JOHANNES KEPLER
UNIVERSITY LINZ

Author
Elena Bogner
12232281

Submission
Institute of
Networks and Security

Thesis Supervisor
Stefan Kempinger, MSc

Assistant Thesis
Supervisor
Maximilian Arthofer, MSc

April 2026

Analysis of the Digidow face sensor in different software architectures

Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

This bachelors thesis presents an analysis of different Operating System environments for the face sensor of the CDL Digidow project. The thesis scores the different environments based on a set of performance and security metrics. This thesis aims to determine the best environment for the face sensor. The paper outlines the different environments, the process of porting the face sensor, the different metrics as well as how to measure them. It also proposes a concept of an ideal setup for the sensor.

Acknowledgements

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World. We gratefully acknowledge financial support by the Austrian Federal Ministry of Economy, Energy and Tourism, the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, and Österreichische Staatsdruckerei GmbH.

Contents

Abstract ii

Acknowledgements iii

List of Figures vi

List of Tables vii

Listings viii

1 Introduction 1

2 Background 3

2.1 System Architecture of DigiDow 3

2.2 The Face sensor environment and design 5

3 Porting the face sensor software 7

4 Software environments subject to analysis 11

4.1 Raspberry Pi OS Lite 11

4.2 RedoxOS 17

4.3 Alpine Linux 21

4.4 PiCore 29

4.5 DietPi 32

5 Analysis 34

5.1 Performance 34

5.1.1 RAM Usage 35

5.1.2 Disk Usage and Image size 38

5.1.3 Boot speed 39
5.1.4 Image Processing speed 43
5.1.5 Sensor-arx startup time 48
5.2 Security 50
5.2.1 Minimal Attack Surface 51
5.2.2 Hardening the kernel 55
5.2.3 Package manager security 60

6 Conclusion **64**
6.1 Environment results 64
6.2 An Ideal System Environment 66
6.3 Future Work 67

Bibliography **68**

List of Figures

2.1 Overview of Digidow's system design [2] 5

5.1 RAM Usage statistics during Sensor-arx execution . 37
5.2 Disk Usage statistics 38
5.3 Image size statistics 40
5.4 Boot speed statistics 43
5.5 Image processing speeds 47
5.6 Sensor-arx startup speeds 50

List of Tables

5.1 Available security-critical programs in all 4 environments 55
5.2 Overview of kernel security features that are enabled in at least one environment 59
5.3 Overview of package manager security measures . . 62

Listings

3.1 Script to install components for cross-compilation . 8
3.2 Contents of .gitconfig 9

4.1 Bash Script to install Edge TPU runtime 13
4.2 Bash script to install PyCoral and Python 3.9.21 . . . 14
4.3 Important configurations in settings.json 16
4.4 Creating the RedoxOS image 18
4.5 Preparing boot process for RedoxOS 20
4.6 Bash script to setup user privilege escalation 22
4.7 File modifications for makefile 24
4.8 File modifications for usb_device_interface.h 24
4.9 Dockerfile for building the image 26
4.10 Commands to compile libedgetpu 28
4.11 Shell script to download extensions and their dependencies 30

5.1 Bash script to create a swap file 36
5.2 Script to record RAM usage 36
5.3 Systemd service file for boot-record service 41
5.4 OpenRC service file 42
5.5 Modifications to src/pipeline.rs 44
5.6 Modifications to src/config.rs 45
5.7 Modifications to flake.nix and Cargo.toml 45
5.8 AWK script to calculate average from benchmark log 46
5.9 AWK script to calculate standard deviation from benchmark log 46
5.10 C program to determine startup speed of sensor-arx 48

5.11 Bash script to install Vuls on Debian systems or Alpine Linux 63

Glossary

Address Space Layout Randomization Security feature which randomizes memory section to make important memory addresses harder to predict. 56

Bazel An open source extensible build system developed by Google. 23

Coral Framework that allows creation of customized AI Models. 6

Cross-Compilation Building software from source for a different target system. 7

Docker Framework to create containerized software environments. 6

Dockerfile Instruction file docker uses to build a container image. 6

Git An Open source versioning system. 8

Glibc An implementation of Libc primarily focused on dynamic linking. 23

Go An open source programming language developed by Google. 62

Gstreamer Open source library used for audio/video processing. 6, 48

Hyper Text Transfer Protocol Communication protocol designed for the transfer of files. 8

HyperText Transfer Protocol Secure Secure version of HTTP using TLS. 9

Integrity Measurement Architecture Linux kernel subsystem used to appraise files. 6

Make An open source build system developed by the GNU project. 23

Mandatory Access Control An access control system, which restricts users from accessing resources within the system based on defined policies. 57

Musl libc A small implementation of Libc focused on static linking. 21

Network Time Protocol Network protocol used to sync time and date. 22

Nix Tool for creating reproducible and reliable software environments. 6

Nix Bundle A binary, built with Nix, that includes all of its dependencies. 7, 9

nix-pkgs Collection of packages available for installation in the Nix utility. 9

OpenRC A small and modular initialization system used in Gentoo and Alpine Linux. 41

Pluggable Authentication Modules Security framework which allows for authentication based on specific policies. 54

Podman A container manager similar to Docker. 18

Position Independent Executable Executables that do not need memory address reconfiguration. 56

Pyenv Utility to manage multiple python versions on the same system. 14

QEMU Open source emulator and virtualizer. 7

Raspberry Pi Imager Utility provided by the creators of Raspberry Pi to flash an Operating System image onto a storage medium. 12

Rust A memory safe programming language that prevents common memory safety issues. 5

Seccomp A Linux kernel security feature that can restrict the system calls a program can make. 57

Secure Shell Communication protocol that can be used to remotely control a system through a shell. 54

Sensor-arx The bundled face sensor binary. 35, 48

Swap file / partition Storage on a physical storage medium that acts as virtual memory should the system run out of physical memory. 35

Systemd A common initialization and service management system. 40

TCE-load The package manager of TinyCore. 62

TianoCore A Cross-platform firmware development environment. 19

Tor Open source networking technology used to encrypt and anonymize TCP Communications. 5

List of Acronyms

APK Alpine Package Keeper. 21, 22

APT Advanced Package Tool. 13

ASLR Address Space Layout Randomization. 54, 56

BIOS Basic Input/Output System. 19

CDL Christian Doppler Laboratory. 1

DHCP Dynamic Host Configuration Protocol. 42

HTTPS HyperText Transer Protocol Secure. 8

IMA Integrity Measurement Architecture. 6

INS Institute for Networks and Security. 8

MAC Mandatory Access Control. 57

NTP Network Time Protocol. 22

OS Operating System. 11, 17

PAM Pluggable Authentication Modules. 54

PIE Position Independent Executable. 54, 56

RAM Random-Access Memory. 35

SSH Secure Shell. 8

TLS Transport Layer Security. 9

TPU Tensor Processing Unit. 5, 12

UEFI Unified Extensible Firmware Interface. 19

Chapter 1

Introduction

Documents, tickets, keys, and other forms of authentication have one major problem: They are too easily lost, stolen, or somehow damaged. They are not persistent and could potentially be forged by malicious actors. In this digital age, it is remarkable that a better solution does not meaningfully exist. Two major issues would need to be solved: Identification/Authentication without the use of physical documents and ensuring the collection of user data doesn't intrude on users' privacy. This is the problem that CDL Digidow aims to solve.

CDL Digidow's role, therefore, is one of a system that organizations or governments can use in order to verify the identity of the user, which then receives appropriate rights/access to the given entity. Due to this fact, Digidow's highest priority is security. This necessitates that all possible attack vectors have to be guarded against all types of cybersecurity attacks. One of these attack vectors is the face sensor that is operated by a Raspberry Pi. Although CDL Digidow also allows for other types of sensors, the subject of this thesis is the face sensor.

Chapter 2 gives an explanation of what Digidow is, in what environment the sensor has to work, as well as additional theory needed to comprehend the following chapters. Chapter 4 describes the different environments and process of adapting the

face sensor software to the new environment. Chapter 5 Describes the different performance and security metrics and discusses their results.

Chapter 2

Background

As mentioned earlier, the goal of CDL Digidow is to establish a service which can be used by other organizations to verify user identities. The data needed for this task includes incredibly sensitive information and is vulnerable to malicious actors seeking to use it for their own gain. CDL Digidow uses a decentralized design to ensure fundamental human rights on data protection are not infringed upon. This next section dives into this design and in what environment the face sensor operates.

2.1 System Architecture of DigiDow

CDL Digidow's design puts a special emphasis on a few key philosophies which are:

1. The system is voluntary, and data should only be collected from participating members
2. The system is decentralized to prevent failures of a single device, which could compromise the data of all users.
3. Information on sensors is publicly available, and sensors are verified through a central registry.

Digidow fulfills these through its system design. An overview of the system design can be seen in Figure 2.1. It depicts the interactions (authentication request, mutual verification to establish trust and identity certification) between the three main entities described below.

There are three main entities involved during the identity verification process:

1. The Verifier - The entity that is trying to establish the identity of a user. This could be a company regulating employee access through cameras or a government official attempting to verify the identity of a person trying to cross the border.
2. The Biometric Sensor - The sensor that records the biometric data of the user. Any kind of biometric sensor is possible, but face recognition, fingerprint, or retina scan would be the most common.
3. The Personal Agent - This is the digital representation of the user who is trying to be verified. The data for the personal agent is not stored centrally but rather in a distributed system.

An example for the identification process can be seen at the Johannes Kepler University in Linz, where a camera, equipped with face recognition software, is set up to automatically control access to various laboratories. The face sensor of this project is the subject of this thesis.

The identification works as follows: First, the face sensor (biometric sensor) detects that a person is walking in the frame and whether or not they are attempting to go into the laboratory. If the person is participating in the study, the sensor tries to authenticate the user. If the person is not participating, any data collected is immediately discarded. The sensor authenticates the user by communicating with the personal agent, both parties verify each other's authority, and the biometric sensor sends authentication

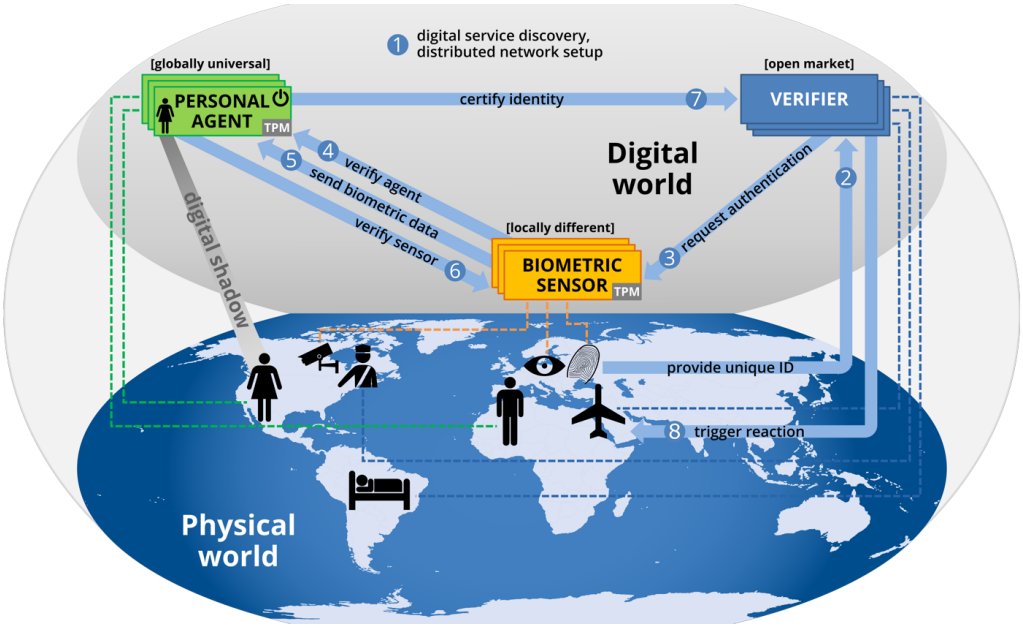

Figure 2.1: Overview of Digidow's system design [2]

proof to the personal agent. The personal agent then certifies the identity of the user and requests the verifier to grant access for that particular person.

2.2 The Face sensor environment and design

The face sensor is the central part of the software that runs on the Raspberry Pi. It is responsible for detecting and recognizing faces captured by the USB camera. It uses the following technologies:

1. Rust - A memory safe programming language that prevents common memory safety issues like: Buffer Overflows, Memory leaks, double frees, etc.
2. Tor - A free open source network technology used to encrypt and anonymize TCP Communication within a network to enhance security and deter network attacks.
3. Coral USB Accelerator - An accelerator (TPU) used for speeding up artificial intelligence models. It is connected using the

USB ports on the Raspberry Pi. Coral is a technology that allows developers to create customized AI models for their operations.

4. Nix - A tool that allows developers to create reproducible software environments through declarative file syntax.
5. Integrity Measurement Architecture - IMA is a subsystem in the Linux kernel that can be enabled to calculate hashes and appraise files before read/write system calls.
6. Docker - A tool that uses virtualization to package software environments into isolated containers which only interact through specified ports and volumes. Docker containers are based on images that are built with Dockerfiles.
7. Gstreamer - An open source library for audio/video processing. The Gstreamer pipeline refers to the set of sequential modifications that are done to the video feed.

