Implementing Secure QNX OTA Firmware Updates on Raspberry Pi 4 with Post-Quantum Cryptography

Prepared by:
Muntaha Attef, 101236246
Fraser Rankin, 101192297
Jerick Liu, 101225819
Melissa Rand, 101223291
Xuan Nguyen, 101228417

Prepared for:
Jun (Steed) Huang
COMP 4900 E
Carleton University

April 9, 2025

1 Abstract

This paper explores the usage of a quantum-resistant cryptographic hash function into OTA update mechanisms, addressing security concerns in a post-quantum age. An incredibly important component to IOT, automotive, and industrial systems, OTA are vulnerable while using traditional cryptographic hashes due to advancements in quantum computing [1]. With the never ending race between new vulnerabilities and implementing security upgrades, the importance of safe updates has never been greater, and quantum computing can threaten the current model of OTA updates. We propose a framework that would replace the standard cryptographic hash functions that are used today, and instead use a cryptographic hash function that is designed to endure attacks from quantum computing systems as well as traditional attacks.

2 Introduction

OTA Updates

An OTA (Over The Air) update is an update for an embedded system that is sent via a wireless network (WI-FI or Cellular Network). These embedded systems include devices such as cars, IOT devices, mobile devices, and many more. Various components can be updated within an OTA update and are often critical for the peak performance of embedded systems. These OTA updates can be used to update software such as the application software, the configuration settings, the operating software, encryption keys or API keys. This format of update allows for the distribution of wide-spread updates in an efficient and cost effective format, as it minimizes the necessity for client-side operations to be conducted to perform the update. There are also a lot of accessibility options with this style of update, as it has the option for users to choose whether they wish to update or not and there are often easy ways to have the users acquire an older version of an update. OTA updates often need to fit very specific parameters as the embedded systems that are being updated are often incredibly sensitive and can be very vulnerable if the updates are not configured correctly. For example, if the embedded system relies on batteries for its power source, then the OTA updates need to be very energy efficient. OTA updates need to be incredibly precise, and if they are tampered with, it could have devastating effects for the embedded systems [2].

Quantum Resistant Cryptographic Hash

When we send an update via OTA updates, we want our updates to not be modified in transmission, as the updates to embedded systems are often very precise and any modifications can have devastating effects. To prevent modifications by other entities to their updates, software developers will use cryptographic hash functions for integrity checks so that they remain untouched in transmission. Once the software has arrived at the embedded system, it uses a decryption key that it already knows to decrypt the update so it may use the update. As technology develops and the capacity to break these encryptions develops, software developers often will need to update their cryptographic hash functions with more advanced and complicated hash functions. With rapid technological advances, many cryptographers are concerned with the future when our current methods of cryptography are not enough and will be vulnerable to new methods of attack. The Leighton-Micali signature scheme (LMS) is a hash based signature scheme that is often considered for usage when it comes to advanced defenses and many cryptographers believe to be a very sound defense against future quantum computing based attacks [3]. The LMS scheme works by implementing a public key that is then used to sign a second public key that is then distributed with the signatures created with the second key. The LMS scheme is often a favourable pick for advanced schemes as it performs significantly faster than other similar schemes, while keeping similar levels of security [3].

This Project

Our project focuses on the usage of over the air updates with the added security of Post-Quantum Cryptography (PQC) code signing. We aim to create a framework that would allow users to send wide-spread OTA updates to embedded systems in a safe and secure manner, without any future concerns from quantum computing systems. We also want our program to be efficient and to have very little overhead, as any excessive overhead can significantly slow down an OTA update, which negates a lot of the security value. Therefore, we are using a PQC code signing scheme that is significantly faster than most of the other PQC code signing schemes, LMS. LMS is well regarded in the cryptographic community for being one of the fastest PQC code signing schemes and is comparable in speed to standard code signing schemes [3]. This allows us to use PQC signing without needing to worry about excessive overhead. The embedded system we plan to update with our framework, is a Raspberry Pi 4 using QNX 7.1 for its RTOS, this setup is able to be developed at a fairly low cost and is able to run our framework without issue. Hopefully, others can follow suit to our model and framework, so that they too can have a reliable and safe embedded system, without excessive overhead.

