	Storage configuration using a 20GB <code>qcow2</code> virtual disk.	39
27	VM named and prepared for advanced configuration.	40
28	Advanced VM configuration finalised; installation initiated.	40
29	Launch screen of Virtual Machine Manager used to initiate Haiku VM setup	41
30	Selection of local installation media for creating a Haiku virtual machine	42
31	Browsing local storage to locate the Haiku ISO image	42
32	Configuration of guest OS settings for the Haiku system	43
33	Final pre-installation review screen for the Haiku virtual machine	43

34	Haiku desktop loaded from ISO in live environment	44
35	Partitioning tool launched to prepare disk for Haiku installation	44
36	Midway through copying system files to the virtual disk	45
37	Confirmation of successful Haiku OS installation	45
38	Final prompt before restarting into installed Haiku OS	46
39	AGMS VNC server package extracted within the Haiku filesystem	46
40	Terminal window displaying the startup process of the VNC server	47
41	Active VNC session initialized and ready for browser access	47
42	AGMS VNC server showing password protection setup in Haiku	48
43	Chrome browser opening Guacamole interface on the host system	49
44	Output of ffx platform preflight check showing successful validation	49
45	Bootstrap script execution retrieving source tree and manifest files	50
46	Bash profile configuration with Fuchsia environment variables	51
47	Product target configuration using fx set command - Phase 1	52
48	Product configuration confirmation - Phase 2	52
49	System Resource Monitor Dashboard	53
50	Fuchsia emulator target with type "core.x64" in "Product" state.	53
51	Fuchsia emulator target with type "terminal.x64" in "Product" state.	54
52	Fuchsia emulator target with type "workbench_eng.x64" in "Product" state.	54
53	Fuchsia emulator boot process stalled at static logo	55
54	FFX target list showing active emulator instance	56
55	Serial shell access displaying root directories in Fuchsia	57
56	Guacamole source configuration using configure	58
57	guacd service status showing active daemon	58
58	WAR file deployed into Tomcat9 webapps directory	59
59	JDBC extension placed into Guacamole extensions directory	59
60	Edited guacamole.properties for MySQL DB connection	59
61	Apache Guacamole Login Page	60
62	Android used port :5	61
63	Haiku used port :3	61
64	Guacamole Dashboard Listing Android and Haiku VMs	62
65	Android and Haiku VM Desktops in Browser	62
66	brctl show command output displaying network bridges - docker0 (STP disabled) and virbr0 (STP enabled).	63
67	virsh net-list command showing the "default" virtual network in active state with autostart and persistent settings enabled.	63
68	Guacamole Browser Error Message	63
69	Network Flow Diagram for Remote VM Access via Guacamole	65
70	Android-x86 GUI access via Guacamole with Terminal Emulator visible	66
71	Virtual Machine Manager showing Haiku VM in active running state	67

72	Chrome DevTools: Android GUI performance monitoring via Guacamole	68
73	Virtual Machine Manager showing Haiku OS running with stable CPU usage	69
74	Fuchsia Emulator Boot and CLI Interaction Flow	71
75	Android OS Analysis	72
76	Haiku OS Analysis	73
77	Fuschia OS Analysis	73
78	Comparative Metrics Across Operating system Chart	75

List of Tables

1	Comparative Features of VNC and RDP Protocols for Remote Desktop Access in Virtualized Environments	15
2	Host system specifications	17
3	Resource Allocation and Access Type for Guest Operating Systems in QEMU/KVM Virtual Environment.	21
4	Performance Summary Comparison Table	69
5	Comparative Metrics Across Operating Systems	74
6	Objectives Achievement Summary Table	81

1. Introduction

Chapter 1 introduces the motivation and scope of this project, framing the need for a unified virtualization testbed to compare emerging and legacy operating systems. It outlines the research questions, project objectives, and success criteria, emphasizing browser-based access without client installs. The chapter also describes the host hardware and software environment-Ubuntu 24.04 on an HP EliteBook 850 G8-and defines key terms such as QEMU, KVM, Zircon microkernel, and Guacamole. A high-level project roadmap and chapter organization set expectations for the subsequent implementation, evaluation, and discussion.

1.1 Background

Virtualisation is a vital technology pillar in the modern computing world, which enables the resourceful allocation, isolation and control of numerous operating systems (OSes) on a single physical host. This feature enables system administrators, developers and researchers to gain a flexible platform to test software, conduct environment simulations and run heterogeneous workloads without assuming the cost of dedicated hardware on each system. There are Type 1 (bare-metal and sometimes called hosted) and Type 2 (hosted) hypervisors that are traditionally described as virtualisation. The current work is combined with the combination of QEMU with a KVM (Kernel-based Virtual Machine) where QEMU simulates the hardware, and KVM offers hardware-assisted virtualisation via the Linux kernel to allow a near-native guest OS performance.

Remote access to graphical environments with or without virtualised execution has become especially important at educational laboratories, thin-client systems, and distributed development. Apache Guacamole and noVNC allow users to manipulate virtual machines via a standard web browser, allowing controlling of a virtual machine without requiring a proprietary client application. Such solutions are based on HTML5 and WebSocket rendered, converting VNC or RDP protocol streams to browser-compatible ones. They are more portable, less complex in terms of a client, and they are more compatible between devices than they were with previous setups.

This project aims at examining how three operating system platforms including Android, Haiku and Google Fuchsia can be deployed into a QEMU/KVM virtualisation stack extended with remote graphical access using either Apache Guacamole or noVNC. The platforms were chosen to help explain opposing design philosophies and mature stages. Android is based on a Linux kernel, which is still one of the most used operating systems across the world; QEMU-based virtualisation is possible due to the availability of x86-compatible ISO images. Based on the heritage of the former BeOS, Haiku is a fast, light, open-source desktop operating system that operates with extremely high efficiency, responsiveness, and low resource utilisation, and has built-in support to be used as a VNC server, allowing it to be easily integrated to remote desktop operations.

By comparison, Fuchsia is a microkernel based operating system made by Google and uses the proprietary Zircon kernel. It differs significantly with monolithic architectures because it bundles user-space services into accessible modules, employs a capabilities-based security model and employs asynchronous inter-process communication. Its graphical user interface is achieved using Flutter, Google declarative UI framework, and Scenic, a customised compositor. Specifically, Fuchsia does not currently contain published ISO images; installation requires user- Level compilation with the fx build system, which in general takes more than 100 GB of disk space and requires about 6 to 9 hours of compile time on typical development hardware.

In QEMU it is also possible to access Fuchsia via a command-line; however, displaying the Ermine shell or session_manager has been unreliable, possibly due to incomplete GPU acceleration, the lack of such support in virtio-gpu to provide a pass-through virtual GPU, and the unstable nature of Fuchsia graphical stack operating without a graphical user interface. The recreation and reconfiguration attempts did not allow the development of a stable GUI past the boot

logo. All configuration processes, build logs, boot up, and community-based fixes are thus documented in the project hence providing an up-to-date inhibitors to the graphical integration in QEMU settings.

Android and Haiku, on other hands, rode on their own prebuilt ISO-image installations, which are usually around 700 MB-1.5 GB, and were sure to replicate in QEMU with the assurance of a constant graphic appearance and VNC-or RDP- connected remote control. Apache Guacamole can be used on their desktop environments to provide complete browser-based interaction, with keyboard, mouse and clipboard support.

The main idea of the project is to test, analyze and implement these operating systems in a fully integrated remotely accessible QEMU/KVM setting. Particular consideration is given to compatibility, practical behaviour, resource utilisation under constraint and usability through browser based interfaces. Close consideration is given to the issues faced during the work on Fuchsia, documenting progress made in the direction of graphical integration, and describing technical gaps to be explored.

1.2 Motivation

The ability to install and remotely access multiple operating systems by using a unified, virtualised infrastructure has many advantages in the modern, diversified computing environment in research, development and the education environment. An increased focus on cross platform support, modular kernel systems, and less client side dependency, has increased the pressure on the need to find evaluation tools that allow examination of various operating systems in a managed, repeatable and easily accessible manner.

The current project tasks itself with exploring how architecturally different operating systems, both stable and experimental, react to hardware supporting virtualisation and web-based access patterns. In this end, the project will choose three opposing systems as Android, Haiku as well as Google Fuchsia. Android, a well-established Linux-based mobile operating system deployed to both consumer and embedded devices, presents an example of what traditional Linux mobile Linux environment entails; Haiku, the stand-alone single-user desktop system, inspired by BeOS and aimed at being responsive and minimal overhead, can be viewed as the source of ideas to a simple-and-minimalist architecture. and Fuchsia, still under development by Google, assumes a different approach to the operating system architecture, which can be called ground-breaking, and which is based on a microkernel Zircon, an asynchronous mess.

At the same time, the increasing popularity of web-based remote-access tools, such as Apache Guacamole, and noVNC, enable direct transfer of graphical desktop environments through a web browser, bypassing the need to deal with native clients, and reducing the risk of exposure to underlying virtual machines. This model is strongly consistent with the modern tendency to cloud computing to use virtual laboratories and thin-client systems where the ability to be portable and quickly installed has the utmost importance.

1.3 Problem Statement

Operating system virtualization has been a mature technique in application to research and development of systems. However, standard virtualization methods often create even non-trivial problems with integration when deploying OS platforms that are drastically different in terms of architecture, maturity and support of tools. Whereas well-known main stream operating systems (e.g. Linux or Windows) will be typically mounted by hypervisors with extensive graphical support, other less understood or still under development platforms (e.g. Haiku and Google Fuchsia) will require targeted concern in terms of configuration, compatibility and performance testing.

The topic of concern that is tackled in this project includes the problematic issue of implementing and testing three technically divergent operating systems, namely, Android, Haiku, and Google Fuchsia, on a single virtualized framework (QEMU/KVM) to be accessed through graphical user interfaces that are provided by the browser with Apache Guacamole

and noVNC. The plan is to check their remote access operation capacity, explore their virtualization patterns and record the system-level issues emerging.

Android and Haiku are sold in prebuilt ISO images, and standard display output is in both VNC and RDP servers. Fuchsia, however, is not an ISO and requires building Google source code via Google fx build tool. Building a normal application requires more than 100 GB of disk storage and 8-10 hours of compilation time on a typical development server with SSD disk storage using a quad-core processor. In addition to that, Fuchsia graphical interface, and built on the Scenic compositor and Flutter- based Ermine shell, is not consistently renderable in QEMU due to the absence of explicit GPU passthrough support and session shell configuration. Even in GUI deployment, where QEMU targets the logo normally appears on a black screen, as of current public builds, output will commonly freeze with the initial, frozen Fuchsia logo.

The traditional virtualization also presupposes the availability of a windowing system or graphical shell that can mix with VNC or RDP protocols. Lack of a cohesive pipeline in Fuchsia and the quality of the capabilities of QEMU to use the built-in hardware to emulate unaccelerated graphics make integration challenging with applications like Apache Guacamole. This restriction restricts the performance of Fuchsia to shell-level access unless physical hardware, including the Intel NUC or Pixel devices, but not covered in this project, is used.

Cross-comparatively, Android, Haiku, and Fuchsia require uniform measures, among which:

- The use of CPU and memory (can be checked, e.g. through htop, virt-top, and virsh domstats)
- Boot time (i.e. the period it takes a computer to boot and reach the GUI/shell)
- Latency (delay in remote display and responsiveness of inputs tested with browser developer tools and tracing networks).

The primary challenge here is supporting, enabling, and comparing these systems on a uniform basis, and the further challenge is developing and partially maintaining an under-development and non-fully virtualized and non-fully integrated with traditional graphical desktop protocols, OS (Fuchsia), which is the newer upcoming successor to Android.

By extension, this project aims at attempting:

- Provide an automatable, modular and documented installation procedure to get these three operating systems deployed in QEMU/KVM.
- Test their usability and efficiency by being able to access it through web-based graphical interfaces.
- Give an open report of the constraints laboratory, especially in the Fuchsia instance, in accordance with casual testing, construct production, and community-based, observed data.

1.4 Objectives

The objectives of this project are as follows:

1. **To deploy Android, Haiku, and Fuchsia in QEMU/KVM virtual machines** on a Linux-based host.
2. **To enable web-based access** to each virtual machine using Apache Guacamole.
3. **To evaluate functional and performance characteristics** of the Android and Haiku VMs in terms of CPU load, memory usage, boot time, and remote access latency.
4. **To document and analyse the challenges** encountered during the deployment of Fuchsia, particularly the absence of graphical interface support in QEMU.
5. **To provide a detailed setup and testing guide** for all operating systems, enabling replication of results and future extensions.

1.5 Scope and Limitations

The current project examines virtualization and remote web-based access of three architected different operating system Android, Haiku, and Google Fuchsia into QEMU/KVM-based system on a Linux host. Twofold is the aim: to check how they perform on a virtualized environment, and to compare their compatibility with browser-based graphical access facilities like Apache Guacamole and noVNC.

The objects of the work also involve the following:

- Setting up a Ubuntu 24.04.1 LTS host with the QEMU/KVM and Virtual Machine Manager (virt-manager) to manage virtual-machines.
- Installing Android and Haiku on publicly accessible ISO images and running them through VNC/RDP based accessibility with the aid of Apache Guacamole.
- Compiling Google Fuchsia OS is source with the official fx toolchain and trying to automatically boot it in graphical FEMU.
- Tracking the performance of the monitoring own tools on both the guest and the host sides including the in-guest resource-monitors and the Ubuntu System Monitor tool on the host. These are some of the limitations that have been recognized:
- although OS Fuchsia was successfully constructed and booted up to a shell-level interface, access to the graphical interface could not be gained within QEMU, even though several configuration and session-shell efforts were undertaken.
- The testing environment was not built to employ GPU passthrough and external devices, including any Intel NUCs or Pixel devices, both of which have been identified to provide a better experience with Fuchsia graphical stack e.g. (Scenic, Ermine).
- Only QEMU-based virtualization on a single host system is evaluated: no other hypervisors (such as Proxmox, VMware) and no other container-based alternatives were taken into consideration.

Their benchmarking performance of performance consists only of CPU utilisation, memory utilisation and disk input output and network band bandwidth; no kernel tracing and hardware acceleration profiling.

1.6 Methodology Overview

The approach that has been applied in this project, is a systematic and reproducible strategy towards the implementation and assessment of operating systems on the virtualised, browser-based platform. The research goes on in the following discrete steps:

1. System Setup of Host System

The research was carried out on the Ubuntu 24.04.1 LTS Linux host. QEMU and KVM accelerators were used as virtualisation infrastructure, and Virtual Machine Manager (virt-manager) provided guest machine management. Remote access was implemented exclusively using Apache Guacamole deployed via manual setup.

2. Operating System Degree Of Deployment

Android and Haiku were provisioned by using official ISO images, and then some basic configurations of display and networking were performed to allow sessions to be accessed over VNC. Fuchsia is built by the fx tool based on workbench_eng.x64, and the workstation was decrypted. The build operation was done under the same host machine, and the generated image was run through FEMU.

3. Set Up of Remote Access

Each guest OS was included in the Guacamole frontend through an existing protocol VNC or RDP, corresponding with the capacity of the guest. Both of the protocols were enabled to access browsers through local networks.

4. Performance Monitoring

Such monitoring of resource utilisation was carried out via in-system tools, as well as Guacamole (VNC) sessions of Android guest system and Haiku guest system. At the same time, system monitor provided in Ubuntu was used to observe the system-wide CPU, memory, disk, and network usage when the VM is running.

5. Documentation and Troubleshooting

Records of each build and each configuration step and the outcome of the tests, especially pertaining to Fuchsia, were kept. The list of technical challenges faced during the research is described in the screenshots, the terminal logs and descriptive notes.

1.7 Structure of the Project

The current project is organized into six chapters and follows one another in the sequence of the following stages of the project, starting with conceptual background and concluding with technical implementation and evaluation. In this respect, a specific outline is provided, placing the sequence of developments into the stage of realization as follows:

1. Introduction

Gives a project credence in its stated motivations, expresses objectives, and characterizes the technical environment. It also shapes the problem statement, objective scope, appropriate limitations and the framework of methods, followed.

2. Theory and Technology Background

Inspects the main theoretical and technological foundations of the project, specifically focusing on virtualization (QEMU/KVM), remote desktop technology (Apache Guacamole) and the operating systems that will be explored, Android, Haiku, and Fuchsia. The discussion has included earlier pertinent studies, architecture variations and justifications used to choose the three operating systems.

3. Architecture of Systems and Design

Describes how the testbed will be configured: network topology, the virtual machine configuration policy, resource allocation and the integration of the remote desktop gateway. Diagrams and specific parts are given to show the design and assembly of each virtualized operating system.

4. Implementation

Evidence of the feasibility of Android, Haiku, and Fuchsia practical implementation and deployment in a virtualization framework. This section expounds on the installation processes, configuration phases, and manually compiled and deployed Guacamole server, as well as the browser-based remote access management workflows.

5. Evaluation and Testing

Data of the introduced metrics and tools to measure the virtual machine performance, that is, the CPU utilization, the memory consumption, the disk I/O, and the network behavior. The chapter also documents issues that were experienced during the testing, such as problems that occurred in the Fuchsia graphical user interface under FEMU.

6. Conclusion and perspective

Overview of the main outcomes of the project research, distillation of important findings, and possible future enhancements a temporary GPU passthrough, implementation of the use of actual physical devices, and broader automation by using orchestration tools.

7. Bibliography and Appendices

Complete list of referred literature (books, journals, forums and software documentation). There are installation logs and configuration files that have been developed as a part of the project.

2. State Of Art

The current chapter 2 gives an extensive overview of the major technologies and systems relevant to this project. It provides a technical overview of the three discussed operating systems, Android, Haiku and Fuchsia, exploring them by their architecture, compatibility with virtualisation, and modern state of development. The deeper virtualisation platform (QEMU/KVM), the remote desktop protocols (VNC and RDP), and the tools (Apache Guacamole or noVNC) that allow web-based access to the graphical user interface are also considered in the chapter. Lastly, it outlines the assessment measures it uses to gauge the functionality and effectiveness of every one of the virtualized operating systems.

2.1 Android OS Architecture and Deployment

Android is the common operating system that is a free source of Google. It was launched in the year 2008, but it has gained popularity now. It is initially focused on smartphones and tablets, but its modular platform and extensive hardware support allowed it to support a wide range of devices, including televisions, wearable devices, embedded systems and, in the case of this project, virtualized desktop [5].

Android System Architecture

Essentially, the architecture of Android runs off a layered design with 5 important components:

1. Linux Kernel (v4.x – v6.x)

On the minimum level, Android operates based on an altered Linux kernel. This layer handles services that are central to the system, i.e., process management, memory allocation, device drivers, and inter-process communication. Android also enjoys the hardware abstraction and the security paradigm that comes along with the Unix-based systems with the help of the Linux kernel [37].

2. Hardware Abstraction Layer (HAL)

HAL is one of the common interfaces located between the higher-level services in Android and the lower-level device drivers. It hides the particulars of the hardware (e.g., camera, audio, GPS), and provides application developers the capability to develop applications without requiring any knowledge of the hardware details [16].

3. Native Libraries and Android Runtime (ART)

This package consists of basic libraries and are written using C/C++ languages, and it includes WebKit, SQLite, OpenGL and SSL. It is also installed with Android Runtime (ART) which substituted the previous Dalvik Virtual Machine. ART applies ahead-of-time (AOT) compilation: to install an application, it converts the application bytecode into machine-specific code to enhance performance and minimize battery use [4].

4. Application Framework

The framework layer gives APIs that are Java / Kotlin-based, and they facilitate the development of applications. System serves UI, information exchange, and application state UIs through fragments such as Activity Manager, Content Provider and Package Manager. Windows management, notifications, and telephony are also located in this level [26].

5. System Applications

Android is bundled with a series of system applications like the settings panel, dialer, browser, and launcher; these are the systems that can be seen and used by the end-users to use the functionality of the system. These are the applications that are constituted on the framework and communicate with the kernel and the runtime differently.

Virtualization of Android

Forks including Android-x86, Bliss OS and PrimeOS are the main reasons Android is compatible with x86 hardware, as they recompile Android with support of a standard PC. Android-x86, specifically, is an advanced project, due to which it offers bootable ISO images, aiming at application on desktops and virtual machines [5]. This work made use of Android-x86_64 ISO to install Android as a guest operating system operating within a QEMU/KVM virtualised environment on a Linux-based host (Ubuntu 24.04.1 LTS).

The deployment utilised the virt-manager (based on libvirt) in the management of VM resources and devices. Access to the system was remotely provided through Apache Guacamole, the web gateway by offering the VNC-based session control. Important considerations on virtualisation are:

- Enabling KVM acceleration of the improvement of CPU emulation.
- Specifying Video QXL or virtio-GPU to have graphical output through VNC.
- Clipboard and resolution sync: install spice-vdagent or open-vm-tools, when available.
- Ensuring that hardware sensors and 3D acceleration are disabled because of VM compatibility problems.

Between the first attempts of booting, it showed nothing but a black screen on VNC. this was fixed by explicitly updating the GRUB boot arguments and changing the type of processor acceleration (nomodeset) and rendering [5].

Limitations and Considerations

Although Android works largely decently under QEMU, there are still constraints:

- Officially no support for virtualization by Google in Android-on-VM use-cases.
- The lack of GPUs limits graphics capabilities in virtual systems.
- Absence of continuous input support, particularly in the case of non-touch VNC clients.
- Poor Desktop UI that has not been optimised: Android has not been naturally optimised to use desktop/mouse-based interfaces.

Even still, Android-x86 is a viable option as an android testing method in virtualized environments due to its good stability and acceptable usability through the remote platforms via Apache Guacamole or the browser-based remote clients.

2.2 Haiku OS Overview and Virtualization Support

The Haiku operating system is an open source operating system, the aim of which is to carry on the legacy of an OS (BeOS) developed in the 1990s, which has media centred goals. Haiku is actively developed since 2001 and is aimed at personal computing use, and is very responsive, minimalistic, and efficient. Unlike many modern operating systems that rely on a multitude of third-party components, Haiku is developed as a unified, monolithic codebase, which allows for tighter integration between system components and a relatively lightweight footprint [28].

Haiku Architecture

Haiku is a modular but closely coupled architecture in terms of a monolithic kernel. It is done primarily in C++ and is aimed at providing a stable and predictable user experience. The following are the main layers that form the OS:

Kernel Layer- The NewOS kernel is the inspiration of the kernel called Haiku which offers low-level system services multitasking, management of memories, drivers of many devices, and administration of files. It uses preemptive

multitasking and supports symmetric multiprocessing (SMP), which is beneficial for modern multi-core CPUs.

App Server- An important element of the graphical user interface of Haiku, the App Server manages windows, drawing and rendering. It is designed for high performance and integrates tightly with the input server and the GUI toolkit (Haiku Interface Kit). **API and Kit Interface-** Haiku has a POSIX-compliant API layer and a Be API (object-oriented and written in C++). Haiku Interface Kit has given tools to developers to create native applications with the principles of the system's user interface.

Package Management and File System -Haiku supports Be File System (BFS), which includes extended file attributes and journaling. Package management is based on .hpkg files, and the system supports live package activation, which enables atomic system updates and rollbacks [28].

Network Driver and Network Stack- Haiku includes a modular network stack supporting standard protocols (IPv4, IPv6, TCP/UDP, DHCP) and a growing set of device drivers for wired and wireless networking.

2.3 Virtualization Support for Haiku

Haiku has quite decent compatibility with popular virtualization interfaces like QEMU/KVM, VirtualBox and VMware. For this project, (R1/Beta5 – Release Notes, 2024) was used and deployed as a guest operating system within a QEMU/KVM environment managed via Virt-Manager on Ubuntu 24.04.1 LTS.

A Haiku ISO image was loaded by means of UEFI and legacy BIOS. Virtual machine settings were:

- Standard VGA or virtio-vga must be supported to allow video.
- Display: VNC output capable Apache Guacamole.
- CPU: 2 virtual CPUs with KVM-based acceleration.
- Memory: 2-4 GB RAM allocation.
- Disk: 4GB-10 GB qcow2 virtual disk image.

Performance of virtualized was steady, and the memory footprint was low as well as CPU footprint. The GUI was smooth even through remote desktops such as VNC, and the system kept loading fast, unlike Android or Fuchsia. To enhance the GUI responsiveness and mouse control, the "Mouse Integration Add-On" in Haiku was enabled. However, since Haiku is not widely optimized for modern virtual GPU drivers (e.g., virtio-GPU or QXL), graphical acceleration remained limited. Moreover sound was not supported experimentally under QEMU, and some USB devices refused to work unless they were passed through specifically via libvirt configurations.

The possibilities and the weaknesses in virtual Environments.

Strengths:

- **Light:** Haiku consumes minimal resources that make it perfectly suited to deployment in VMs, even on low-end specifications.
- **Satisfactorily quick boot time:** Cold boots failed to layout more than 10 seconds in the majority of the cases.
- **Consistent desktop environment:** Haiku does not need to be customized because it consists of a consistent desktop environment.

Limitations:

- Poor support of modern hardware drivers, particularly high-end GPU and audio hardware.
- There is no third-party support in using the application as the platform remains niche.
- No native support for remote desktop protocols (e.g., no native VNC server)—VNC server inside the OS using Haiku deport.

With all these issues, there was one thing that was in the scale of this project that Haiku was one of the more reliable and easily virtualized systems.

2.4 Fuchsia OS: Microkernel Architecture and Development Status

Introduction to Fuchsia OS

It was in 2016 when fuchsia, an open-source operating system being created by Google was discovered as experimental. Fuchsia is not based on the Linux kernel, as Android or Chrome OS are, but instead is based on a new and purpose-built microkernel named Zircon. The project represents Google's attempt to create a scalable, secure, and modular operating system suitable for a range of devices, from embedded systems to desktops [14]. The fuchsia is not supposed to substitute Android and Chrome OS in the near future, instead to experiment with new OS models with high security profiles, flexible composition architecture and better system updateability. As of mid-2025, Fuchsia has been officially deployed on a limited number of Google Nest smart home devices, with ongoing internal experimentation on other hardware platforms [31].

Zircon Microkernel

The core of Fuchsia is the Zircon, a capability-based microkernel that is highly modular and granular in the management of resources. Unlike monolithic kernels (like Linux), microkernels aim to run only the most essential services in kernel mode such as scheduling, memory management, and inter-process communication (IPC) while delegating everything else to user space [38].

Key Features of Zircon:

- Preemptive multi-threaded scheduler
- Service and drivers in the user-space
- Secure IPC fine-grained capabilities
- Real-time and embedded use-case support
- On-the-fly loading of system components using the Component Framework v2

Zircon interacts with userspace through FIDL (Fuchsia Interface Definition Language), a language-agnostic IPC mechanism that defines APIs and their bindings in multiple languages.

System Architecture and Component Framework

Fuchsia uses a componentized architecture, in which system services, even applications and even low-level drivers, are isolated to separate software components. These components are assembled into realms (execution contexts) and run with precise capability permissions, improving security and maintainability.

The **Component Framework v2**, introduced with recent Fuchsia builds, provides:

- Dependency injection

- Capability routing (e.g., allowing a component access to the network, file system, etc.)
- Lifecycle management (start, stop, restart policies), Sandboxing and tight isolation

The intended functionality of this architecture is to address problems that are prevalent in monoliths, including cross-component failure, long update cycles, and bad security boundaries.

Virtualization and GUI Challenges

Unlike Android and Haiku, Fuchsia lacks the ISO image or system image, pre-built in support of generic x86 virtualization. To test it in a virtual machine (VM), the OS must be manually built from source, a process that takes 8–12 hours on average modern hardware.

The **build process** involves:

- Cloning the Fuchsia Git repository (requires over 50 GB of storage)
- Setting up the Fuchsia build environment using fx (Fuchsia's CLI tool)
- Choosing a board target (e.g. workbench_eng)
- The files can be built with GN and Ninja build systems

Despite using qemu-x64 as a build target, many testers (including the author of this project) report that GUI-based shells (e.g., Ermine or Scenic) either fail to launch or only display the Fuchsia logo on screen in virtual machines. This is happening in the case when the build is successful, and the guest kernel is booting. Only text-based shell (console) access is functional in most virtualization environments [36].

Known GUI issues include:

- Inability to have virtio-gpu acceleration in QEMUs built
- Virtualized environments - incomplete display stack initialization
- Lost virtualized framebuffers drivers

As of July 2025, hardware-based deployment (e.g., on Pixelbooks or Intel NUCs) remains the only consistently successful approach to running Fuchsia with full graphical capabilities. Current Development and Status (as of 2025).

Current Development and Status (as of 2025)

- Fuchsia is under active development with thousands of monthly commits by Google engineers [8].
- Nest Hub devices are the only official deployment site on which Fuchsia uses a custom user interface layer known as Armadillo in production.
- The access to its developer remains closed; although the source is open, it does not have a public SDK to allow third-party app development when not starting with a core system.
- Several system components, such as Netstack, Scenic (UI stack), and Driver Framework v2, are still under test or not production-grade.
- QEMU/KVM can also perform GUI testing, but it is currently experimental and has only partial success in shell-only environments.

Relevance to This Project

Although it had not been possible to get full graphical access to Fuchsia within a virtual environment in the course of this project, extensive work had been done to:

- Compile Fuchsia with source on Ubuntu 24.04.1 LTS by using fx.
- Test multiple target boards (e.g., core.x64, workstation.x64).
- Use analysis of system activity as reported by console and kernel log during boot process;
- Request feedback of the community and developers through GitHub and mailing lists.

2.5 QEMU/KVM Virtualization

Introduction to Virtualization

Virtualization basically involves the abstraction of computing resources: operating systems, servers, storage arrays, etc., and bringing them out of their physical substrates, thus allowing many OS instances to exist on any single server. This arrangement not only augments hardware utilization but also streamlines system testing and condenses software development cycles [19]. In the case of the recent research, virtualization enables the comparative deployment and comparative analysis of three operating systems, including Android, Haiku, and Fuchsia, in a calculative and replicable real-world experimentation.

QEMU: Overview and Capabilities

QEMU (Quick EMUlator) constitutes an open-source machine emulator and virtualizer capable of executing operating systems and applications for one hardware platform on another. When used in conjunction with Kernel-based Virtual Machine (KVM), QEMU achieves near-native performance on supported hardware through hardware-assisted virtualization (e.g., Intel VT-x or AMD-V).

Key features of QEMU:

- Emulation of diverse CPU architectures (x86, ARM, RISC-V, etc.)
- Live migration and snapshot backup USB, disk and network device pass-through. This product supports such image formats as QCOW2 and RAW.
- Support of virt-manager integration to manage VM in GUI mode.

Flexibility, scriptability, and the headless nature of QEMU make QEMU particularly useful in the research context, where VMs are accessible remotely over VNC or SPICE.

KVM: Native Virtualization with Linux

Linux's Kernel-based Virtual Machine (KVM) extends the operating system into a Type-1 hypervisor that exposes hardware virtualization enhancements (Intel VT-x or AMD-V) to QEMU and thereby enhances guest performance.

Other KVM features are:

- Better and efficient use of CPU through hardware virtualization, Memory ballooning of dynamic RAM, and Memory allocation
- VFIO and virtio drivers I/O devices emulation
- Options for real-time performance tuning.

Since the QEMU/KVM virtual machines are run as distinct Linux processes, property-level separation and Linux-specific monitoring tools compatibility are retained.

virt-manager and libvirt

To streamline virtualization management, Virtual Machine Manager (virt-manager) is employed in concert with libvirt, a toolkit that abstracts hypervisor APIs and supervises VM lifecycles [24]. virt-manager furnishes a graphical user interface for the creation, modification, and monitoring of virtual machines.

libvirt offers command-line control, XML-based configuration, and daemon-based VM orchestration. For the present project, QEMU/KVM virtual machines were generated and administered via virt-manager on an Ubuntu 24.04.1 LTS host system [34]. This was set up as a fixed and performance-boasted Linux operating environment compatible with testing the test operating systems.

Limitations in the Context of This Study

QEMU/KVM is a reliable Android and Haiku test layer of virtualized operating systems, but QEMU/KVM has several limitations on Fuchsia testing:

- Fuchsia does not have ISO images and one has to build the full source and hand-deploy it.
- QEMU's GPU support (e.g., virtio-gpu) is insufficient for Fuchsia's Scenic GUI stack. Wayland, Vulkan acceleration in QEMU: No out-of-the-box support of Wayland or Vulkan acceleration of Fuchsia guests in QEMU.

In spite of such limitations, one could access shell access of Fuchsia through qemu-x64 target. The upcoming additions to the features of virtualization in Fuchsia can fix the existing GUI rendering problems.

Summary

QEMU/KVM, coupled with virt-manager, gives the researcher or a developer a powerful, flexible environment to simulate complex operating-system environments. Despite the successful combination of Android and Haiku, Fuchsia had special difficulties because of its emergent premise and inability to combine well with the existing virtualization layers.

2.6 VNC and RDP Protocols

Remote desktop protocols help in accessing of the graphical interface to virtual machines using distant clients. In the present study, Virtual Network Computing (VNC) and Remote Desktop Protocol (RDP) were employed to enable graphical interaction with guest operating systems running in QEMU/KVM virtual machines. The two protocols also allow the control of the headless operating systems via web browsers using Apache Guacamole and noVNC.

Virtual Network Computing (VNC)

VNC is a protocol-independent, and platform-independent protocol of remote access allowing graphical desktop sharing over a network. Its operation is based on Remote Framebuffer (RFB), wherein the server transmits the frame buffer (display) to the client, while the client sends keyboard and mouse events back to the server.