The software is written in Rust, and all communication is done through a Tor service. The face sensor binary is a self-extracting binary that contains all its dependencies (except drivers and runtimes). The binary is created with Nix using cross-compilation tools to build for the Raspberry Pi's target architecture (aarch64). The USB Coral Accelerator is used to speed up the face recognition model. The created binary first extracts all dependencies at runtime and reads the supplied settings file. Afterwards, the GStreamer pipeline and communication tools are set up, and the detection/recognition loop begins.

Chapter 3

Porting the face sensor software

To test out these different approaches, a reliable way of reproducing the same software environment is needed. For this purpose, the face sensor uses Nix to create an isolated Nix Bundle. This circumvents the typical `nix-copy` procedure, which can be the source of many errors (`nix-copy` allows users to copy programs and their dependencies between two Nix environments). Due to the size of the sensor software, a Cross-Compilation approach was chosen. The compilation was done on a Lenovo Legion Laptop using Arch Linux. A description of the compilation and porting process will be presented.

To compile the face sensor binary both Nix and QEMU are required. QEMU is an open-source emulator and virtualizer for other platforms. For the purposes of cross compilation, qemu-user-static-binfmt is also needed. It adds and adjusts binary rules for qemu static user mode execution.

Additionally, Nix requires extra configuration for building binaries targeted at different platforms and new features like flakes and subcommands. Flakes allow for a more resilient and deterministic way of building programs. They consist of `flake.nix` and `flake.lock` file. The `flake.nix` file is a description of how inputs (dependencies) are used to build outputs (programs). `flake.lock` records exact versions of dependencies and is created when exe-

cuting any Nix command that requires building the flake. The face sensor program uses such a setup to produce outputs for different architectures (x86, aarch64, riscv). Listing 3.1 shows how to set up the Nix building environment. First Nix and qemu are installed and afterwards the `extra-platforms` settings is specified in `/etc/nix.conf`.

```
# Install nix
sh <(curl --proto '=https' --tlsv1.2 -L https://nixos.org/nix/install) --daemon
# Install the qemu packages
sudo pacman -S qemu-full qemu-user-static qemu-user-static-binfmt
# Check if the binfmt has been registered and if not restart the binfmt
service
if [[ ! -e "/proc/sys/fs/binfmt_misc/qemu-aarch64"]]; then
sudo systemctl restart systemd-binfmt.service
fi
# add required nix config
sudo echo "trusted-users = ellie" >> /etc/nix/nix.conf
sudo echo "extra-platforms = aarch64-linux" >> /etc/nix/nix.conf
sudo echo "extra-experimental-features = flakes nix-command" >> /etc/nix/nix.
conf
```

Listing 3.1: Script to install components for cross-compilation

To confirm that the system is able to cross-compile to the aarch64 platform `arch-test` can be installed. It can be used to easily check if a system supports building for different architectures. If aarch64 is listed when executing `arch-test` the host system is able to cross-compile for the Raspberry Pi 4/5.

Before the system is ready to start the Nix build process, one last issue needs to be addressed. During the build process Nix will pull various dependencies from within the CDL Digidow project. By default the Git pull request is set up to use Secure Shell (SSH) instead of an HyperText Transer Protocol Secure (HTTPS) pull. This causes the repository to reject the request due to a missing public key that it is not privy to. To remedy this issue the section shown in Listing 3.2 needs to be added to the local `.gitconfig`. It specifies to use HTTPS instead of SSH when pulling from the Institute for Networks and Security (INS) institute Git server. Hyper Text Trans-

fer Protocol is a network communication protocol for sending and receiving files through requests. HyperText Transfer Protocol Secure is HTTP which uses TLS for advanced security.

```
1     [url "https://git.ins.jku.at"]
2     insteadOf = ssh://git@git.ins.jku.at
```

Listing 3.2: Contents of .gitconfig

The system is now ready to start the Nix bundle process. Nix Bundles are self-extracting executables that can work in an environment where Nix is not installed. A specific bundler can be specified when creating a Nix bundle to target a specific architecture or platform (e.g, Docker). After creating the Nix bundle, it can be copied to the target system either by using scp (SSH copy) or copying it directly onto the SD card of the Raspberry Pi. As of March 2026, the Nix bundle utility is broken due to an update in arx, which `nix-bundle` depends on. To alleviate this problem, a specific patch of `nix-pkgs`[1] needs to be used. nix-pkgs is the collection of packages that can be installed by the Nix utility. The bundle can be built using: `nix bundle --bundler github:NixOS/bundlers/dce9249b74500ef75110153e6da455ffec2532ba#bundlers.aarch64-linux.default --refresh ".#aarch64"`. This command (described in this GitHub issue[2]) uses a specific version of nixpkg's to downgrade the version of arx and remedy the breaking issue.

It is important to mention a minor difference to the currently running setup on the JKU Campus, before examining the different software environments. All of the testing was done on a Raspberry Pi 4 Model B in comparison to the previously named system, which uses a Raspberry Pi 5. The major differences come down to a few factors:

1. CPU - The Raspberry PI 4 has a Cortex A72 @ 1.5GHz processor which implements the armv8-A architecture. The Rasp-

[1] https://github.com/nixos/nixpkgs
[2] https://github.com/NixOS/bundlers/issues/27

berry Pi 5 has an ARM Cortex A76-CPU @ 2.4GHz, which implements the ARMv8.2-A Architecture. This difference gives the Raspberry Pi 5 a vast increase in performance, as can be seen in [3]. Here, the Raspberry Pi 5 achieves a performance increase of about 200-300%. This improvement does not require a different processor architecture, which would require different versions for the Raspberry Pi 4 and Raspberry Pi 5.

2. The microSD card slot supports high-speed SDR104 mode for doubling the read/write speeds.
3. The Raspberry Pi also includes other improvements like a power button, higher quality video feeds (4k60), PCIE support, Real time clock and a Fan connector, among many others.

The Raspberry Pi 5 overall is vastly more powerful than the Raspberry Pi 4 Model B. Notably the Cortex A76 Processor improves specific mathematical operations needed for Digidow's use case of facial recognition and AI model execution. However, there aren't any fundamental changes that would impact the environment analysis. The relative performance changes between environments should stay the same, regardless of target hardware. This is not guaranteed and may be a topic of further study in the future.

It should be noted that the face sensor program has the ability to be tested on a specific video. All of the environment setups are, however, done with a USB camera to better replicate the actual environment of the face sensor. The USB camera that was used is a Logitech C922 Pro Webcam. It delivers a video feed of 1080p at 60 frames per second, which is enough for testing purposes.

Chapter 4

Software environments subject to analysis

This next section presents the different environments that will be subject to analysis in chapter 5. Each section represents a different software environment on a different operating system and/or different kernel. In each section, an overview of the system is first presented, followed by an explanation of why this environment was chosen as a potential candidate for further analysis. Then, the process of setting up the system and porting the bundle to the target system is explained.

4.1 Raspberry Pi OS Lite

Raspberry Pi OS is the official and most widely supported operating system for the Raspberry Pi. It is based on the Debian distribution but differs in some factors from a normal Debian setup. The current version is based on Debian Trixie with a customized kernel including drivers for the GPIO pins on the board, a serial console, and various other optimizations for the single-board computer. Generally, any packages used on Debian also work on Raspberry Pi OS.

Raspberry Pi Lite was chosen as a baseline on which to compare other environments, because it is the currently used setup in the 'face recognition on JKU Campus' [2] project. In comparison to the base Raspberry Pi OS, it ships with significantly fewer pre-installed applications and without a desktop environment. Due to the familiarity of the system this was the first environment to be set up as a means of testing the porting process.

Like some of the other environments, Raspberry Pi OS can be easily flashed onto an SD card with the use of the Raspberry Pi Imager. The Imager is the official tool provided by the creators of Raspberry Pi. It allows users to easily flash an operating system environment onto an SD Card. It also enables the user to customize the system before it is booted. Customization options include: user and wifi setup, static IP configuration, and SSH daemon configuration. The Imager also automatically chooses appropriate firmware based on the model selected in the GUI.

Before the Nix bundle can be executed, a few issues need to be addressed:

1. TPU Edge Runtime
2. Sensor configuration files
3. USB camera setup
4. Tor server setup

The Coral USB Accelerator provides the Coral Edge Tensor Processing Unit (TPU), which vastly enhances the performance of specific mathematical operations used in AI models. For the TPU to be able to speed up the processing of AI models, the TPU Edge Runtime needs to be installed as well as the PyCoral Library with a Python version ranging between Python 3.6-3.9[1] . It should also be noted that the Coral Edge TPU only works with TensorFlow Lite models and is no longer being maintained, with its last update

[0]This is only technically accurate and will be revisited later in section 4.3

being targeted at TensorFlow Lite 2.16.1, which was released in March of 2024.

Because the Debian ecosystem is by far the most popular Linux ecosystem, Google Coral provides a package ready to be installed after configuring the repository. The installation process is described by the official Coral documentation [7]. Since the coral project is no longer being maintained, the documentation is no longer applicable. It still uses `apt-key`, which is deprecated due to a security vulnerability where gpg signatures were not being checked thoroughly [11]. Vanilla Debian itself was not affected, but downstream distributions like Raspberry Pi OS were [8]. Even though Debian was not affected, apt-key was later removed in Debian 11. The new way to add distributions is by specifying the key directly in the sources list. Listing 4.1 presents the script used to register the package sources and install the TPU Edge runtime. To register the package source the gpg file needs to be downloaded and referenced in the APT sources list with `signed-by`. Advanced Package Tool (APT) is the package manager of debian.

```
1 # Get gpg key and register archive
2 wget -qO - https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo gpg --
  dearmor -o /usr/share/keyrings/google-coral-archive-keyring.gpg
3 sudo echo "deb [signed-by=/usr/share/keyrings/google-coral-archive-keyring.gpg
  ] https://packages.cloud.google.com/apt coral-edgetpu-stable main" > /etc/apt/
  sources.list.d/google-coral.list
4 # Update the package repo lists and install the edge tpu runtime
5 sudo apt update
6 sudo apt install libedgetpu1-std
```

Listing 4.1: Bash Script to install Edge TPU runtime

Another requirement[1] is the PyCoral API for TensorFlow models. As PyCoral only supports 3.6 to 3.9, the Python version provided by a default installation of Raspberry Pi OS needs to be downgraded to 3.9. To install the needed Python version, Pyenv was

[1] Not a "requirement" see section 4.3

used. Pyenv is a tool that can manage multiple different Python versions on the same system. Pyenv can set which Python versions other programs/shells should use, execute a command using a specific Python version, and manage virtual Python environments with specific versions. For the purposes of this thesis, it is used to install Python 3.9 and the pyCoral wheel files.

To install Pyenv, all native Python build dependencies need to be installed, because Pyenv builds Python versions from scratch. After Pyenv is installed through the official shell script, some environment variables need to be set up (`$PYENV_ROOT`). Because global Python packages on Debian are installed using apt, APT cannot be used to install an old version of `python3-pycoral` without downgrading all other Python packages and dependencies and possibly breaking the entire system. Instead, PyCoral is installed using the pip wheel files provided by the Google coral github page. Python wheels are prebuilt modules and package files that can be installed by the python package manager pip.

```
# Install Pyenv
sudo apt update
sudo apt install make build-essential libssl-dev zlib1g-dev libbz2-dev
libreadline-dev libsqlite3-dev curl git libncursesw5-dev xz-utils tk-dev
libxml2-dev libxmlsec1-dev libffi-dev liblzma-dev
curl -fsSL https://pyenv.run | bash
# Setup bashrc for Pyenv
echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.bashrc
echo '[[ -d $PYENV_ROOT/bin ]] && export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.
bashrc
echo 'eval "$(pyenv init - bash)"' >> ~/.bashrc
# reload shell and setup correct pyenv version
exec $SHELL
pyenv install 3.9.21
pyenv global 3.9.21
# install tflite wheel and pycoral wheel file
sudo apt install python3-pip python3-wheel python3-setuptools
pip install https://github.com/google-coral/pycoral/releases/download/v2.0.0/
tflite_runtime-2.5.0.post1-cp39-cp39-linux_aarch64.whl
pip install https://github.com/google-coral/pycoral/releases/download/v2.0.0/
pycoral-2.0.0-cp39-cp39-linux_aarch64.whl
```

Listing 4.2: Bash script to install PyCoral and Python 3.9.21

During testing of other software environments (e.g., Alpine Linux and PiCore) it was discovered that the PyCoral API is not a requirement to run the Nix bundle, as the Tensorflow lite runtime is already included as a dependency in the Nix bundle. So this step can be ignored when replicating the environment.

The Nix bundle depends on a set of static files that need to be provided to the program through a configuration file specified by `--settings`. These files include configuration files but also AI models used for face detection. It should be noted that the program can only handle global paths and does not work when using relative file paths. The lines that need to be changed in the configuration file can be seen in Listing 4.3.

Next, the camera needs to be set up correctly. This includes:

1. Ensuring the user is allowed to read from `/dev/video0`.
2. Setting the correct resolution and orientation in the settings file.
3. Confirming the settings through `ffmpeg` or other utility like `v4l2-ctl`. To check if the camera is being recognized, execute `v4l2-ctl -A` or check dmesg for driver problems.