Problem Significance

At the rate that quantum computing is developing at, it is a very real possibility that newer systems and frameworks will need to be put in place that will be able to defend against attacks from quantum computing systems [1]. Therefore, for future embedded systems to be able to receive safe and secure OTA updates from the appropriate groups, they need to have a method that is able to resist attacks from quantum computing systems. This is why our proposed framework is of utmost importance, the ability to send an OTA update to an embedded system is crucial to the modern age, and anything that can interrupt that can lead to devastating effects. Without a secure and efficient method to conduct OTA updates to our embedded systems, everyone is potentially vulnerable to attack from a quantum-computing system. Therefore, we need a method to utilize a PQC code signing scheme that is both fast, safe and efficient, hence our framework is vital to safe future OTA updates.

3 Approach

To address the growing need for secure, quantum-resistant OTA update, we developed a flexible and forward-compatible framework tailored to embedded systems.This section outlines the tools and strategies used in our project, key differences from standard OTA implementations, and the rationale behind our design decisions.

Overview

Our framework is deployed on a Raspberry Pi 4 running QNX 7.1, a commercial RTOS widely used in industry. The Raspberry Pi serves an affordable, low-power testbed that represents a class of decises often used in IoT, automotive, and industrial settings.

The goal of our OTA system is to securely deliver software updates wirelessly while ensuring the only verified and authentic packages are applied to the target system. To meet this goal, we used Post-Quantum cryptography (PQC) for update verification, with a focus on minimizing performance overhead and system complexity.

PQC Code Signing

Traditional OTA systems rely on RSA or ECDSA signatures, which are vulnerable to Shor's algorithm once quantum computing becomes scalable [4]. To counter this. Our solution incorporates the LMS scheme, an NIST-approved hash-based signature method. LMS is particularly well suited for OTA updates since it supports fast signature generation and verification, which minimizes the update latency. It is also stateless and provides strong post-quantum security assurances, safeguarding against classical and quantum advisories [3]. LMS was used to sign update packages before transmission. At the target device, the update package was verified using a pre-distributed public key, ensuring the integrity and authenticity of the code.

Secure Update Delivery Pipeline

A core component of our project is the secure delivery of update packages through a simplified OTA pipeline. The process begins with the creation of update packages on a trusted host system. These packages are signed using the LMS signature scheme. Signing occurs before any distribution to ensure that only verified and approved software is deployed to the embedded target device. The signed update is then uploaded to a distribution server, which may be local or cloud-based, depending on the deployment environment.

The embedded system, in this case a Raspberry Pi 4 running QNX 7.1, retrieves the update package using a secure communication channel. Once the package has been downloaded, a built-in verification mechanism checks the digital signature against a known public key that is stored securely on the device. Only if the signature is valid does the system proceed to apply the update using custom update scripts. This pipeline ensures the integrity and authenticity of software throughout the delivery process, effectively mitigating risks such as tampering, replay attacks, or unauthorized update injection.

Efficiency of Resources and Design Trade Offs

When designing OTA frameworks for embedded systems, one of the most important considerations is the efficient use of system resources. These devices often have limited CPU power, memory, and storage space, making the selection of cryptographic schemes and update strategies particularly important. In this project, the LMS signature scheme was chosen not only for its quantum-resistant properties but also for its performance characteristics. LMS offers fast signature generation and verification, which reduces computational load and minimizes latency during the update process. This is especially valuable in real-time or safety-critical systems, where delays in applying updates can lead to operational disruption.

Storage constraints were also a primary concern. Update packages were kept small by only including modified files. Temporary storage usage was optimized to avoid overconsumption of the device's flash memory. Although energy efficiency was not the primary focus of this implementation, the lightweight nature of the LMS scheme naturally lends itself to lower power consumption when compared to more computationally intensive alternatives, such as XMSS or SPHINCS+ [3]. In future iterations, this framework could be extended with energy-aware scheduling to further enhance applicability to battery-operated embedded devices.

Comparing Our Approach with Traditional Systems

Traditional OTA update mechanisms typically rely on classical cryptographic algorithms such as RSA or ECDSA to verify the authenticity of update packages. While these algorithms have served well in the past, they are susceptible to attacks by quantum computers, particularly through Shor's algorithm, which could break their security foundations [4]. Our framework addresses this issue by implementing LMS, a quantum-resistant alternative that maintains strong security guarantees even in the presence of quantum adversaries. This transition ensures that the integrity of updates can be preserved well into the future, even as cryptographic threats evolve.