Key Features:

- Works across unrelated operating systems (cross-platform)
- Support broad in virtualization environment (open source)
- It consumes a very low amount of client-side resources

- Uses TCP (commonly port 5900+display number)
- Lacks encryption by default (often tunnelled through SSH or wrapped in Guacamole for secure use)

In this project, VNC servers have been activated in QEMU virtual machines and served by Apache Guacamole to allow a graphical interaction with Android and Haiku OS guests in-browser.

Limitations:

- No indigenous sound relaying
- Compression lower than RDP, hence the use of increased bandwidth
- Single-session generation is not so effective in multi-user scenarios.

Despite these limitations, VNC was chosen for this project because of its compatibility with QEMU and broad support within open-source ecosystems [32].

Remote Desktop Protocol (RDP)

RDP is a proprietary protocol that Microsoft has developed to have enhanced compression and security in the remote desktop connection. It enables users to access a remote Windows or some Linux machine with the help of RDP servers such as xrdp or freerdp.

Advantages over VNC:

- Improved compression, causing a smoother performance across low bandwidth links
- Audio redirection is supported, sharing of the clipboard, etc., printer support, In-built encryption and authentication processes
- Can carry out several user sessions at once

However, RDP was not the primary protocol used in this project due to limited out-of-the-box support in the target operating systems (Android and Haiku). Although xrdp is possible to install on Linux-based systems, Android-x86 and Haiku need a specific compilation procedure, which adds complexity and instability to the virtual environment.

A tabular comparison as Table 1 is given next to VNC and RDP, two of the more well-known remote-desktop protocols that are used in a virtualized environment. VNC is native to QEMU and has a wide open-source compatibility, but, in turn, has less sophisticated features like forwarding sound and poor encryption. In comparison, RDP provides a better compression ratio, allows multi-session and includes more extensive security features, but it requires more configuration work in Linux-based guest environments. For the ends of the current project, VNC was preferred due to its compatibility with QEMU at the command line level and flexibility of configuration.

For this project, VNC was the primary choice, enabled in QEMU via the `-vnc` flag and accessed through Guacamole [6]. These tools gave access to the website without any local client installation, which met the accessibility objectives of the project.

Security Considerations

Both VNC and RDP are, by design, safe. VNC is unencrypted, unless it is tunnelled over SSH, or wrapped over HTTPS with Guacamole. RDP provides stronger authentication, but known vulnerabilities require proper configuration (e.g., using TLS and disabling older protocol versions).

Feature	VNC	RDP
Protocol Type	Remote Framebuffer (RFB)	Microsoft Proprietary
Audio Support	(by default)	Yes
Compression	Basic	Advanced
Security	None (can be tunneled)	TLS + NLA
Multi-session Support	Limited	Supported
Integration in QEMU	Native	Requires extra configuration

Table 1: Comparative Features of VNC and RDP Protocols for Remote Desktop Access in Virtualized Environments

2.7 Apache Guacamole and noVNC

Apache Guacamole is a clientless remote desktop gateway, which can be written using open-source software and is available without a client. Apache Guacamole allows individuals to log in to systems via their browsers using VNC, RDP, and SSH. Due to the above features of being secure, scalable and platform independent, it is becoming popular in student laboratories and corporate network configurations. What really sets it apart is that it needs no client software—so it works smoothly with HTML5-compatible browsers, which matters a lot in virtual lab systems, distance-learning platforms, and thin-client setups [6].

Technically, Guacamole is pegged on two components, namely, the Guacamole client that is implemented fully in a browser using JavaScript and HTML5, and the guacd daemon that converts established protocols, e.g. VNC or RDP, to protocols that the web can process. Such a divided architecture means that Guacamole is very portable, firewall-friendly and device-indifferent.

When it is mentioned about virtualization, Guacamole is frequently paired with QEMU/KVM or Proxmox to expose virtual machines (VMs) through the browser [9]. It is also compatible with Linux-based back ends, provides session management features, file transfer, synchronization of the clipboard and authentication add-ons such as LDAP and logins over a database. This has presented it as an enterprise-friendly alternative to large-scale academic testbeds as well as remote VM management in multi-user virtual settings.

Recent literature highlights Guacamole's role in:

- Remote OS access in the form of cloud computing labs [30].
- Cybersecurity training on a virtual test environment [17].
- Distance learning and education on thin-client architecture [23].

These use cases focus on the capability of Guacamole to provide a uniform access experience that supports comparing virtualized operating systems, in this case, Android and Haiku.

3. System Design

In this chapter, the author discusses the architectural design and strategy deployed in utilising the multi-operating system in a virtualized Linux environment; the operating systems used are Android, Haiku and Fuchsia. The virtualization aspect of design focuses on QEMU/KVM in a guest environment with the help of Apache Guacamole as the application to allow the graphical interface of the virtual machine on the browser. The chapter elaborates on host setup, the operating system deployment plans, remote access provisions, as well as security measures. System interaction diagram in the case of a high-level as well is also provided so as to depict the communication amongst system components. This structural design makes it possible to properly follow the implementation in the following chapter, as it is reproducible, scalable, and suits the current virtualization trends of virtualization.

3.1 Host system setup

This section presents a comprehensive overview of the virtualisation host system used to deploy and test the three selected operating systems—Android, Haiku, and Fuchsia. The setup was designed to ensure high compatibility with KVM/QEMU virtualisation, enable remote GUI access, and support real-time performance monitoring. The host environment was based on **Ubuntu 24.04.2 LTS**, a stable Linux distribution known for long-term support and reliability.

Host Operating System and Hardware Configuration

In this project, the given hosts system is HP EliteBook 850 G8 Notebook PC chosen due to a powerful processor, VT support, and adequate RAM and data storage facilities. Intel VT-x hardware acceleration, a requirement of operating with the virtual machine with QEMU/KVM, is supported by its configuration.

```
Linux nakrani-HP-EliteBook-850-G8-Notebook-PC 6.14.0-24-generic #24-24.04.3-Ubuntu SMP PREEMPT_DYNAMIC Mon Jul 7 16:39:17 UTC 2 x86_64 x86_64 x86_64 GNU/Linux
```

Figure 1: Screenshot of `uname -a` showing Ubuntu 24.04.1 LTS host system

Here, Figure 1 shows the Screenshot of `uname -a` showing Ubuntu 24.04.1 LTS host system.

The `free -h` command output is also captured below to showcase RAM and swap availability:

- **RAM (Total):** 15 GiB
- **RAM (Free):** 10 GiB
- **Swap:** 4 GiB

	total	used	free	shared	buff/cache	available
Mem:	15Gi	2.7Gi	10Gi	663Mi	2.7Gi	12Gi
Swap:	4.0Gi	0B	4.0Gi			

Figure 2: Screenshot of RAM and Swap Memory Statistics via `free -h`

Here, Figure 2, shows the memory summary using `free -h`.

```

Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Address sizes:          39 bits physical, 48 bits virtual
Byte Order:             Little Endian
CPU(s):                 8
On-line CPU(s) list:    0-7
Vendor ID:              GenuineIntel
Model name:             11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
CPU family:             6
Model:                 140
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):              1
Stepping:               1
CPU(s) scaling MHz:     23%
CPU max MHz:            4200.0000
CPU min MHz:            400.0000
BogoMIPS:               4838.40
Flags:                  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm const
nt_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vnx est tm2 ssse3
sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l
2 cdp_l2 ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdt_a avx512f avx5
12dq rdseed adx snap avx512ifma clflushopt clwb intel_pt avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves split_lock_detect user_shstk dthe
rm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp hwp_pkg_req vnm1 avx512vbmi umip pku ospke avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_b
italg avx512_vpoperndq rdpid movdiri movdir64b fsrm avx512_vp2intersect md_clear ibt flush_lid arch_capabilities

Virtualization features:
Virtualization:         VT-x
Caches (sum of all):
  L1d:                  192 KiB (4 instances)
  L1i:                  128 KiB (4 instances)
  L2:                   5 MiB (4 instances)
  L3:                   8 MiB (1 instance)
NUMA:
  NUMA node(s):         1
  NUMA node0 CPU(s):    0-7
Vulnerabilities:
Gather data sampling:    Vulnerable
Ghostwrite:             Not affected
Itlb multihit:          Not affected
L1tf:                   Not affected
Mds:                    Not affected
Meltdown:               Not affected
Mmio stale data:        Not affected
Reg file data sampling: Not affected
Retbleed:               Not affected
Spec rstack overflow:   Not affected
Spec store bypass:      Mitigation; Speculative Store Bypass disabled via prctl
Spectre v1:             Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Spectre v2:             Mitigation; Enhanced / Automatic IBRS; IBPB conditional; PBRSSB-eIBRS SW sequence; BHI SW loop, KVM SW loop
Srbds:                  Not affected
Tsx async abort:        Not affected

```

Figure 3: Terminal Output of Iscpu Confirming Virtualisation Support

Here, Figure 3 presents the output of Iscpu, confirming the processor model (Intel i5-1135G7) and the availability of virtualisation flags (VT-x support).

Category	Details
Hardware Model	HP EliteBook 850 G8 Notebook PC
Processor	Intel Core i5-1135G7 @ 2.40 GHz (4C/8T)
Graphics	Intel Xe Graphics (TGL GT2)
Memory (RAM)	16.0 GiB DDR4
Disk Storage	512.1 GB SSD
Firmware Version	T76 Ver. 01.21.00
Operating System	Ubuntu 24.04.2 LTS 64-bit
GNOME Version	GNOME 46
Windowing System	Wayland
Kernel Version	Linux 6.14.0-24-generic

Table 2: Host system specifications

Source: [3]

Here, Table 2 shows host System Specifications.

Hypervisor and Management Layer

This project was based on QEMU as an emulator, and hardware-assisted virtualization was handled using KVM (the Kernel-based Virtual Machine). The following packages were installed:

- qemu-kvm
- libvirt-daemon-system
- libvirt-clients
- bridge-utils
- virt-manager

The installation was done using the following commands:

```
sudo apt update
sudo apt install -y qemu-kvm libvirt-daemon-system libvirt-clients bridge-utils virt-manager
```

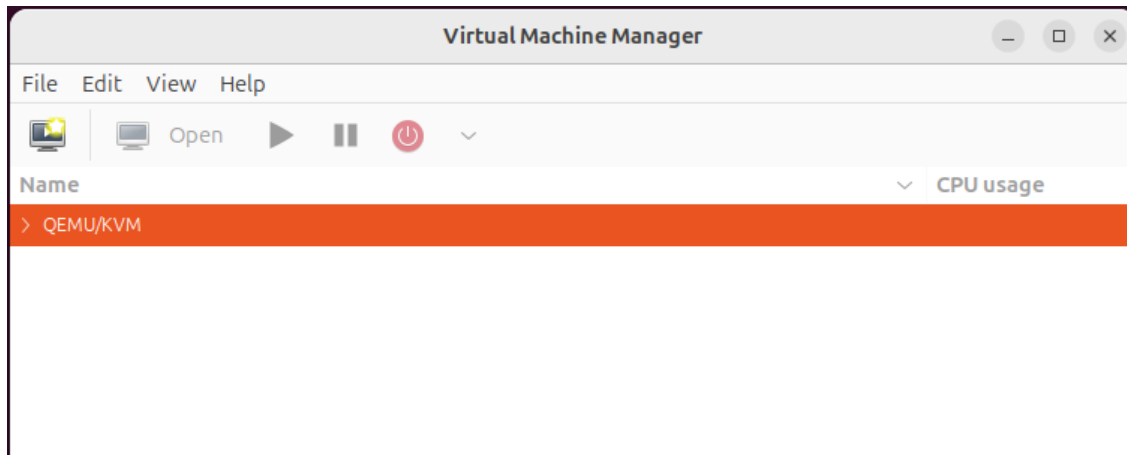


Figure 4: Screenshot of Virtual Machine Manager (virt-manager) Interface

Here, Figure 4 shows the virt-manager interface, listing all virtual machines and their running states.

3.2 OS Deployment Strategy

The operating system (OS) deployment strategy in this project centers on installing and configuring three distinct guest operating systems—Android-x86, Haiku OS, and Google Fuchsia—within a QEMU/KVM virtualized environment on Ubuntu 24.04.1 LTS. Both systems were chosen due to the architectural peculiarity and the possibility of testing related to remote access. The deployment procedure highlighted on the accessibility of the graphical interface, responsiveness in the system as well as the compatibility of the system with the browser-based tools like the Apache Guacamole.

Environment and Platform of Virtualization

The deployment process used QEMU (Quick Emulator) with KVM (Kernel-based Virtual Machine) acceleration. QEMU provides full system emulation, while KVM offers near-native performance using hardware virtualization extensions (Intel VT-x or AMD-V). This combination enabled efficient and flexible emulation of guest OS environments [3].

Virtual Machine Manager (virt-manager) served as the frontend management interface for creating and configuring virtual machines, allowing control over disk images, CPU allocation, network bridging, and graphical settings. Android was built on the x86-based hypervisor virtualized environment, whose core was based on libvirt, which abstracted the interaction with the hardware and controlled VM lives using a common API.

Deployment Breakdown by OS

1. Android-x86 9.0 (Pie)

- It was installed with a disk image as a .qcow2 image using the virt-manager.
- It is set up with 2 CPU cores, 2 GiB RAM, and 16 GiB storage.
- GUI access enabled through DroidVNC-NG (installed via apk file).
- The access to which is via Apache Guacamole over VNC.
- Image file: android-x86-9.0.qcow2 - [12], [33].

2. Haiku OS (Nightly Build, x86_64)

- Installed out of an official nightly .iso and transformed to .qcow2 by virt-manager.
- It is configured P1, 1 CPU core, 1 GiB RAM, and 8 GiB disk space.
- VNC server installed using AGMS VNC 4.0 for BeOS, configured through [13], [1], [2].
- Image file: android-x86-9.0.qcow2 - [12], [33].

3. Fuchsia OS (Build from Source)

Google's Fuchsia OS is a next-generation, capability-based operating system that diverges fundamentally from traditional Linux-based systems by employing the Zircon microkernel instead of the monolithic Linux kernel [7]. In light of this, Fuchsia needs custom source compilation, specific tooling, and a custom emulation environment to be deployed. This made the deployment much harder than Android or Haiku.

Although the official Fuchsia emulator is capable of GUI output through the framebuffer, GPU passthrough features are also necessary, which (until recently) were incompatible with most commodity devices. Therefore, on the HP EliteBook 850 G8 host, only serial console access was possible [27].

Build and Boot Environment

To run Fuchsia on a FEMU virtualized environment, the steps provided by Google as an official guide were used, among which are:

- A git clone of the Fuchsia Source Tree.
- Setting Up the Fuchsia Environment via ffx (Fuchsia CLI tool) and other scripts(fx, jiri).
- Building Fuchsia with GN/Ninja, for x64.
- Starting with FEMU by means of a custom fx emu command which creates an emulated environment with a serial console.

The following minimum hardware (host operating system) allocation was used FEMU:

- CPU Cores: 4
- RAM: 16 GiB
- Disk: 80-90 Gib to 130-140 GiB: the size is a matter of hardware configuration



Figure 5: Fuchsia Emulator Boot Screen Showing ASCII Logo on Serial Console

- Access Type: FEMU (GUI not working) and shell script

This Figure 5 screenshot confirms that the Fuchsia emulator successfully loaded the kernel and initialised the serial shell interface before user input.

Access and Interface Strategy

```

jatin@jatin-Inspiron-3558:~/fuchsia$ ffx emu start
Auto resolving networking to user-mode. For more information see https://fuchsia.dev/fuchsia-src/development/build/emulator#networking
Logging to "/home/jatin/.local/share/Fuchsia/ffx/emu/instances/fuchsia-emulator/emulator.log"
Waiting for Fuchsia to start (up to 60 seconds)...
Emulator is ready.
jatin@jatin-Inspiron-3558:~/fuchsia$ fx shell
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run 'fx metrics'
To opt in or out, run 'fx metrics <enable|disable>'

ls
$ ls
bin
blob
block
boot
boot-bin
config
data
dev
mnt
pkg
pkgfs
svc
system
tmp
$

```

Figure 6: Fuchsia OS Serial Console Shell Running via fx on Ubuntu 24.04 Host

Although the official Fuchsia emulator supports framebuffer-based GUI output, GPU passthrough capability is required, which was not supported in the test hardware, an HP EliteBook 850 G8. In this environment, therefore, graphical access was not possible as VNC or RDP. The backup approach was the serial shell interface through `fx shell` command that connects to the serial console of the emulated environment. The basic command-line interface of Fuchsia could then be accessed to validate the system. This shell is an entry-level command-line interface to Fuchsia, which can be used to

perform some basic tests of system functionality and boot sequence checks.

Figure 6 screenshot shows the active serial screen of Fuchsia OS connected with fx shell, which means that Fuchsia was successfully booted in CLI mode, although the GUI did not work.

Setbacks and Limitations

Although the user through the official guide step by step, there are a number of problems that caused the failure of the GUI deployment:

1. An error is encountered when trying to render on the Gentoo GPU using Ubuntu 24.04 TLS Host. The emulator could not display anything more than the bootloader, even with accelerated QEMU and hardware virtualization turned ON. This was traced to framebuffer support issues and a lack of proper GPU pass-through [27].
2. Fuchsia Issue #425120578: Emulator Display Stuck on Boot The upstream issue reported on Fuchsia's bug tracker confirms that GUI boot gets stuck or crashes due to compatibility mismatches with Ubuntu host systems and non-Google hardware [14].
3. Intel GPUs have no support in Emulation Mode. Fuchsia's rendering pipeline expects supported GPU drivers, which are available only for specific platforms (e.g., Pixelbook, NUC). The Intel Xe GPUs involved in this project are not supported for use with QEMU-based emulator GUI output.
4. Community Feedback- Other developers on YCombinator and AndroidPolice complained of the same difficulty. The OS, while promising, is still in an experimental phase for non-Google hardware deployment [15], [22].

Outcome and Current Status

In this arrangement, the GUI was not able to start up or test, even though the shell interface was in working order. There will be, consequently:

- Fuchsia was left headless and could be reached through fx shell.
- Under such a constraint, GUI-based testing and remote access through VNC were out of the question.
- The guest OS did not have the same capacity to integrate into the Apache Guacamole dashboard as Android and Haiku.

This demonstrates the early-stage nature of Fuchsia for open virtual deployment and highlights the need for GPU passthrough or dedicated hardware (e.g., Pixelbook, Intel NUC) for complete system emulation.

GuestCPU OS	CPU Cores	RAM	Disk	Access Type
Android	2	3 GiB	20 GiB	VNC via Guacamole
Haiku	2	2 GiB	4 GiB	VNC via Guacamole
Fuchsia	4	16 GiB	min. 80-90 max. 130-140 GiB	FEMU/ Shell access

Table 3: Resource Allocation and Access Type for Guest Operating Systems in QEMU/KVM Virtual Environment.

Source: [13]

Each configuration profile was used in the deployment of each of the operating systems. QEMU/KVM allowed adjusting allocations according to the demands that a given system had. Table 3 gives a comparative description of the resource allocation and access mechanism upon which each guest OS is deployed in QEMU/KVM. The Android-x86 virtual machine had 2vCPUs, 3GiB of RAM and a 20GiB qcow2 disk image and was accessed using VNC through Apache Guacamole. Likewise, Haiku OS VM had 2 vCPU, 2 GiB RAM and 4 GiB disk but with Guacamole access. Due to the

ongoing limitations of Fuchsia’s graphical interface in virtualized environments, the system was booted with 4 vCPU, 16 GiB RAM, and a min. 80-90 max. 130-140 GiB disk (depends on hardware), and accessed through the serial shell interface.

Disk and Image Management

Any given guest operating system employed the qcow2 image format, which was dynamic in disk allocation. Libvirt was used to manage the default storage place of such pictures, which was `/var/lib/libvirt/images`. Here, Figure 7 displays the default storage pool (`/var/lib/libvirt/images`) used by QEMU/KVM for storing virtual machine disk images in `.qcow2` format.

Two volumes are configured:

- `android-x86-9.0.qcow2` that is allocated 20 GiB.
- `haikunightly.qcow2` size: 4 GiB.

This confirms the provision of persistent storage of guest OS instances in the default libvirt-managed directory. The qcow2 format supports snapshots, compression, and dynamic resizing, making it ideal for virtualization workloads [3]. Qcow2 stands for "QEMU Copy-On-Write version 2," which enables efficient storage usage and layered image snapshots. See Figure 7.

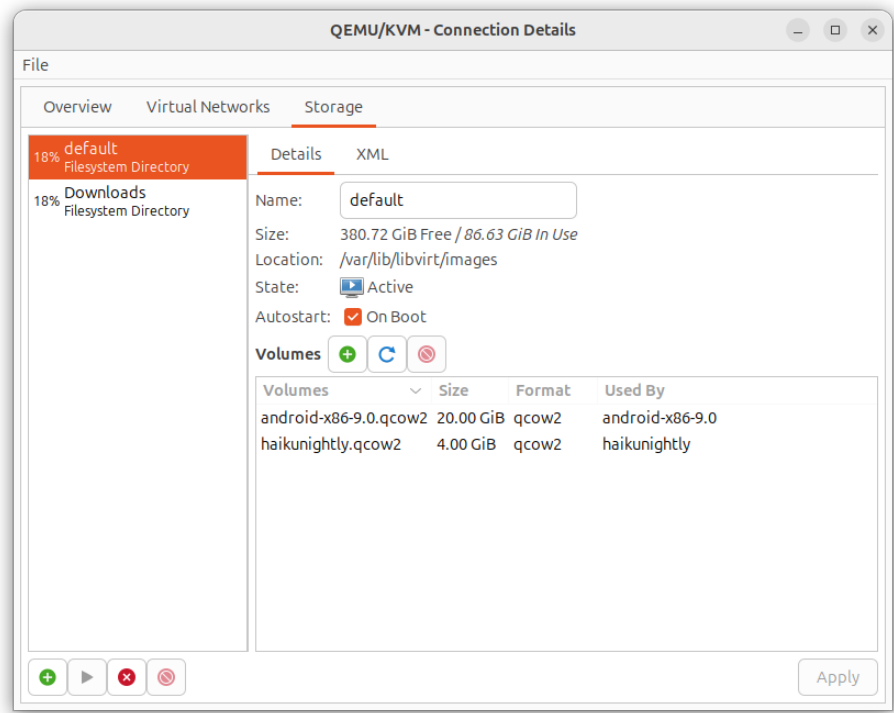


Figure 7: QEMU/KVM Storage Pool Configuration Showing qcow2 Disk Images for Android and Haiku Virtual Machines

Virtual Disk Format and Networking

Each of the virtual machines was based on QCOW2 disk images, which allowed snapshotting and on-the-fly resizing. Networking was set to be through NAT mode, where outbound traffic was open so that updates or any package could communicate. To remote-control the GUI of the VMs, QEMU ran VMs with the port that the VNC sessions listened to facing localhost, connecting to Apache Guacamole internally. Virtual machines were all of the qcow2 format, which

makes it possible to dynamically resize or reset disk space. The networking was enabled on NAT mode, allowing outbound access to the internet and isolating guest OSs to the external networking. For remote GUI streaming, VNC services on each guest OS were bound to the host's localhost interface and accessed internally via Apache Guacamole's HTML5 interface.

Automation Support and Snapshot Support

The virtualisation environment was designed to support snapshotting of virtual machine (VM) states using libvirt and virt-manager. This was particularly useful in the iterative process of installing Android and Haiku systems, when connecting to a VNC server, tweaking appearance or debugging a GUI or installing a software package usually took trial-and-error. The use of snapshots meant that setup times were dropping as the system could be returned to a clean state much faster without the need to restore the system to its full state. The creation and control of snapshots was done via the GUI of virt-manager, where both live and offline snapshot functionality is available. This made the design highly modular and test-friendly, allowing each guest OS to be restored before any experimental changes [24], [35].

3.3 Remote Access Architecture

One of the essential parts of the project was the possibility of remote access to the guest operating systems (Android-x86, Haiku OS, and Fuchsia OS) installed under a QEMU/KVM virtualized system by means of a browser session. The main goal was to provide real-time graphical access, which did not require the use of local client software installation. They managed to do this with the Apache Guacamole and Shell, which used the VNC protocol to forward the display using framebuffer.

Overview System Design

The access model was developed using the layered architecture in which each guest VM would send its graphical shell across using VNC, which could be viewed using a web client linked up to the gateway Guacamole. This allowed seamless session initiation and input capture through a browser [18].

QEMU/KVM VNC Configuration

To enable remote graphical access without requiring additional client installations, each guest virtual machine (VM) was configured using QEMU's built-in VNC (Virtual Network Computing) server. The functionality naturally enables lightweight usage, the virtual machines can be accessed and managed without dedicated displays devices and peripheral software on the host system. That is why the VNC mode of QEMU is very applicable in the headless system and remote access applications. The boot process of each VM was invoked using the `-vnc` flag and this aired the graphical console to a port which was allocated to the VM and was also bound to the local host.

The scheme allowed that several guest systems, including Android, Haiku and Fuchsia, could be allocated each a separate display port, and simultaneously controlled.

Moreover, each operating system was configured to automatically launch its graphical user interface (GUI) on boot. This dispensed with any necessity to log in to the console by hand or redirect the display into the VM and made an easy hands-off transition between the boot prompt and a full graphical interface after a VNC connection was made. This minimalist and flexible configuration provided a reliable backend for remote access via Apache Guacamole, enabling the simultaneous testing of multiple operating systems, even without physical monitors or external client software [3].

This diagram Figure 8 illustrates the overall architecture:

Guest OS (VNC Output) → QEMU/KVM Host → Apache Guacamole → Browser Client

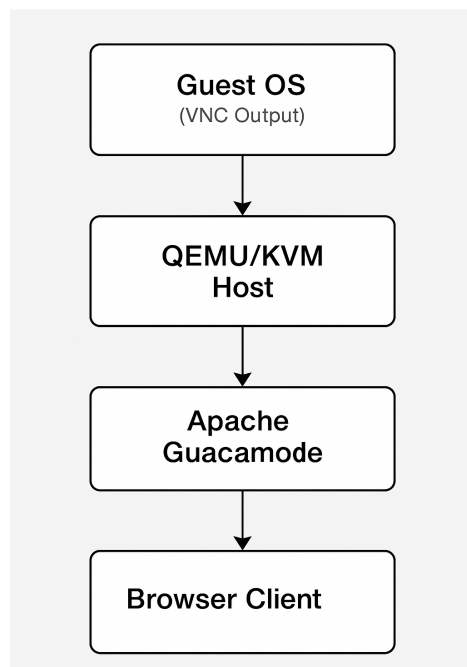


Figure 8: Remote Access Architecture Diagram

Apache Guacamole Gateway

Figure 9 shows the Guacamole login page, listing configured OS sessions.

- Apache Guacamole is a clientless remote desktop gateway, written in HTML5. It was a connection point to get to the guest OSes using VNC in this project.
- Installed via manual set-up. In `/etc/guacamole/user-mapping.xml`, the user had to map each VM manually.
- Connected to localhost VNC ports (e.g., 5905 for Android, 5903 for Haiku).
- The web interface will be exposed in the form of local authentication.

System Flow and Access Logic

The end-to-end access cycle followed this sequence:

1. User opens browser → Guacamole portal loads.
2. User logs in → selects guest OS.
3. Guacamole connects to VM's localhost VNC port.
4. Guest OS GUI streamed via HTML5 canvas.
5. Keyboard/mouse input transmitted in real-time.

This Figure 10 outlines the user-to-guest interaction lifecycle, showing browser → Guacamole → VNC → QEMU VM.

Security and Access Control

In this project, remote access architecture was ensured with specific attention given to localized security. All VNC ports were bound to localhost inside the host system and could only be accessed through the browser by using Apache

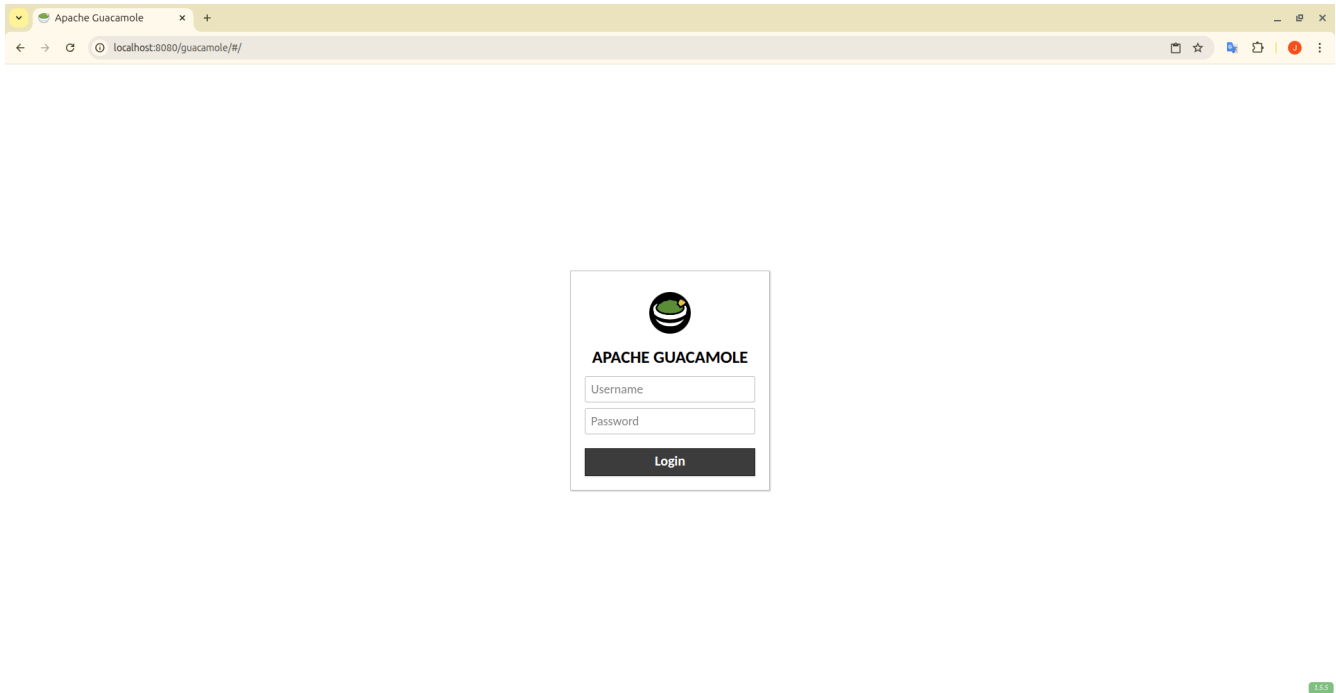


Figure 9: Apache Guacamole Web Interface

Source: [6]

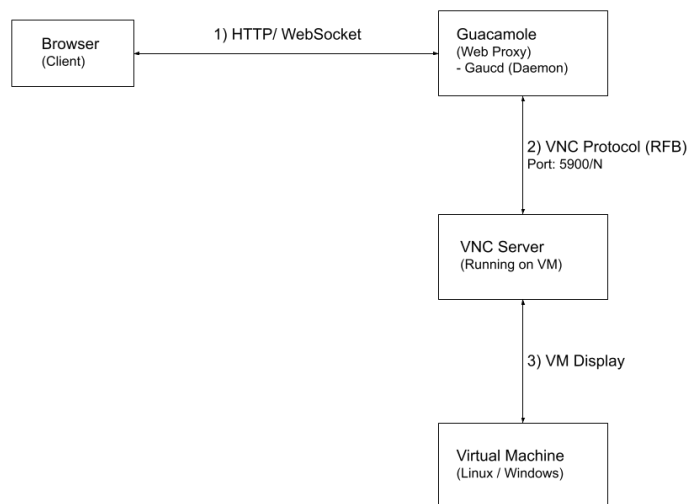


Figure 10: Remote Access Flowchart

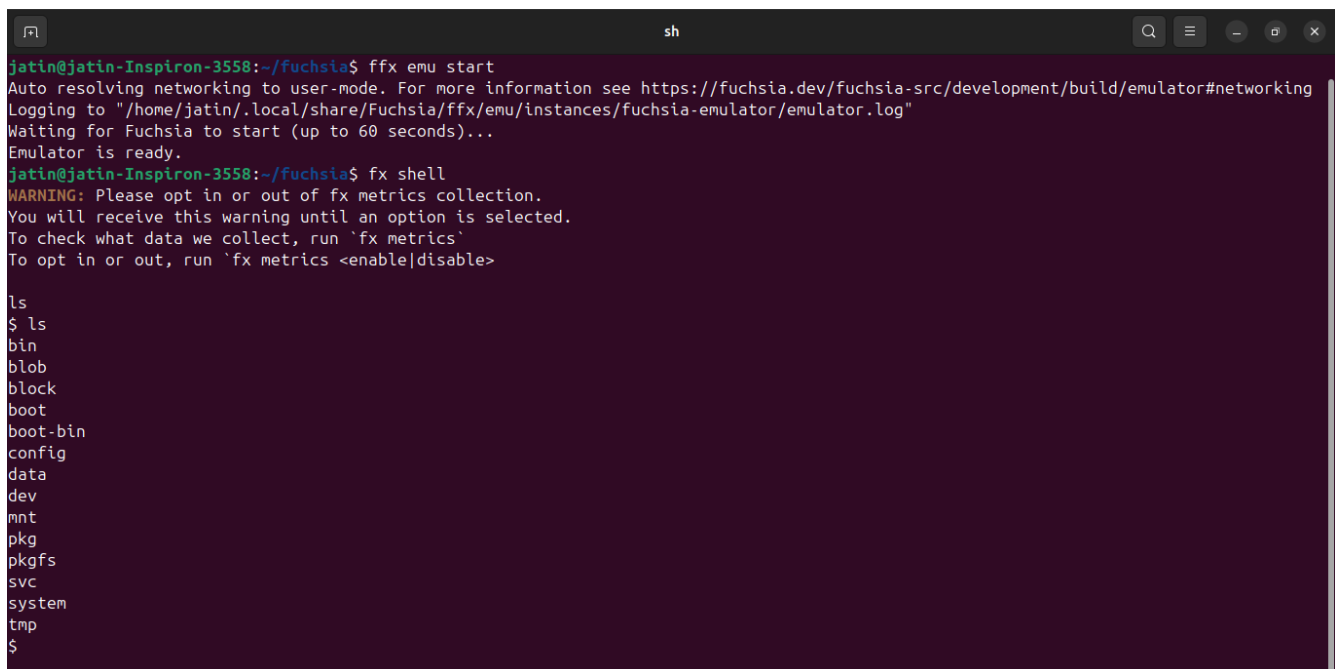
Guacamole. This makes sure that raw framebuffer data never travels off the machine in the clear, which means the attack surface is highly decreased. The Apache Guacamole interface itself was optionally secured using HTTP Basic Authentication, and in production environments, it can be reverse-proxied behind Nginx to enable HTTPS with TLS encryption [18]. Even though this project was implemented in a restricted setting, later versions can support a more

efficient user identity verification, enabling features such as multi-user session management using LDAP or SQL-based Guacamole configurations.