The setup of the Tor server also produces an error if the `/home` directory is not owned by the executing user. This error is not fatal. To alleviate this issue and prevent errors/warnings, change ownership of `/home` to the executing user with `sudo chown {user} /home`.

```
{...
    "fastdet_path": "/home/ellie/sensor/static/detection
↪ /fastdet_640.onnx",
    "retinaface_path": "/home/ellie/sensor/static/
↪ detection/retinaface-150x150_edgetpu.tflite",
    "retinaface_anchors": "/home/ellie/sensor/static/
↪ detection/retinaface_anchors-150x150.json",
    "arcface_path": "/home/ellie/sensor/static/
↪ recognition/arcface_edgetpu.tflite",
    "width": 1080,
    "height": 1920,
    "name": "
↪ djsmen243lqmqh4aj7ydawhnujeweqa3eedkmazjx6cn5hqb2chz7aid
↪ ",
    "lat": 48.335318,
    "lng": 14.323952,
    "local_port": 7999,
    "max_distance_closest_poi": 3000.0,
    "raw_image_storage_path": null,
    "gstreamer": "v4l2src device=/dev/video0 ! image/
↪ jpeg,width=1920,height=1080 ! jpegdec ! videoconvert !
↪  appsink max-buffers=1 drop=True",
    "default_registration_mins": 60,
    "image_rotation": 180,
    ...
}
```

Listing 4.3: Important configurations in settings.json

Now the Nix bundle can be executed. The Nix bundle can be executed using this command `RUST_LOG=DEBUG ./sensor-arx --settings /home/ellie/sensor_run/settings.json`. After the Nix bundle extracts all of the dependencies, it will start the Gstreamer pipeline and try to detect faces that the USB camera sees. If the camera rotation/alignment is of,f it can be set in the config file with `image_rotation` (only 90, 180, 270 and 0 are allowed). The face detection now works.

4.2 RedoxOS

Redox OS is an operating system written in Rust or unsafe Rust (where possible). Unsafe Rust refers to Rust that is not guaranteed by the borrow checker. This allows access to raw pointers, which are sometimes needed in low-level development. The OS is Unix-like and follows some similar design philosophies to Linux and BSD. Its goal is to one day be a more secure and reliable alternative. In comparison to the Linux kernel, it follows a microkernel architecture and all of its drivers are run in userspace. RedoxOS is still in the early stages of development and has a lot of issues with supporting different types of hardware and architectures. Nevertheless, the advantages RedoxOS could provide in theory are promising for the face sensor's use case. Security is the top priority of Digidow at all times. This is also the reason why the face sensor is written in Rust.

The major problem in setting up the RedoxOS environment is that all of the fundamental groundwork that the kernel typically provides is different in RedoxOS. This means that even minor differences in system calling conventions or the standard library could cause problems. Another issue is the lack of online documentation for setting up lesser-known architectures like the Raspberry Pi. Even though Nix bundles can execute across different environments and architectures, they are mostly meant for reproducing

```
# Install podman container and debian container
mkdir -p redoxOS
cd redoxOS
curl -sf https://gitlab.redox-os.org/redox-os/redox/raw/master/podman\
_bootstrap.sh -o podman\_bootstrap.sh
bash -e podman_bootstrap.sh
# Update PATH
source ~/.cargo/env
cd redox
# Set configuration
echo "PODMAN_BUILD=1" >> .config
echo "ARCH=aarch64" >> .config
echo "CONFIG_NAME=minimal" >> .config
echo "FILESYSTEM_SIZE=2048"
make all
```

Listing 4.4: Creating the RedoxOS image

stable environments between Linux distributions, but not across different operating system boundaries.

Even though the factors described above pose massive potential problems, a setup of RedoxOS was attempted. RedoxOS does not provide plug-and-play images, which is why self-compilation is needed. RedoxOS uses Podman to create a replicable environment during the build process. Podman is a container manager like Docker, which can execute Linux images. After the necessary building requirements are installed through Podman, the desired configuration needs to be established. A `.config` file in the root directory of RedoxOS will overwrite any configuration entries in `mk/config`. RedoxOS has a variety of different settings that can be adjusted, but it also provides configuration presets that can be used. Afterwards, the hard disk target is compiled with make. Listing 4.4 describes how the RedoxOS image can be compiled. First the Podman environment is created and the `.config` file is modified to ensure a minimal system targeting the aarch64 architecture is created.

The built image is located at `build/aarch64/minimal/harddrive.img` and needs to be copied to the SD card. The problem is that the normal BIOS bootloader that RedoxOS uses has problems when interacting with the Raspberry Pi firmware. However, this issue can be avoided by using the Raspberry Pi port of the Tianocore EDK2 UEFI firmware. TianoCore is a cross-platform firmware development environment that allows developers to create firmware for almost any system.

To understand how and why this works, it is important to first understand the difference between the BIOS and UEFI boot processes. BIOS and UEFI refer to the basic firmware that resides within non-volatile memory. BIOS is a legacy system that was established within IBM PCs as a standard together with MBR. This is how the boot process on a system with BIOS firmware works: The firmware looks into the first 512 bytes of the Master Boot Record table, which contains the bootloader (and description of partitions). The firmware then executes the bootloader, which is responsible for starting the operating system. This design is limiting because modern OS bootloaders require more space than 512 bytes, and the bootloader always needs to reside within the first 512 bytes, which means they cannot be swapped out easily. UEFI solves this problem by requiring all firmware following its specification to be able to read FAT12, FAT16, and FAT32 partitions and their contents. The bootloaders are then contained within .efi files, which can be any size up to the limitations of FAT. The EFI boot manager then determines which boot entries (.efi Files) are executed. However, UEFI systems can also boot from partitions utilizing an MBR layout.

RedoxOS can boot in 2 different ways: BIOS or UEFI. However, UEFI booting is only supported on x86-64, ARM64 (aarch64), and RISC-V 64-bit machines. The BIOS bootloader is separated into 3 stages, with only the last stage being written in Rust. When compiling the RedoxOS image, both the BIOS and UEFI bootloaders are

included. However, because the boot sector does not exist within the GPT partition layout, a BIOS partition is needed so that systems that still have BIOS firmware can boot. The UEFI bootloader is included in `BOOT/EFI/BOOTAA64.EFI` and is the entry point which will be used to boot RedoxOS. Listing 4.5 shows how to boot RedoxOS on the Raspberry Pi. The image is flashed onto the SD card with the `dd` utility and the EFI partition is increased to hold the bootloader and the TianoCore UEFI Firmware. Then the TianoCore firmware is copied onto the EFI partition.

```
1  # Setup SD card (replace sdX with corresponding device identifier)
2  sudo dd bs=4M status=progress if=build/aarch64/minimal/harddrive.img of=/dev/
   sdX
3  # Enter gparted setup, delete BIOS partition, resize EFI Partition to be 512MB
     and move REDOX partition accordingly
4  sudo gparted /dev/sdX
5  # Gather TianoCore UEFI firmware for raspberryPi and put it on the EFI
   partition
6  curl -o UEFI_firmware.zip https://github.com/pftf/RPi4/releases/download/v1
   .50/RPi4\_UEFI\_Firmware\_v1.50.zip
7  sudo mkdir -p /mnt/efi_boot
8  sudo mount /dev/sdX2 /mnt/efi_boot
9  sudo cp UEFI_firmware.zip /mnt/efi_boot/
10 cp /mnt/efi_boot/
11 sudo unzip UEFI_firmware.zip
12 sudo rm UEFI_firmware.zip Readme.md
```

Listing 4.5: Preparing boot process for RedoxOS

The SD card is now ready and should be able to boot. In the boot setup menu a manual entry for `BOOTAA64.EFI` needs to be added. It can be done through (ESC) Setup → Boot Maintenance Manager → Boot Options → Add Boot Options → SD Card identifier → EFI → BOOT → BOOTAA64.EFI. The system then boots and is able to get through the first initialization stages. RedoxFS then encountered an unexplainable error, which would have needed more investigation. However, after talking to notable contributors to the RedoxOS Project, the process of getting RedoxOS running for this

purpose was deemed not to be within the scope of this thesis. It should be noted, however, that it is possible to get RedoxOS to post to the login screen according to [12].

4.3 Alpine Linux

Alpine Linux is a lightweight Linux distribution that focuses heavily on security. It uses Busybox and Musl libc to be as small as possible. Busybox provides a collection of basic GNU utilities in a single program. Musl libc is an implementation of the C standard library, which has vast static linking support [2]. Musl is focused on being lightweight, fast, secure, and portable. Due to the static linking support, binaries compiled with musl can be deployed in almost any environment. Static linking here refers to building the required libraries directly into the binary. Alpine Linux uses the OpenRC init system for a more modular and faster init system and has its own package manager: Alpine Package Keeper (APK). This distribution excels in terms of security because all of its userspace binaries are compiled with heavy hardening protections like PIE, read-only relocation table, Stackguard, etc.

All of these factors make Alpine Linux a perfect candidate for the face sensor environment. Due to musl binaries also being able to be compiled statically, it makes all binaries created with Musl portable and easy to use in other environments. In comparison to other environments, Alpine Linux promises to be a more lightweight, secure, and modular alternative. It is more difficult to insitall the Edge TPU runtime, because Alpine Linux is a different ecosystem with a different package manager, init system, and standard library.

The installation of Alpine Linux is similar to Raspberry Pi OS. The Raspberry Pi Imager offers an image of Alpine Linux under Gen-

[2] This is important. See section 4.4

eral Purpose OS → Alpine Linux. In comparison to the Raspberry Pi OS, which comes with a pre-configured user, SSH setup, etc. Alpine Linux offers setup utilities through `setup-alpine`. The setup enables the user to easily: set locales and keyboard layout, hostnames, timezone, SSH, disk setup, Network Time Protocol (NTP), and APK caching and mirrors. Network Time Protocol is a protocol for syncing date and time using reliable NTP servers. Alpine Linux also provides separate setup utilities for each of the configuration steps. This is especially useful if the only available network connection is wireless, because the automatic setup will overwrite the interface and networking setup otherwise. To set up wireless network access, use `setup-interfaces` and restart the networking service afterwards through rc-service networking restart. Alternatively, the same effect can be achieved by manually editing `/etc/network/interfaces` and `/etc/resolv.conf`. DNS configuration is specified in `/etc/resolv.conf` and interface setup in `/etc/network/interfaces`. While both wired and wireless configurations typically work out of the box, complications may sometimes be encountered.

Alpine Linux does not create a user after a basic installation or install any privilege escalation utilities. However, both `sudo` and `doas` can be installed to fit this purpose. Alpine Linux recommends doas as it is smaller, but still covers almost all of `sudo`'s use cases. The difference in size also means that doas has less attack surface. Listing 4.6 shows how to achieve a working `doas` setup with a given user.

```
1  # Add a user (replace ellie with appropriate user)
2  adduser -G wheel -m -h /home/{user} -s /bin/bash {user}
3  su -l root
4  apk add doas
5  echo "permit :wheel" > /etc/doas.d/doas.conf
6  su -l {user}
```

Listing 4.6: Bash script to setup user privilege escalation

The package distributed by Google Coral cannot be used, because

Alpine Linux is fundamentally different from Debian. Therefore, the TPU edge runtime (also referred to as libedgetpu) needs to be compiled manually. The method used to compile the TPU edge runtime are outdated, because libedgetpu repository has been archived since October of 2024. The repository describes three possible ways to build the library: Docker with Bazel, Bazel, and native. However, all of these processes no longer work as described. Bazel is a build tool similar to `make`, which Google Coral uses to build its runtime.

Before the repair instructions are illustrated below, it should be noted that all of these modifications have been discovered through testing and inspiration from the Arch user repository package libedgetpu-git [3].

The Docker + Bazel build process is examined first. In total, 3 files need to be modified. These modifications are needed for updates to `abseil-cpp-dev`, `gcc`, and `flatbuffers`. 4.3 shows the needed modifications. - and + symbols signify the removal and addition of lines. Numbers following @ indicate the line at which this modification can be found within the file. The modifications tell make to build with a newer standard of c++, include needed .cc files and apply required flags for `abseil-cpp`.

These modifications are sufficient for the Docker build process to work. `libedgetpu.so.1` is compiled with: `DOCKER_CPUS=aarch64 DOCKER_IMAGE ="debian:bookworm" DOCKER_TARGETS=libedgetpu make docker-build`. This command instructs Make to cross-compile libedgetpu on Debian Bookworm for use on the aarch64 architecture (armv8). Even though the binary works and is ready to be copied to /usr/lib on a target machine, it won't work for Alpine Linux. This is because Alpine uses Musl Libc, whereas Debian relies on Glibc. This necessitates building the library in an environment that uses Musl Libc.

The second possible way to build libedgetpu is through Bazel na-

[3] https://aur.archlinux.org/packages/libedgetpu-git

```
1         # Makefile changes in makefile_build/Makefile
2         @ 22
3         LIBEDGETPU_CXXFlAGS := \
4         -fPIC \
5         -fWall \
6         -   -std=c++14 \
7         +   -std=c++17 \
8         ...
9         @ 28
10        LIBEDGETPU_LDFLAGS := \
11        -Wl, -Map=$(BUILDDIR)/output.map \
12        -shared \
13        -Wl, --soname, libedgetpu.so.1 \
14        -Wl,--version-script=$(BUILDROOT)/tflite/public/libedgetpu.lds \
15        -fuse-ld=gold \
16        -lflatbuffers \
17        -   -labsl_flags \
18        +   -labsl_flags_usage \
19        ...
20
21        @ 62
22        - LIBEDGETPU_CSRCS := $(TFROOT)/tensorflow/lite/c/common.c
23        + # LIBEDGETPU_CSRCS := $(TFROOT)/tensorflow/lite/c/common.c
24
25        @ 142
26        $(BUILDROOT)/tflite/edgetpu_delegate_for_custom_op_tflite_plugin.cc \
27        -   $(TFROOT)/tensorflow/lite/util.cc
28        +   $(TFROOT)/tensorflow/lite/util.cc \
29        +   $(TFROOT)/tensorflow/lite/core/c/common.cc \
30        +   $(TFROOT)/tensorflow/lite/array.cc
31        LIBEDGETPU_CCOBJS := $(call TOBUILDDIR,$(patsubst %.cc,%.o,$(
    LIBEDGETPU_CCSRCS)))
32
```

Listing 4.7: File modifications for makefile

```
1         @ 18
2         +#include<cstdint>
3         +
4         #include "port/array_slice.h"
5         #include "port/integral_types.h"
6
```

Listing 4.8: File modifications for usb_device_interface.h

tively. However, the Bazel setup is broken, due to package updates messing with declared dependencies, without any way to remedy the issue without diving deeper into the Bazel setup. This may be an area for possible further research.