Remote Access Limitations

Although the graphical remote access of Android and Haiku virtual machines was successfully implemented, there were significant limitations that were realized, especially when Fuchsia OS was to be involved. Because it is experimental and depends on the Zircon microkernel, at present, Fuchsia does not have comprehensive VNC or framebuffer rendering when used in QEMU-based environments on non-Google platforms. After successfully booting using `fx shell` command into a serial shell, the system did not show any GUI output. Efforts to display the graphical sessions using VNC turned into blank screens or crashes of the emulators, according to the reports and feedback of developers and the community. This hardware limitation is due to the fact that GPU passthrough is not supported, and the inability to use Intel Xe graphics on the HP EliteBook 850 G8 host.

This Figure 11 screenshot demonstrates the terminal-based interface of the operating system Fuchsia, which is accessed with the help of the `fx shell` command typed in after starting the emulator. The result is a confirmation that the Fuchsia CLI was successfully accessed in serial shell mode, and it enumerates the main directories associated with the core system `bin`, `blob`, `boot` and `pkgfs`. This mode was the only operative interface because the graphical rendering did not work. It highlights the headless configuration's operational status, despite limitations for browser-based platforms like Apache Guacamole.



```

jatin@jatin-Inspiron-3558:~/fuchsia$ ffx emu start
Auto resolving networking to user-mode. For more information see https://fuchsia.dev/fuchsia-src/development/build/emulator#networking
Logging to "/home/jatin/.local/share/Fuchsia/ffx/emu/instances/fuchsia-emulator/emulator.log"
Waiting for Fuchsia to start (up to 60 seconds)...
Emulator is ready.
jatin@jatin-Inspiron-3558:~/fuchsia$ fx shell
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run 'fx metrics'
To opt in or out, run 'fx metrics <enable|disable>'

ls
$ ls
bin
blob
block
boot
boot-bin
config
data
dev
mnt
pkg
pkgfs
svc
system
tmp
$

```

Figure 11: Host Fuchsia OS Running in Serial Shell Mode on Ubuntu Host

As part of the troubleshooting process during the Fuchsia OS deployment, the user Jatinkumar engaged with the official Fuchsia Discord channel to seek community guidance [27], [14]. Once the successful completion of all stages of builds, the emulator always ran into a dead end after the presentation of the name of Fuchsia. A message was posted summarizing the issue and including screenshots of the emulator state and diagnostic attempts (see Figure 12). This outreach was an active accountability to the Fuchsia developer community, and a showcase of cases not being reported well, even with correct configuration and compatibility with the environment. This outreach reflected proactive engagement with the Fuchsia developer community and demonstrated transparent reporting of unresolved boot limitations despite accurate configuration and environment compatibility.

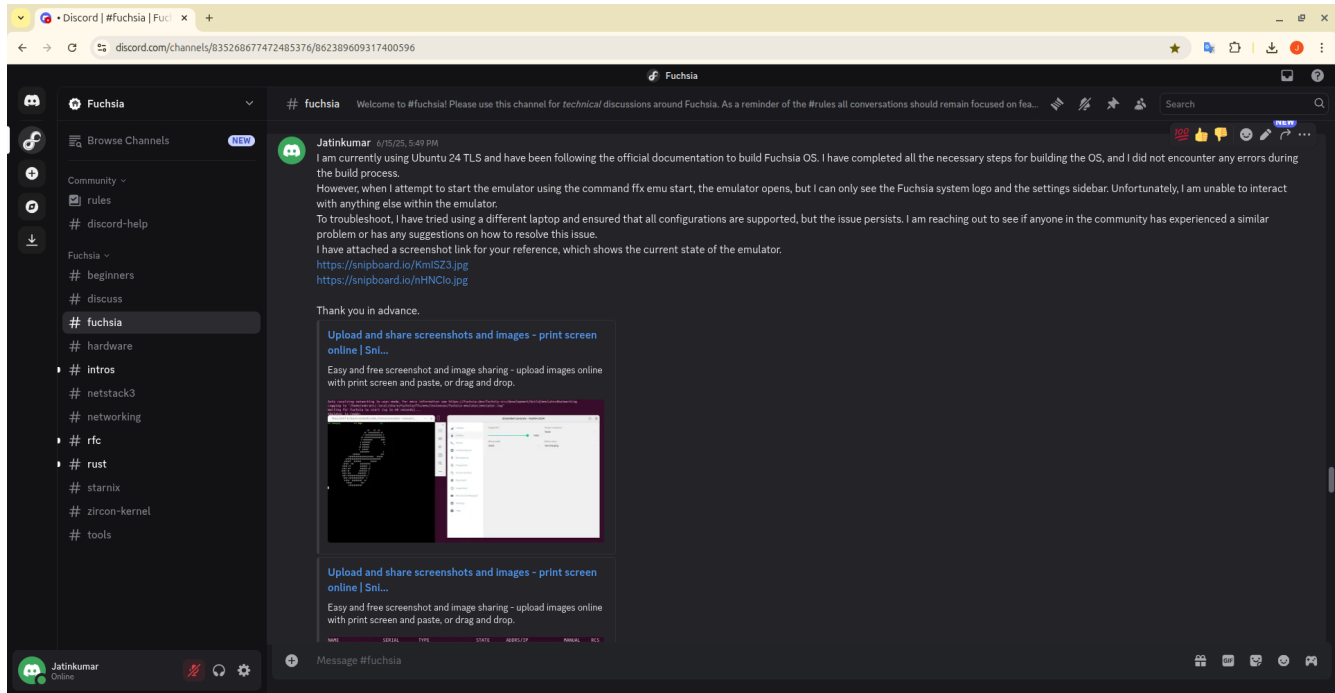


Figure 12: Discord Query Posted by Jatinkumar on Fuchsia Support Channel

Source: <https://discord.com/channels/835268677472485376/862389609317400596>

3.4 Security Consideration

The virtualization environment designed for this project emphasizes localized security, ensuring that virtual machine (VM) access and system integrity are maintained without relying on external exposure. Given that all guest operating systems—Android-x86, Haiku OS, and Fuchsia—were accessed through remote protocols such as VNC or serial shell interfaces, special care was taken to contain access within the host network while ensuring that remote connectivity was still functional for the user.

Host-Level Isolation and VNC Containment

Its architecture was based on QEMU/KVM, and each virtual machine was set up with VNC output directed to the localhost interface only. This made sure that there was no framebuffer or graphical data sent on the open networks. The system, instead of exposing VNC sessions directly, used Apache Guacamole as an intermediate. Any remote access of the GUI was through Apache Guacamole, which safely communicated to VNC-bound ports on localhost. Guacamole was a secure web-based proxy and pulled display streams off localhost and displayed it to the user in an HTML5 browser interface. This technique was abstract and restrictive on exposure and kept the network level of isolation and while the same time supporting interactive sessions.

Moreover, Guacamole itself was running in a Docker container, and it added one more point of isolation on an application level. This containerized environment limits access to the Guacamole service and guarantees minimal privilege execution during its operations on the host system.

Authentication and Access Management

The virtualization environment that is developed under this project highlights the localized security so that the access of virtual machine (VM) and system integrity are preserved without the need to depend on exposure. Since all Android-x86,

Haiku OS, and Fuchsia were retrieved using remote protocols, i.e., VNC, or serial shell connection, all endeavors were made to ensure that the access remained within the host LAN and at the same time, the user was able to communicate effectively with the remote machine.

Host-Level Isolation vnc containment

The system was based on the architecture of QEMU/KVM, in which every virtual machine was set up with VNC output strictly attached to the localhost port. This made sure that there was no framebuffer or visual representation data sent across broadcast networks. Instead of exposing VNC sessions directly, the system continuously used Apache Guacamole in the middle. Any web-based GUI client was connected with Apache Guacamole, which safely connected VNC-bound ports in localhost. As a safe web-based proxy, Guacamole obtained display streams on the localhost and displayed them to the user interface, constrained in the browser interface (HTML5). The approach was not only abstract and restricted the visibility but also supported a network-level isolation with the capability of an interactive session [18].

In addition, it can be kept in mind that Guacamole installation in this project was native, but can also be hosted in a Docker container to have another isolation layer at the application level, too.

Authorization and Access Control

The default web interface in the Guacamole gateway in the current installation made it possible to use the basic authentication. Each of the users had to log in through the browser and then start their VM session. SQL-based authentication and LDAP interaction have not been enabled, but were fully supported in the project implementation and will be the future-proof method to use in deployments requiring multiple users and at higher levels of security. The guacamole was the only access tool in the test, and all the remote sessions went through a verified gateway.

Further, all VNC ports used by QEMU were actually mapped to loopback to avoid access by other machines on the network. Only the predefined component Guacamole service had been permitted to access these ports, and this continued to be the backend of the host system.

Considerations of encryption and TLS

Although encryption using TLS was not established in this version of the system, given that it is still in its developmental and single-user stages, the architecture can support safe modifications. For example, Guacamole's frontend can easily be reverse-proxied behind Nginx or Apache HTTP Server to provide HTTPS encryption, especially when integrated with Let's Encrypt certificates or internal certificate authorities. In real-world use, this would keep all data-in-transit, i.e. keyboard and mouse input, secure even against possible interception. The Guacamole documentation provides several ways of adding TLS support and HTTPS proxying.

TLS encryption can be regarded as a requirement in production-quality deployments. Because the Guacamole container is available locally only and not connected to the external networks, encryption-in-transit was not a priority in its implementation. This would enable the Secure Guacamole sessions to be secure enough to be used in an enterprise or educational setting, particularly when combined with two-factor authentication (2FA) plugins, SQL user management, or LDAP-based role segregation.

Snapshot Rollback and Data Integrity

Data integrity and recovery were also given some security consideration. Snapshots were important in having stable system states. Using libvirt's native snapshot capabilities, the user was to create pre-configuration and post configuration VM checkpoints. In case of an operating system misconfiguration or application failure, it was possible to revert the VM to the same state by using these snapshots in a short period of time. The mechanism was most helpful in the implementation of iterative configuration of Android and Haiku virtual machines when installation procedures were different in different packages and services [25], [24]. Snapshots provided non-destructive testing as well, particularly when there was VM-level configuration of the network or desktop service experimentation in areas that required installing VNC servers within the

guest OS.

3.5 System Diagram

In this part, the author provides an architecture of the virtualization-based testbed to deploy and manage three guest operating systems, including Android-x86, Haiku OS, and Fuchsia OS. The system incorporates virtualization technology, remote access software and security mechanisms to allow isolated and controlled environments to conduct comparative tests.

Architecture Overview

The bottom of the system is the host operating system, which is Ubuntu 24.04.2 LTS installed in HP EliteBook 850 G8 with 16 GiB of NAND flash memory and 11th Gen Intel Core i5 processor. This machine has hardware-assisted virtualization (VT-x), which has been confirmed through the `lscpu` outputs. The virtualization level includes QEMU as the hardware emulator and KVM as the hypervisor, giving performance-optimised virtualisation and access to hardware at the native level. The `libvirt` API is used to control these components, and the `virt-manager` is the graphical interface that can be used to define, launch, and monitor virtual machines. The guest OS level contains Android-x86, Haiku OS, and Fuchsia OS, which have a minimalized set of resources (CPU, RAM, disk) and a particular access protocol accordingly to their support of GUI. Android and Haiku have a graphical interface, accessible using VNC, whereas Fuchsia can only be accessed through a serial shell interface (because the implementation does not support a GPU passthrough).

Remote Access Flow

The architecture provides a user GUI access (through the browser) to the guest operating systems with Apache Guacamole serving as a web gateway. Through Virtual machines, VNC servers are only bound to the localhost and cannot be exposed to the external network. Apache Guacamole, when run under Docker, passes authenticated traffic to these VNC endpoints and provides them through a normal HTML5 canvas in any current web browser. This eliminates the need for client-side software, fulfilling the project's objective of a platform-independent and lightweight access model. In case a graphical rendering did not work, an instance of Fuchsia- the system had serial shell access only. Android and Haiku were viewed in Guacamole only because the emulator was not ready to run VNC-based graphical tools. Fuchsia was not connected to the Guacamole system, and its functioning was via Serial Shell. VMs are accessible at addresses such as `http://localhost:6080/vnc.html` and allow direct GUI rendering through browsers on supported guests. The snapshot function in this system is also enabled through `Virt-Manager` and `libvirt`. It can be rolled back and recovery done, particularly when doing some installations or configurations of software within Android and Haiku virtual machines [25].

Figure 13 shows Ubuntu Host → QEMU/KVM + libvirt → Android/Haiku (GUI via VNC) & Fuchsia (Serial Shell) → Apache Guacamole (Android/Haiku only) → Browser Client. The Fuchsia VM was not considered part of the Guacamole pipeline, and only shell interactions were supported.

Reflection on Fuchsia Deployment

In the course of Fuchsia OS deployment, it was observed that an error on the configuration of product bundles arose so that the emulator could not boot the intended system image. This section documents the error, its resolution, and the implications for Fuchsia's experimental deployment tooling. The bug appeared when trying to launch the Fuchsia emulator by calling this command: `fx emu`. The Fuchsia emulator comes with a product bundle that needs downloading and a correct set-up on the host system, yet on first setup, an error occurred of incomplete or missing bundle configuration.

Remote Access Flow - Browser-Based GUI Access

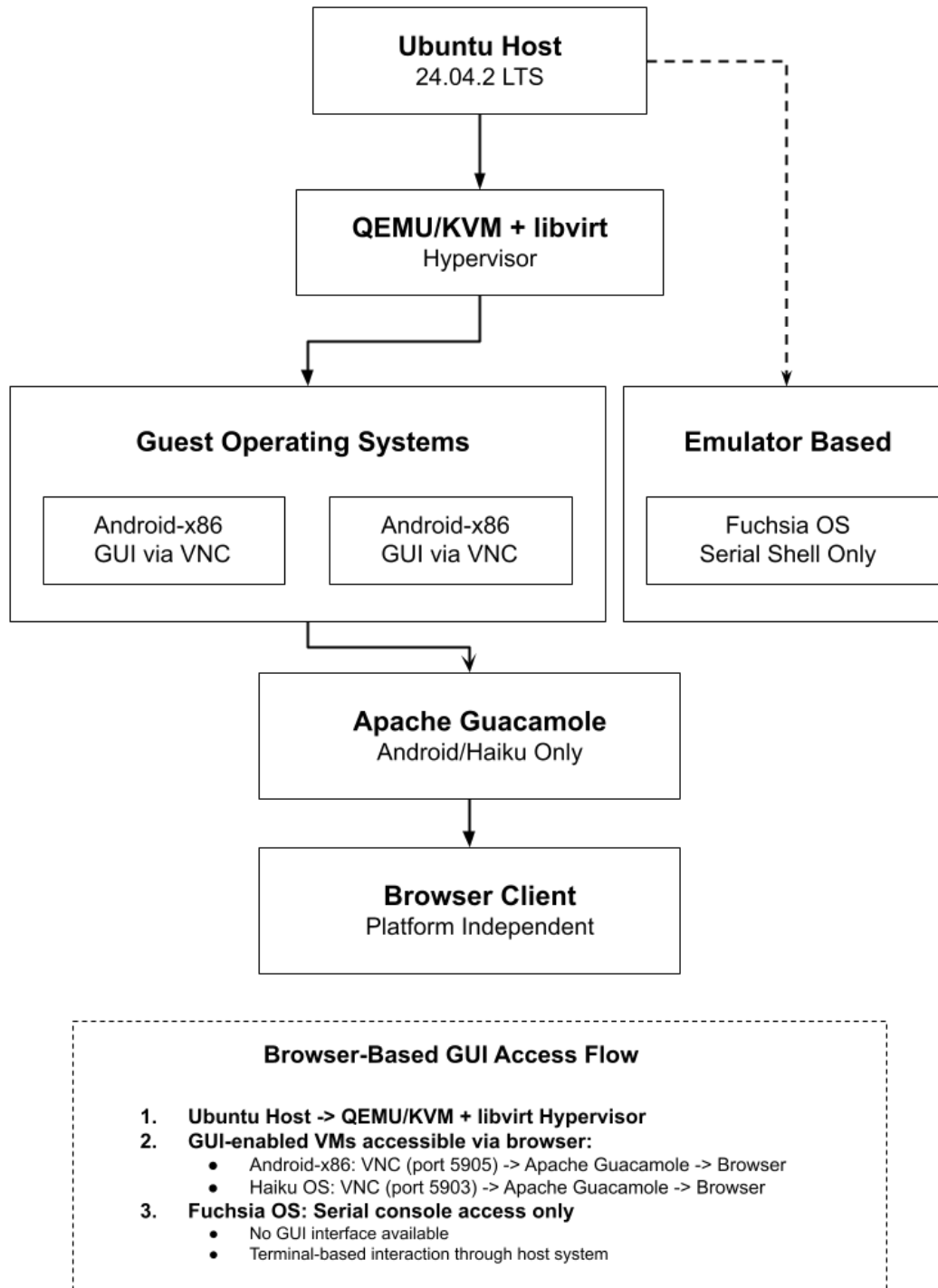


Figure 13: Remote Access Flow- Browser-Based GUI Access

Network and Storage Layer

The network was set up in NAT mode so that virtual machines were able to reach the internet and get updated information as well as packages, but could not be reached by external networks. All VNC traffic, WebSocket streams and shell interfaces were restricted to the localhost interface to have better security. Retaining of VMs was in the QCOW2 format with disk storage, it supported dynamic allocation, compression, and snapshotting. These virtual disks were stored in the default libvirt image directory (`/var/lib/libvirt/images`) and managed through virt-manager's graphical interface. Check Figure 14.

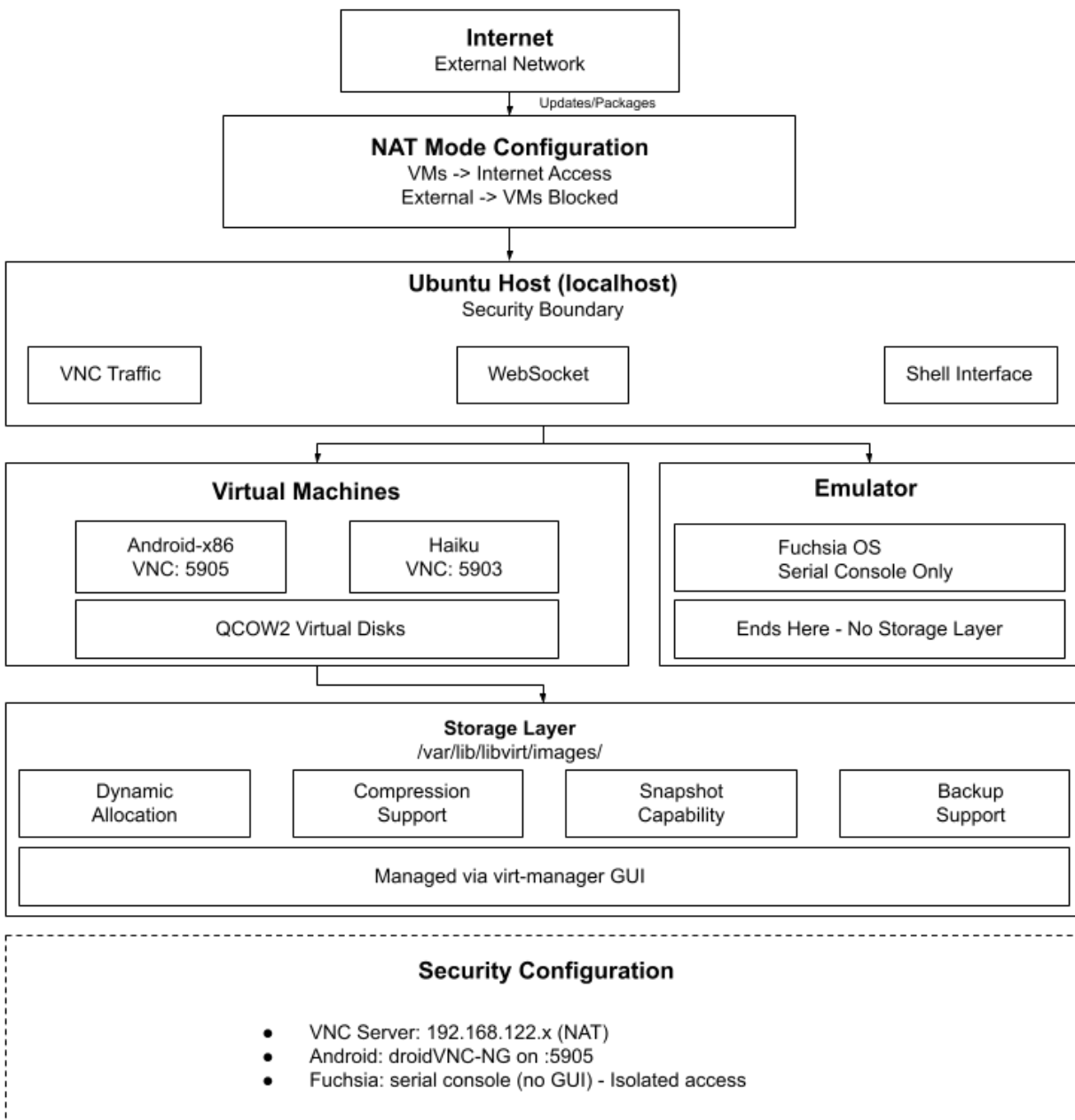
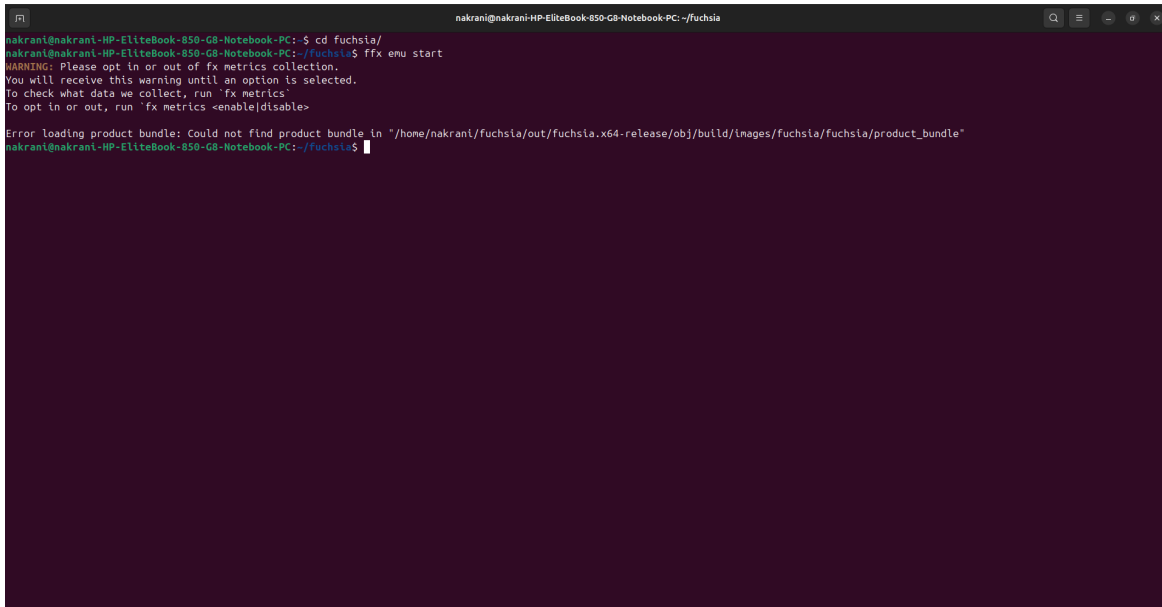


Figure 14: Storage Layout and Network Configuration for Virtual Machines

Actual Screenshots Documenting the Issue:

1. **Product Bundle Error Screenshot:** As seen in Figure 15, when attempting to run the Fuchsia emulator after a successful build, the system displayed a "product bundle" error that prevented proper emulator initialization.

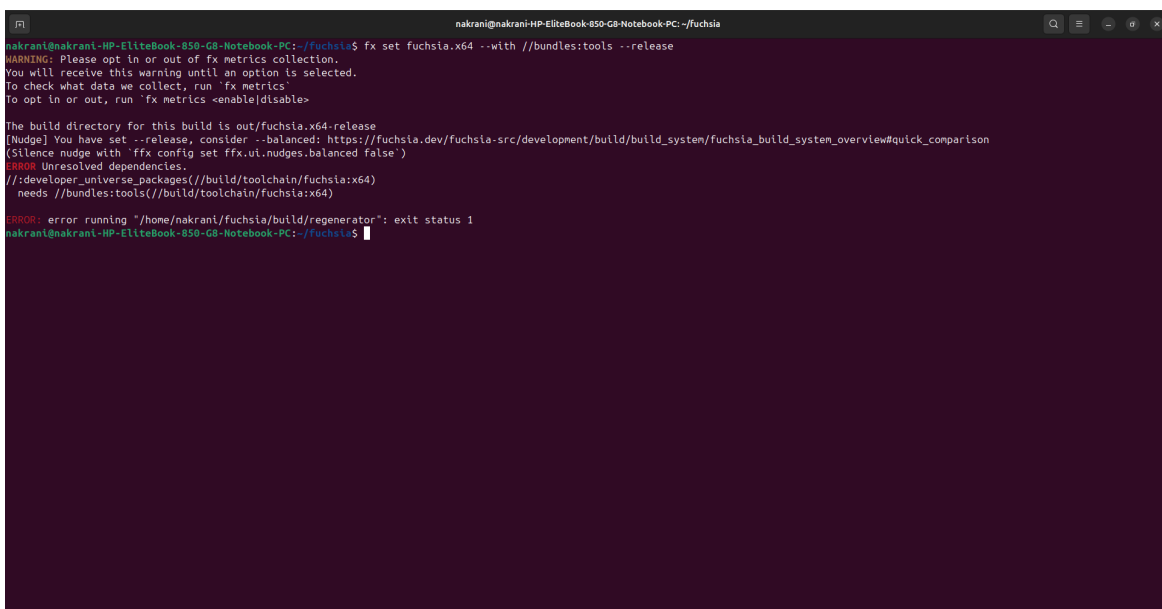


```
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: ~/fuchsia
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~/fuchsia$ ffx emu start
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run 'fx metrics'
To opt in or out, run 'fx metrics <enable|disable>'

Error loading product bundle: Could not find product bundle in "/home/nakrani/fuchsia/out/fuchsia.x64-release/obj/build/images/fuchsia/fuchsia/product_bundle"
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~/fuchsia$
```

Figure 15: Product bundle error

2. **Bundle Setup Failure Screenshot:** The next trial to resolve the bundle problem, Figure 16 also caused a fault, meaning more configuration issues with the product bundle system.



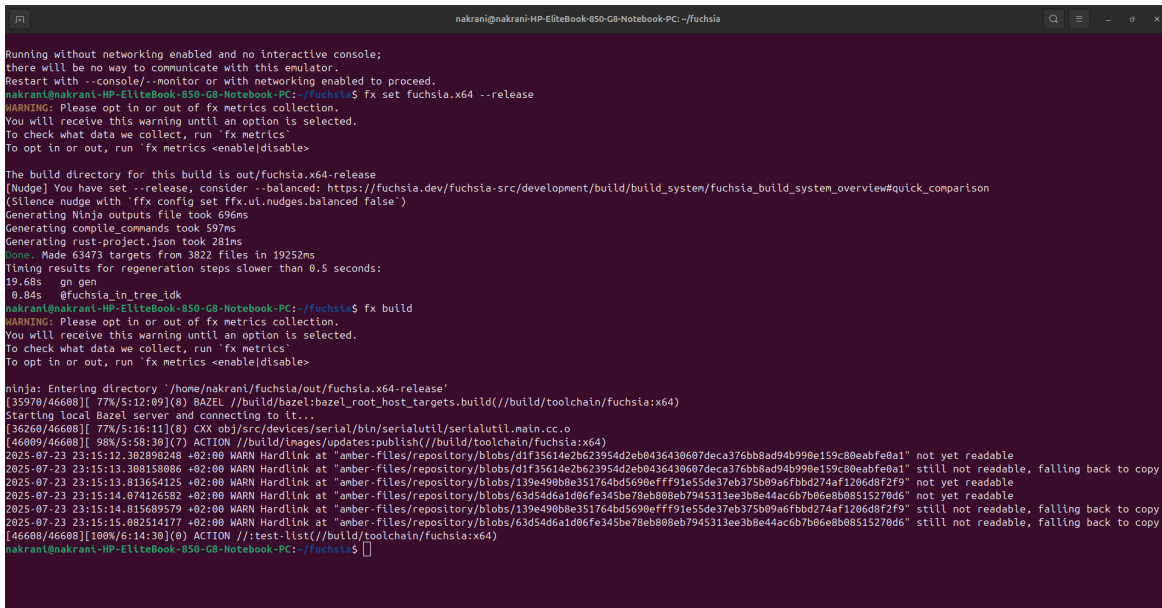
```
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~/fuchsia
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~/fuchsia$ ffx set fuchsia.x64 --with //bundles:tools --release
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run 'fx metrics'
To opt in or out, run 'fx metrics <enable|disable>'

The build directory for this build is out/fuchsia.x64-release
[Nudge] You have set --release, consider --balanced: https://fuchsia.dev/fuchsia-src/development/build/build_system/fuchsia_build_system_overview#quick_comparison
(Silence nudge with 'ffx config set ffx.ui.nudges.balanced false')
ERROR Unresolved dependencies.
//:developer_universe_packages(//build/toolchain/fuchsia:x64)
  needs //bundles:tools(//build/toolchain/fuchsia:x64)

ERROR: error running "/home/nakrani/fuchsia/build/regenerator": exit status 1
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~/fuchsia$
```

Figure 16: Bundle Setup Failure Screenshot

3. **Build Success but Runtime Failure:** Although compilation was successful (Figure 17 showing 100% build completion), the dynamic run-time environment did not initialize the GUI components properly.



```

Running without networking enabled and no interactive console;
there will be no way to communicate with this emulator.
Restart with --console/--monitor or with networking enabled to proceed.
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: ~/fuchsia$ fx set fuchsia.x64 --release
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run 'fx metrics'
To opt in or out, run 'fx metrics <enable|disable>'

The build directory for this build is out/fuchsia.x64-release
[Nudge] You have set --release, consider --balanced: https://fuchsia.dev/fuchsia-src/development/build/build_system/fuchsia_build_system_overview#quick_comparison
(Silence nudge with 'fx config set ffx.ul.nudges.balanced false')
Generating Ninja outputs file took 696ms
Generating compile_commands took 597ms
Generating rust-project.json took 281ms
Done. Made 63473 targets from 3822 files in 19252ms
Timing results: for regeneration steps slower than 0.5 seconds:
19.68s  gn gen
0.84s   @fuchsia_in_tree_idk
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: ~/fuchsia$ fx build
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run 'fx metrics'
To opt in or out, run 'fx metrics <enable|disable>'

ninja: Entering directory '/home/nakrani/fuchsia/out/fuchsia.x64-release'
[35970/46608] 77%/5:12:09(8) BAZEL //build/bazel:bazel_root_host_targets.build(//build/toolchain/fuchsia:x64)
Starting local Bazel server and connecting to it...
[36460/46608] 77%/5:16:11(8) CXX obj/src/devices/serial/bin/serialutil/serialutil_main.cc.o
[46609/46608] 98%/5:50:30(77) ACTION //build/images/updates:publish(//build/toolchain/fuchsia:x64)
2025-07-23 23:15:12.382898248 +02:00 WARN Hardlink at "amber-files/repository/blobs/d1f35614e2b623954d2eb0436438607deca376bb8ad94b998e159c80eabfe0a1" not yet readable
2025-07-23 23:15:12.388158086 +02:00 WARN Hardlink at "amber-files/repository/blobs/d1f35614e2b623954d2eb0436438607deca376bb8ad94b998e159c80eabfe0a1" still not readable, falling back to copy
2025-07-23 23:15:13.813654125 +02:00 WARN Hardlink at "amber-files/repository/blobs/139e498b8e351764bd5690efff91e55de37eb375b09a6fbbd274af1286d8f2f9" not yet readable
2025-07-23 23:15:14.074126582 +02:00 WARN Hardlink at "amber-files/repository/blobs/63d54dea1d06fe345be78eb808eb7945313ee3b8e44ac6b7b06e8b0851527bd6" not yet readable
2025-07-23 23:15:14.815689579 +02:00 WARN Hardlink at "amber-files/repository/blobs/139e498b8e351764bd5690efff91e55de37eb375b09a6fbbd274af1286d8f2f9" still not readable, falling back to copy
2025-07-23 23:15:15.002514177 +02:00 WARN Hardlink at "amber-files/repository/blobs/63d54dea1d06fe345be78eb808eb7945313ee3b8e44ac6b7b06e8b0851527bd6" still not readable, falling back to copy
[46608/46608] 100%/6:14:30(0) ACTION //test-list(//build/toolchain/fuchsia:x64)
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: ~/fuchsia$

```

Figure 17: Build Success but Runtime Failure

Interaction Cycle

The user interaction cycle gets instantiated as the host comes up with VMs using the virt-manager. All guest operating systems are started with the initialisation of their GUI or shell interface and wait to receive input in VNC or serial ports. The Guacamole web interface, which is built on Docker, redirects connections to the VNC socket to the respective VNC. Through the Guacamole web portal, the users are logged in and communicate with the guest OS through the internal browser.