The last available build process is building natively on Alpine Linux. Building natively means building the runtime on an system with the same architecture. Due to the fact that Musl has wide static linking support, the runtime can be used on any Linux environment, regardless of the standard library it employs. The build process requires the same modifications mentioned in the Docker build, as well as other modifications that are shown below. The requirements to build libedgetpu are: `makefile` and `usb_device_interface.h` modifications, `flatbuffers` with version <= 23, `libusb binutils` and `abseil-cpp-dev` installed, and a tensorflow directory with a matching commit corresponding to `workspace.bzl`.

To make future installation of libedgetpu easier, a Docker build process was created. Docker uses Dockerfiles to describe how an image is built. This image provides an environment in which the edge TPU runtime can be compiled. Listing 4.9 shows the Dockerfile used to build the image.

The Dockerfile builds on top of the basic Alpine Docker image. Build arguments are used to determine what architecture is targeted and what version of the Docker image is used. When building a Dockerfile, a specific image of Alpine can be supplied with build arguments with `docker build --build-arg VAR=value .` . If no build arguments are supplied, the Dockerfile will create an image targeting the aarch64(armv8) architecture with the latest Alpine release. The image creates an environment in which all of the libraries, dependencies, and configuration changes needed to compile the runtime are installed.

To understand how the Dockerfile works, an understanding of

```
1  ARG DOCKER_PLATFORM=arm64v8
2  ARG TARGET_ARCH=aarch64
3  ARG TENSORFLOW_COMMIT=5bc9d26649cca274750ad3625bd93422617eed4b
4
5  FROM ${DOCKER_PLATFORM}/alpine:latest
6  RUN apk add doas bash git abuild \
7  && adduser -G abuild -h /home/builder -s /bin/bash -D builder \
8  && echo ``permit nopass builder'' > /etc/doas.conf
9  USER builder
10 WORKDIR /home/builder
11 RUN doas apk add alpine-sdk
12 RUN abuild-keygen -ain \
13 && git clone -b 3.21-stable --depth=1 https://gitlab.alpinelinux.org/alpine/
   aports.git \
14 && cd aports/main/py3-setuptools/ \
15 && abuild -r \
16 && doas apk add \$HOME/packages/main/$TARGET_ARCH/py3-setuptools*.apk
17 WORKDIR /home/builder/aports/community/flatbuffers
18 RUN git remote set-branches --add origin 3.19-stable \
19 && git fetch --depth=1 origin 3.19-stable \
20 && git checkout -b 3.19-stable origin/3.19-stable \
21 && abuild -r \&\& cd \$HOME/packages/community/$TARGET_ARCH/ \
22 && doas apk add flatc*.apk flatbuffers*.apk
23 WORKDIR /home/builder
24 RUN git clone https://github.com/google-coral/libedgetpu \
25 && cd libedgetpu \
26 && git clone https://github.com/tensorflow/tensorflow \
27 && cd tensorflow \
28 && git checkout $TENSORFLOW_COMMIT
29 WORKDIR /home/builder/libedgetpu
30 COPY Makefile makefile\_build/Makefile
31 COPY usb\_device\_interface.h driver/usb/usb\_device\_interface.h
32 RUN doas apk add make g++ bash abseil-cpp-dev libusb-dev binutils-gold
```

Listing 4.9: Dockerfile for building the image

package building process on Alpine is required. Alpine Linux uses `APKBUILD`, which is similar to Arch Linux's `PKGBUILD`. It contains build instructions, dependencies, checksums, and package metadata that `abuild` uses to build the package file `\${pkgname}-\${version}.apk`. The built package file can then be installed with `doas apk add ${pkgname}-${version}.apk`. In order to build packages with `abuild`, the executing user has to be part of the abuild group and have a private/public key setup in `~/.abuild/`. This keypair can be generated with `abuild-keygen -ai`. All official packages for Alpine Linux are available through the aports repository. To install a package with a specific version, the following steps are necessary:

1. Clone the aports git repository
2. Using the Alpine Package index search for the wanted version and architecture using the filters.
   - For example if flatbuffers version <= 23 is to be installed the v3.19 branch or older is needed.
3. Checkout to desired branch and navigate to package folder (Note: aports is divided into main, community and testing. This means the package path is: `${REPO_NAME}/${PKG_NAME}`)
4. Build the package with `abuild -r`
5. The built .apk files are then found in `~/packages/${ARCHITECTURE}/${PKG_NAME}-${PKG_VERSION}` and are added with `doas apk add ${PKG_NAME-${PKG_version}}.apk`

The Dockerfile creates the `abuild` setup because the edge TPU runtime can no longer be compiled with `flatbuffers` version >= 23. Flatbuffers, in turn, requires `py3-setuptools` with version < 78 (Due to a structural change of how py3-setuptools relative addressing works [9]). First `py3-setuptools` is compiled and installed. Afterwards `flatbuffers` is compiled and installed together with `flatc` (abuild generates both). Then the correct modifications to the runtime build sources are applied, and the correct TensorFlow version is in-

stalled. The static commit corresponds to TensorFlow version `2.16.1`. The Dockerfile only provides the environment; for the compilation process, see Listing 4.10. The image is first built based on the provided Dockerfile and the container is run with the specified image and platform. Afterwards the native build command is issued and afterwards the result is copied out of the container.

The multi-architecture support in QEMU needs to be enabled, because the Dockerfile builds an image for the aarch64 architecture. This can be done with the following command: `sudo docker run --rm --privileged multiarch/qemu-user-static --reset -p yes --credential yes`. By specifying `--credential yes`, utilities (like Sudo or Doas) can use suid (This is a requirement).

```
1  docker build -t libedgetpu .
2  docker run --platform=linux/arm64/v8 -it --name edgetpu_compiler libedgetpu /
   bin/bash
3  TFROOT=tensorflow-dir/ make -f makefile_build/Makefile -j$(nproc) libedgetpu
4  # in a new shell use docker to copy the libedgetpu outside of the container
5  docker cp edgetpu_compiler:/home/builder/libedgetpu/out/direct/aarch64/
   libedgetpu* .
```

Listing 4.10: Commands to compile libedgetpu

Afterwards, the runtime library needs to be copied to `/usr/lib` on the target system. To run the face sensor program, the Nix bundle is copied together with the settings.json and static files from the Raspberry Pi OS setup. Then the camera is verified with v4l2-utils/ffmpeg, and the Tor server issue is remedied as described in section 4.1. If the created username deviates from the raspberry Pi OS username, file paths in `settings.json` need to be adapted. Afterwards, the Nix bundle can be executed with `RUST_LOG=debug ./sensor-arx --settings /home/ellie/sensor_run/settings.json`.

4.4 PiCore

PiCore is the Raspberry Pi port of TinyCore. TinyCore is a Linux distribution that focuses on extreme modularity and minimalism. TinyCore is designed to be a minimal system shipped with just the bare necessities. This allows TinyCore to be run on any system capable of holding the incredibly small image (10MB). Its default operation mode is diskless (cloud mode), which means any changes made to the system are not permanent. In diskless mode it runs completely from RAM, only taking up 46MB while idle. Due to its size, TinyCore is restricted in the amount of utilities and programs it offers out of the box. Basic GNU utilities from Busybox are included, but most other tools need to be installed. This is what TinyCore refers to as an extension. Extensions are typically installed into RAM and are only saved if the user tells the system to do so. Otherwise, TinyCore will always revert to a default clean install after a shutdown. Options to save changes and automatically load extensions are included but require some setup. TinyCore also has its own package manager and uses BusyBox and the BusyBox init system to save space.

TinyCore is the best choice for a system that takes up minimal space without getting into Linux-from-scratch territory. With this minimal setup, TinyCore should also provide some of the fastest boot times. A smaller image also reduces the possible attack surface. Due to TinyCore's size, any additional security enhancements would need to be done through extensions and heavy manual configuration.

TinyCore provides official images for the Raspberry Pi, which can be flashed onto the SD card with `dd bs=4M if=${img.tar.gz} of=/dev/sdX status=progress`. The image ships with the main partition only being 16MB large in comparison to the 80MB that the EFI partition takes up. This is because 10MB is enough for the entire TinyCore system to reside on the partition and operate in diskless mode after

booting. To make space for the Nix bundle, expanding/recreating the partition is necessary to fit the unoccupied space. This can be done easily with `fdisk /dev/sdX`. Afterwards, the filesystem needs to be resized to match boundaries with `e2fsck` and `resize2fs`.

Before diving into extra configuration, it is important to further understand a central concept of TinyCore: FileBackups and bootcodes. As mentioned earlier, most changes made on a running tc system are not persistent and need to be backed up using the `filetool.sh` utility. The filetool utility allows the user to restore old files or create backups when needed. **Any changes that are not stored in a backup at shutdown will be lost permanently if a diskless setup is being used!** The filetool reads `/opt/.filetool.st` to decide which files to backup into `mydata.tgz`.

The base TinyCore image only provides enough utilities to install extensions from wired internet access. If a wired setup is not possible, a lot more configuration is needed to install additional extensions. There is little up to date documentation for this issue, but [5] shows how to set it up. Listing 4.11 shows how to download the drivers for wifi and management tools on a host machine with an internet connection. `.tcz` file is the compiled extension file, which can be loaded. `.dep` names dependencies if any exist, and `.md5.txt` is required for checking the extension hash. The script downloads all required extensions from a mirror of TinyCore using `wget`.

```
1     echo "ca-certificates.tcz
2     ca-certificates.tcz.md5.txt
3     libnl.tcz
4     libnl.tcz.md5.txt
5     firmware-rpi-wifi.tcz
6     firmware-rpi-wifi.tcz.md5.txt
7     ncurses.tcz
8     ncurses.tcz.md5.txt
9     readline.tcz
10    readline.tcz.dep
11    readline.tcz.md5.txt
```

```
12      wifi.tcz
13      wifi.tcz.dep
14      wifi.tcz.md5.txt
15      wireless_tools.tcz
16      wireless_tools.tcz.md5.txt
17      wireless-6.12.25-piCore-v8.tcz
18      wireless-6.12.25-piCore-v8.tcz.md5.txt
19      wpa_supplicant.tcz
20      wpa_supplicant.tcz.dep
21      wpa_supplicant.tcz.md5.txt" > extensions.lst
22      cat extensions.lst | (while read -r line; do wget https://mirror.cpsc.ucalgary
        .ca/mirror/tinycorelinux/16.x/aarch64/tcz/$line; done)
```

Listing 4.11: Shell script to download extensions and their dependencies

To install the extensions, move the downloaded files to `/mnt/mmcblk0p2/tce/optional/` and change `tce/onboot.lst` to autoload `wireless-6.12.25-piCore-v8.tcz firmware-rpi-wifi wifi.tcz` on boot. Then return the SD card to the Raspberry Pi and execute `sudo wifi.sh` and connect to the Wi-Fi. A working internet connection should now be established, and the credentials will be saved to `wifi.db`. There are two ways to make changes in tinyCore permanent: 1. Setting boot options `opt`, `home`, and `tce` to the target SD card device (`mmcblk0pX`) or 2. Adding the files to the backup with `filetool.sh -b`. It is important to mention that including a large number of files in the backup causes a significant slowdown in boot time because it needs to be decompressed and unpacked at startup.

The set of files that will be included in the filetool.sh backup is determined by two files: `/opt/.filetool.lst` and `/opt/.xfiletool.lst`. `.filetool.lst` adds to the set of backup files and `.xfiletool.lst` removes them from the set. It is very important that files in `/home` are not included in this backup. (Files are already saved on the SD card due to `cmdline.txt setup`).

The porting process of the face sensor is very similar to Alpine Linux after the runtime compilation. Because libedgetpu was

compiled statically with Musl Libc it can be ported to tinyCore by just copying it to `/usr/lib/`. Then the static files are copied together with the Nix bundle and the file paths in `settings.json` are changed to adapt to the new environment (`/home/tc`). By installing ffmpeg with `tce-load -wi ffmpeg`, the permissions and drivers of `/dev/video0` can be confirmed. To remedy the Tor Service issue, change ownership of `/home` to `tc` and install bzip2 so the Nix bundle can extract itself. Afterwards, the Nix bundle is ready to be executed. To save these change,s edit `/opt/.filetool.st` to include `/usr/lib/libedgetpu.so.1` and execute `filetool.sh -bv`.

4.5 DietPi

DietPi is a lightweight distribution that is based on Raspberry Pi OS. It aims to reduce the resource usage of a typical Raspberry Pi OS setup. Due to it also being a Debian distribution (like Raspberry Pi OS Lite), the process of porting the face sensor is almost identical to the Raspberry Pi OS process. In Comparison to Raspberry Pi OS lite, DietPi achieves a reduction in image size of about 61% according to [13]. This is a significant reduction, but DietPi still relies on APT, Systemd, and other systems that increase image size when compared to Alpine and TinyCore. Still, DietPi is one of the most common choices when trying to get more performance out of a Raspberry Pi project.

DietPi is another distribution that promises to deliver a lighter system and higher speeds. It is specifically built for the Raspberry Pi and could therefore offer some more optimizations that other distributions would not. Due to its similarity with Raspberry Pi OS, it is also easy to set up and use.

DietPi can also be flashed onto the SD card with Raspberry Pi Imager (General purpose OS → DietPi). The install process for DietPi

is almost identical to Raspberry Pi OS Lite (See section 4.1). First, libedgetpu-std is installed by registering the package repository from Google Coral and installing it with `sudo apt install libedgetpu-std`. Afterwards, the Nix bundle and static files are copied over from previous configurations, the video feed is verified with v4l2-ctl and ffmpeg, and the Tor error is remedied by changing the owner of `/home` to the executing user. Because DietPi doesn't include `hexdump` and `bzip2` by default, they need to be installed with `sudo apt install bsdmainutils bzip2`. The Nix bundle can then be executed with `RUST_LOG=debug ./sensor-arx --settings /home/ellie/sensor_run/settings.json`.

Chapter 5

Analysis

In this section, the thesis examines the performance and security details of the presented environments. It separates the evaluation into two sections: Performance analysis and security analysis. The performance evaluation is conducted using a series of metrics, which will be presented later. The security evaluation relies on the availability of security features/issues which cannot easily be quantified. Making an exact ranking difficult.

5.1 Performance

Performance can have varying definitions, but for the purposes of this thesis, it refers to how quickly a system completes a specific task and how many resources it requires to do so. The performance is measured by a set of metrics presented in a separate subsection. Each metric is first explained, and then its results are presented.

The following metrics will be presented:

1. RAM Usage during execution
2. Disk usage and Image size
3. Boot speed

4. Time between face recognition
5. Sensor-arx startup time

All results have been gathered by testing out the previously mentioned software environments on a Raspberry Pi 4B. Processor details:

1. Processor: Broadcom BCM2711, quad-core Cortex-A72(ARMv8) 64bit SoC @ 1.8GHz
2. Memory: 4GB LPDDR4

5.1.1 RAM Usage

Random-Access Memory (RAM) usage is a method designed to measure how much memory needs to be allocated at a time for the system to function. If the memory cap is reached (4GB in Raspberry Pi 4B) the entire system will halt due to unrecoverable memory allocation errors. A swap file can prevent a complete halt of the system, but it significantly reduces the speed of the system.

A Swap file / partition is an extension to the system's memory. The kernel moves inactive pages out of RAM and to the swap file if the memory capacity is close to being reached. A swap file can be created with the script shown in Listing 5.1. First a 4GB file is allocated and the correct permissions for it are set. Afterwards the swapfile is made and turned on with `mkswap` and `swapon`. If the change should be made permanent an entry in `/etc/fstab` is needed.

The swapfile behaviour can be changed by setting `sudo sysctl vm.swappiness=$VAL` according to the system's needs. This controls how eager the kernel is to move inactive pages to swap space. The value ranges from 0 to 100 and defaults to 60.

Generally, RAM usage is not a strong measure of performance, but rather an indicator that the system might be hitting its limits if

```
# Allocate a file with 4 Gigabytes and change its permission so only the owner (root) can read/write
sudo fallocate -l 4G /swap
sudo chmod 600 /swap
sudo mkswap /swap
sudo swapon /swap
# Make Changes permanent by registering the swapfile in /etc/fstab
sudo echo "/swap none swap defaults 0 0" > /etc/fstab
```

Listing 5.1: Bash script to create a swap file

```
#!/bin/bash
for ((i = 0; i < 100; i++)); do
usage=$(free -b | awk '/^{}Mem:/{print $3}')
echo $usage > $1
sleep .25
done
awk '{ sum += $1; n++} END {if (n>0) print (sum / n) / (1024*1024*1024) "GiB" }
' $1
```

Listing 5.2: Script to record RAM usage

RAM usage is high. A comparison of RAM usage is still useful to determine if 1. RAM is a limiting factor in some environments, resulting in a possible slowdown 2. Which environment offers the biggest reduction in RAM Usage?

This metric specifically measures the amount of RAM in GB that is currently being used by the system at face sensor execution. The metric is measured 100 times with a delay of 250ms, and the average is returned. Listing 5.2 shows the script used to capture the metric. The script does not differentiate between RAM and swap. The script records the measurements by using the `free` utility, which shows an overview of the systems available and used memory. After all 100 measurements are taken it uses AWK to print the average.

Figure 5.1 shows the results captured in the different software en-

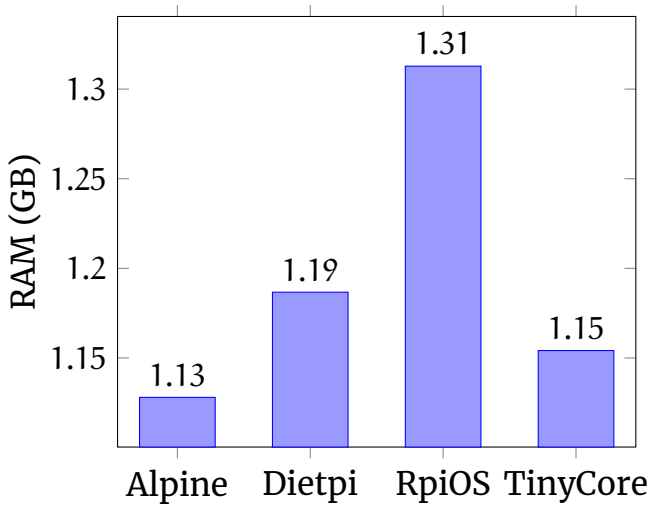

Figure 5.1: RAM Usage statistics during Sensor-arx execution

vironments.

Alpine uses the least amount of RAM at 1.13GB, with TinyCore, at 1.14GB, being a close second. DietPi, at 1.19GB, occupies about 40MB more than TinyCore, but beats Raspberry Pi OS Lite RAM usage, at 1.31GB, by about 120MB.

Discussion

The higher RAM usage on Debian systems (Raspberry Pi OS and DietPi) was expected due to their usage of Systemd. The surprising revelation is that Alpine actually uses the least amount of memory, even though TinyCore is the smallest system that was tested. This difference can be explained by TinyCore's methodology of loading extensions. TinyCore mounts all extensions as loop devices. Loop devices are virtual devices that allow files to be read as block files. These loop devices are mounted on `/tmp/tcloop/$EXTENSION`. This takes up additional RAM because `/tmp` resides on tmpfs, which is directly stored in RAM. This sacrifices RAM usage for an increase in performance.

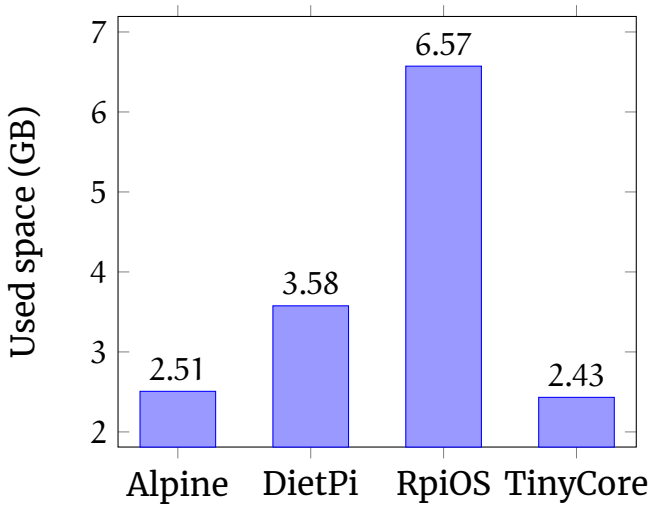

Figure 5.2: Disk Usage statistics

5.1.2 Disk Usage and Image size

The disk usage metric specifies how much space the environment occupies on the hard disk and determines on what hardware the architecture can be deployed. Disk usage is defined as the amount of space occupied by files on the main disk (Only one disk is used in the environment setup). The image size metric is the amount of space that the raw OS image takes up before being deployed. It impacts how fast a deployment can occur and what storage medium can be used.

The disk usage is determined by: `df -h | grep /dev/mmcblk0`. It captures the amount of used space on the boot and root partitions. The measurement is done on a minimal setup that contains the edge TPU runtime, the Nix bundle, the required static files from `static/detection` and `static/recognition`, and required libraries to run the executable, like Hexdump and Bzip2. The metric is captured after the first execution of the Sensor-arx executable, because the first execution adds about 1.5GB of extra dependencies on the disk. The results are shown in 5.2

TinyCore has the lowest disk usage with 2.43GB followed by Alpine Linux at 2.51GB. DietPi's disk usage is significantly higher at

3.58GB, but far removed from RaspberryPi OS's disk usage at 6.57GB.

Discussion

The disk usage of Alpine and TinyCore is barely different. This is due to these systems being so minimal that almost all of the space is taken up by the Sensor-arx executable, the static files, and their dependencies. DietPi cuts back on a lot of the unneeded programs that the Raspberry Pi OS includes by default. However, due to DietPi still relying on Systemd, Sudo, and other Debian systems, it cannot achieve the same size that Alpine and TinyCore can achieve. Raspberry Pi Lite OS uses the largest amount of space (almost 3GB more than DietPi).

A similar story is told by the image size metric, which is measured with `ls -l $IMAGE_FILE | awk '{print \$5}'`. The results can be seen in 5.3. It should be noted that Raspberry Pi OS lite and DietPi use `xz` compression while Alpine and TinyCore use `gzip`. `gzip` is generally faster but produces files that are about 1.5x larger than `xz` [6]. Because of this discrepancy, all images were first decompressed and then recompressed using `gzip` to establish a common compression algorithm.

TinyCore has the smallest image at 38.45MB, closely followed by Alpine at 86.63MB. DietPi's image, at 283.55MB, is significantly smaller than RaspberryPi OS's image at 771.96MB.

5.1.3 Boot speed

The boot speed metric indicates how fast the system can reach a state from which the face sensor service could be started after a reboot. This metric may be an important factor when considering power-saving options. If the face sensor is configured to be powered down when nobody is in the facility at nighttime, the boot

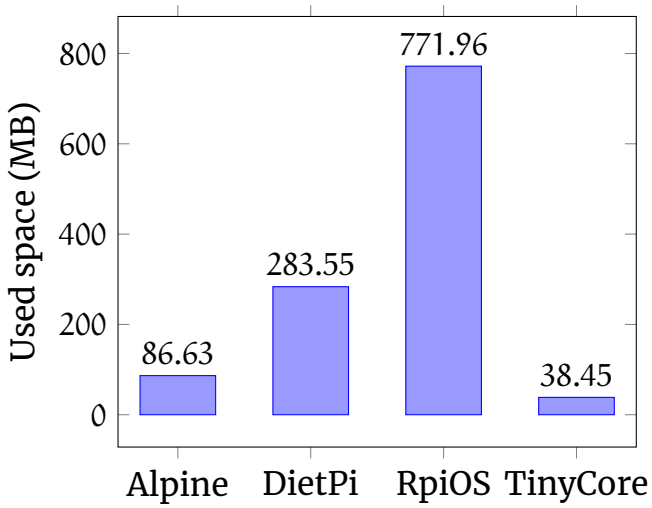

Figure 5.3: Image size statistics

speed could be a big factor in the decision of what environment to employ.

Boot speed is not easy to measure because the early stages do not provide file access. On systems with Systemd initialization, `systemd-analyze` can be utilized. It determines exact times of UEFI loading, kernel startup, and Systemd initialization. However, TinyCore and Alpine use different systems, which makes a consistent measurement process across environments difficult. Because of this issue, this thesis relies on the one common denominator across all environments: The Linux Kernel. The `/proc/uptime` from procfs measures the time since the kernel has been initialized. This is the closest approximation to boot time that is possible on all systems.

The boot speed is measured by hooking into the different initialization systems that the environments provide. Even though the initialization systems differ, the principle remains the same across all systems: A service is registered with the initialization system that starts at the latest runlevel after a network connection has been established.

On the Debian systems (DietPi and Raspberry Pi OS), the boot

```
[Unit]
Description=boot-record-service
After=network-online.target

[Service]
ExecStart=/usr/bin/boot-record.sh

[Install]
WantedBy=multi-user.target
```

Listing 5.3: Systemd service file for boot-record service

recording service can be achieved using the following steps: 1. Create a custom executable bash script at `/usr/bin/boot-record` which logs the startuptime to `/var/boot-time.log` 2. Create the Systemd unit file at `/etc/systemd/system/boot-record.service` according to Listing 5.3. 3. Enable the service with `sudo systemctl enable boot-record.service`. After a reboot, `/var/boot-time.log` shows the startup time.

`After` and `WantedBy` dictate that this service is required to reach the multi-user target of Systemd and that it is started only after the network-online target is met.

On TinyCore the same can be achieved by simply adding `wifi.sh -a 2&>1 > /tmp/wifi.log && cat /proc/uptime | awk '{print $1}' > /var/boot-time.log` line to `tce/bootlocal.sh`. This tells the system to connect to the first entry in the `wifi.db` file, redirect the output to `/tmp/wifi.log`, and after a network connection has been established, print the uptime to a log file. This shell script is the last step in the startup process before loading `.profile` files [10].