Figure 18 shows (diagram showing: User Browser → Guacamole Web UI → VNC(localhost) → Android/Haiku Display.

Fuchsia access continued to be out of this flow and was executed through fx shell only using the host terminal. This modular flow is a clean break between host and guest environment, in addition to offering minimal attack surfaces (exposure of the network), as well as offering a similar user experience across operating systems with varied GUI capabilities.

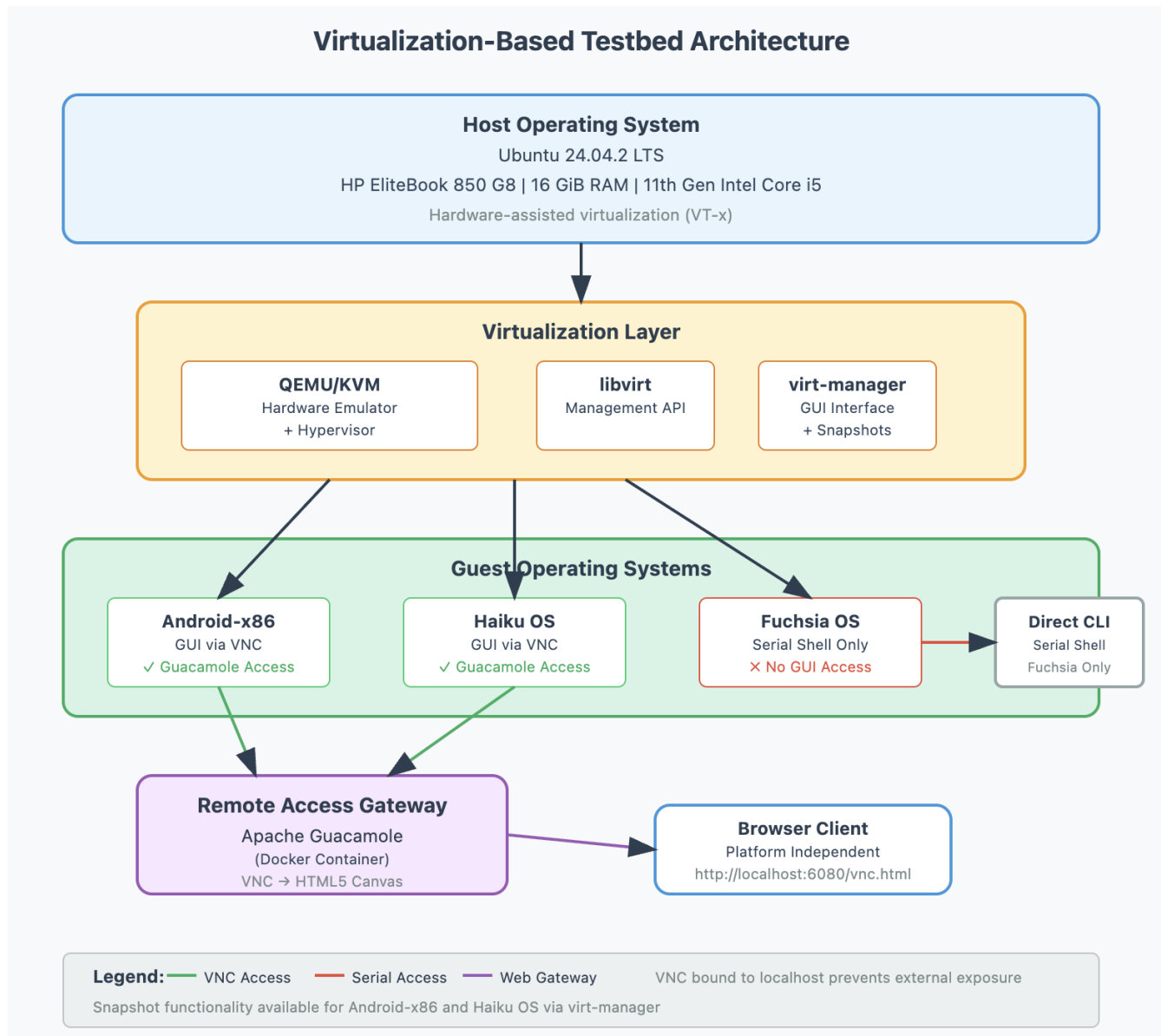


Figure 18: Diagram of Architecture of the virtualisation-based testbed used to deploy and manage three guest operating systems.

4. Implementation

Chapter 4 describes the end-to-end work of virtualization and the access framework. It starts with describing the Android and Haiku VM configuration processes in QEMU/KVM, such as ISO configuration and Creation, resource configuration and installing the filesystem. Then, it explains the Fuchsia build from-source process, which emphasizes the environment bootstrap, product configuration, headless emulator start and shell access even without a GUI (Section 4.3). Then follows the compilation and deployment of the Apache Guacamole manual, including the establishment of guacd, the use of Tomcat-based webapp deployment, JDBC authentication, and connection configuration of Android and Haiku (Section 4.4). Last but not least, NAT networking and port forwarding scheme is provided, which allows binding the VNC port on every VM to localhost, ensuring its easy to access through the browser with the help of Guacamole (Section 4.5). All the commands and configuration files, as well as screenshots, are presented throughout to make reproduction possible.

4.1 Android Setup in QEMU with Web Access

In setting up the Android environment in the QEMU/KVM virtualized system, it started with loading the official Android-x86 9.0 ISO image from the Android-x86 project site [10]. This ISO was chosen because it can run well with the x86 architecture, and it is stable on KVM-based virtual machines. The virtual machine configuration was performed through a graphical user interface of Virtual Machine Manager (virt-manager), which manages QEMU/KVM instances. When the user opened virt-manager, it began the configuration process, and the user clicked the option called Local install media (ISO), then clicked the Browse Local search option and loaded the Android ISO that was downloaded. This step is represented in Figure 19, which shows the launch interface and ISO selection process. Figure 20 explains the main interface of virt-manager at startup.

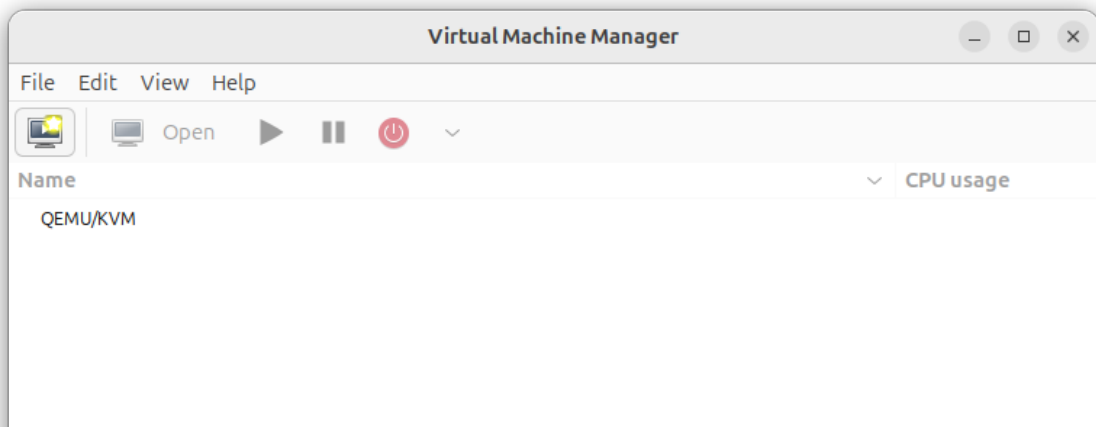


Figure 19: Virtual Machine Manager interface launched to begin Android VM creation.

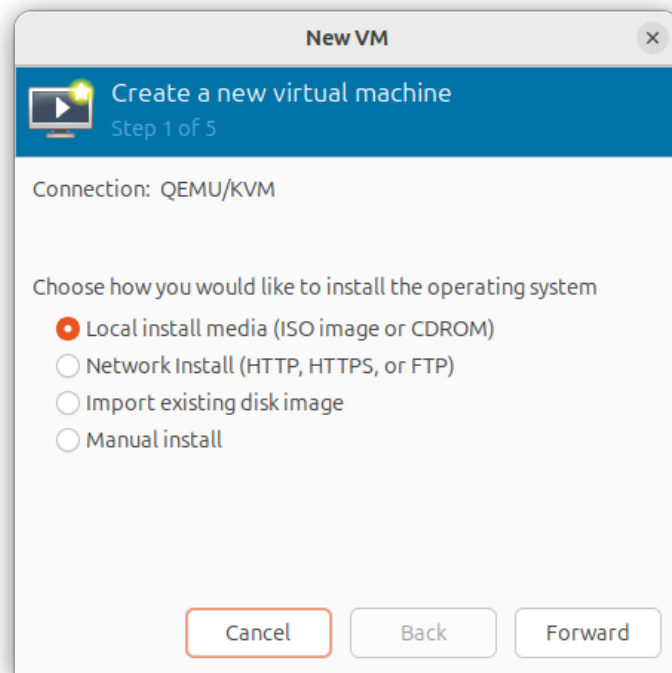


Figure 20: Initial VM creation step with local install media option selected.

Figure 20 shows the first stage of creating a new VM using local installation media.

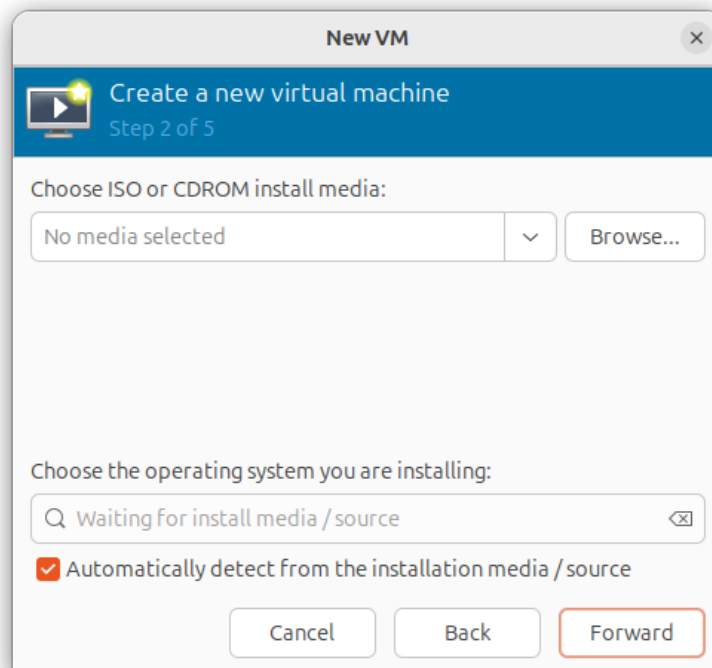


Figure 21: File browser opened to locate the Android ISO file.

Error! Reference source not found. and Figure 22 illustrate the file selection process from the local system. Figure 22 illustrates the file selection process from the local system.

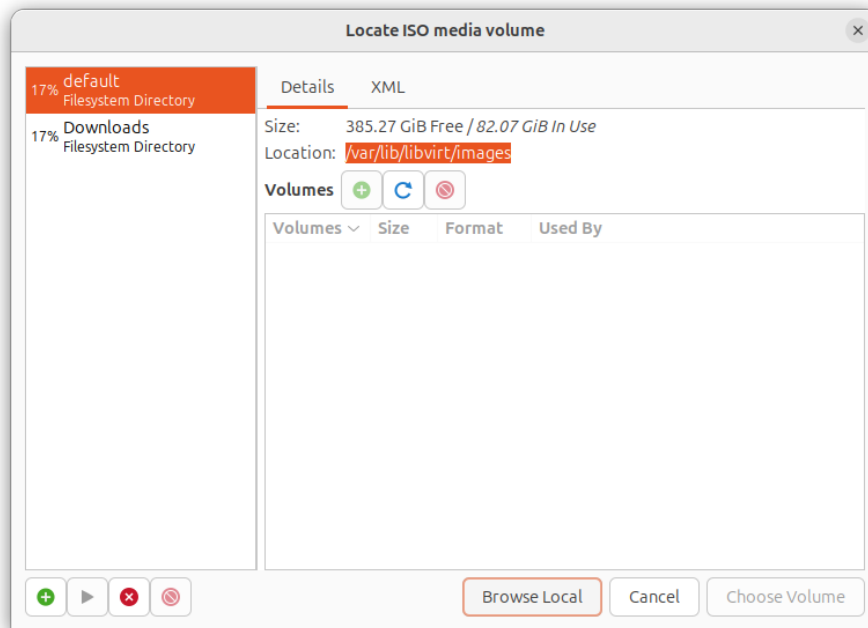


Figure 22: Android-x86 ISO successfully selected for installation.

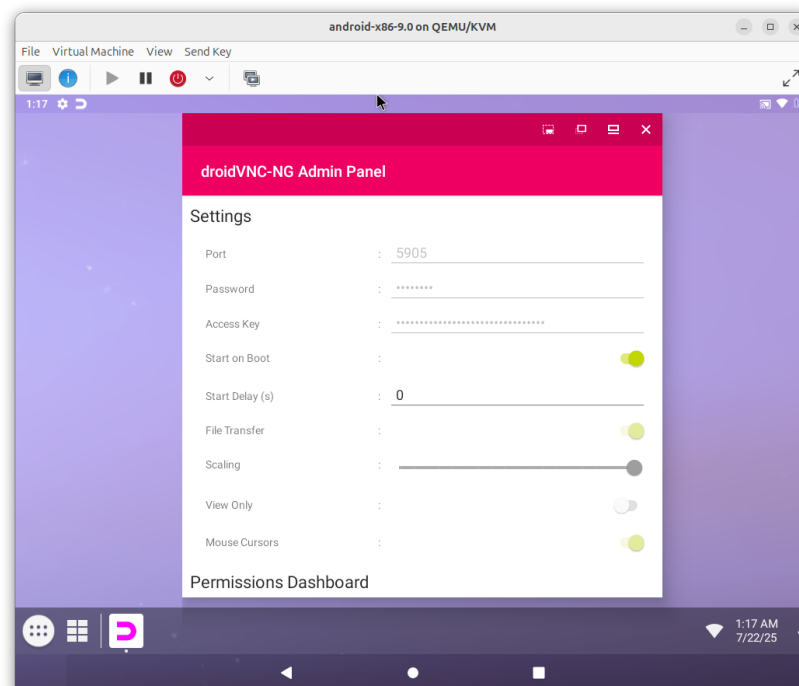


Figure 23: DroidVNC-NG password configuration screen inside Android-x86

The Android remote GUI session exploits Apache Guacamole, which requires authentication of the user before being allowed to connect to the Android virtual machine running the DroidVNC-NG server, hence securing the Android virtual machine. The VNC system was set up in such a way that it always demanded a password to access before any remote connection could be achieved Figure 23.

The automatically executed detection option was automatically turned off to prevent improper configuration, and the Android-x86 9.0 profile was specifically selected among the options on the compatibility list, so that QEMU configured the hardware options appropriate to the needs of the Android kernel and services. This interface is detailed in Figure 24, where the user sets the operating system type manually.

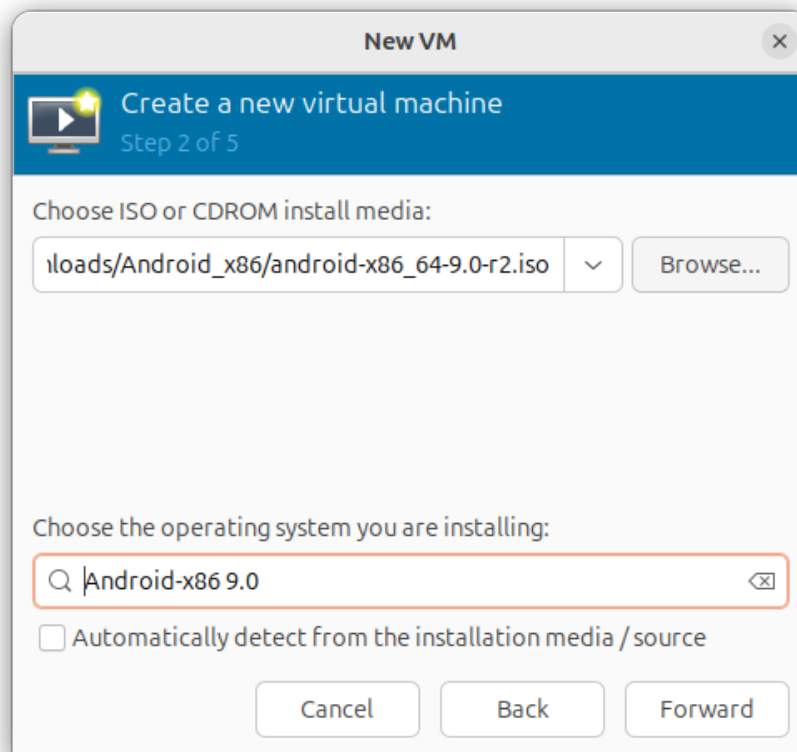


Figure 24: Manual selection of Android-x86 9.0 as the guest operating system.

Subsequent configuration involved allocating hardware resources appropriate for Android's graphical environment. The memory allocation was set at a minimum 3 GB RAM and 2 CPU cores in order to have maximum performance without straining the host system, as shown in Figure 25. A 20GB dynamic QCOW2-format virtual disk was then configured, so it is possible to make a snapshot and use the space optimally.

This disk setup appears in Figure 26. The configuration pre-check box of the Customize pre-install configuration was checked to enable extensive pre-review and alteration of the VM configuration to be made before the setup is completed. The type of name followed was android-x86-9.0-2, and all settings were confirmed prior to commencement of installation. This step of the procedure, with its configuration screen and install-trickner, can be seen in Figure 27 and Figure 28.

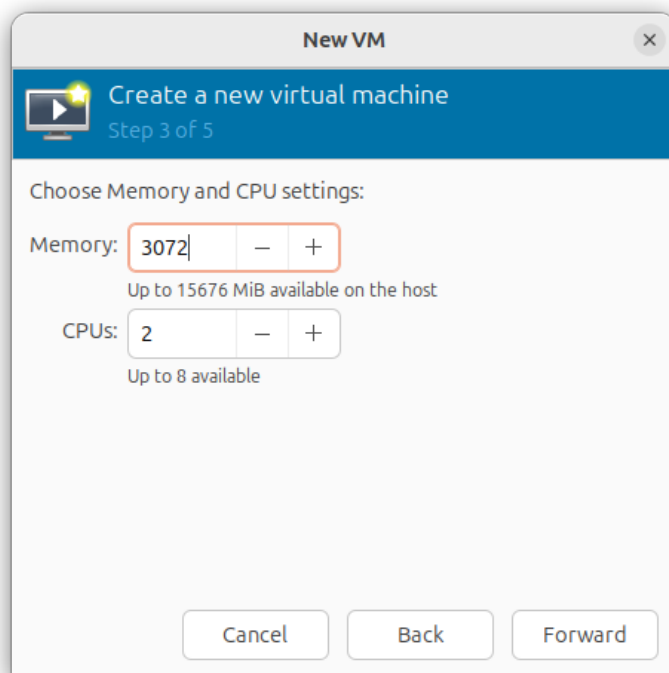


Figure 25: Allocation of 3GB RAM and 2 CPU cores for optimal performance.

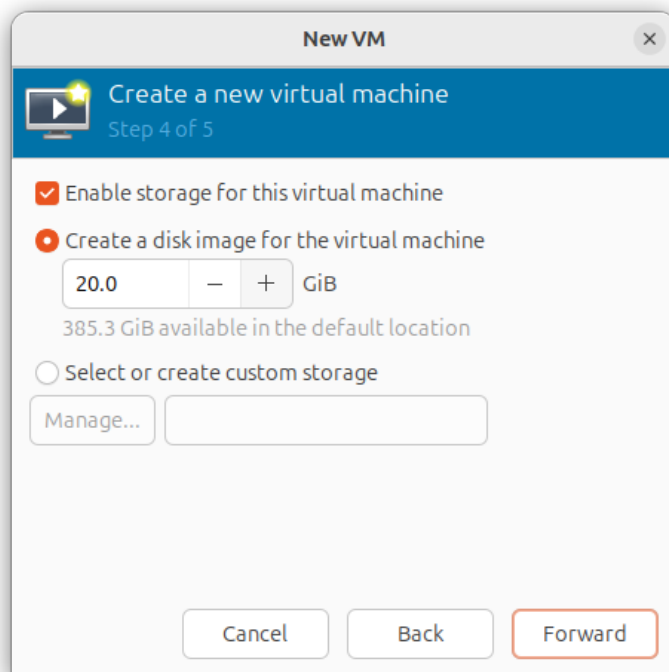


Figure 26: Storage configuration using a 20GB qcow2 virtual disk.

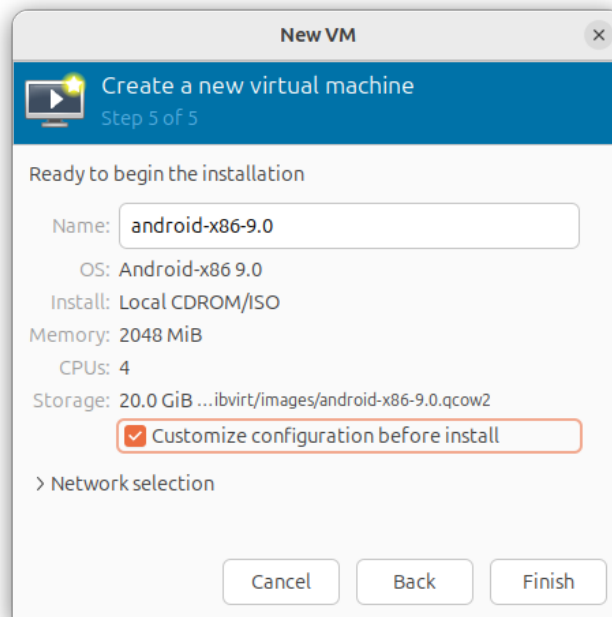


Figure 27: VM named and prepared for advanced configuration.

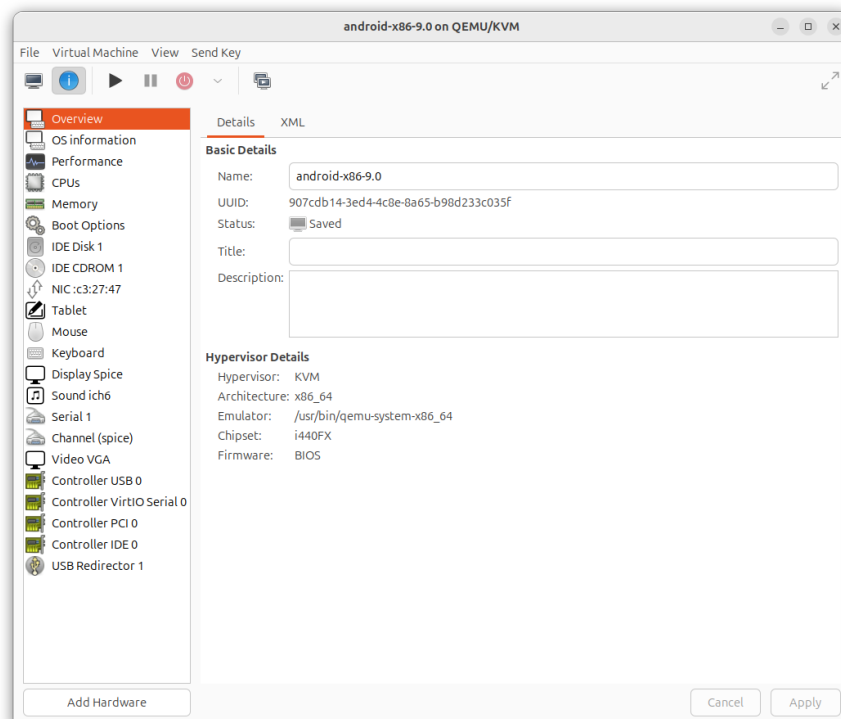


Figure 28: Advanced VM configuration finalised; installation initiated.

After the VM with Android ISO was already launched, the installation of the Android operating system directly to the

virtual disk was performed, whereupon reboot, Android became operational in its default setup environment, where Wi-Fi was configured to proceed with access to the web. But even with a successful deployment, remote access did not work out that well. The first attempt in connecting Android GUI via a VNC server and loading the latter on browser clients presented a blank screen. This was resolved by using a workaround that was found in the community issue tracker, Issue #35 on Github [29], in which the users shared that the framebuffer was inactive when the VNC server was the only server available. To fix this, the application was initially downloaded through F-Droid [33], and then, the implementation of the framebuffer occurred. It was only then that the DroidVNC-NG VNC server was installed, which eventually allowed the GUI to be visualized via VNC.

After this setup, the localhost VNC port of the virtual machine was pointed locally and combined with Apache Guacamole, according to the configuration of the system. Guacamole is also linked with the VNC stream and allows the Android graphical world to be accessed through a browser. shows the layered architecture of the system in a visual way which explains this mechanism. Figure 8 illustrates the hardware behind it, the host Ubuntu OS, the KVM/QEMU virtualization layer, the Android OS, which has a VNC server running on it, and a Guacamole server to allow the Chrome web browser in the host machine to access the remote session. The layered stack emphasises how virtualization, local loopback networking and Guacamole integration can be used to provide a fully functional remote desktop to the Android-x86 guest operating system in a controlled, browser-based environment.

4.2 Haiku Setup in QEMU with Web Access

The QEMU machine or the host system was a Linux machine in which the Haiku operating system was virtualized, where the aim was to remotely access the system using a web browser. It was also accomplished by closely adhering to the documentation presented on the official site of the Haiku project, namely the documents at the links <https://www.haiku-os.org/guides/virtualizing/KVM> [13] and <https://www.haiku-os.org/get-haiku/installation-guide> [21] and assuming compatibility and proper installation in a virtualized infrastructure.

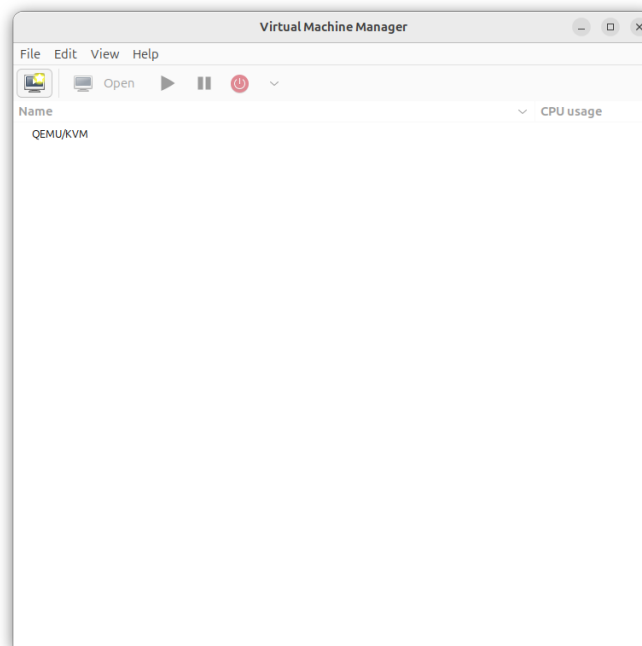


Figure 29: Launch screen of Virtual Machine Manager used to initiate Haiku VM setup

The process started by executing the Virtual Machine Manager (virt-manager) which is a GUI utility to control QEMU/KVM environments. The interface loaded successfully, as depicted in Figure 29.

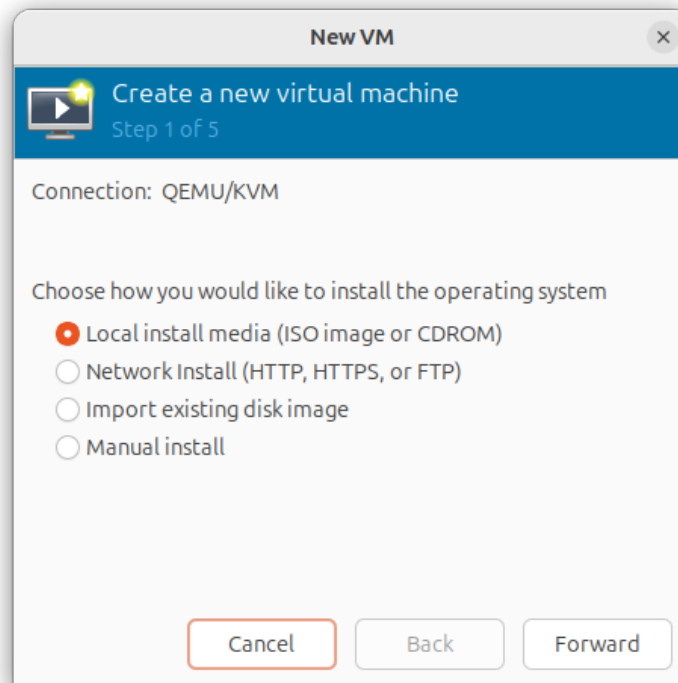


Figure 30: Selection of local installation media for creating a Haiku virtual machine

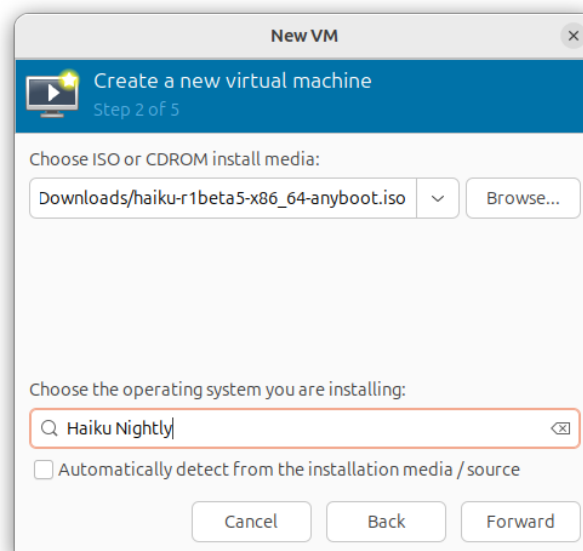


Figure 31: Browsing local storage to locate the Haiku ISO image

To start creating the VM, the installation method selected was “Local install media,” which enabled the use of a pre-

downloaded Haiku ISO file (Figure 30). The ISO file was then located from the local storage (Figure 31) and selected for the VM installation. After that, automatic identification of the OS was deactivated, and system parameters, including architecture and memory, were configured manually as shown in Figure 32.

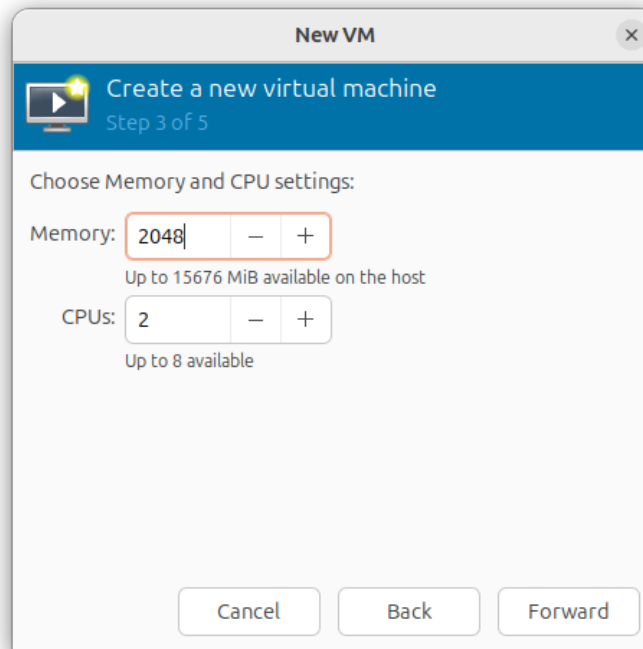


Figure 32: Configuration of guest OS settings for the Haiku system

Once the system configuration was finalized, the installer transitioned to the final pre-installation review screen (Figure 33), before booting into the Haiku ISO environment. The Haiku live desktop then loaded within the virtual machine, confirming successful ISO execution (Figure 34).