Alpine Linux uses the OpenRC init system. Similar to Systemd, a service, which runs at a specific time in the initialization process, can be registered with OpenRC. A service in OpenRC is represented by a service file that declares dependencies, file accesses, start/stop mechanisms, etc. In this case, a simple service file, as shown in Listing 5.4, is sufficient.

```
description="Boot record service"

depend() {
    after=net-online
}

start() {
    ebegin "Starting boot record service"
    cat /proc/uptime | awk '{print $1}' > /var/boot-time.log
    eend $?
}
```

Listing 5.4: OpenRC service file

The measurements are captured 10 times, and the average is presented in Figure 5.4. DietPi has the fastest startup time at 12.61 seconds, barely edging out Alpine Linux at 14.16 seconds for the top spot. TinyCore places at a close third with 15.75 seconds. Raspberry Pi OS however occupies the last place with the slowest boot time at 23.76 seconds.

Discussion

DietPi's high placing may be due to its usage of Systemd which heavily uses parallelization to reduce startup times. There are configuration options which could potentially speed up the startup process in other environments (OpenRC, Busybox init) as well (e.g: `rc_parallel=YES`); however, most of the slowdown seems to originate from connecting to the wireless network and the Dynamic Host Configuration Protocol (DHCP) discovery process. The boot speed could be improved by using a static IP and an Ethernet connection, which is not available and lies outside the scope of this thesis. The only outlier in this metric is Raspberry Pi OS due to its broad range of services. TinyCore's startup time is also slower than Alpine and DietPi due to its extension load-

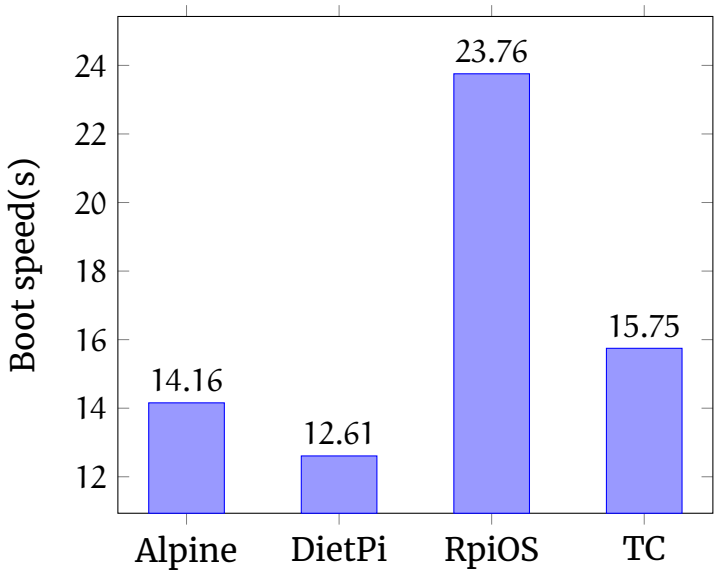

Figure 5.4: Boot speed statistics

ing and `mydata.tgz` extraction. Further study would be required to investigate exactly why DietPi is faster in this area.

5.1.4 Image Processing speed

This metric measures how fast the system can process an image from the video feed. This significantly impacts how many decisions per second can be made by the system. A faster face detection results in more images that can be processed per second and therefore less time that the user needs to spend interacting with the system.

This metric was implemented by developing a custom patch to the face sensor. If the `benchmark` feature is enabled when building the face sensor Nix bundle, a time measurement before and after the `process_image` call is taken. `process_image` is responsible for running face detection on the image and making an authentication decision based on it. Afterwards, a duration is calculated and written to a benchmark log file that needs to be specified in `settings.json`. The required modifications to the face sensor are described in List-

ing 5.5, 5.6, and 5.7. The modifications introduce a new feature flag called `benchmark`. If this feature is specified in the cargo build, sections of code within the `pipeline.rs` and `config.rs` are activated. The code sections within `pipeline.rs` record a timestamp before and after the `process_image` call and store the difference in a buffer. This buffer is flushed out into the benchmark log file specified by the modifications in `config.rs`.

```
    @ 274
    +#[cfg(feature = "benchmark")]
    +let mut count: u16 = 0;
    +#[cfg(feature = "benchmark")]
    +let mut buffer: Vec<u128> = Vec::new();

    loop {...}
    @ 303
    +#[cfg(feature = "benchmark")]
    +let start_time = SystemTime::now();
    self.process_image(&im);
    +
    +#[cfg(feature = "benchmark")]
    +{
        +   let duration_ms = match SystemTime::now().duration_since(start_time) {
            +       Ok(duration) => duration.as_millis(),
            +       Err(_error) => {
                +           error!("Error when trying to get time");
                +           0
                +       }
            +   };
        +   buffer.push(duration_ms);
        +   if count == 25 {
            +       count = 0;
            +       if let Some(benchmark_log_path) = &self.settings.
    benchmark_log_path {
                +           if let Ok(file) = File::create(benchmark_log_path).
    as_mut() {
                    +               let mut content = String::new();
                    +               for duration in &buffer {
                        +                   content.push_str(&format!("Processing
    duration: {}ms\n", duration));
                        +               }
                    +               file.write_all(content.as_bytes());
```

```
32                  +           } else {
33                  +               error!("Could not create file for benchmarking
    ");
34                  +           }
35              +       }
36          +       else {
37              +           error!("No log file path for benchmarking was
    specified!")
38              +       }
39          +       buffer.clear()
40          +   } else {
41          +       count += 1;
42          +   }
43       +}
```

Listing 5.5: Modifications to src/pipeline.rs

```
1   @ 262
2   #[cfg(feature = "singledoor")]
3   pub door_id: i64,
4   +#[cfg(feature = "benchmark")]
5   +pub benchmark_log_path: Option<String>,
```

Listing 5.6: Modifications to src/config.rs

```
1   # flake.nix
2   @ 71
3   + cargoExtraArgs = "--features␣'benchmark'";
4   strictDeps = true;
5   doCheck = stdenv.buildPLatform.canExecute stdenv.hostPlatform; # add check for
      emu
6
7   # Cargo.toml
8   @ 13
9   singledoor = []
10  save_images_on_sensor_push = ["face/vis"]
11  debug_vis = ["face/vis"]
12  +benchmark = []
```

Listing 5.7: Modifications to flake.nix and Cargo.toml

By specifying the benchmark log file with `"benchmark_log_path"` in the `settings.json` file, the Rust patch will log the time spent for face de-

tection and authorization. It is important to note that the processing takes longer if there is a face visible. This is why the result of any measurement that takes longer than 750 ms is seperated from the others. The metrics are captured using the AWK script shown in Listing 5.8. The results are shown in Figure 5.5. The error bars indicate any value that would fall within the 95% confidence interval (assuming a normal distribution). The standard deviation was calculated using the script seen in Listing 5.9 (The mean values need to be provided with `-v mean1=... -v mean2=...`).

```
1   match($3, /([^ms]*)/, val) {
2       if(val[0] < 750) {
3           sum1 += val[0]; n1++
4       } else {
5           sum2 += val[0]; n2++
6       }
7   } END {
8       if(n1 > 0) print "Face detection: " sum1 / n1 "ms";
9       if(n2 > 0) print "Face detection+recognition: " sum2/n2 "ms"
10  }
```

Listing 5.8: AWK script to calculate average from benchmark log

```
1   match($3, /([^ms]*)/, val) {
2       if(val[0] < 750) {
3           sum1 += (mean1-val[0])^2; n1++
4       } else {
5           sum2 += (mean2-val[0])^2; n2++
6       }
7   } END {
8       if(n1 > 0) print "Face detection stddev: " (1/n1)*sqrt(sum1) "ms";
9       if(n2 > 0) print "Face detection+recognition stddev: " (1/n2)*sqrt(sum2) "
    ms"
10  }
```

Listing 5.9: AWK script to calculate standard deviation from benchmark log

It should be noted that the environments are not the only variable influencing the processing speed. To reduce the impact of

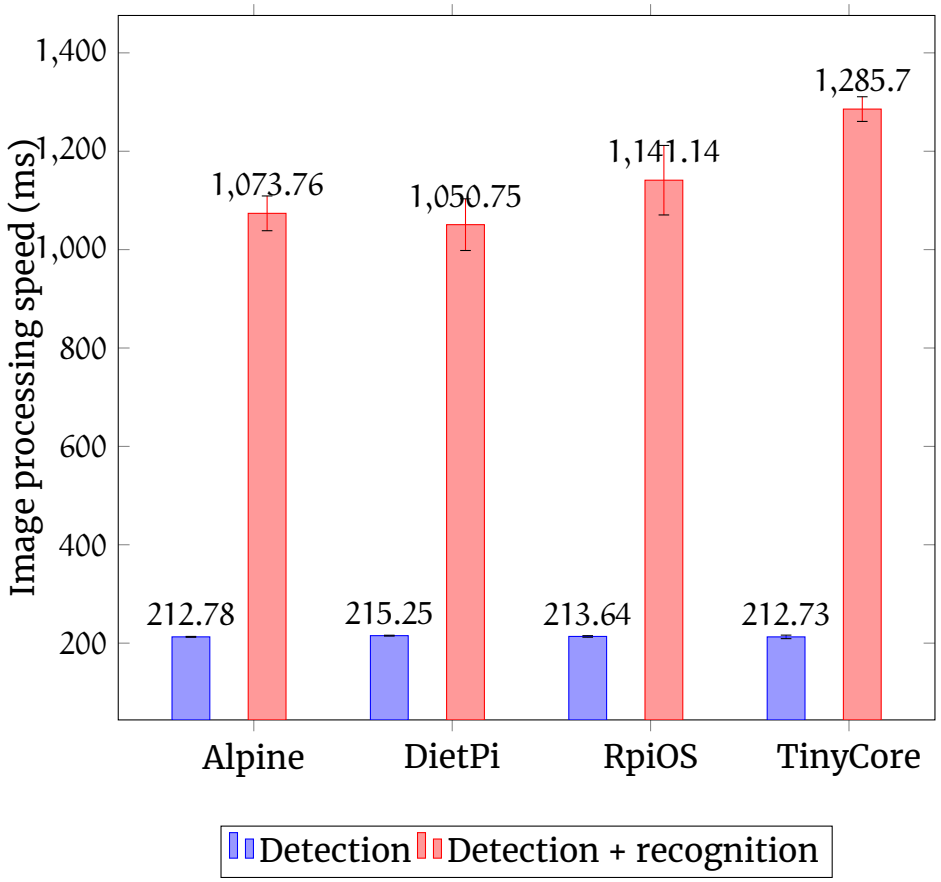

Figure 5.5: Image processing speeds

outside circumstances, the sample size was set at 300 individual measurements.

Detection speeds are similar across the board, with just a few milliseconds difference. A more notable difference can be seen in the face recognition segment, where DietPi and Alpine Linux have the best speeds at 1050.75ms and 1073.76ms. With TinyCore and RaspberryPi OS having slightly worse processing speeds at 1141.14ms and 1285.7ms. Another difference is the standard deviation: TinyCore and Alpine Linux have the lowest, resulting in more reliable timing.

To understand why these speed differences exist, a thorough analysis would be required, which lies outside the scope of this thesis.

5.1.5 Sensor-arx startup time

The Nix bundle (also called Sensor-arx) has a long startup time. This is because the Nix environment needs to be prepared before the face sensor execution. This (similar to boot time) affects how fast the system can be started from a non-executing state. This is mainly affected by the speed of binary extraction from the Nix bundle.

This metric is defined as the time between issuing the execution command of Sensor-arx and the start of the Gstreamer pipeline. TinyCore presents a special case here: It does not provide the `%N` option in the `date` utility, which means it does not have access to millisecond precision in measurements. To remedy this issue, a custom C program was written (See Listing 5.10) which monitors the logs of a running program to detect the start of the GStreamer loop. It works by reading lines from stdin and every time a search term, which is provided as an argument, is found, the elapsed time is written to stdout.

The metric can then be measured by: `RUST_LOG=debug ./sensor-arx --settings /home/tc/sensor_run/settings.json 2>&1 > /dev/null | ./startup_speed "Starting stream, using gstreamer string"`. This command first sets the `RUST_LOG=debug` environment variable to tell the Rust logger to redirect `!debug()` calls to stderr. Because the C program only captures input from stdin, stderr needs to be rerouted to stdout with `2>&1`. The old stdout is routed to `/dev/null` because it is irrelevant to the Sensor-arx startup measurement. Then the new stdout, which contains the Rust logs, is piped into `./speed`.

```
1  #include <stdlib.h>
2  #include <time.h>
3  #include <stdio.h>
4  #include <string.h>
5
6  #define MAX_LINE_LENGTH 4096
7
8  int main(int argc, char *argv[]) {
```

```
9           struct timespec starttime, stoptime;
10          if (clock_gettime(CLOCK_REALTIME, &starttime) == -1) {
11              perror("clock_gettime");
12              exit(EXIT_FAILURE);
13          }
14
15          double starttime_s = starttime.tv_sec + ((double)starttime.tv_nsec /
    1000000000);
16          printf("Starttime: %.6fs\n", starttime_s);
17
18          char *linebuffer = malloc(MAX_LINE_LENGTH);
19          if (!linebuffer) {
20              // Not enough space for line in memory
21              perror("malloc");
22              exit(EXIT_FAILURE);
23          }
24          while (fgets(linebuffer, MAX_LINE_LENGTH, stdin)) {
25              // If the log line detects the string provided -> Log it to the
    console
26              if (strstr(linebuffer, argv[1])) {
27                  if (clock_gettime(CLOCK_REALTIME, &stoptime) == -1) {
28                      perror("clock_gettime");
29                      free(linebuffer);
30                      exit(EXIT_FAILURE);
31                  }
32                  double stoptime_s = stoptime.tv_sec  + ((double)stoptime.tv_nsec /
     1000000000);
33                  printf("Elapsed Time: %.6f, line = %s\n", stoptime_s - starttime_s
    , linebuffer);
34              }
35          }
36          free(linebuffer);
37      }
```

Listing 5.10: C program to determine startup speed of sensor-arx

The first execution of sensor-arx always takes about 2x longer than subsequent executions. In this metric, only succeeding executions are measured. The results are shown in Figure 5.6.