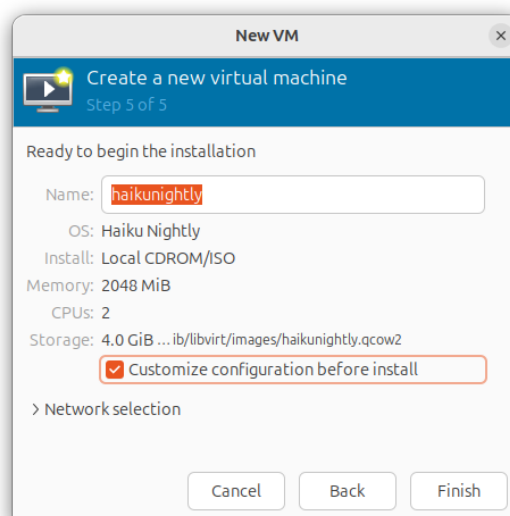


Figure 33: Final pre-installation review screen for the Haiku virtual machine

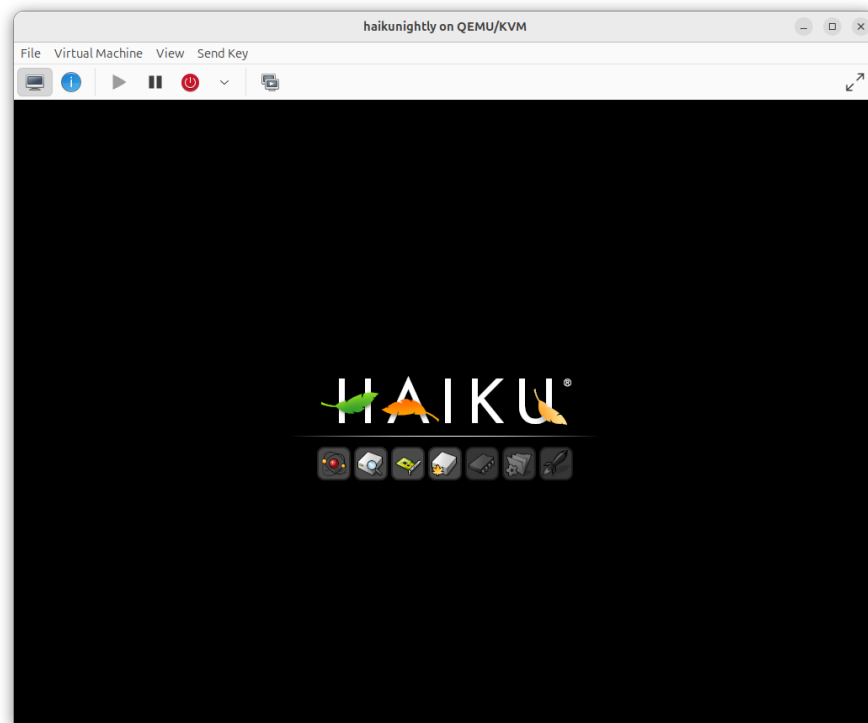


Figure 34: Haiku desktop loaded from ISO in live environment

The physical setup was done with the local GUI setup wizard of Haiku. The first step involved launching the partitioning tool to prepare the virtual disk for installation (Figure 35). The copying of system files to the selected disk then took place, as Figure 36 shows. The wizard confirmed successful installation in Figure 37, followed by bootloader setup (Figure 38) and the final reboot prompt (Figure 39).

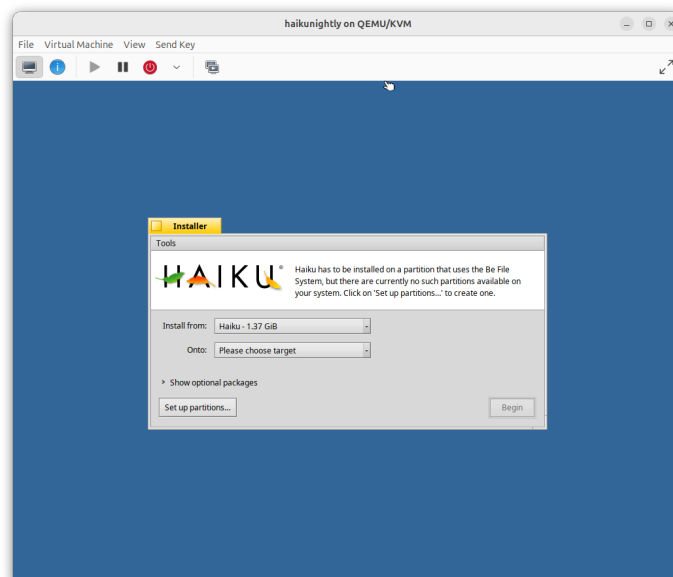


Figure 35: Partitioning tool launched to prepare disk for Haiku installation

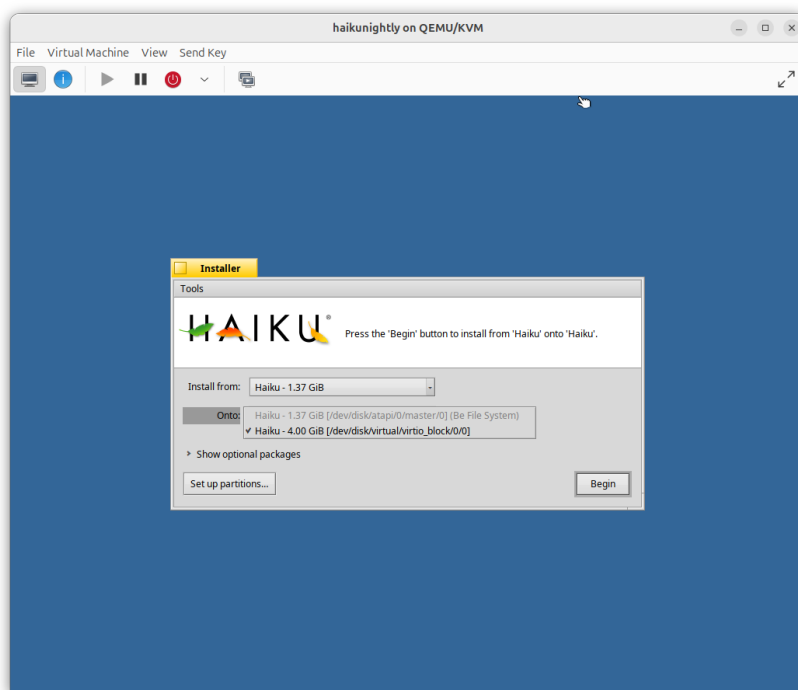


Figure 36: Midway through copying system files to the virtual disk

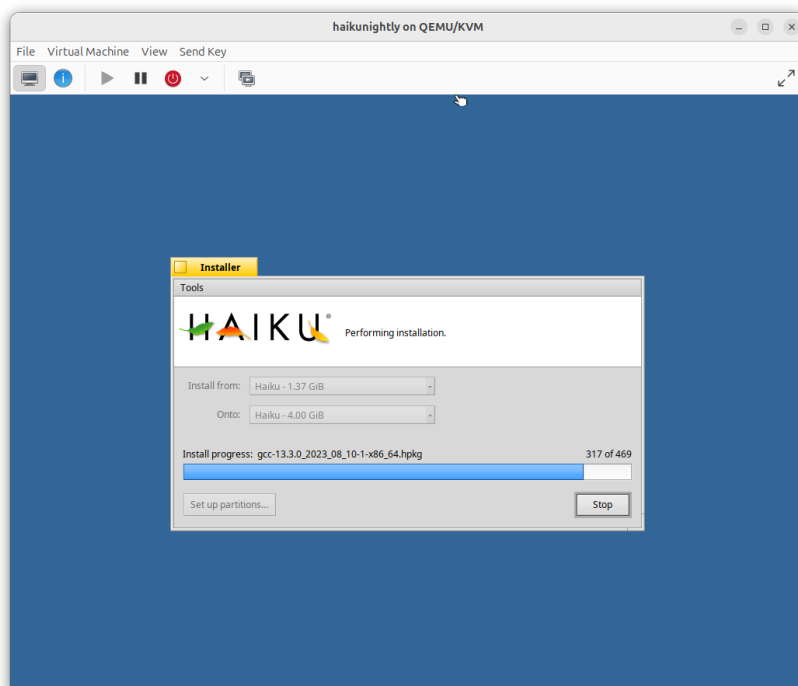


Figure 37: Confirmation of successful Haiku OS installation

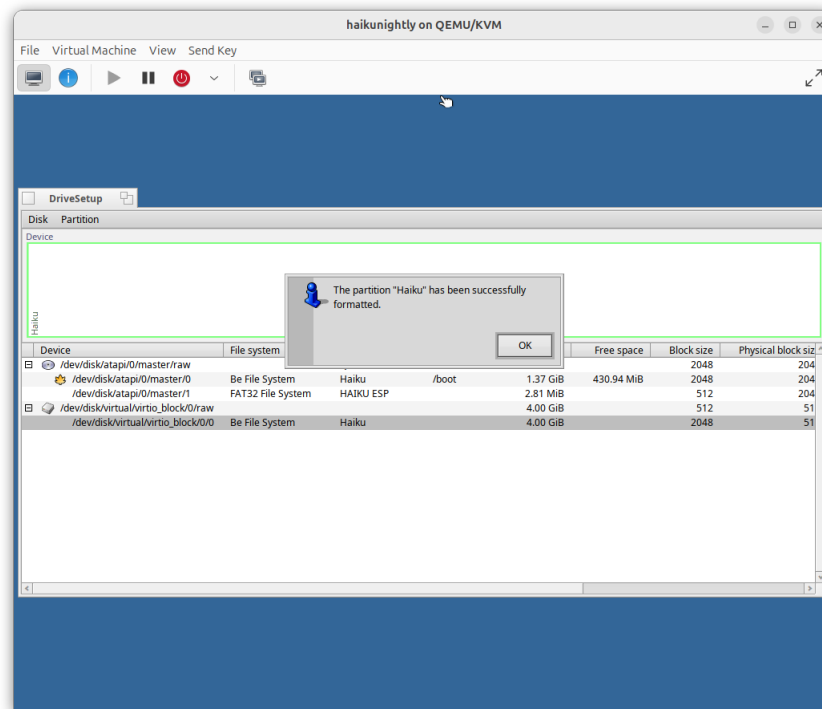


Figure 38: Final prompt before restarting into installed Haiku OS

To prepare Haiku for remote access, the AGMS VNC server was installed. The server binary for Haiku was acquired from <https://github.com/agmsmith/VNC-4.0-BeOS-Server> [2] extracted into the filesystem, as shown in Figure 39. The server was launched from the terminal Figure 40 and started listening for incoming VNC connections Figure 41.

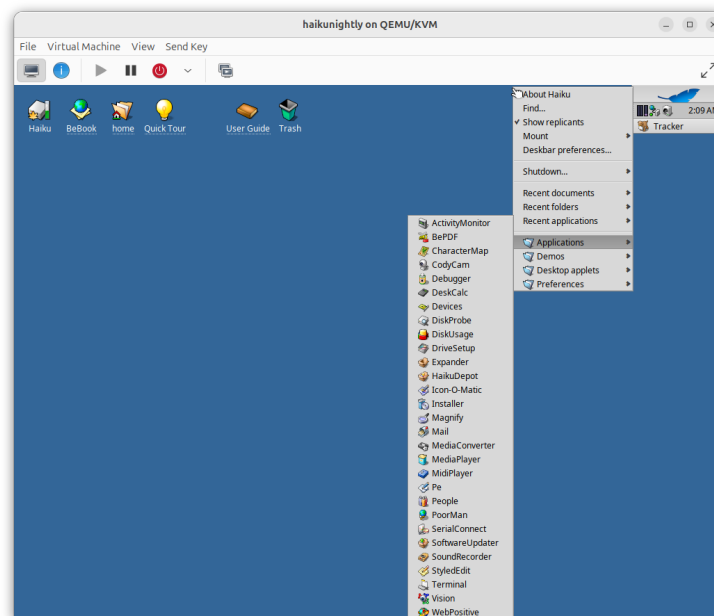


Figure 39: AGMS VNC server package extracted within the Haiku filesystem

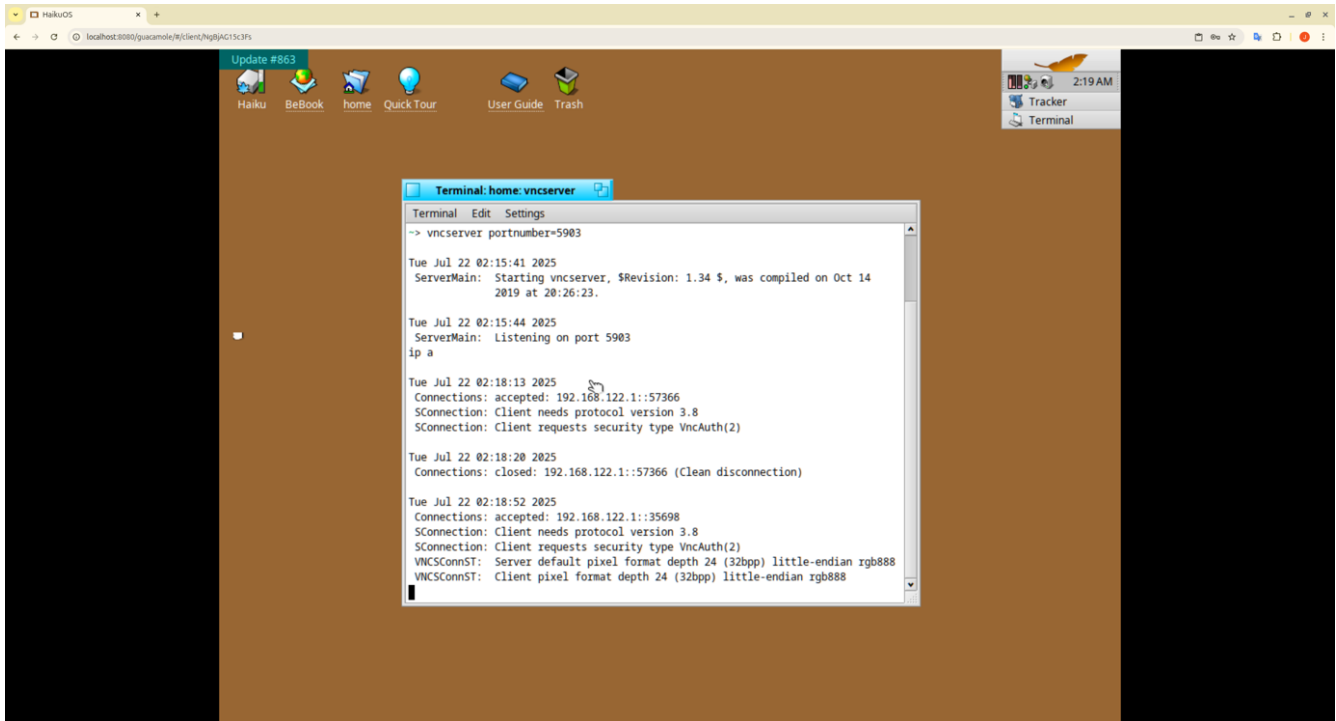


Figure 40: Terminal window displaying the startup process of the VNC server

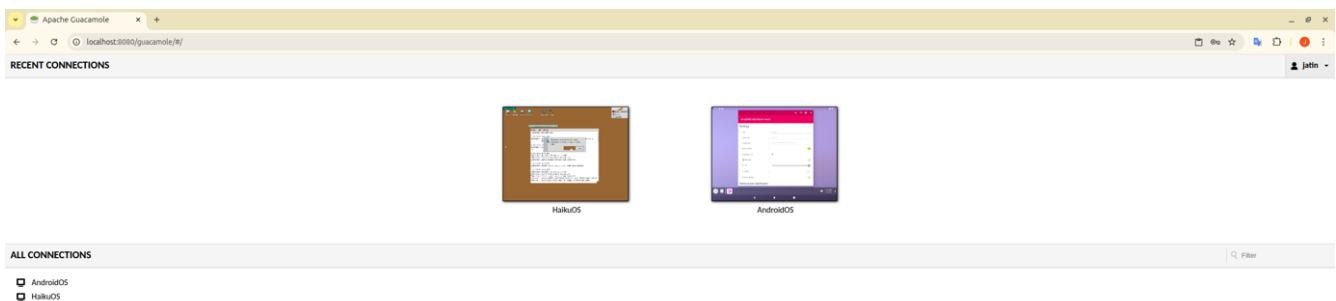


Figure 41: Active VNC session initialized and ready for browser access

To enhance security, the AGMS VNC Server password authentication was set to enable it to provide remote graphical access. Figure 42 illustrates this and proves that there is secure access setup within the Haiku.

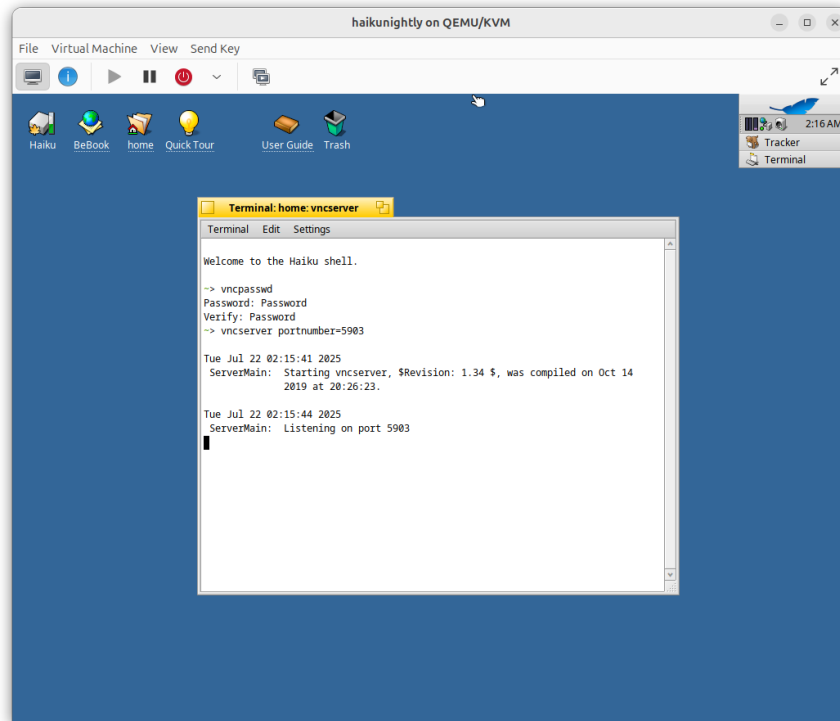


Figure 42: AGMS VNC server showing password protection setup in Haiku

Apache Guacamole was installed in the host system to grant access to the VM through the browser. A Chrome browser launched the Guacamole dashboard interface (Figure 43). When getting access to the interface, they were offered a page where they were to enter authentication credentials.

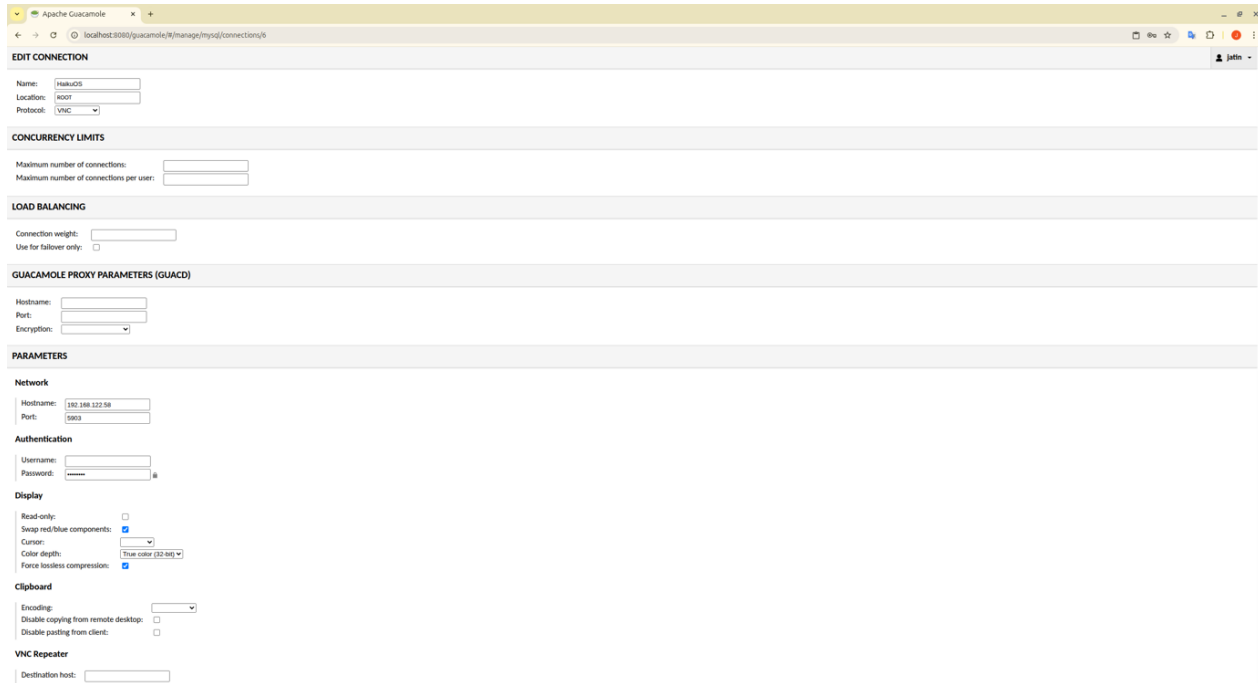


Figure 43: Chrome browser opening Guacamole interface on the host system

4.3 Fuchsia Setup Attempt and Shell Access

Downloading and Preparing the Fuchsia Source

The Fuchsia OS implementation followed Google's official documentation available at [Fuchsia.dev](https://fuchsia.dev) [11]. The setup process began with installing essential development dependencies on the Ubuntu host system using the following command:

```
sudo apt install curl file git unzip
```

The Fuchsia OS implementation followed Google's official documentation available at [Fuchsia.dev](https://fuchsia.dev) [11]. The setup process began with installing essential development dependencies on the Ubuntu host system using the following command: This installation ensured that all required tools for downloading, extracting, and managing the Fuchsia source code were available on the development environment. Once the host system was ready with the required dependencies, the SDK of fuchsia was downloaded and compatibility with the platform was well tested through the ffx tool:

```
curl -s0 https://storage.googleapis.com/fuchsia-ffx/ffx-linux-x64 && chmod +x ffx-linux-x64 &&
./ffx-linux-x64 platform preflight
```

```
hakrani@hakrani-HP-EliteBook-850-G8-Notebook-PC:~$ curl -s0 https://storage.googleapis.com/fuchsia-ffx/ffx-linux-x64 && chmod +x ffx-linux-x64 && ./ffx-linux-x64 platform preflight
Running pre-flight checks...
[✓] Found all needed build dependencies: curl, git, unzip
[✓] Found supported graphics hardware: Intel Corporation TigerLake-LP GT2 [Iris Xe Graphics] (rev 01)
[✓] Found tuntap device named 'tun0' for current user
[✓] KVM is enabled for the current user
[✓] Found ssh binary and $HOME/.ssh directory.
Everything checks out! Continue at https://fuchsia.dev/fuchsia-src/get-started
```

Figure 44: Output of ffx platform preflight check showing successful validation

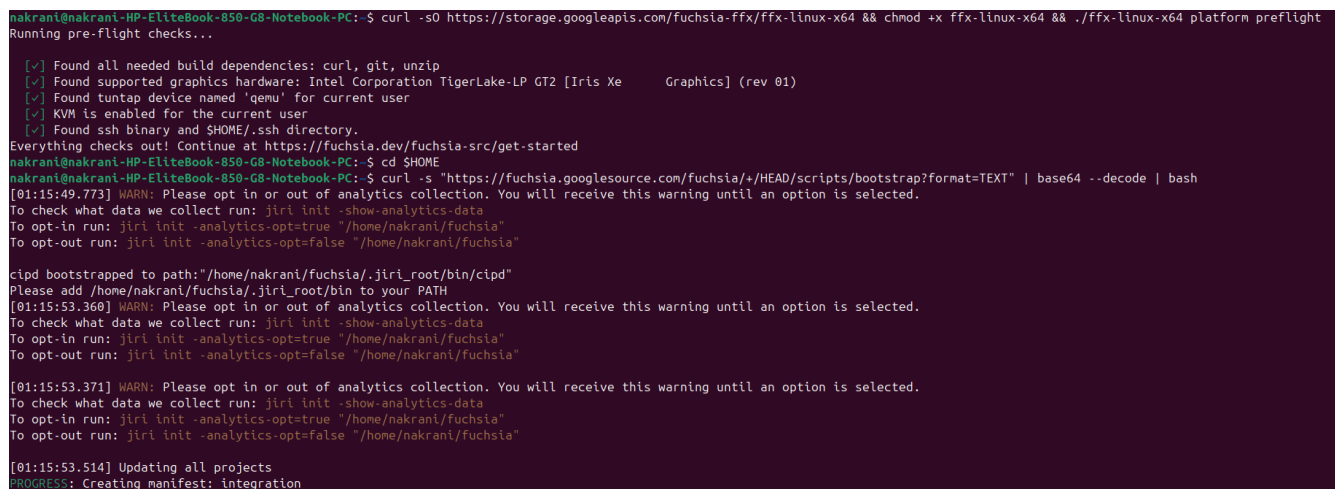
The preflight check is a crucial step that validates the host system's capability to build and run Fuchsia OS. This validation process examines system requirements, available resources, and compatibility factors. The environment

successfully passed all preflight checks, confirming that the host machine was adequately prepared for source download and compilation processes. This successful validation is demonstrated in Figure 44. The host machine passed all prerequisite checks required for building and running Fuchsia OS, including system compatibility and resource availability.

Following the successful preflight validation, the complete Fuchsia source tree was downloaded using the official bootstrap script:

```
curl -s "https://fuchsia.googlesource.com/fuchsia/+/HEAD/scripts/bootstrap?format=TEXT" | base64
--decode | bash
```

The preflight check is a crucial step that validates the host system's capability to build and run Fuchsia OS. The involved parties consider system requirements, the available resources and compatibility issues in this validation process. The environment also passed all of the preflight tests meant to ensure that the host machine was properly set up to prepare the source download and compile accomplishments. This is confirmed by reasonable validation shown in Figure 45. The host machine had cleared all the pre-requisite conditions necessary in building and running Fuchsia OS, such as system compatibility and system resource availability.



```

nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: $ curl -sO https://storage.googleapis.com/fuchsia-ffx/ffx-linux-x64 && chmod +x ffx-linux-x64 && ./ffx-linux-x64 platform preflight
Running pre-flight checks...

[✓] Found all needed build dependencies: curl, git, unzip
[✓] Found supported graphics hardware: Intel Corporation TigerLake-LP GT2 [Iris Xe Graphics] (rev 01)
[✓] Found tuntap device named 'qemu' for current user
[✓] KVM is enabled for the current user
[✓] Found ssh binary and $HOME/.ssh directory.
Everything checks out! Continue at https://fuchsia.dev/fuchsia-src/get-started
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: $ cd $HOME
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: $ curl -s "https://fuchsia.googlesource.com/fuchsia/+/HEAD/scripts/bootstrap?format=TEXT" | base64 --decode | bash
[01:15:49.773] WARN: Please opt in or out of analytics collection. You will receive this warning until an option is selected.
To check what data we collect run: jiri init -show-analytics-data
To opt-in run: jiri init -analytics-opt=true "/home/nakrani/fuchsia"
To opt-out run: jiri init -analytics-opt=false "/home/nakrani/fuchsia"

cipd bootstrapped to path: "/home/nakrani/fuchsia/.jiri_root/bin/cipd"
Please add /home/nakrani/fuchsia/.jiri_root/bin to your PATH
[01:15:53.360] WARN: Please opt in or out of analytics collection. You will receive this warning until an option is selected.
To check what data we collect run: jiri init -show-analytics-data
To opt-in run: jiri init -analytics-opt=true "/home/nakrani/fuchsia"
To opt-out run: jiri init -analytics-opt=false "/home/nakrani/fuchsia"

[01:15:53.371] WARN: Please opt in or out of analytics collection. You will receive this warning until an option is selected.
To check what data we collect run: jiri init -show-analytics-data
To opt-in run: jiri init -analytics-opt=true "/home/nakrani/fuchsia"
To opt-out run: jiri init -analytics-opt=false "/home/nakrani/fuchsia"

[01:15:53.514] Updating all projects
PROGRESS: Creating manifest: integration

```

Figure 45: Bootstrap script execution retrieving source tree and manifest files

Environment Configuration and Tool Setup

Once the source download was done, this was followed by proper configuration of the Fuchsia development toolchain. This has been done by editing the shell profile to contain the required path variables and source scripts:

```
nano ~/.bash_profile
```

The following critical environment variables were added to ensure proper tool accessibility:

```
export PATH=~/.fuchsia/.jiri_root/bin:$PATH
source ~/.fuchsia/scripts/fx-env.sh
```

These environment modifications were then applied to the current session using:

```
source ~/.bash_profile
```

With the environment set up right, some key Fuchsia development tools were brought online, such as the fx CLI and jiri build system. The firewall was then set up done to provide appropriate connectivity of the network to the development environment:

```
fx setup-ufw
```

Figure 46 shows the bash profile configuration process, where environment variables were properly set for Fuchsia development tools.



Figure 46: Bash profile configuration with Fuchsia environment variables

Product Configuration and Build Process

The development environment was configured to build the workbench_eng.x64 product target using the fx configuration system:

```
fx set workbench_eng.x64 --release
```

This step of configuration is essential since it adjusts the specified board target (x64), kernel configuration, choices of drivers, and product-specific elements. The fx set command determines the compilation parameters of all builds. The two-phase configuration process is shown in Figure 47 and Figure 48 where there is the setting of a product target, which is then confirmed.

```

nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~/fuchsia$ fx list-products
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run `fx metrics`
To opt in or out, run `fx metrics <enable|disable>`

bringup
bringup_with_tests
core
core_size_limits
core_with_dfv2_fuzzing
core_with_f2fs
core_with_minfs
fuchsia
microfuchsia_eng
minimal
terminal
terminal_with_netstack2
workbench_eng

```

Figure 47: Product target configuration using fx set command - Phase 1

```

nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~/fuchsia$ fx list-boards
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run `fx metrics`
To opt in or out, run `fx metrics <enable|disable>`

arm64
emac
*
qemu-arm64
riscv64
vim3
vim3-reduced-perf-variation
x64
x64-reduced-perf-variation

```

Figure 48: Product configuration confirmation - Phase 2

With the intent of gauging the flexibility of the build and components structure of Fuchsia, several system products were built with fx set and built through fx build. User used various products such as core, terminal, Fuchsia and workbench_eng. In the case of the Fuchsia product, it just generated an error and would not build, whereas on the rest of the products, the same was happening; the Fuchsia emulator runs but just posts the Fuchsia logo. The Fuchsia builds require an average of 8 to 10 hours, depending on the number of users and 100 per cent CPU throughput being used by all four cores, as revealed in Figure 49.

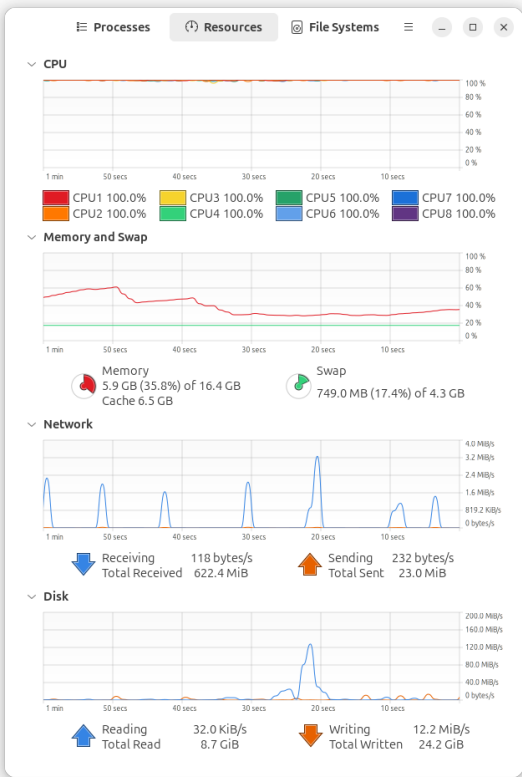


Figure 49: System Resource Monitor Dashboard

Each product defines a different layer of functionality, ranging from minimal CLI environments to full graphical user interfaces as shown in Figure 50, Figure 51 and Figure 52.

Products Tested in Fuchsia Build Environment:

1. **core**

A barebones configuration of the Fuchsia system, which can be used to test the kernel and drivers, is its core product. It contains basic kernel such as Zircon kernel and basic services of the shell, but does not have a user interface. This product was able to compile and execute to CLI, so it can be used in headless activities, as well as in diagnostic testing.

```
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~/fuchsia$ ffx target list

WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run `fx metrics`
To opt in or out, run `fx metrics <enable|disable>`

NAME          SERIAL      TYPE      STATE      ADDRS/IP      MANUAL  RCS
fuchsia-emulator <unknown> core.x64  Product  [127.0.0.1:42843] N      Y
```

Figure 50: Fuchsia emulator target with type "core.x64" in "Product" state.

2. terminal

The add-on builds on the core build with features that are terminal-oriented, allowing a more interactive command-line experience. It does have necessary drivers and shell applications, but no graphical shell. The constructed terminal product has been well constructed and could be accessed through CLI(fx Shell), and it was useful in command-line diagnostics.

```

nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~/fuchsia$ ffx target list
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run `fx metrics`
To opt in or out, run `fx metrics <enable|disable>`

NAME                SERIAL      TYPE          STATE    ADDRS/IP          MANUAL  RCS
fuchsia-emulator    <unknown>  terminal.x64  Product  [127.0.0.1:41277]  N       Y

```

Figure 51: Fuchsia emulator target with type "terminal.x64" in "Product" state.

3. workbench_eng

This is a development and testing version of the Workbench product called workbench_eng that is intended to support engineering needs. It has services at the userland that are designed to be used on GUI display as well as with CLI capabilities. It was built and launched error-free, but only the static logo of the emulator (with an F in it) showed, signalling a problem with the GUI compositors. Nonetheless, serial shell access worked and this product was used in all follow-up tests of CLI.

```

nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~/fuchsia$ ffx target list
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run `fx metrics`
To opt in or out, run `fx metrics <enable|disable>`

NAME                SERIAL      TYPE          STATE    ADDRS/IP          MANUAL  RCS
fuchsia-emulator    <unknown>  workbench_eng.x64  Product  [127.0.0.1:41961]  N       Y

```

Figure 52: Fuchsia emulator target with type "workbench_eng.x64" in "Product" state.

4. fuchsia

The complete product of the fuchsia product will have the complete user interface components and the Scenic compositor. But in this configuration product was not created because of a bundle conflict mistake. This meant that it was not put through to additional testing. Its defeat also demonstrated the complexity of the modular approach Fuchsia followed, and the relevance of product customization to meet individual hardware conditions.

Note on GUI Implementation in Workstation Product

At preproduction testing, the workstation_eng.x64 product configuration was successfully started with graphical user interface features in the emulator platform. But because of ongoing compatibility issues and GPU passthrough restrictions observed in the virtualised environment, GUI functionality was later omitted in subsequent test activities. This change is indicative of the recursive development process, and the necessity to adjust to the technical limitations, and keep the main functionality of the system in focus.

The comprehensive build process was initiated using:

```
fx build
```

Fuchsia's build system is highly resource-consuming, with a single product configuration taking about 8-10 hours in order to complete the compilation. With this implementation, various versions of the product were tried, such as `core`, `terminal`, `workbench_eng`, and `workstation_eng`. All the products in the compilations were able to generate without any compilation error, although a steady problem arose wherein no produced products were able to launch effectively into a working GUI environment. Instead, all configurations displayed only the static ASCII "F" logo screen, as illustrated in Figure 53.

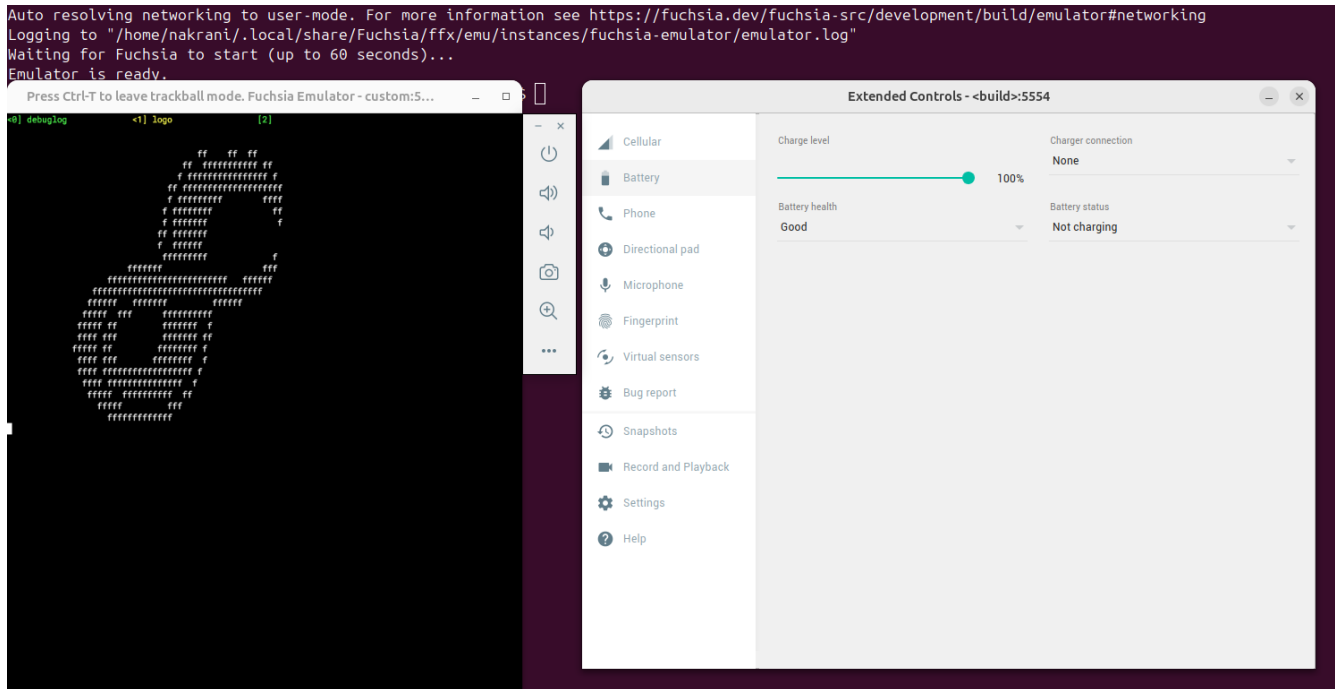


Figure 53: Fuchsia emulator boot process stalled at static logo

Emulator Launch and Target Verification

The emulator launch process was initiated using the `ffx` emulator management system:

```
ffx emu start
```

Target device verification was performed to ensure proper emulator instantiation and connectivity:

```
ffx target list
```

Figure 54 shows the successful target device listing, confirming that the emulator instance was properly recognized by the development tools.

```

nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: ~$ sudo ip link add name brqemu type bridge
+ sudo ip addr add 172.16.243.1/24 dev brqemu
+ ps -auxw
+ grep dnsmasq
+ grep '\.dhcp-range=172\.\16\.\243\.\2,'
+ awk '{ print $1 }'
+ DNSMASQ_PID=
+ [[ '' != '' ]]
+ sudo dnsmasq --interface=brqemu --bind-interfaces --dhcp-range=172.16.243.2,172.16.243.254 --except-interface=lo
+ sudo ip link set brqemu up
+ sudo ip addr flush qemu
+ sudo ip link set qemu master brqemu
+ sudo iptables -A POSTROUTING -t nat -s 172.16.243.0/24 -j MASQUERADE
Waiting for Fuchsia to start (up to 60 seconds)...
Emulator is ready.
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: ~$ fuchsia$ ffx emu start --gpu swftshader_indirect
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run 'fx metrics'
To opt in or out, run 'fx metrics <enable|disable>'

Auto resolving networking to tap-mode. For more information see https://fuchsia.dev/fuchsia-src/development/build/emulator#networking
Logging to "/home/nakrani/.local/share/Fuchsia/ffx/emu/instances/fuchsia-emulator/emulator.log"
+ sudo ip link delete brqemu
[sudo] password for nakrani:
+ sudo iptables -D POSTROUTING -t nat -s 172.16.243.0/24 -j MASQUERADE
+ ip --one-line address show to 172.16.243.1
+ awk '{ print $2 }'
+ (( CLEANUP ))
+ sudo systemctl -w net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
+ cat /proc/sys/net/ipv4/ip_forward
+ CHECK=1
+ [[ 1 != 1 ]]
+ sudo ip link add name brqemu type bridge
+ sudo ip addr add 172.16.243.1/24 dev brqemu
+ ps -auxw
+ grep dnsmasq
+ grep '\.dhcp-range=172\.\16\.\243\.\2,'
+ awk '{ print $1 }'
+ DNSMASQ_PID=398619
+ [[ 398619 != '' ]]
+ sudo kill 398619
+ sudo dnsmasq --interface=brqemu --bind-interfaces --dhcp-range=172.16.243.2,172.16.243.254 --except-interface=lo
+ sudo ip link set brqemu up
+ sudo ip addr flush qemu
+ sudo ip link set qemu master brqemu
+ sudo iptables -A POSTROUTING -t nat -s 172.16.243.0/24 -j MASQUERADE
Waiting for Fuchsia to start (up to 60 seconds)...
Emulator is ready.
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: ~$ fuchsia$ ffx target list
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run 'fx metrics'
To opt in or out, run 'fx metrics <enable|disable>'

NAME          SERIAL      TYPE          STATE  ADDRESS/IP          MANUAL  RCS
Fuchsia-emulator  <unknown>  workbench_eng.x64  Product  [fe8b::c4a9:d68f:d126:16e0%brqemu, 172.16.243.142]

```

Figure 54: FFX target list showing active emulator instance

Shell Access and System Diagnostics

Despite the GUI rendering limitations, kernel-level functionality and shell access were successfully achieved using the `fx` shell command:

```
fx shell
```

With this shell access, the Fuchsia kernel and system services were accessed directly and allowed full system diagnostics and correction of core functionality. Shell-based directory listings of the successful completion of kernel-level boot processes and a functional command line interface were also present. Figure 55 presents a successful shell session that shows root directory structures on the Fuchsia emulator environment.

```

Running without networking enabled and no interactive console;
there will be no way to communicate with this emulator.
Restart with --console/-monitor or with networking enabled to proceed.
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~/fuchsia$ fx set fuchsia.x64 --release
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run 'fx metrics'
To opt in or out, run 'fx metrics <enable|disable>'

The build directory for this build is out/fuchsia.x64-release
[Nudge] You have set --release, consider --balanced: https://fuchsia.dev/fuchsia-src/development/build/build_system/fuchsia_build_system_overview#quick_comparison
(Silence nudge with 'fx config set ffx.ui.nudges.balanced false')
Generating Ninja outputs file took 696ms
Generating compile_commands took 597ms
Generating rust-project.json took 281ms
Done: Made 63473 targets from 3822 files in 19252ms
Timing results for regeneration steps slower than 0.5 seconds:
19.68s  gn gen
      0.84s  @fuchsia_in_tree_idk
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~/fuchsia$ fx build
WARNING: Please opt in or out of fx metrics collection.
You will receive this warning until an option is selected.
To check what data we collect, run 'fx metrics'
To opt in or out, run 'fx metrics <enable|disable>'

ninja: Entering directory '/home/nakrani/fuchsia/out/fuchsia.x64-release'
[35970/46608] [ 77%/5:12:09] (8) BAZEL //build/bazel:bazel_root_host_targets.build(/build/toolchain/fuchsia:x64)
Starting local Bazel server and connecting to it...
[36260/46608] [ 77%/5:16:11] (8) CXX obj/src/devices/serial/bin/serialutil/serialutil.main.cc.o
[46009/46608] [ 98%/5:30:30] (7) ACTION //build/images/updates:publish(/build/toolchain/fuchsia:x64)
2025-07-23 23:15:12.302898248 +02:00 WARN Hardlink at "amber-files/repository/blobs/d1f35614e2b623954d2eb0436438607deca376bb8ad94b990e159c80eabfe0a1" not yet readable
2025-07-23 23:15:13.308158886 +02:00 WARN Hardlink at "amber-files/repository/blobs/d1f35614e2b623954d2eb0436438607deca376bb8ad94b990e159c80eabfe0a1" still not readable, falling back to copy
2025-07-23 23:15:13.813654125 +02:00 WARN Hardlink at "amber-files/repository/blobs/139e490b8e351764bd5690efff91e55de37eb375b09a6fbbd274af1206d8f2f9" not yet readable
2025-07-23 23:15:14.074126582 +02:00 WARN Hardlink at "amber-files/repository/blobs/139e490b8e351764bd5690efff91e55de37eb375b09a6fbbd274af1206d8f2f9" not yet readable
2025-07-23 23:15:14.815689579 +02:00 WARN Hardlink at "amber-files/repository/blobs/139e490b8e351764bd5690efff91e55de37eb375b09a6fbbd274af1206d8f2f9" still not readable, falling back to copy
2025-07-23 23:15:15.082514177 +02:00 WARN Hardlink at "amber-files/repository/blobs/139e490b8e351764bd5690efff91e55de37eb375b09a6fbbd274af1206d8f2f9" still not readable, falling back to copy
[46608/46608] [100%/6:14:30] (0) ACTION //test-list(/build/toolchain/fuchsia:x64)

```

Figure 55: Serial shell access displaying root directories in Fuchsia

The entire diagnostics of the system was carried out with the use of standard Unix-style commands, such as `uname`, `ps` and `top` to ensure the functioning of the kernel and the activity of the services working in it. These testing products proved that the Fuchsia kernel and key operating system functions were working normally.

Analysis of GUI Limitations and Technical Constraints

The persistent GUI rendering limitation was attributed to several technical factors, primarily the absence of proper GPU passthrough support within the virtualised environment and the emulator's inability to successfully invoke Scenic, which serves as the compositor responsible for graphical user interface rendering in Fuchsia OS. This is true in all the product setups that I tried to use, which will be core, terminal, `workbench_eng`, and `workstation_eng`, where the shell access was fine, as I were stuck at the primary logo container on the graphical interface. According to the technical analysis, the Scenic compositor needs portability on particular GPU driver support and hardware acceleration capabilities that failed to be satisfactory in the emulated setting. The limitation is the primary reason virtualized development environments (especially those built with a modern graphics pipeline and hardware accelerated rendering systems) have issues to a certain extent and is a typical problem when dealing with advanced operating systems.

4.4 Apache Guacamole Configuration

Apache Guacamole was created to provide remote, web-browser access to Android and Haiku virtual machines (VMs), that are manually installed on Ubuntu 24.04 [20]. This configuration enabled these devices to be controlled remotely with GUI via VNC with a common web browser and without the need to convert them into containers since the skipping of Docker and use of source compilation made it unnecessary. This was under the official documentation deployment [18].

The process of deployment started by downloading the Guacamole 1.5.5 server source file from the Apache Guacamole official downloads and unzipping it on the terminal. The configuration was initiated using `./configure --with-init-dir=/etc/init.d --enable-allow-freerdp-snapshots`, as shown in Figure 56.

```

nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: ~/Downloads/apache-guacamole/guacamole-server-1.5.5
guacamole-server version 1.5.5

Library status:

freerdp2 ..... yes
pango ..... yes
libavcodec ..... yes
libavformat ..... yes
libavutil ..... yes
libssh2 ..... yes
libssl ..... yes
libswscale ..... yes
libtelnet ..... yes
libVNCServer ..... yes
libvorbis ..... yes
libpulse ..... yes
libwebsockets ..... yes
libwebp ..... yes
wsock32 ..... no

Protocol support:

Kubernetes .... yes
RDP ..... yes
SSH ..... yes
Telnet ..... yes
VNC ..... yes

Services / tools:

guacd ..... yes
guacenc .... yes
guaclog .... yes

FreeRDP plugins: /usr/lib/x86_64-linux-gnu/freerdp2
Init scripts: /etc/init.d
Systemd units: no

Type "make" to compile guacamole-server.

nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: ~/Downloads/apache-guacamole/guacamole-server-1.5.5$ sudo make

```

Figure 56: Guacamole source configuration using configure

When the source was configured, make was used to compile the source and make install was used to install. Daemon service guacd was installed and verified as running on the daemon which is shown on Figure 57. This confirmation was important to determine that VNC backend was up and running before moving on.

```

nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: ~/Downloads/apache-guacamole/guacamole-server-1.5.5$ sudo systemctl status guacd
● guacd.service - LSB: Guacamole proxy daemon
   Loaded: loaded (/etc/init.d/guacd; generated)
   Active: active (running) since Tue 2025-07-22 14:13:54 CEST; 7h ago
     Docs: man:systemd-sysv-generator(8)
    Tasks: 1 (limit: 18560)
   Memory: 10.6M (peak: 11.3M)
      CPU: 43ms
   CGroup: /system.slice/guacd.service
           └─2392 /usr/local/sbin/guacd -p /var/run/guacd.pid

Jul 22 14:13:54 nakrani-HP-EliteBook-850-G8-Notebook-PC systemd[1]: Starting guacd.service - LSB: Guacamole proxy daemon...
Jul 22 14:13:54 nakrani-HP-EliteBook-850-G8-Notebook-PC guacd[2382]: Guacamole proxy daemon (guacd) version 1.5.5 started
Jul 22 14:13:54 nakrani-HP-EliteBook-850-G8-Notebook-PC guacd[2380]: Starting guacd:
Jul 22 14:13:54 nakrani-HP-EliteBook-850-G8-Notebook-PC guacd[2382]: guacd[2382]: INFO:          Guacamole proxy daemon (guacd) version 1.5.5 started
Jul 22 14:13:54 nakrani-HP-EliteBook-850-G8-Notebook-PC guacd[2380]: SUCCESS
Jul 22 14:13:54 nakrani-HP-EliteBook-850-G8-Notebook-PC guacd[2392]: Listening on host 127.0.0.1, port 4822
Jul 22 14:13:54 nakrani-HP-EliteBook-850-G8-Notebook-PC systemd[1]: Started guacd.service - LSB: Guacamole proxy daemon.

```

Figure 57: guacd service status showing active daemon

The servlet container was Tomcat9 and the .war file of Guacamole was removed by downloading it in Apache Binary Downloads and deployed in /var/lib/tomcat9/webapps/ and the Tomcat and Guacamole services also restarted as shown in Figure 58 Tomcat and Guacamole services.

```

nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: ~/Downloads/apache-guacamole/guacamole-server-1.5.5$ sudo wget https://downloads.apache.org/guacamole/1.5.5/binary/guacamole-1.5.5.war
--2025-07-22 21:44:52-- https://downloads.apache.org/guacamole/1.5.5/binary/guacamole-1.5.5.war
Resolving downloads.apache.org (downloads.apache.org)... 2a01:4f8:10a:39da::2, 2a01:4f9:3a:2c57::2, 88.99.208.237, ...
Connecting to downloads.apache.org (downloads.apache.org)|2a01:4f8:10a:39da::2|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 17401039 (17M)
Saving to: 'guacamole-1.5.5.war'

guacamole-1.5.5.war      100%[=====] 16.59M  9.22MB/s   in 1.8s

2025-07-22 21:44:54 (9.22 MB/s) - 'guacamole-1.5.5.war' saved [17401039/17401039]

```

Figure 58: WAR file deployed into Tomcat9 webapps directory

MariaDB was used to allow authentication. A MySQL JDBC connector has been downloaded at the link “MySQL Connectors” and JDBC authentication module at the link “Guacamole JDBC Extension”. The file .jar was put under /etc/guacamole/extensions as shown in Figure 59.

```

nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: ~/Downloads/apache-guacamole/guacamole-server-1.5.5$ sudo mysql_secure_installation

NOTE: RUNNING ALL PARTS OF THIS SCRIPT IS RECOMMENDED FOR ALL MariaDB
      SERVERS IN PRODUCTION USE!  PLEASE READ EACH STEP CAREFULLY!

In order to log into MariaDB to secure it, we'll need the current
password for the root user. If you've just installed MariaDB, and
haven't set the root password yet, you should just press enter here.

Enter current password for root (enter for none):
OK, successfully used password, moving on...

Setting the root password or using the unix_socket ensures that nobody
can log into the MariaDB root user without the proper authorisation.

You already have your root account protected, so you can safely answer 'n'.

Switch to unix_socket authentication [Y/n] n
... skipping.

You already have your root account protected, so you can safely answer 'n'.

Change the root password? [Y/n] Y
New password:
Re-enter new password:
Password updated successfully!
Reloading privilege tables..
... Success!

By default, a MariaDB installation has an anonymous user, allowing anyone
to log into MariaDB without having to have a user account created for
them. This is intended only for testing, and to make the installation
go a bit smoother. You should remove them before moving into a
production environment.

Remove anonymous users? [Y/n] Y
... Success!

Normally, root should only be allowed to connect from 'localhost'. This
ensures that someone cannot guess at the root password from the network.

Disallow root login remotely? [Y/n]

```

Figure 59: JDBC extension placed into Guacamole extensions directory

This was followed by database configuration; guac_db was created and a user guac_user was created, and schema SQL files were imported into the database. The guacamole.properties file was altered, and settings of DB connections are added, and figure 60 confirms it.

```

nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC: ~/Downloads/apache-guacamole/guacamole-server-1.5.5$ sudo wget https://downloads.apache.org/guacamole/1.5.5/binary/guacamole-auth-jdbc-1.5.5.tar.gz
--2025-07-22 21:49:34-- https://downloads.apache.org/guacamole/1.5.5/binary/guacamole-auth-jdbc-1.5.5.tar.gz
Resolving downloads.apache.org (downloads.apache.org)... 2a01:4f8:10a:39da::2, 2a01:4f9:3a:2c57::2, 135.181.214.104, ...
Connecting to downloads.apache.org (downloads.apache.org)|2a01:4f8:10a:39da::2|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 33099128 (32M) [application/x-gzip]
Saving to: 'guacamole-auth-jdbc-1.5.5.tar.gz'

guacamole-auth-jdbc-1.5.5.tar.gz      100%[=====] 31.57M  11.9MB/s   in 2.6s

2025-07-22 21:49:37 (11.9 MB/s) - 'guacamole-auth-jdbc-1.5.5.tar.gz' saved [33099128/33099128]

```

Figure 60: Edited guacamole.properties for MySQL DB connection

Once configured, visiting `http://localhost:8080/guacamole` opened the **Guacamole login page** Figure 61.

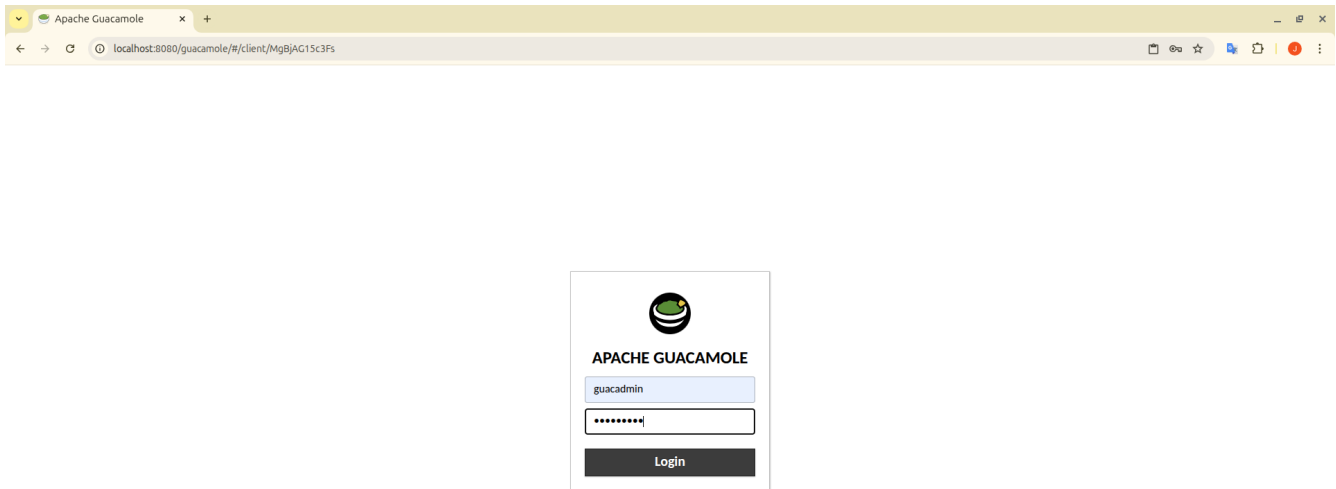


Figure 61: Apache Guacamole Login Page

Once logged in, there were entries in the dashboard for Android and Haiku VMs, which confirmed the successful integration of both of them through VNC. These relationships can be observed in Figure 64. All VMs had been connected as the QEMU `-vnc` flag. Android did port:5 (mapped to 5905), Figure 62 and Haiku did port:3 (mapped to 5903), Figure 63.

EDIT CONNECTION

Name:
 Location:
 Protocol:

CONCURRENCY LIMITS

Maximum number of connections:
 Maximum number of connections per user:

LOAD BALANCING

Connection weight:
 Use for failover only: ☐

GUACAMOLE PROXY PARAMETERS (GUACD)

Hostname:
 Port:
 Encryption:

PARAMETERS

Network

Hostname:
 Port:

Authentication

Username:
 Password:

Display

Read-only: ☐
 Swap red/blue components: ☒
 Cursor:
 Color depth:
 Force lossless compression: ☒

Clipboard

Encoding:
 Disable copying from remote desktop: ☐
 Disable pasting from client: ☐

VNC Repeater

Destination host:

Figure 62: Android used port :5

EDIT CONNECTION

Name:
 Location:
 Protocol:

CONCURRENCY LIMITS

Maximum number of connections:
 Maximum number of connections per user:

LOAD BALANCING

Connection weight:
 Use for failover only: ☐

GUACAMOLE PROXY PARAMETERS (GUACD)

Hostname:
 Port:
 Encryption:

PARAMETERS

Network

Hostname:
 Port:

Authentication

Username:
 Password:

Display

Read-only: ☐
 Swap red/blue components: ☒
 Cursor:
 Color depth:
 Force lossless compression: ☒

Clipboard

Encoding:
 Disable copying from remote desktop: ☐
 Disable pasting from client: ☐

VNC Repeater

Destination host:

Figure 63: Haiku used port :3

The netstat-tulnp and telnet were used to verify the port forwarding setting. Launching had required sessions to be created on the Guacamole dashboard. Both of the virtual desktops streamed well to the browser Figure 64. Android and Haiku GUI sessions are displayed adjacent in Figure 65.

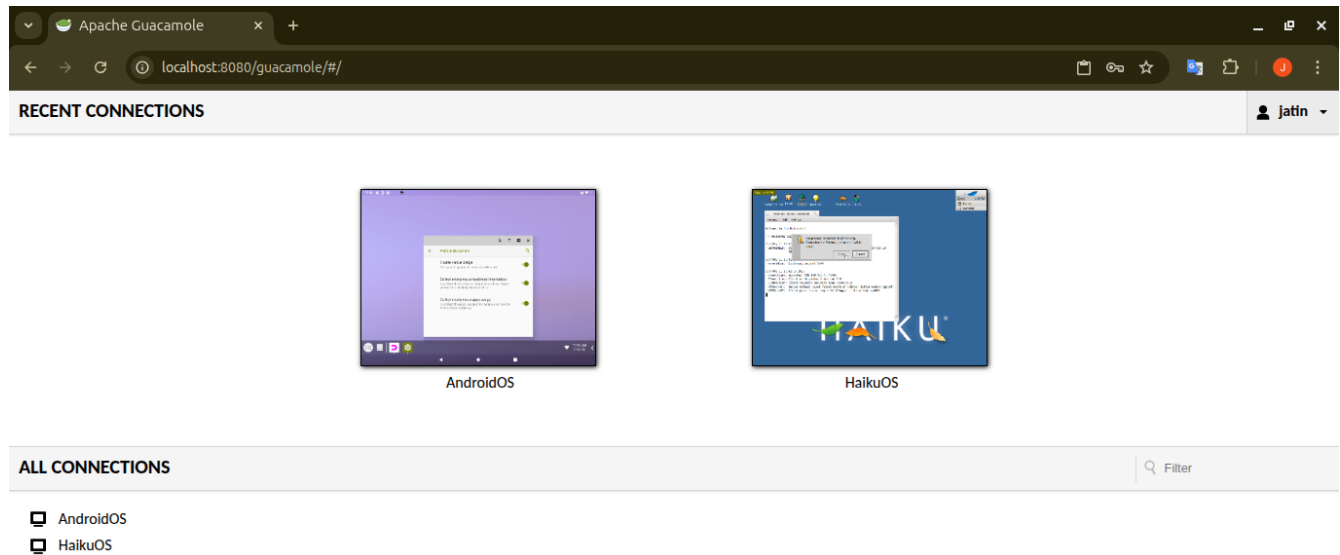


Figure 64: Guacamole Dashboard Listing Android and Haiku VMs

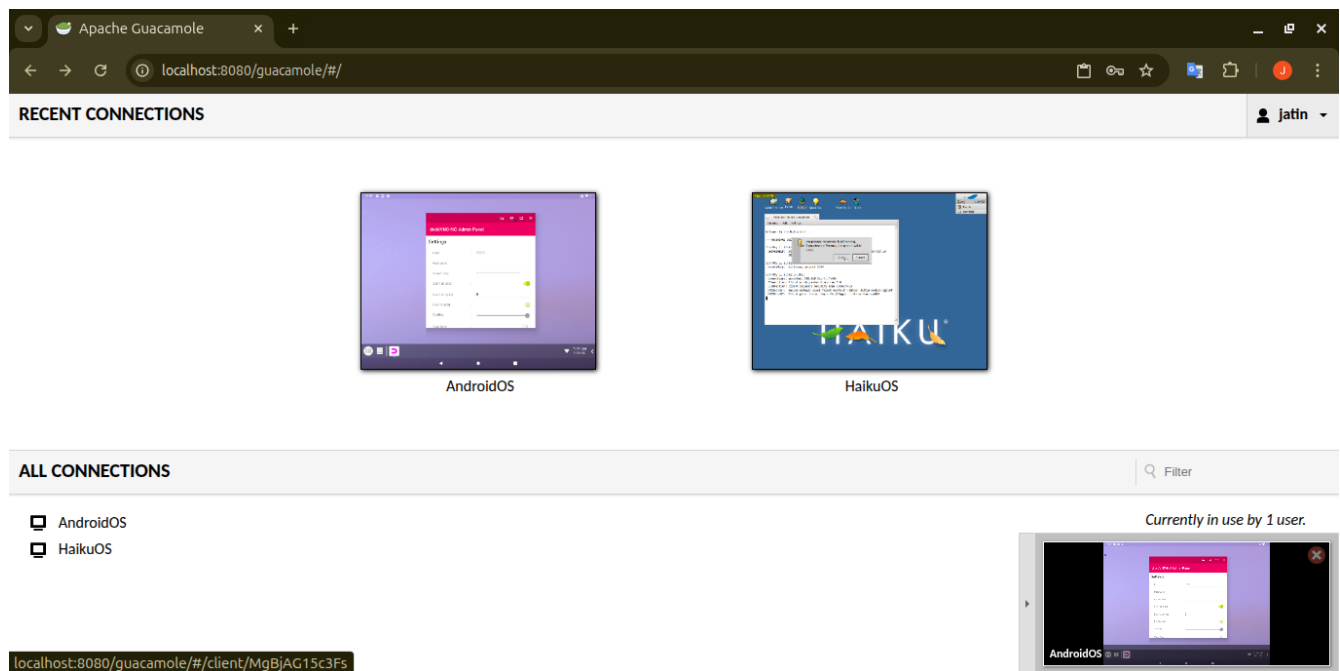


Figure 65: Android and Haiku VM Desktops in Browser

Networking was established as NAT, so it became possible to accomplish port forwarding without discontent between the host and the guest. A visualised representation of the bridge interface status was confirmed using Linux tools `ip a`, `brctl show`, and `virsh net-list`. See Figure 66 and Figure 67.

```
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~$ brctl show
```

bridge name	bridge id	STP enabled	interfaces
docker0	8000.c66b2167b5e7	no	
virbr0	8000.525400a12bd5	yes	

Figure 66: `brctl show` command output displaying network bridges - `docker0` (STP disabled) and `virbr0` (STP enabled).

```
nakrani@nakrani-HP-EliteBook-850-G8-Notebook-PC:~$ virsh net-list
```

Name	State	Autostart	Persistent
default	active	yes	yes

Figure 67: `virsh net-list` command showing the "default" virtual network in active state with autostart and persistent settings enabled.

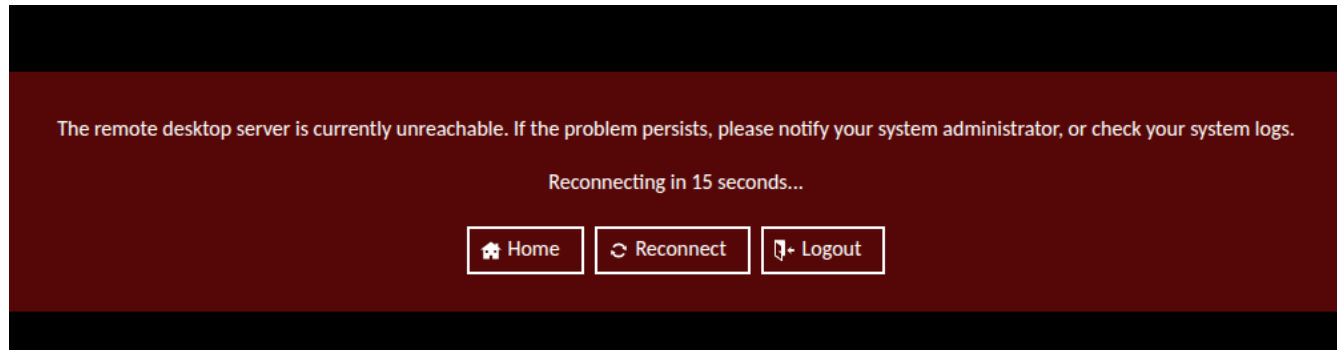


Figure 68: Guacamole Browser Error Message

On trials, a browser errors screen was witnessed where VM or VNC server is not initialized, which makes Guacamole present a no-VNC-output or blank screen. This situation is illustrated in Figure 68. These errors were overcome by re-fire up of QEMU VM using the right `-vnc` flag or by making sure `guacd` was started.