Debian-based operating systems (Raspberry Pi OS and DietPi) vastly outperform TinyCore and Alpine Linux in this metric. Rasp-

berry Pi OS is has the fastest startup time of 11.23 seconds, just slightly faster than DietPi's 11.24 second startup time. TinyCore and Alpine linux have the longest startup times at 26.42 seconds and 30.76 seconds respectively.

Discussion

This may be due to a difference in the packages provided by the OS and how they were compiled (possible support for parallelization). This may also be subject to change due to `nix-bundle` being an experimental feature within Nix. A definitive answer would require further study and lies outside the scope of this thesis.

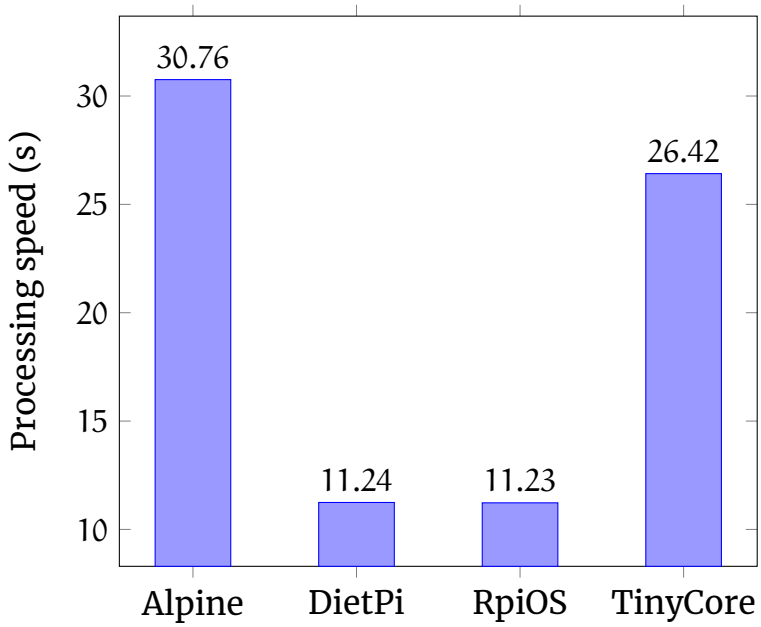

Figure 5.6: Sensor-arx startup speeds

5.2 Security

Security is the top priority of the Digidow project. For this reason, it is important to evaluate the security implications of these different environments. In this section, a number of different security factors that determine how safe the face sensor would be

if they were applied are introduced. Each subsection describes a security category/feature, what security implications it has, and which environments implement it.

5.2.1 Minimal Attack Surface

Small and simple systems are easier to manage and have fewer entry points for attackers to exploit. Reducing the attack surface is one of the most effective methods of preventing future attacks. Each additional program (or a larger alternative to a program) poses a threat as another vulnerable point in a systems defense against attackers. These additional points of attack force the system to spend additional resources to verify their correctness and safety. For this purpose, Alpine Linux and TinyCore use small modular systems like Busybox and OpenRC, which vastly reduce their image size as seen in Figure 5.3.

It is important to note that not all attack surfaces are equal. This means that some programs are stationed at a critical location within the operating system, making them especially prone to attacks. These programs are privilege escalation programs like `sudo`, `su`, and `doas`, initialization systems like Systemd and OpenRC, standard libraries that interact directly with the kernel through system calls, and any other kind of program that requires root privileges. In order to evaluate the score of the environments in this security aspect, the following program types will be examined in further detail: Privilege escalation systems, initialization systems, the standard libraries, and SSH daemons.

Intialization Systems

This thesis examines Systemd, OpenRC, and Busybox init.

Systemd is not only an initialization system but rather a collection of utilities and programs that provide the system with

a central tool for managing logs through `journald`, services with `systemctl`, event management, as well as timers and resource control through control groups. Most distributions use Systemd as it is an easy tool that covers a lot of use cases for which other distributions would need separate utilities. Because Systemd has such a broad catalog of features, its size is unparalleled when compared with other init systems like OpenRC, Sysvinit, or runit. Due to its size, a smaller system is preferred even if it does not possess the same number of features.

OpenRC is an alternative to Systemd that provides service management, resource control, and an optional init system. It is designed to be a fast and modular system that is non-intrusive and typically works alongside other utilities (e.g., Syslog). While, accroding to the tests in this thesis, OpenRC does not provide a big benefit in terms of startup performance, it is vastly smaller and more adaptable. OpenRC was originally developed as part of Gentoo but has since evolved into a standalone project.

Busybox init is a smaller alternative to Sysvinit, which is minimalistic in nature. It provides a base functionality of executing scripts from `/etc/init.d/`. Any advanced configuration, like runlevels and targets, needs to be configured through `/etc/inittab`. Busybox init is the bare bones init system, which does not provide anything more than is necessary for a simple startup.

While Systemd provides advantages in its ease of use and huge catalog of features, it has too many drawbacks for Digidow's security-sensitive use case. However, Busybox init does not provide enough features to warrant its use. OpenRC is the optimal choice for the face sensor's use case.

Privilege escalation programs

For the privilege escalation programs, this thesis examines Sudo and Doas. Sudo is the privilege escalation tool that most Linux distributions use by default. It provides a great number of features for fine-grained control over executing commands as root. While Sudo is hardened against attacks and has stood the test of time, it is not without its flaws. Sudo in comparison to Doas is larger and has a higher attack surface. Doas instead focuses on being as small as possible while still allowing for necessary security features like rule-based privilege escalation.

The fundamental problem with privilege escalating programs is that they allow users to gain access to a root environment without actually possessing the root password. This removes a second layer of defense against attackers. This is why, if it can be avoided, Sudo and Doas should not be installed to prevent an attacker from compromising the entire system if possible. If this is not possible, the configurations of `sudo` and `doas` need to be fine-tuned to only allow specific commands and arguments to be used. A blank cheque should never be granted.

All software environments that have been tested (except for RedoxOS) have `sudo` and `doas` packages available for installation.

C Standard library

The C standard library (also called Libc) is a library that provides macros, functions, types, and constants for the C programming language. Because Libc interacts directly with the kernel through system calls, it is a safety-critical library. Two candidates for the C standard library will be examined: Musl and Glibc.

Glibc is the industry standard and implements the functionality required by POSIX, parts of ISO C11, and more OS-specific interface specifications. It has been thoroughly tested and has stood

the test of time as the most common standard library. However, due to its complexity and size, vulnerabilities have been detected in the past. The most notable one being CVE-2015-0235 (also known as GHOST), which allowed arbitrary code execution within Glibc. Glibc mostly relies on dynamic linking of libraries.

Musl also implements the POSIX and C11 standard. It is designed to strictly adhere to the standards and be as simple as possible. This significantly reduces the attack surface of the standard library. Musl uses static linking to compile binaries with its needed dependencies. This increases portability and compatibility with safety features like static PIE.

While both Libcs use standard security features like stackguard, ASLR, and memory protections. Musl Libc is the better option due to the attack surface reduction and static linking support.

SSH daemon

The last security-critical application that this thesis examines is the SSH server. Secure Shell (Secure Shell) is a protocol that allows a system to remotely interact with a different system through a shell. The usage of SSH makes the system inherently vulnerable to attacks because the attacker no longer has to be near the physical location of the Raspberry Pi. This is why an SSH daemon should generally be avoided in a production environment if possible. If this is not feasible, sufficient defensive countermeasures should be employed to thwart attacks seeking to exploit the SSH daemon. Countermeasures include: Public key authentication, refusing root logins, using PAM for two-factor authentication, and sufficient password protection on public key and user login. Pluggable Authentication Modules is a security framework that defines policies for the authentication process.

With this knowledge, an investigation into the choice of an SSH

Program Type	Alpine	DietPi	Raspberry Pi OS	TinyCore
Init System	OpenRC	Systemd	Systemd	Busybox-init
Privilege Escalation	Doas/Sudo	Doas/Sudo	Doas/Sudo	Sudo
C Standard Library	Musl	Glibc	Glibc	Glibc
SSH daemon	Db/OpenSSH	Db/OpenSSH	Db/OpenSSH	Db/OpenSSH

Table 5.1: Available security-critical programs in all 4 environments

daemon can be conducted. There are two possible candidates: OpenSSH(`sshd`) and `dropbear`. OpenSSH is the industry standard and has the most amount of security features. Dropbear, in comparison, uses fewer resources and is smaller in terms of size, but does not offer the same amount of encryption support as OpenSSH does. Even though it is generally important to minimize the possible attack surface, the drawback of losing security features outweighs the benefits dropbear provides.

Table 5.1 shows which security-critical programs are available in what environment.

5.2.2 Hardening the kernel

Another important security consideration is the kernel. An exploitable kernel is a prime target for attackers who seek to gain control of the entire system. This is why it is especially important to harden the kernel against all possible attacks. In this section, a set of measures that can be taken to harden the kernel is described.”Hardened” means that the system has specific measures that make it harder to penetrate or exploit. Afterwards the default security features of each environment’s kernel are examined.

Generally, every kernel (from our tested environments) can be hardened against attacks. But some environments make the pro-

cess easier than others. Important hardening options include:

1. ASLR
   Address Space Layout Randomization is a safety feature that randomizes the layout of memory regions to make attacks that rely on predictable memory addresses (ROP/return into Libc) harder to execute. The safety feature is determined by `kernel.va_randomize_space=2`. The kernel can further be hardened against these attacks by choosing: 1. `CONFIG_RANDOMIZE_BASE`, which randomizes the physical address at which the kernel is mapped to. 2. `CONFIG_RANDOMIZE_MEMORY` (only available in x86), which randomizes the layout of different memory regions in the kernel. 3. `CONFIG_RANDOMIZE_MODULE_REGION_FULL`, which randomizes the module section in the kernel.

2. Stack Guard
   Stack Guard [4] is a `gcc` plugin that forces each method to be safeguarded against stack smashing attacks through the use of a randomly chosen canary word that is checked before jumping to the return address. It is controlled by the option `CONFIG_STACKPROTECTOR`.

3. Position Independent Executable (PIE)
   Position Independent Executables are executables that are compiled to run and load at an arbitrary memory address. They do not need memory address recalculation like other binaries. This feature is needed for user binaries to take advantage of ASLR. Static PIE is a position-independent executable that is statically linked.

4. Kernel pointer hiding and kernel log hiding
   Some kernel pointers are exposed through the `/proc` interface and can be abused by attackers for write vulnerabilities or calculating other kernel addresses. To hide these pointers, set `kernel.kptr_restrict`. Kernel logs can be abused by attackers to gain information about the system. They can be restricted with

kernel.dmesg_restrict and CONFIG_SECURITY_DMESG_RESTRICT.

5. Custom kernel patches for advanced security
   Some organizations provide custom Linux kernel patches that target safety issues like memory integrity (e.g. GRSecurity, OpenPaX))

6. System call filtering
   Seccomp, or "secure computing," is a component that enables applications to create environments in which certain system calls are prohibited. This is useful for any applications that need to process some form of input that could be malicious. This can be enabled using CONFIG_SECCOMP CONFIG_SECCOMP_FILTER.

7. Memory protections
   The Linux kernel includes multiple options to fortify specific aspects of memory safety. These options include: hardening usermode copy with CONFIG_HARDENED_USERCOPY, detection of buffer overflows with CONFIG_FORTIFY_SOURCE, denying access of root to /dev/mem by disabling CONFIG_DEV_KMEM, preventing write operations to a memory page, that is marked as executable, with CONFIG_STRICT_KERNEL_RWX and CONFIG_STRICT_MODULE_RWX, etc.

8. Enforce a strict userspace / kernelspace boundary
   There are some interfaces that can be used by root to modify the Linux kernel. This presents a serious threat to the integrity of the Linux kernel and should be turned off. This can be achieved by setting lockdown=confidentiality.

9. SELinux and AppArmor
   SELinux is a framework developed by the NSA that allows systems to enforce Mandatory Access Control (MAC) policies on files and processes. Most distributions implement MAC using the AppArmor kernel module.

8. Audit system
   audit is a framework that captures security-related events and

allows users to analyze potential security problems. Rules can also be set up to automatically forward certain security events to another system. This can be enabled with `CONFIG_AUDIT`.

This is not an exhaustive list of all options, but it provides a good baseline for additional security. Most operating systems generally provide examples for hardened configurations or guides on how to compile a hardened kernel. For example, Arch Linux provides a precompiled hardened kernel in its package repository. If such a configuration is not available, a custom kernel needs to be built using tools like `linux-hardened`.