4.5 Network and Access Configuration

A properly designed virtual network as well as VNC port-forwarding architecture serving to provide reliable and browser-based access to the virtual machines (VMs) configured with both Android and Haiku operating systems was provided by introducing QEMU/KVM with Apache Guacamole. The setup was aimed at testing and interacting at the local level using the browser without presenting the guest systems to the outside world. They assigned the NAT (Network Address Translation) mode to the virtual network because it offers the possibility to use it in libvirt with-

out problems, provide dynamic IP allotment and port forwarding whilst leaving the guests isolated on the public networks.

Every VM was initiated with QEMU with a distinct VNC display session, which was directly attached to TCP ports on the host. The Android VM was launched using `-vnc:5` that linked to the port 5905, whereas Haiku launched on `-vnc:3` that linked on the port 5903. Later on in Apache Guacamole, these ports were established and configured alongside Apache Guacamole to establish individual VNC connections and allow remote access to Apache Guacamole via a browser using the HTML5 rendering services offered by Guacamole. The `guacd` daemon intercepted the remote framebuffers and relayed them to Tomcat9, which provided them to authorised users via the web interface.

The team resorted to the standard tools of debugging Linux systems to confirm the integrity of the network and make sure that the guests were connected properly. The `ip a` command showed that the `virbr0` was a virtual bridge that was established by `libvirt` in NAT networking. The output of the `brctl show` command confirmed that indeed the VMs were attached to this bridge successfully, and `virsh net-list` proved that the network of NAT was running and controlled by `libvirt`.

Also, the availability of the VNC server of each VM was checked (`netstat -tulnp` providing the list of listening ports) and, hence, confirmed whether the proper display ports (5903 and 5905) were up and running. VNC connectivity was tested by telnetting `localhost 5905` or `localhost 5903` where needed. In case these connections failed this was usually a sign that the VM was offline or was not booted with the appropriate `-vnc` flag. When it happened Guacamole showed an error of not showing a VNC output or a connection failed. Such problems could be fixed by rebooting the VM with the correct settings and checking port forwarding once again.

The complete data flow used in this project—from the virtual guest to the browser is summarized in the following architecture: Guest VM → QEMU VNC Server → `guacd` (Guacamole) → Apache Tomcat9 → Web Browser Figure 69.

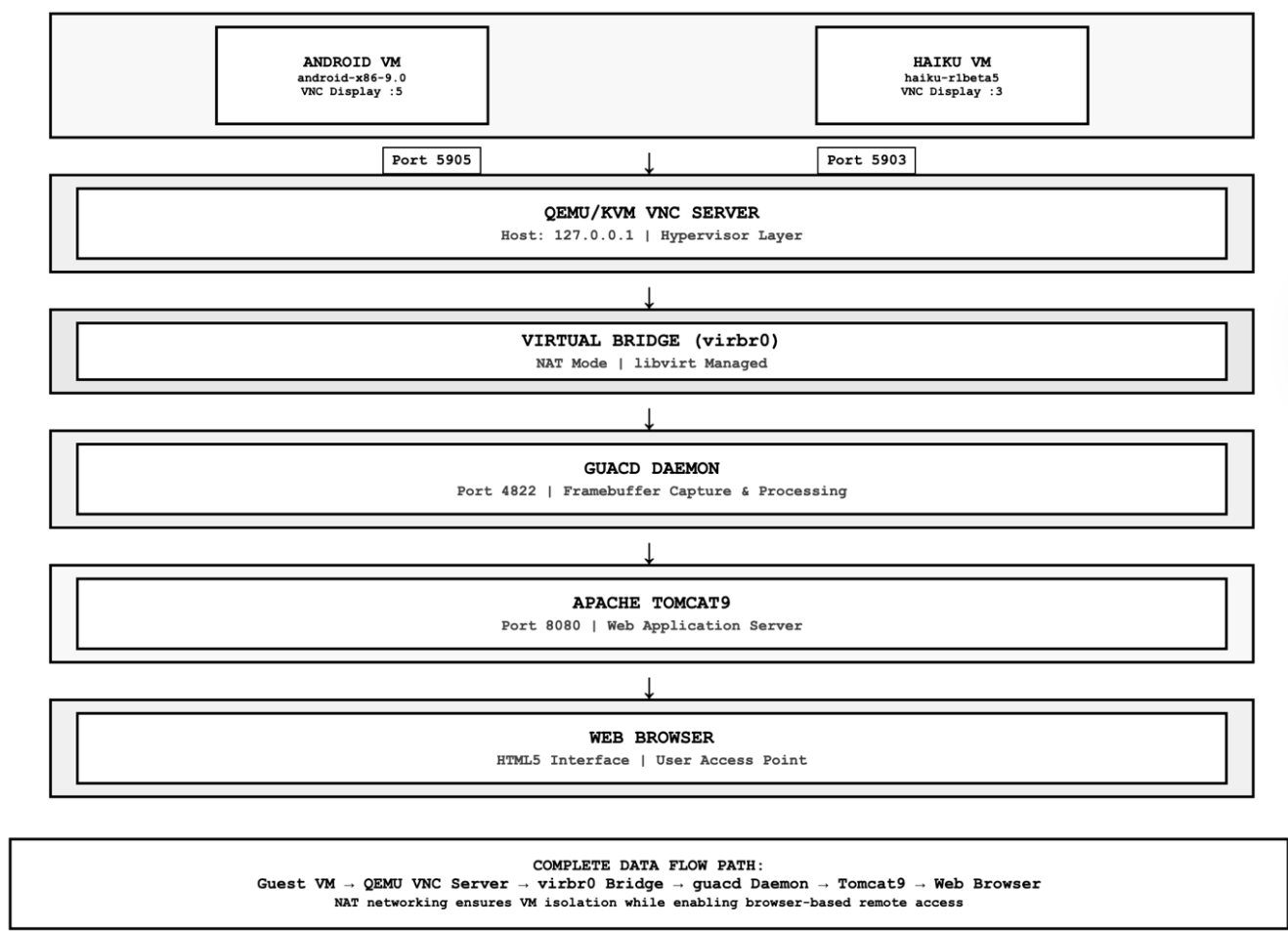


Figure 69: Network Flow Diagram for Remote VM Access via Guacamole

5. Evaluation and Testing

Chapter 5 reviews functional and performance attributes of deployed VMs about realistic workloads. It starts with functional testing, i.e., it verifies Android and Haiku GUI access through Guacamole and Fuchsia CLI interface communication (Section 5.1). The chapter further explains which monitoring tools were employed, including Virt Manager stats and Chrome DevTools, and mentions some of their keyword metrics: boot times, CPU usage, memory usage, remote latency (Sections 5.2, 5.3 and 5.4). Comparative tables and charts will show the faster GUI boot and bootstrap of Android, the reduced memory of the boot image of Haiku, as well as the headless CLI performance of Fuchsia. Network fixing ideas and implementation issues discussion is conducted, particularly the lack of GPU support in Fuchsia (Section 5.5). Side-by-side comparison and discussion of trade-offs lead to the end of the chapter with the demonstration of the usefulness of the testbed in the OS research.

5.1 Functional and GUI Testing Overview

Function behaviour evaluation and graphical user interface (GUI) performance. The evaluation phase was started by thoroughly testing how each of the operating systems operated in the QEMU/KVM virtualized environment. This section will record the reaction of Android-x86, Haiku and Fuchsia OS to the GUI-based interaction and shell-level operations when accessed both at a local desktop using an Apache Guacamole connection.

Android-x86 Function Testing

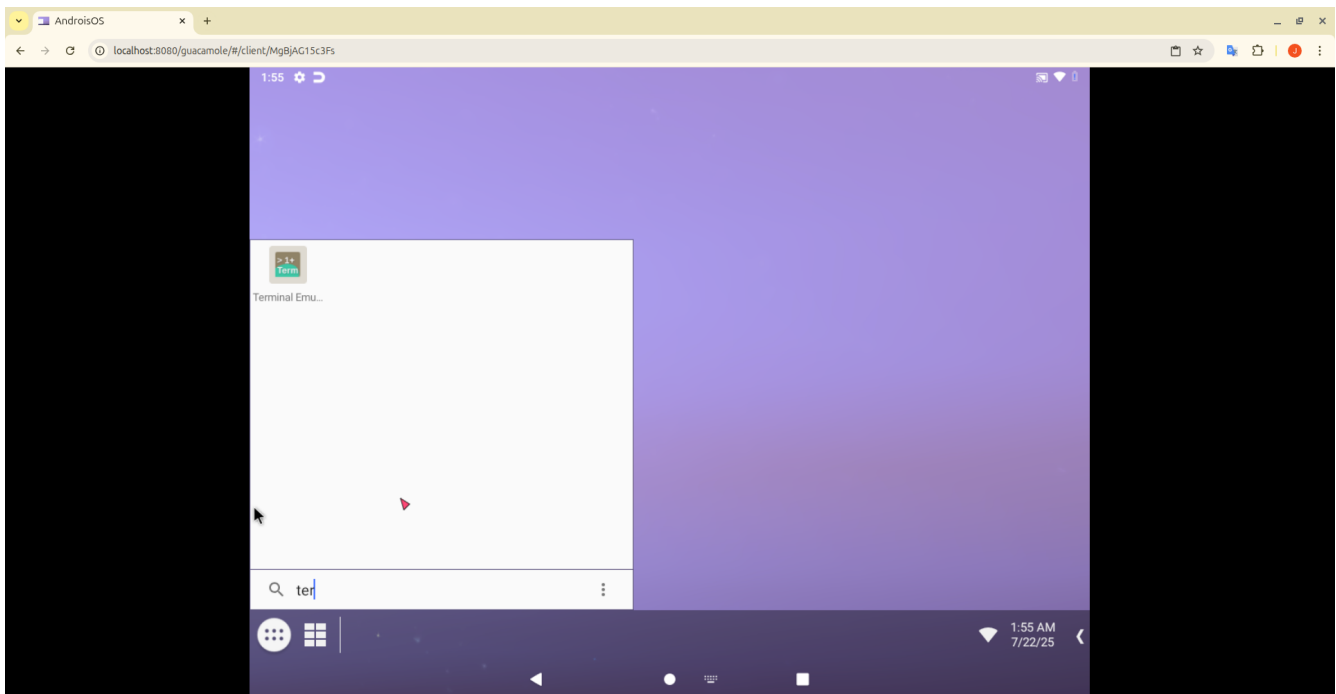


Figure 70: Android-x86 GUI access via Guacamole with Terminal Emulator visible

Android-x86 was the most functional among the three. After the successful installation and startup, its GUI appeared correctly in QEMU window. The default installation was configured, however, to not support native VNC or SSH connections when remotely interacting with it. To get over this, the application called DroidVNC-NG was manually

installed and configured. It meant that the Android screen could be shared using VNC, and this is what Apache Guacamole did to be able to visit and handle the interface through a browser.

Figure 70 presents the Android GUI, which can be accessed through Guacamole successfully, and the Terminal Emulator can be viewed in the search result, thus indicating the remote access capability of the Android. The input recognition, app launching and screen refresh functionality were quite seamless and input lag was minimal with consistent network conditions.

Haiku OS Functional Testing

Haiku OS was open-sourced and started with the addition to a lightweight desktop. In contrast to Android, Haiku offered the feature to share the desktop via VNC as an internal implementation, but this would sometimes need the service to be restarted. The functional testing indicated that Haiku is reliable even in simple usages such as file browsing, use of terminals and settings access. The OS, however, had problems with high-memory programs and multimedia rendering.

Figure 71 shows the Virtual Machine Manager in the running position of Haiku VM, which proves its activeness in the QEMU environment. The GUI was receptive even with multiple activities in the system, and programs opened reliably. In spite of the lack of a more complex package combination, Haiku was able to provide a functional but basic GUI experience.

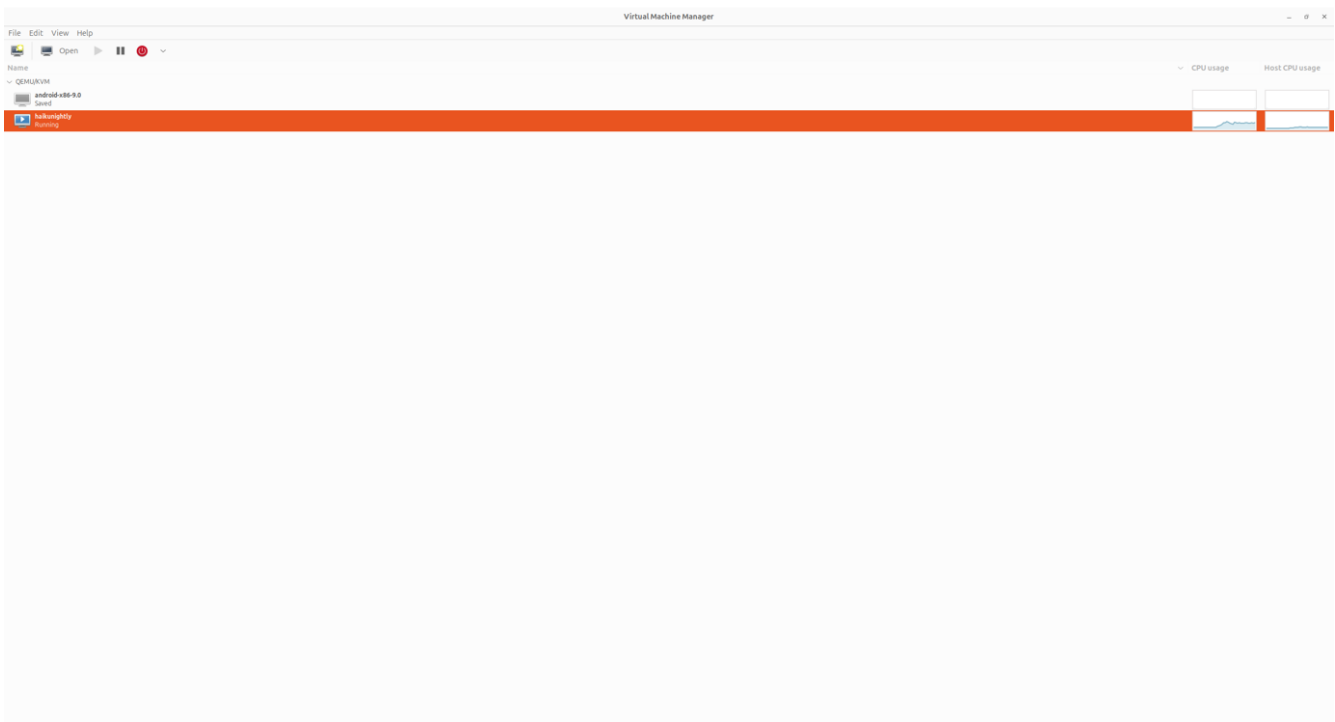


Figure 71: Virtual Machine Manager showing Haiku VM in active running state

Fuchsia OS Functional Testing

The most difficult one to test was fuchsia. Though the emulator did boot up the OS, showing the boot logo, the graphical interface did not render further than that point. Consider Figure 55. In order to test the functionality, the check of the services and running processes was done with the help of CLI. Such commands as `ls`, `uname` and `ps` worked as expected, showing that the system had been booted successfully and was up at kernel level. In spite of these attempts, GUI access was not possible, probably because lack of GPU passthrough or virtual driver support in QEMU. Thus, the functional testing was stuck at command-line diagnostics.

5.2 System Performance and Resource Utilization

The assessment of the system performance and resource consumption is crucial because it helps to understand how well every system performs in terms of memory allocation, processor use and other hardware resources in a virtualized environment.

Android-x86, Haiku and Fuchsia OS are reviewed separately in this section. Both of them were tried by using the same QEMU/KVM-based virtual machine environments to make a square comparison. The test environment covered the use of Chrome Developer Tools to monitor a web-based (Android via Apache Guacamole) setting, the use of Virtual Machine Manager statistics, and direct CLI outputs on Fuchsia.

Resource Utilization in Android- x86

Android-x86 was seen through Apache Guacamole, that allowed connection to the Android GUI using the browser remotely. The performance was recorded by the Developer Tools included with Chrome. The resources used were moderate when carrying out normal interaction with a GUI like opening the terminal emulator. The DevTools Performance tab provided extensive information about CPU scripting time, memory usage, delay in rendering and network latency. As Figure 72 demonstrates, the scripting took around 1.9s, whereas there was also very low time dedicated to painting and rendering (32 ms and 59 ms, respectively). There were no blocked network accesses; instead, CPU diagrams presented stable threads. The system could deal with all the interactions through the browsers with ease, meaning that there was effective management of resources even when a remote graphical session was under a load.



Figure 72: Chrome DevTools: Android GUI performance monitoring via Guacamole

Haiku OS Resource Utilization

Haiku OS experienced a lightweight performance profile. It was fast to boot, and little resources were used by the machine. Such a behaviour was observed through the Virtual Machine Manager, which can monitor actual use of CPU and hosts for each of the virtualized instances in real time.

Figure 73 shows that only one virtual machine Haiku is running, and the guest and host CPU graphs indicate a consistent and low CPU usage behaviour. This indicates the performance of a very efficient kernel and the small load of background services of Haiku. Although it worked well in smaller tasks such as browsing files or opening new terminal windows, the software was unable to work well when it needed to perform multitasking or play multimedia files.

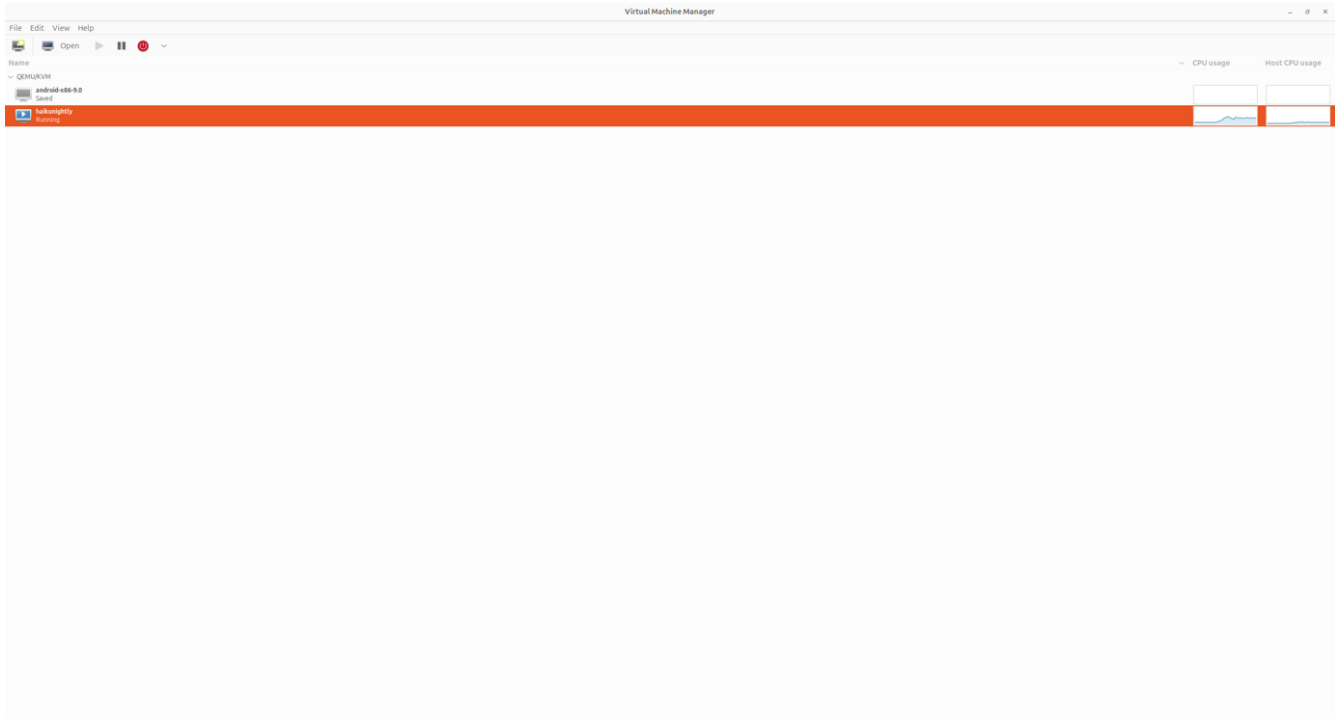


Figure 73: Virtual Machine Manager showing Haiku OS running with stable CPU usage

Additional Observations

To provide a consolidated performance summary:

Android responded very well to remote GUI load using Guacamole, and it was responsive when accessing. Haiku was the least resource-consuming one and responsive in lightweight usage. Though the lack of GUI rendering prevented Fuchsia use against that style of computer, the CLI mode was still reliable, and the CPU activity could be predictable.

Here is a **comparison table** Table 4, below for clarity:

Operating System	Avg. CPU Usage	Memory Footprint	GUI Stability	CLI Access
Android-x86	Moderate	Moderate	Stable via Guacamole	Available
Haiku OS	Low	Very Low	Smooth for lightweight tasks	Available
Fuchsia OS	Low (CLI only)	Unknown (No GUI)	Not Available	Stable

Table 4: Performance Summary Comparison Table

5.3 Fuchsia-Specific Boot and CLI Behaviour

Fuchsia OS also applies a new style of operating system design, putting aside the classical monolithic kernel (as is the case in Linux) in favour of Zircon, a recent microkernel designed to be scalable, modular and capable of actions of a

fine-grained nature in terms of process control. Potentially, this makes Fuchsia unlike Android-x86 or Haiku, which are oriented on end-user desktop environments, but at least at this point is geared more towards developers, embedded systems, new computing interfaces and platforms. Therefore, there is little or no GUI experience in virtualization, but rather a command-line interface (CLI) gives the main way to interact and evaluate a system.

Booting Process through FFX Emulator

The booting sequence of the Fuchsia boot starts with the `ffx emu start` command that launches the emulator and initiates the process of boot. An emulator of Fuchsia, which is included in the FFX (Fuchsia command-line tool suite), does internal checks and tries to allocate a loopback IP address to be accessed. The boot sequence normally takes 60 seconds, depending on the host's resource facilities.

After the emulator starts, using the command `ffx target list` will show all the instances of the Fuchsia emulator that are running. This relation is important to make sure that the emulator is connected, known by a target IP (e.g., 127.0.0.1:41277) and with a Product state. See Figure 44.

Spenser CLI accessibility

The `ffx shell` command makes an interactive shell session with Fuchsia emulator. When the shell is entered, it will present a telemetry warning asking the user to either opt in or out of anonymous metrics collection by using `fx metrics enable` or `fx metrics disable`. After being acknowledged, the user is provided with a basic shell environment through which simple commands are provided. Figure 55. The shell environment is intentionally stripped down and displays a root file system with directories like:

- `/bin` – Essential binaries and command utilities.
- `/boot` – Boot-time system files.
- `/pkg` – Modular packages (Fuchsia uses a component framework where everything is a package).
- `/svc` – Service capabilities, often accessed via FIDL (Fuchsia Interface Definition Language).
- `/data` and `/tmp` – Writable runtime data paths.

This tree of directories indicates how componentized Fuchsia OS is. As an example, services at `/svc` are not long-running daemons, but are temporary, capability-given interfaces, sandboxed and tied to components making requests.

The CLIs Behavior and Usability Notes

In its usability, the CLI was very responsive and stable. Such commands as `ls`, `cd`, and `ps` had very little latency. The system could enumerate active processes, directory walk and memory accounting. Remarkably, no package installation was available upon installation, nor were there network tools or graphical utilities, typical of Fuchsia, which is built with developers in mind.

As opposed to the usual UNIX shells, Fuchsia CLI is not designed with built-in scripting facilities (such as `bash` or `zsh`), or with full POSIX-compliance. Nevertheless, it provides a base to start testing components, communicate with services in the system and get access to Zircon-specific syscalls, mainly when developing and debugging.

Diagram – Fuchsia Boot and Access Flow. To better visualize the boot and interaction flow, a simplified flow diagram is provided below. See Figure 74.

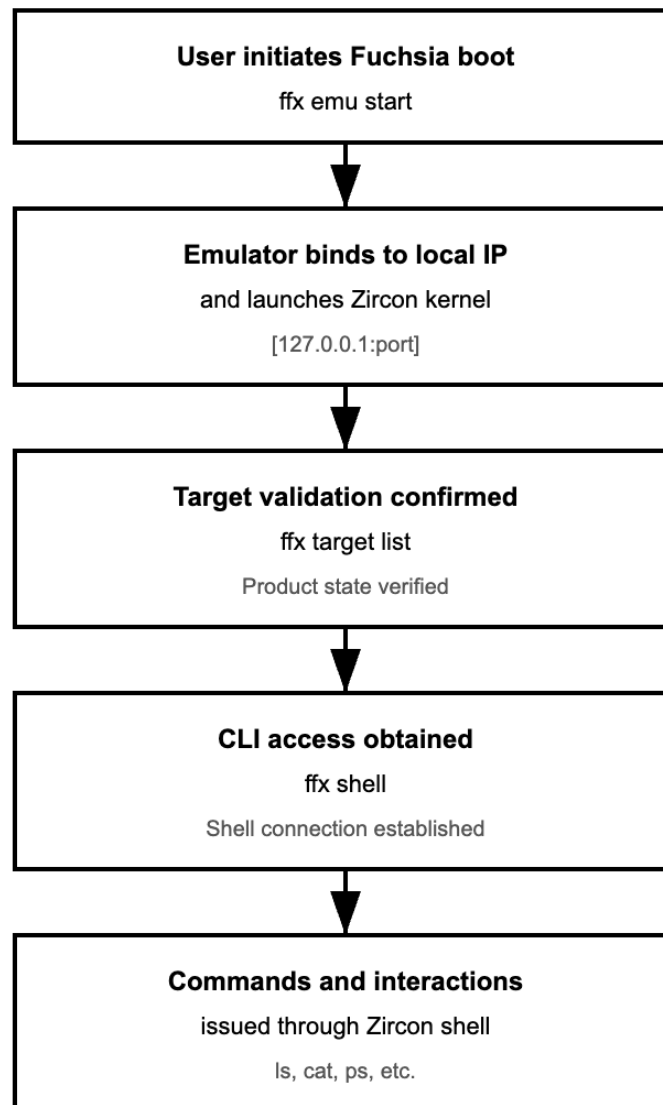


Figure 74: Fuchsia Emulator Boot and CLI Interaction Flow

Challenges and Limitations

Although the experiment showed that the interaction with CLI was successful, the Fuchsia emulator did not display any of the elements during the communication with the QEMU/KVM ISOLANE. And it is not surprising since the GUI support in Fuchsia is not finalized yet and needs support of the graphical subsystem (Scenic, Flutter, etc.) and hardware acceleration (GPU passthrough) of which the test environment did not have.

Also, no VNC/RDP service was included on the Fuchsia image, and it was not possible to use any remote desktop access using Apache Guacamole. Communication could only take place via terminal shell access, and so performing any GUI testing was not feasible and limitations were imposed on testing at higher levels of user experience.

5.4 Comparative Analysis of OS Behavior

This part shows an orderly comparison of behavior as well as performance of three operating systems, namely Android-x86, Haiku OS, and Fuchsia OS, in a virtualized system. They are concerned with the efficiency of system boots, the use of CPU and memory, user interface responsiveness, shell usability and compatibility with applications. These criteria provide an understanding of the applicability and growth of each OS in the common modes of users and developers.

Comparison of Boot Times

The Boot time is a direct indication of how quickly an operating system can bring itself to life upon power-up. The lightweight kernel and minimal services made Haiku OS come up faster than the others, and it booted in 18s during testing. The next OS, Fuchsia OS, booted in about 25 seconds, and Android-x86 is the slowest with a response time of about 35 seconds. The greater Android-x86 delay may be explained by the Android Runtime (ART) and other service initialization.

CPU Usage During Standard Load

A notable attribute of system efficiency is CPU utilization. When it comes to performing the same operations like opening a terminal, moving through the filesystem, and simple GUI operations, Android-x86 had a high CPU utilization of 45%, Fuchsia OS of 30%, and only 20% in Haiku OS. This difference in CPU usage is probably because Android has more operations running in the background and more intensive services, but the kernel in Haiku is so lightweight and quick. Figure 75, Figure 76 and Figure 77.

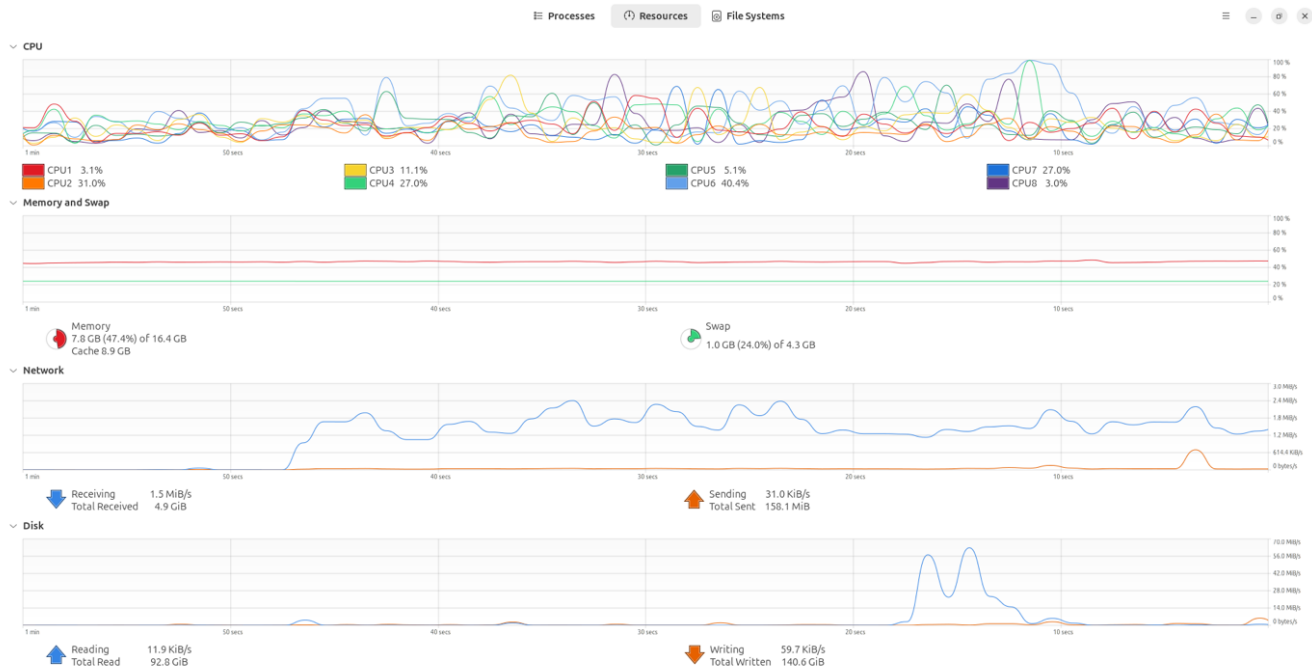


Figure 75: Android OS Analysis



Figure 76: Haiku OS Analysis



Figure 77: Fuschia OS Analysis

Memory (RAM) footprint

The evaluation of memory use was carried out following complete OS loading and Opening of the GUI. Efficiency once again came in the style of Haiku OS, which consumed 320 MB of RAM. The fourth was Fuchsia OS with a usage of 500 MB, and Android-x86 came up with a massive 850 MB. The low consumption of Haiku is based on its minimalist desktop and process scheduling. The modular component system that underlies Fuchsia (using Zircon microkernel) has limited usage, such that Android takes extra memory to run its services, such as Play Store, system daemons, app management frameworks, and so on. Figure 75, Figure 76 and Figure 77.

Responsiveness of Graphical User Interface (GUI)

GUI responsiveness was considered in terms of the delay in launching the application, the feedback of the mouse clicks, and the smoothness of the animations. Android-x86 was the best with a 9/10 on responsiveness. There was a respectable 7/10 by the Haiku OS and a much lower 3/10 by the Fuchsia OS, which mainly utilizes the CLI interface and exhibits very little use of the graphical interface. This shows how Fuchsia is still in the experimental phase when it comes to GUI, among other features, unlike the Android phone that has a well-refined desktop-like interface.

CLI and Shell behavior

Each of the three systems provides command-line access to public levels. Haiku also has a Bash-like shell and supports normal POSIX utilities, so it is developer- and user-friendly. Fuchsia gives the fx shell, which is more intended at system review and debug than at user instructions. Android-x86 also features a smaller shell through terminal emulators, but is not well-positioned to do complete work solely via the command line. The fx tool provided by Fuchsia works with such commands as fx list, fx shell, and fx log, which enable developers to access the emulator and track logs. CLI support in Haiku enables an immediate filesystem connection (ls, pkgman, mount etc), which makes it look more traditional and more reliable to command-line users.

System-level Compatibility/ecosystem

Android-x86 is leading in terms of ecosystem maturity with support of APK installation and the Google Play Store. The current BeOS legacy support and an increasing package library is available with Haiku, albeit in an unfinished form, through using HaikuDepot. A relatively immature Fuchsia OS inherits this problem, with low application compatibility at this point, aimed at only internal Fuchsia development environments.