The environment's kernel security will be evaluated based on two factors:

1. The security features of the default kernel
2. The ease of creating a hardened kernel

To estimate how secure the default kernels are, an overview of the enabled security features is shown in Table 5.2. These options have been determined by checking the `kconfig` and default `sysctl` options of the various kernels manually and in combination with `kernel-hardening-checker`.

Discussion

In this comparison, Alpine Linux is the clear winner. OpenPax is a custom Linux kernel patch that solves common memory safety errors.TinyCore and Raspberry Pi's kernel are similar in terms of important security features and differ in MAC, hardened SLAB and userspace copy, and audit support.

All Linux kernel modifications follow a similar pattern. This thesis demonstrates this pattern based on the process of creating a hardened Linux kernel for Alpine Linux. To create a hardened

Security Feature	Alpine	DietPi	RpiOS lite	TinyCore
PIE user binaries	✔(static)	✔	✔	✔
StackGuard	✔	✔	✔	✔
ASLR	✔	✔	✔	✔
Kernel pointer hiding	from non-root	✖	✖	✖
Safety kernel patch	✔(openpax)	✖	✖	✖
System call filtering	✔	✔	✔	✔
Hardened SLAB and usercopy	✔	✖	✖	✔
Restrict root memory access	✔	✖	✖	✔
W xor X	✔	✔	✔	✔
MAC architecture	✔	✔	✔	✖
Audit kernel support	✔	✔	✔	✖

Table 5.2: Overview of kernel security features that are enabled in at least one environment

Linux kernel with the OpenPax patch, the following steps should be followed:

1. Set up the default Alpine building environment as described in section 4.3.
2. Checkout to the latest stable version of Alpine with `git checkout -b $VERSION-hardened origin/$VERSION` navigate to the linux-openpax package in `community/linux-openpax`.
3. Download and add the latest hardened linux-kernel patch from linux-hardened to the kernel sources.
4. Change the configuration file, depending on your needs (use kernel-hardening-checker to verify and menuconfig for an easy configuration).
5. Regenerate the checksum with `abuild checksum` and build the package.

6. Add the built package with `doas apk add ~/packages/community/$ARCH/linux-openpax`.
7. Reboot to apply and run the new kernel (initramfs should be already generated by `apk add`).

Debian generally discourages and has little support and guidance for compiling a hardened kernel because it breaks the automatic maintenance of the kernel. Some packages still provide tools for hardening the Debian kernel, like `hardening-runtime`. Debian also has hardening guides for configuring building tools that harden the user-mode binaries.

TinyCore provides guides for building your own kernel. TinyCore's kernel configuration can be modified with the `kernel-hardening-checker` for maximum security options. TinyCore also provides the needed kernel patches that it needs to function.

Alpine Linux provides the most amount of tools to customize the kernel, followed by TinyCore.

5.2.3 Package manager security

The package manager is one of the most fundamental utilities that a distribution has to provide. It allows the user to download and install software packages, manage their installation, version, settings, and incorporate the packages into the target environment using a structure determined by the package manager (e.g., APKbuild for Alpine's APK, PKGbuild in Arch's pacman). Package managers are security-critical applications because they represent a single point of failure for installing potentially malicious software. There are several security measures most package managers take to ensure package integrity and safety:

1. Package and repository signing
   To ensure the integrity of the package, it is signed by a private master key of the repository, which is trusted in advance.

This prevents any tampering of the package file that will be installed. It also prevents attacks where the attacker poses as a mirror of the package repository.

2. Package checksums
   Here, a checksum of the package's contents is generated and repeatedly checked to prevent any form of tampering after the package has been installed. This can be checked periodically or upon execution.

3. Public Security Issue Tracker
   If a security vulnerability or bug arises in one of the packages in the most up to date repo, it will be tracked on such a tracker. This allows vulnerability scanners to check if a specific system using this package manager has a vulnerability and how severe it is.

4. Stable release vs Rolling release
   Stable release distributions create a new distribution/package repository version after a specific time period (e.g. 6 months for Alpine Linux) and will keep supporting older versions for some time after it has been overhauled (2 years for Alpine Linux). Rolling release distributions update their packages continuously, possibly breaking interactions between packages. These distributions are less stable and reliable than stable distributions.

5. Communication protocol
   By using HTTPS as an additional security measure, an attacker would need to have access to both the SSL certificate and the repository master private key. It also prevents man-in-the-middle attacks as mentioned in [1].

The package managers of the different environments are now compared with each other using these security measures as a base. Table 5.3 shows an overview of which security measures are implemented in which package manager. Alpine Linux uses APK, De-

Security Measure	APK	APT	Tce-load
Package Signing	✔	✔	✖
Package Checksums	✔	✔	✔
Public Security Issue Tracker	✔	✔	✖
Stable Release	✔	✔	✔
Uses HTTPS	✔	✔	✖

Table 5.3: Overview of package manager security measures

bian uses APT, and TinyCore uses TCE-load.

TinyCore's package manager does not have many security features preventing man-in-the-middle or malicious mirror attacks. APK and APT have similar security features, with both stable and a rolling release branch (unstable for Debian and edge for Alpine). They also both separate packages into different repositories. The only major difference is that Alpine Linux compiles and builds its packages with more hardening features than Debian (e.g. Read-only relocation table), and it enforces secure self-compilation with encryption keys through `abuild`.

Vuls - Vulnerability Scanner Vulnerability scanners, as mentioned before, can scan the entire system for packages that have publicly known vulnerability issues. This specific vulnerability scanner can be installed and has been tested out on DietPi, Raspberry Pi OS, and Alpine Linux. Listing 5.11 describes how to install Vuls on the Raspberry Pi. Vulsio can easily be installed using the `vulsctl` utility provided by the Vuls project. To install Vuls natively, the default install-host script needs to be adapted to select the arm64 target of Go. Go is an open source programming language developed by Google.

Before the installation can be done, a swapfile needs to be created and active (See Listing 5.1). Testing has determined that the swap

file needs to be at least 4GB large on the Raspberry Pi 4B.

```
    git clone https://github.com/vulsio/vulsctl && cd vulsctl/install-host
    sed -i -e "s/linux-amd64/linux-arm64/" install.sh
    sudo bash install.sh
    cd $HOME
    echo '
    [servers]
    [servers.localhost]
    host = "localhost"
    port = "local"' > config.toml
```

Listing 5.11: Bash script to install Vuls on Debian systems or Alpine Linux

After installation, the scanner can be started with `vuls scan -format -one-line-text`. The result will be displayed as a one-line summary indicating the different amounts of vulnerabilities with their corresponding severity score. To get a full case-by-case summary, use `vuls tui` or `vuls report`.

Chapter 6

Conclusion

This thesis presents the face sensor performance and security results of different environments and establishes a porting process using Nix and Docker. 4 different environments were tested: Alpine Linux, Raspberry Pi OS Lite, DietPi, and TinyCore. Each has its own benefits and drawbacks; one even is uniquely adapted for use in CDL Digidow. It also establishes a new process of building the edge TPU runtime to ensure portability.

6.1 Environment results

Raspberry Pi OS Lite's advantages lie in its popularity and easy-to-navigate setup utilities. Raspberry Pi OS works out of the box without any problems and is easy to configure. The face sensor porting process was by far the easiest across all environments. It offers a strikingly fast `nix-bundle` startup time with average image processing speeds but comes at a cost of large disk usage and slow boot times. While Debian implements some common security features, most of its security advantages stem from the mass adoption of the Debian ecosystem. Security patches and updates are delievered quickly. Raspberry Pi OS is a reliable system that works well as a baseline, but it suffers in terms of a lack of ad-

ditional security needed for Digidow's requirements and high resource usage.

DietPi is a lightweight version of Raspberry Pi OS focused on resource efficiency and eliminating redundancies. This allows DietPi to achieve significantly lower disk/RAM usage and faster boot times. DietPi, like Raspberry Pi OS, also contains user-friendly setup scripts that make the bloat-removal process easier. It also inherits security patches and updates upstream from Debian and has thorough documentation due to relying on Raspberry Pi OS systems. DietPi offers an increase in performance in comparison to Raspberry Pi OS, and has no significant drawbacks related to the use case of the face sensor. There is no major difference between DietPi and Raspberry Pi OS's security issues.

TinyCore is a distribution focused on absolute minimalism, with its default operation mode being diskless. This allows TinyCore to have the lowest disk usage and the second-lowest RAM usage. TinyCore is a niche operating system, and almost all of its documentation is either out of date or unavailable. With some configuration, a stable setup with low resource usage can still be achieved. However, there are serious safety issues with the package manager and basic security features like SELinux integration.

Alpine Linux is a security-first distribution that excels in small environments due to its use of Busybox and Musl Libc. Alpine Linux, according to the tests and security investigations conducted in this thesis, the most secure environment of all tested candidates. This is due to the vast amount of security measures applied: Hardened userspace binaries, openpax kernel security patch, AppArmor and auditing support, abuild and APK safety features, etc. The environment also uses the least amount of RAM (Figure 5.1), has fast boot and image processing speeds. But it has the slowest Sensor-arx startup time. Alpine has great documentation and good setup scripts aiding an initial setup, and provides

a lot of utilities and guidance materials for customizing the Linux kernel.

6.2 An Ideal System Environment

Now that all possible benefits and drawbacks have been thoroughly discussed, this section describes an ideal environment based on the requirements of CDL Digidow as well as the results presented in chapter 5. This thesis proposes an Alpine Linux environment with the following security and performance measures:

1. A custom hardened kernel using the Openpax and Linux-hardened kernel patches.
2. AppArmor profile for Nix bundle which restricts file access to the needed set of files like Tensorflow lite models, video feed, settings file, and Nix dependencies.
3. OpenRC service which starts the Nix bundle and is restricted by AppArmor profile and Seccomp filtering.
4. OpenSSH configuration requiring public key authentication and denying any root login.
5. Restricted access to Doas for all users, especially the executing user.
6. Audit framework setup which monitors files critical for the operation of the face sensor and reports any changes
7. Automatic package update script which uses Vuls to periodically scan the entire system for safety vulnerabilities and updates any vulnerable packages if a fix is available.
8. Parallelized implementation of Bzip2 to reduce Sensor-arx startup time.
9. Custom OpenRC configuration aimed at performance to reduce boot time to a minimum.

This setup should achieve maximum security while maintaining a good performance, both in terms of resource usage and image processing speeds.

6.3 Future Work

This thesis does not cover why certain environments outperform each other. Examples of this are Alpine Linux having a lower RAM usage compared to TinyCore, Debian-based environments being able to start the Nix bundle a lot faster than Alpine and TinyCore, as well as DietPi having the fastest boot time.

Another target worthy of investigation is the feasibility of reducing energy consumption during low usage. This could be accomplished using CPU frequency scaling or a standby state. Combined with a customized kernel configuration for increased security and performance, a lot of the metrics described in chapter 5 could be improved.

Bibliography

[1] Anish Athalye, Rumen Hristov, Tran Nguyen, and Qui Nguyen. 2014. Package Manager Security. Course Project. Massachusetts Institute of Technology. https://css.csail.mit.edu/6.858/2014/projects/aathalye-rhristov-viettran-qui.pdf.

[2] CDL Digidow. 2026. Digidow: Christian Doppler Laboratory for Private Digital Authentication in the Physical World. https://www.digidow.eu/. Accessed: 27-02-2026. (2026).

[3] Core Electronics - Jaryk. 2023. Raspberry Pi 5 Vs Raspberry Pi 4 Model B | Comparison and Benchmarking. https://core-electronics.com.au/guides/raspberry-pi-5-vs-raspberry-pi-4-model-b-comparison-and-benchmarking/. Accessed: 03-03-2026. (2023).

[4] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*. USENIX Association, San Antonio, TX, pp. 63–78. https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf.

[5] Julianno Fancisco do Canto Silva. 2022. TinyCore Linux Cheat Sheet. https://gist.github.com/juliannojungle/febb80e292616c65acda65c448f14510. Accessed: 11-03-2026. (July 2022).

[6] Jarrod Farncomb. 2015. Gzip vs Bzip2 vs XZ Performance Comparison. RootUsers. Accessed: March 26, 2026. (September 2015). https://www.rootusers.com/gzip-vs-bzip2-vs-xz-performance-comparison/.

[7] Google LLC. 2024. Get started with the USB Accelerator. https://coral.ai/docs/accelerator/get-started/. Accessed: 03-03-2026. (2024).

[8] Julian Andres Klode. 2011. Debian Bug report logs - #642480: apt-key doesn't check if the key is actually in the keyring. Debian Bug Tracking System. Accessed: 03-03-2026. (2011). https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=642480.

[9] Karl Knechtel. 2025. Recent disruptive changes from Setuptools. *LWN.net*, (May 2025). https://lwn.net/Articles/1020576/.

[10] TinyCore Wiki Maintainers. 2013. The boot process - TinyCore wiki. https://wiki.tinycorelinux.net/doku.php?id=wiki:the_boot_process. Accessed: 18-03-2026. (2013).

[11] National Institute of Standards and Technology. 2011. CVE-2011-3374. National Vulnerability Database (NVD). Accessed: 03-03-2026. (2011). https://nvd.nist.gov/vuln/detail/CVE-2011-3374.

[12] Redox OS Developers. 2024. This Month in Redox - October 2024. https://www.redox-os.org/news/this-month-241031/. Accessed: 05-03-2026. (October 2024).

[13] StephanStS. 2021. Comparison to other Debian based distributions - Why should you use Dietpi? https://dietpi.com/blog/?p=888. Accessed: 11-03-2026. (September 2021).