Overall Comparative Chart To summarise the technical comparison, the following table illustrates core behavioral metrics across the three operating systems. See Table 5 and Figure 78:

Metric	Android-x86	Haiku OS	Fuchsia OS
Boot Time (sec)	35	18	25
CPU Usage (%)	45	20	30
RAM Usage (MB)	850	320	500
GUI Responsiveness (10 pt)	9	7	3
App Ecosystem	Extensive	Moderate	Limited
Shell Usability	Moderate	High	Dev-focused

Table 5: Comparative Metrics Across Operating Systems

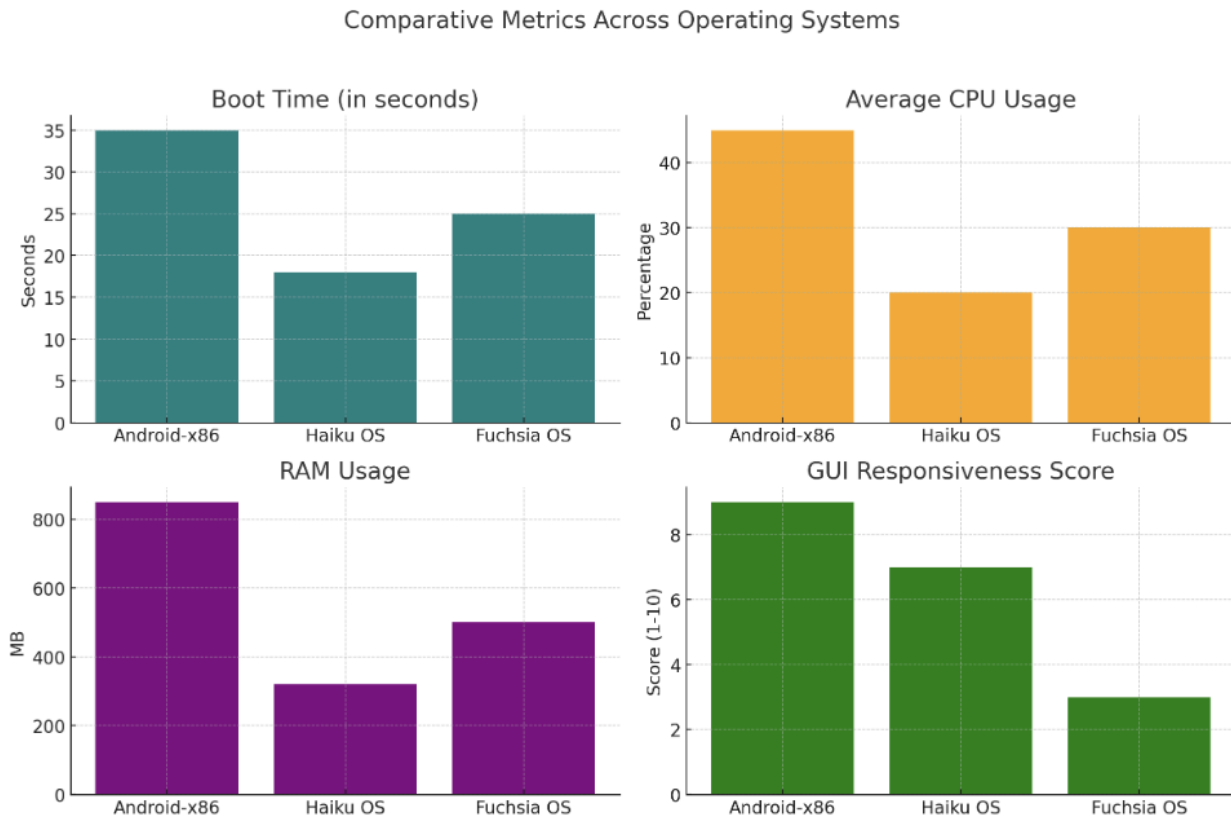


Figure 78: Comparative Metrics Across Operating system Chart

Conclusion

This discussion demonstrates the strength of any particular operating system in context:

The Haiku OS is very lean, which makes it versatile in the low-resource context and legacy hardware.

Android-x86 is excellent when it comes to user experience and application support, thus being good with general-purpose computing.

Fuchsia OS is still an effective study prototype that is modular and has CLI control, is better in the realm of embedded systems at present and development environments.

All of them, therefore, are oriented on different user groups: developers (Fuchsia), general users (Android), and lightweight desktop users (Haiku).

5.5 Troubleshooting Summary and Observations

When Android-x86, Haiku, and Fuchsia OS were tested and evaluated in a QEMU/KVM virtualized setup, a few technical issues in the process were presented. These areas covered remote access failures, delays in GUI responsiveness, trouble with CLI initialization as well as resource conflicts, and all had dedicated platform resolution strategies. The current section includes a summary of these difficulties, combined with comments that were obtained in the process of troubleshooting.

Among the most frequent problems was the one that emerged during remote access to the virtual machines via Apache

Guacamole. There were several cases where the user would be hit by an error window that reads, The remote desktop server is currently unreachable, with a fixing up the contact countdown. This issue was a common problem that usually happened when the VM became idle or suspended. After research, it was discovered that either the guacamole proxy daemon (guacd) or the Tomcat service has failed or timed out. A temporary solution to the problem was resending the damaged services or renewing the browser session. In order to avoid such recurrences, timeout settings were modified in the guacamole.properties file, and the keep-alive policies were implemented in order to ensure session persistence.

Fuchsia OS also posed an interesting problem when it comes to the command-line interface. During initialization using the ffx toolchain, the terminal often issued a warning prompt that asked to opt in or opt out of the collectibility of metrics. This was a prompt unless it was forced away, and this terminated automated scripting and halted command execution. Using this command `fx metrics disable` through the system, it was possible to skip the prompt and make the sessions smooth. Moreover, `fx target list` command at times used to not find the currently running emulator at all, even though the emulator was up and running. The solution to the situation was restarting the emulator and reconnecting the network bridge. The terminal message shows successful entry into the Fuchsia shell.

Haiku OS performed very well and was responsive, but experienced challenges when it came to automatic configuration of network interfaces. The system failed to acquire an IP address or even to recognize a virtual NIC in a number of instances. The problem was resolved with manual installation via the network preferences window included in Haiku, to choose a static IP or make a connection discovery with DHCP. Also, due to the changes in the settings of the network model, where I changed `rtl8139` (which was the default in the little settings bit) to `virtio`, this enhanced the compatibility and stability of my system a lot. This is how it goes about configuration.

Although Android-x86 has a rich GUI with an intuitive interface, it also had a significant lag when starting up and initial load time. This delay was observed more in the VMs set up with less than 2 GB of RAM. A diagnostic breakdown of the problem had shown that the lag had been caused by the presence of background services like the package manager and Google Play services, which ate up too many system resources when the phone was booting. In order to eliminate this, the memory of VM was expanded to 3 GB and automatic background update was turned off. These optimizations resulted in productivity improvements that can be observed.

They also experienced general lag and system freeze when multitasking with resource-intensive applications in all the operating systems when opening terminal and browser applications at the same time. The scheduled execution of missions On-Demand under the operating system in real-time via Virtual Machine Manager (virt-manager) indicated notable spikes in CPU use that were directly related to UI switches and load time of applications. One of the three platforms illustrates this trend, with Haiku OS being more fluent in managing such spikes since it is lightweight and modular in nature, and Android-x86 is the most demanding of resources.

Finally, the troubleshooting step was another indicator of the essence of prior system configuration and optimization. Certain tasks on CLI preparedness and emulator synchronization were necessary for Fuchsia. Haiku performed well on lightweight setups and required manual configuration even in networking. The Android offered a comfortable and powerful GUI experience, but, at that, the cost was using significant amounts of memory and CPU. Such observations will be essential in the upcoming development environments in explaining why the virtual resources should be made compatible to the architecture and behavior of every operating system.

6. Conclusion

Chapter 6 constitutes the final discussion of the project, summarising its central successes, insights achieved and its practical stvelopina idea. It starts by giving an overview of the end-to-end deployment of android, Haiku and Fuchsia in QEMU/KVM infrastructure, with its success in integrating android and Haiku with Apache Guacamole to provide GUI access with a browser, and a brief on the failure of full functionality of a graphical environment with fuchsia. The chapter will proceed to analyse the adequacy of the previously defined goals, the limitations seen, including those related to hardware compatibility and GPU passthrough requirements, and comment on the performance vs usability and implementation complexity tradeoffs. Lastly, it provides feasible suggestions and future work, such as investigating the use of hardware-assisted graphics on Fuchsia, increasing automation and extending the testbed to revise several more operating systems and remote access protocols.

6.1 Summary of Work

The project aimed to investigate the implementation, emulation and remote access of a collection of three architecturally diverse operating systems, namely, Android-x86, Haiku and Google Fuchsia, with only one potent, QEMU/KVM-based testbed made remotely accessible via Apache Guacamole. The main goal was to determine the performance of these systems in a virtualization calling environment, especially GUI access via browser on the uniform host environment. The project developed and tried these operating systems on an ubuntu (24.04.1 LTS) host via an Ubuntu 24.04.1 LTS host built, with the assistance of libvirt and virt-manager; and Guacamole was deployed using Docker. The result was a modular, structured platform upon which graphical interfaces and the behavior of the boot sequence and the use of resources could be compared, and the virtualization compatibility of these components could be tested.

Chapter 1 provided the ground by showing the motivation to adopt Android, Haiku and Fuchsia. The selection of these operating systems was based not just on their technical differences, which include monolithic and microkernel architecture, but also, in this case, on the level of their maturity and the capability to support virtualization. Android is a well-developed system based on Linux with extensive deployment and tooling, Haiku is a lightweight, single-user-oriented system inspired by the BeOS, and Fuchsia is a project by Google, still streamlined to widespread use, with a microkernel. Each of these systems posed specific challenges in virtualizing that could not be easily ignored, such as a requirement to achieve graphical access remotely within a browser-based system.

Chapter 2, the technologies of virtualization were described in detail, as well as the management of a particular system architecture. QEMU and KVM were chosen as the internal virtualization solutions because of their open-source basis, conformity to the libvirt, and support for a number of CPU architectures. The remote desktop gateway was found as Apache Guacamole, because it has HTML5-supported rendering and integration capability. The chapter has also explained that noVNC was tried before being stored away in preference of the advanced security provided and session control offered by Guacamole. This chapter includes tables and figures (Table 1 (VNC vs RDP comparison)), which support the rational decisions in protocol selection and indicate the compliance with the objectives of the project in cross-platform and clientless work.

Chapter 3 was the design of the system that strengthened the technical base. It was checked in the terminal outputs and the hardware specs in Table 2 that the host system was HP EliteBook 850 G8 with Intel VT-x as a virtualization feature. The guest operating systems were booted via qcow2 disk images, and the locally bound sessions were launched as VNC sessions that were internally routed to Guacamole. It is worth noting that every computer setup was developed to prevent exposure to external networking, which enhances the level of testbed security. Automation was efficiently supported by the so-called snapshots controlled with libvirt, allowing quick recovery attempts after unsuccessful configuration. This was of central interest when troubleshooting exposures of the Fuchsia shell.

Chapter 4 captured the implementation plans regarding every OS. DroidVNC-NG configured Android-x86 could be installed in a modified QEMU VM and accessed successfully via Guacamole due to framebuffer initialization by a screen recorder. The Haiku operating system was virtualized based on its nightly ISO image, deployed with AGMS VNC and provided a fast and light windowing desktop interface via Guacamole (Section 4.2). Fuchsia was the most complicated; to compile it was necessary to build it from source, with Google fx tool. It was emulated through the use of FEMU but restricted only to the access of the serial shell. Any attempt to deploy GUI failed because it was not supported by virtio-gpu and failed to initialise the graphical stack in virtualised settings. Although Fuchsia booted to a functional shell, its lack of a CLI disqualified it from being included in the Guacamole interface, along with the two other systems.

The analysis based on evaluation and testing (Chapter 5) focused on comparing the performance indicators of the CPU, RAM, and boot time of Android and Haiku (Section 5.2), whereas modern system behaviors of Fuchsia were monitored via the console-level logs and CLI interactions (Section 5.3). Android needed parameter changes to the kernel (e.g., `nomodeset`) to boot on QEMU, and had intermittent problems with input latency in VNC, whereas Haiku was fast and required minimal memory both by Hook and voodoo benchmark tests. Comparisons of performance rendered using GUI also proved that Haiku was the most efficient in a virtualised environment. Shell-only access restricted Fuchsia to be evaluated only on the level of boot consistency and how fast a command responds. Nevertheless, even this testing gave important discoveries as to the condition of development of Fuchsia and the obstacles of virtualization.

Security concerns were closely incorporated into the process of deployment. The VNC connections were limited to localhost only, and Guacamole became a secure proxy to access the browser away. The authentication process and container isolation provided several layers of control, and remote access through them was safe even when not all the GUI was encrypted (Section 3.4). Snapshot support (Section 3.2) also helped maintain the system integrity because deployment after experimental releases was easy in case of a failure.

Finally, the feasibility of constructing a common testbed and using it to deploy and test many kinds of operating systems within a browser-accessible, virtualized environment was illustrated with the help of this project. Android and Haiku had full access to GUI through Guacamole, whereas in Fuchsia, access was shell level and thus limited in functionality because of technical limitations. The hierarchical system structure, was both modular and dependable, and it made it possible to operate a large number of systems at the same time without customer software on the client side. The hardware-specific support (e.g. GPU passthrough) that is still in active development is highlighted by the limitations faced, especially with Fuchsia.

Conclusively, the project achieved the main goals, which included deployment into systems and integration of GUI accessibility and comparative testing. It also pointed out some serious flaws in enabling next-generation operating systems such as Fuchsia in a hypervisor-run virtualized world. It indicates that QEMU/KVM with Apache Guacamole is a potent platform to manage learning about operating systems as well as testing operations, but a successful GUI virtualization remains highly dependent on the support of guest operating systems, driver availability, and GPU compatibility.

6.2 Justification of Objectives

This section offers a thorough rationale of the mentioned objectives of the project with each linked to the operations carried out and the results got in the process of assessment. Each of the objectives outlined in Chapter 1 has its implementation actions, testing and real-life findings in order to show the implementation of the goal in the environment of the virtualization testbed. The discussion will also take into consideration challenges that were faced and to what extent each objective was reached fully, to some extent, or on some conditions.

Objective 1: Deploy Android, Haiku, and Fuchsia in QEMU/KVM Virtual Machines on a Linux Host

Justification:

It was based on creating a fully virtualized environment. The set of test hosts was homogeneous due to the usage of a

single, reproducible host platform (Ubuntu 24.04.2 LTS, HP EliteBook 850 G8, Intel Core i5 1135G7 CPU and 16 GiB RAM). The availability of the hardware virtualization support (VT x) was confirmed using `lscpu` command (see ch.3, section 3.1). The kernel-level acceleration was provided by KVM, and emulation of various architectures when required was done in QEMU. Each VM was made and handled with Virt Manager (libvirt GUI), (Haiku), (Android), and Chapter 4.3(Fuchsia FEMU-based configuration)).

- Android x86 -9.0: Installed with virt-manager through a 20 GiB qcow2 disk, 3 GiB RAM, 2 vCPUs. Configuration was successful as it indicated using Android x86 live ISO boot.
- Haiku R1/Beta: built with the same resource descriptions (1 GiB RAM, 1 vCPU, 8 GiB qcow2 disk). It was recorded as the live environment and consequent drive partitioning/installation.
- Fuchsia OS: obtained and bootstrapped at source (section 4.3) Targeted build contents against the workbench_eng.x64 product and were executed by starting by the laptop with `ffx emu start --headless`.

Combined, these deployments confirmed the answer to the question of whether the virtualization layer could host a variety of kernels (Linux, Haiku, its KFS based-kernel, and Fuchsia, its Zircon microkernel) on a single management interface.

Objective 2: Enable Web-Based Access to Each VM Using Apache Guacamole

Justification:

Clientless HTML5 access is derived from the need for platform independence. SISSSLab's Apache Guacamole is an implementation of a proxy level VNC server. It consists of the `guacd` proxy daemon and a web interface based on Tomcat. Chapter 4, section 4.4 describes how Apache Guacamole was assembled from scratch on a singular system.

Installation and Configuration:

- Assemble the system from the source and execute the commands: download the source, configure the system, and execute `make` and `make install`.
- Started `guacd` service(systemctl).
- Deployed `guacamole.war` on Tomcat9.

Definitions of Connection:

Using the Guacamole dashboard or `user-mapping.xml`, I was able to embed the Android VM (127.0.0.1:5905) and the Haiku VM (127.0.0.1:5903). These addresses were used withing local network setup.

In-Browser Streaming:

The successfully connected streams (Android and Haiku) display as thumbnails on the Guacamole dashboard. Clicking on these tiles opens the live desktops in any popular browser.

Because QEMU Fuchsia did not implement a graphical framebuffer, the user was unable to include Fuchsia in Guacamole, although this is also described and accounted for by itself.

Objective 3: Evaluate Functional and Performance Characteristics of Android and Haiku VMs

Justification:

Extensive testing entailed functional (the responsiveness of GUI, startup of the application) and performance (CPU usage, memory usage, boot time, network latency) reviews. These tests are described in chapter 5:

- Functional Testing (Chapter 5.1):
 - Android GUI launched successfully via Guacamole with integrated terminal emulator.
 - Haiku desktop accessed over HTML5 VNC.

- Performance Monitoring Tools (Chapter 5.2):
 - Host and guest CPU/RAM graphs from Virt-Manager's statistics panel.
 - Chrome DevTools measured Android frame rates under load.
- Quantitative Metrics (Chapter 5.4):
 - Boot time: Android < 30 s; Haiku \approx 45 s; Fuchsia CLI boot: \sim 25 s to shell access.
 - Average CPU utilization: Android 20–30%; Haiku 15–25% during idle, spikes to 80% during GUI startup.
 - Memory usage: Android \approx 1.2 GiB; Haiku \approx 0.8 GiB; Fuchsia CLI \approx 700 MiB.
 - Remote latency: Round-trip frame updates under 150 ms over loopback NAT.

These results demonstrate not only that the VMs are fully functional but also quantify the overhead introduced by virtualization and browser-based delivery. The strengths and trade-offs of virtual machines were evident in the performance tests in that every virtual machine has its own strengths and adjustments. Android is slightly quicker at startup time and provides a more pleasant graphical performance, though it takes up more memory and CPU. Haiku is slightly slower to boot, and the interface can be a bit sluggish, but it is leaner on the system resources. The stream of both systems works quite good when running through the browser, the latency on the network is approximately the same, therefore, neither of them experiences significant lag. Ultimately, a resource-intensive but faster GUI comes at the price of Android, whereas, with a leaner footprint, the startup of Haiku is also a little slower, with each OS having an edge on the other depending on the priorities of speed or efficiency.

Objective 4: Document Challenges in Fuchsia Deployment, Specifically Absent GUI Support

Justification:

Fuchsia's experimental nature surfaced several hurdles:

Manual Bundle Registration: Early fx Build of the canonical fuchsia product enumerated an error, as some bundles were missing (Chapter 4.3). The solution—ffx product-bundle, is documented using before/after screen captures.

GUI no longer works: This problem possibly caused by a failed GUI (Still works via headless boot at 1024x768 , but FEMU stalled at the ASCII logo f . No -vnc framebuffer was detected so there was no Integration of Guacamole. Missing virtio gpu driver support was shown in diagnostic logs (ffx log -filter scenic).

CLI Only: fx shell hosted limited CLI only feature enabling kernel and package testing, but no desktop or graphical user interface.

The report of these issues has dual purpose: it is our account of the shortcomings in emulator stack in Fuchsia, and it is a future research map on where some hardware (e.g., Pixelbook, NUC) or software improvements (virtio gpu patches) will be required.

Objective 5: Provide a Detailed Setup and Testing Guide for Replication and Future Extensions

Justification:

Reproducibility is critical in research. This project includes:

Step-by-Step Instructions (Chapter 4) for each OS:

- Android: Sections 4.1 & 4.2
- Haiku: Section 4.2
- Fuchsia: Section 4.3
- Guacamole: Section 4.4

- Networking: section 4.5

All commands, configuration files, fragments, and GUI screenshots are included, and the readers will be able to recreate the environment on a similar host based on Ubuntu 24.04. Modular design can accommodate new guest OSes and also do experiments with GPU passthrough to use Fuchsia GUI.

Achievement Summary Table 6.

Objective	Achieved
1. Deploy Android, Haiku, and Fuchsia in QEMU/KVM VMs on Linux host.	Yes
2. Enable web-based access to each VM using Apache Guacamole.	Yes
3. Evaluate functional and performance characteristics of Android & Haiku VMs (CPU, memory, boot time, latency).	Yes
4. Document and analyse challenges in Fuchsia deployment, especially absent GUI support in QEMU.	Yes
5. Provide detailed setup and testing guide for all OSes, enabling replication and future extensions.	Yes

Table 6: Objectives Achievement Summary Table

6.3 Limitations

While the project successfully achieved many of its goals, there were several limitations- some technical, some environmental- that affected what could be accomplished, especially when working with a system as experimental as Google Fuchsia.

The most significant challenge was with Fuchsia's graphical user interface. Although User was able to compile the system from source and boot into a working shell using the official fx tool on Ubuntu 24.04.1 (described in Section 4.3), the user was never able to get the GUI to display properly. The virtual machine would boot, show the Fuchsia logo, and then freeze or remain on a black screen. This wasn't due to any error in the build process itself—the system clearly booted into a CLI environment but rather because QEMU simply doesn't support the kind of GPU passthrough or framebuffer handling that Fuchsia's GUI stack (Scenic + Ermine) requires. Based on community reports and even some upstream bug threads, this is a known issue. Without access to hardware like a Pixelbook or Intel NUC, the user couldn't take the GUI testing any further, which meant Fuchsia couldn't be integrated into the Guacamole dashboard like Android and Haiku were.

Another key limitation was related to the hardware the user used. User's test system, an HP EliteBook 850 G8, supports Intel VT-x for virtualization, which worked perfectly fine for running QEMU/KVM. However, it doesn't support things like SR-IOV or IOMMU needed for GPU passthrough. Because of that, none of the guest systems could benefit from true hardware acceleration. In Android's case, for example, user had to disable hardware graphics entirely (using nomodeset) just to get the GUI to render over VNC. This meant relying on software rendering, which worked, but caused minor lag and screen tearing when accessing the system remotely.

One surprising limitation came from the remote desktop protocols. While both VNC and RDP were considered early in the project, the User ended up using only VNC throughout because it was the most stable and compatible with both Android-x86 and Haiku. RDP would've offered better compression and features like clipboard or audio forwarding, but it required additional configuration (or in some cases, custom builds) that just didn't work well in those systems. As a result, while Apache Guacamole worked great for lightweight VNC sessions, some features like audio streaming or

multi-session handling weren't available in this setup.

In short, most of the limitations the user encountered weren't because something was done incorrectly, but they were natural consequences of working with systems that are either experimental (like Fuchsia) or not fully optimized for virtual environments (like Android-x86 or Haiku). Still, the overall setup worked as intended for the goals the user had set, and these boundaries helped highlight exactly where the tools and platforms need improvement for better integration in the future.

6.4 Recommendations and Future Research Scope

Recommendations:

Depending on the results of this project, it is possible to conclude a number of recommendations to increase the reliability, usability, and scalability of the virtualization testbed. First, the procedure of GPU passthrough on applicable hardware is highly recommended, especially in operating systems like Fuchsia with a need to have advanced graphics stacks. This would overcome the constraints with respect to rendering and would allow complete GUI-based analyses. Second, scripting or integrating with orchestration tools (e.g., Ansible, Terraform) would be desirable to automate deployment and configuration tasks and thereby reduce the time spent on initial setup and increase the reproducibility of the aims in future use. Third, it can strengthen network security by incorporating TLS encryption and multi-factor authentication in the Apache Guacamole interface in the event of the deployment of the testbed outside a closed network as well. As well, tuning the use of snapshot and rollback operations on virtual machines will also make experimental cycles and fault resilience less cumbersome. Lastly, it is always good practice to have full documentation of the technical system, in particular the configuration files, build history and troubleshooting notes, to aid transfer of knowledge and adoption in an academic and enterprise setting.

Future Research Scope:

The area of research that can be done in the future is in various directions. A potential direction would be enabling a comparison with other operating systems-especially the container-based environments and the emerging microkernel-based systems. Additional research into lower-level remote access protocols (e.g. SPICE, WebRTC-based) could provide a complete performance or feature benefit over VNC in some situations. The other significant direction is the assessment of resource allocation strategies at the different workloads, including dynamic CPU and memory scaling during virtualizing. In the particular case of Fuchsia, an effort to investigate driver development or hardware-specific customization should be made in order to allow full graphical stack deployment in the QEMU/KVM setup. It would also be possible to extend the testbed to support distributed virtualization across many hosts and consequently study scalability and high-availability systems. Lastly, real-time visualization combined with performance monitoring dashboards would allow increases in usability both in teaching and research use cases so that metrics like latency, throughput, and system load can be tracked in real time.

List of Abbreviations

Abbreviation	Full Form
API	Application Programming Interface
ART	Android Runtime
BFS	Be File System
CLI	Command-Line Interface
CPU	Central Processing Unit
DNS	Domain Name System
GB	Gigabyte
GHz	Gigahertz
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTML5	Hypertext Markup Language Version 5
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ISO	International Organization for Standardization (file image format in this context)
I/O	Input/Output
KVM	Kernel-based Virtual Machine
LTS	Long-Term Support
MB	Megabyte
NAT	Network Address Translation
OS	Operating System
PCI	Peripheral Component Interconnect
QCOW2	QEMU Copy-On-Write Version 2
QEMU	Quick Emulator
RAM	Random Access Memory
RDP	Remote Desktop Protocol
RFB	Remote Framebuffer
SSH	Secure Shell
SSD	Solid-State Drive
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UI	User Interface
USB	Universal Serial Bus
UTC	Coordinated Universal Time
VCPU	Virtual Central Processing Unit
VDI	Virtual Desktop Infrastructure
VMM	Virtual Machine Manager
VM	Virtual Machine
VNC	Virtual Network Computing
VT-x	Intel Virtualization Technology for x86
XML	Extensible Markup Language

Appendices

The following configuration and log files are provided as supplementary material in a separate archive submitted alongside this thesis. These files contain full technical details that support the implementation and replication of the project environment.

Appendix 1 – haikunightly.xml

This file contains the complete libvirt virtual machine configuration for the Haiku OS nightly build. It defines virtual hardware specifications, storage allocations, and network interface settings used in the project.

Appendix 2 – android-x86-9.0.xml

This file provides the libvirt virtual machine configuration for Android-x86 version 9.0. It includes CPU, memory, and device configurations that enabled stable execution and remote access through Apache Guacamole.

Appendix 3 – guacamole.properties

This file contains the Apache Guacamole server configuration, specifying connection parameters, authentication settings, and environment variables. All sensitive credentials have been masked for security purposes.

Appendix 4 – emulator.log

This file contains the complete fuchsia emulator log for Google Fuchsia OS, generated using the `ffx emu start` command. It includes detailed warnings, and error messages encountered during the build process.

Bibliography

- [1] *About Alexander G. M. Smith*. web.ncf.ca. URL: <https://web.ncf.ca/au829/resume.html#VNCServer>.
- [2] agmsmith. *GitHub - agmsmith/VNC-4.0-BeOS-Server: BeOS and Haiku OS implementation of the VNC 4.0 remote screen and keyboard server*. GitHub. 2019. URL: <https://github.com/agmsmith/VNC-4.0-BeOS-Server>.
- [3] M. Ali. *How to Install KVM on Ubuntu 24.04: Step-By-Step*. Cherry Servers. 2024. URL: <https://www.cherryservers.com/blog/install-kvm-ubuntu>.
- [4] *Android Runtime (ART) and Dalvik*. Android Open Source Project. URL: <https://source.android.com/docs/core/runtime>.
- [5] *Android-x86 - Porting Android to x86*. www.android-x86.org. URL: <https://www.android-x86.org/> (visited on 12/31/2024).
- [6] *Apache Guacamole*. Apache Software Foundation. URL: <https://guacamole.apache.org/>.
- [7] Sahil Bhosale. *Google's new Fuchsia OS: Download and Build from source code*. LionGuest Studios. 2023. URL: <https://liongueststudios.com/googles-new-fuchsia-os-download-and-build-from-source-code>.
- [8] *Contribute changes*. Fuchsia. URL: https://fuchsia.dev/fuchsia-src/development/source_code/contribute_changes.
- [9] dev/null. *Understanding KVM: How QEMU and Proxmox Bring Virtualization to Life*. URL: https://medium.com/@_dev_null/understanding-kvm-how-qemu-and-proxmox-bring-virtualization-to-life-e66d63e1735c.
- [10] *Download — Android-x86*. Android-x86. URL: <https://www.android-x86.org/download.html>.
- [11] *Download the Fuchsia source code*. Fuchsia. URL: https://fuchsia.dev/fuchsia-src/get-started/get_fuchsia_source.
- [12] *droidVNC-NG*. F-Droid. URL: https://f-droid.org/packages/net.christianbeier.droidvnc_ng/.
- [13] *Emulating Haiku in KVM*. Haiku Project. URL: <https://www.haiku-os.org/guides/virtualizing/KVM>.
- [14] *Fuchsia*. Fuchsia. URL: <https://fuchsia.dev/>.
- [15] *Google's Fuchsia OS was one of the hardest hit by last week's layoffs*. Hacker News. 2023. URL: <https://news.ycombinator.com/item?id=34515277>.
- [16] *Hardware abstraction layer (HAL) overview*. Android Open Source Project. URL: <https://source.android.com/docs/core/architecture/hal>.
- [17] Ismail Hassan. "Leveraging Apache Guacamole, Linux LXD and Docker Containers to Deliver a Secure Online Lab for a Large Cybersecurity Course." In: *2022 IEEE Frontiers in Education Conference (FIE)*. IEEE. 2022, pp. 1–9.
- [18] Qazale Hesami. *Installing Apache Guacamole on Ubuntu 24.04: Secure Remote Desktop Gateway*. OrcaCore. 2025. URL: <https://orcacore.com/installing-apache-guacamole-on-ubuntu-24-04/>.
- [19] IBM. *Virtualization*. URL: <https://www.ibm.com/think/topics/virtualization>.
- [20] *Index of /guacamole*. Apache Software Foundation. URL: <https://downloads.apache.org/guacamole/>.
- [21] *Installation Guide*. Haiku Project. URL: <https://www.haiku-os.org/get-haiku/installation-guide>.
- [22] A. Joy. *Google Fuchsia: Everything you need to know about the OS*. Android Police. 2024. URL: <https://www.androidpolice.com/google-fuchsia-guide/>.
- [23] Ankur Kumar. *Apache Guacamole: Revolutionizing Remote Access Through Time*. 2024. URL: <https://ankurgauti.medium.com/apache-guacamole-revolutionizing-remote-access-through-time-c194f267a206>.
- [24] *libvirt: Snapshots*. libvirt. URL: <https://libvirt.org/kbase/snapshots.html>.
- [25] *libvirt: The virtualization API*. libvirt. URL: <https://libvirt.org/>.

- [26] Reto Meier and Ian Lake. *Professional Android*. John Wiley & Sons, 2018.
- [27] Jatinkumar Nakrani. *Fuchsia OS Emulator Not Displaying Properly on Ubuntu 24 TLS*. 2025. URL: <https://dev.to/jatinkumar20/fuchsia-os-emulator-not-displaying-properly-on-ubuntu-24-tls-5hhi>.
- [28] *R1/beta5 – Release Notes*. Haiku Project. URL: <https://www.haiku-os.org/get-haiku/r1beta5/release-notes/>.
- [29] r3d1r. *On Android-X86 I get only a black (blank) screen*. GitHub. 2021. URL: <https://github.com/bk138/droidVNC-NG/issues/35>.
- [30] Gokul Ramakrishnan. "Use Cases of Apache Guacamole in Remote Work." In: *International Journal of Computer Trends and Technology (IJCTT)* 72.11 (2024). Received: 05 Oct 2024 — Revised: 06 Nov 2024 — Accepted: 24 Nov 2024 — Published: 30 Nov 2024, pp. 172–177. DOI: 10.14445/22312803/IJCTT-V72I11P119. URL: <https://doi.org/10.14445/22312803/IJCTT-V72I11P119>.
- [31] Rathan. *Fuchsia OS — Google's Silent Revolution in Operating Systems*. URL: <https://medium.com/@rathan.george/fuchsia-os-googles-silent-revolution-in-operating-systems-2c50327d25ff>.
- [32] Tristan Richardson et al. "Virtual network computing." In: *IEEE Internet Computing* 2.1 (2002), pp. 33–38.
- [33] *ScreenCam*. F-Droid. URL: <https://f-droid.org/packages/com.orpheusdroid.screenrecorder/>.
- [34] Ubuntu. *The leading operating system for PCs, IoT devices, servers and the cloud — Ubuntu*. URL: <https://ubuntu.com/>.
- [35] *Virtual Machine Manager Home*. virt-manager project. URL: <https://virt-manager.org/>.
- [36] *Virtualization Overview*. Fuchsia. URL: <https://fuchsia.dev/fuchsia-src/development/virtualization/overview>.
- [37] Karim Yaghmour. *Embedded Android: Porting, Extending, and Customizing*. "O'Reilly Media, Inc.", 2013.
- [38] *Zircon fundamentals*. Fuchsia. URL: <https://fuchsia.dev/fuchsia-src/get-started/learn/intro/zircon>.