Beyond the cryptographic layer, our approach also introduces key architectural differences. Most notably, signature verification is performed locally on the embedded device, reducing dependency on external validation and increasing overall system trustworthiness. This decentralization of trust ensures that even if the update distribution server is compromised, malicious updates can be detected and rejected at the edge. Additionally, by maintaining a lightweight footprint and using standard QNX utilities where possible, our implementation integrates seamlessly into existing embedded software stacks without requiring significant reconfiguration or added complexity. As a result, our system not only enhances security but also maintains usability, scalability, and adaptability across a broad range of embedded platforms.

4 OTA Tutorial Guide

A link to a video tutorial can be found here.

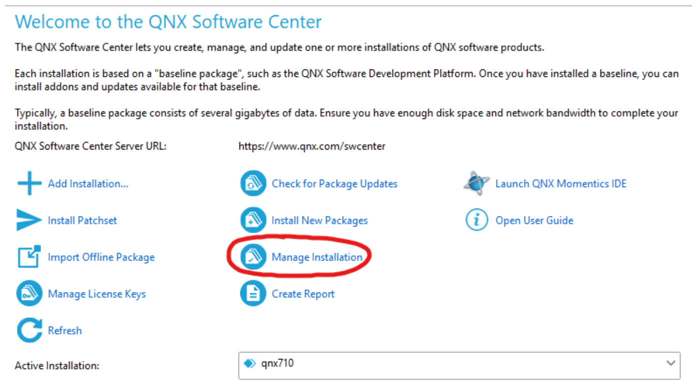
4.1 Hardware & Software Requirements

Github Repo: https://github.com/JerickLiu/RPI4-OTA-with-PQC
Host Server PC: Windows 11 (with WSL Ubuntu)
QNX SDP: Version 7.1 and 8.0 (requires BlackBerry QNX license)
Micro Controller: Raspberry Pi 4
Required Cables/Accessories:

- USB to TTL serial cable (to access the Pi's UART pins for console)
- Micro SD card (8 GB minimum)

4.2 Install QNX SDP and Tools

1. Download and install the QNX Software Center from the official QNX website [5]. After creating an account, request or apply a valid license.
2. Inside the Software Center, select QNX SDP version 7.1 and install the package. Do the same for QNX SDP version 8.0.
3. Under "Manage Installation", you will also need to download the following packages for **QNX version 7.1 only**:
   (a) QNX® SDP 7.1 BSP for Raspberry Pi BCM2711 R-PI4
   (b) QNX® SDP 7.1 Wireless driver for the Broadcom BCM4339 (wpa-2.9)



Welcome to the QNX Software Center

The QNX Software Center lets you create, manage, and update one or more installations of QNX software products.

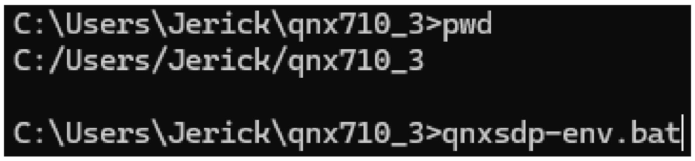
Each installation is based on a "baseline package", such as the QNX Software Development Platform. Once you have installed a baseline, you can install addons and updates available for that baseline.

Typically, a baseline package consists of several gigabytes of data. Ensure you have enough disk space and network bandwidth to complete your installation.

QNX Software Center Server URL: https://www.qnx.com/swcenter

Active Installation: qnx710

4. Locate the download file path for QNX version 7.1 and run the QNX command environment script `qnxsdp-env.bat` on Command Prompt. This ensures `QNX_HOST` and `QNX_TARGET` are set.



```
C:\Users\Jerick\qnx710_3>pwd
C:/Users/Jerick/qnx710_3

C:\Users\Jerick\qnx710_3>qnxsdp-env.bat
```

4.3 Configure the QNX Builds

With the default configuration of QNX, wireless connectivity is unavailable and must be setup in the build file. For simplicity, this build file is available on the Github Repo.

1. Extract the zipped BSP located in the `bsp/` directory then navigate to `bsp/images/`
2. Download the `rpi4.build` from Github and configure the ssid (name) and psk (password) fields with your WiFi details:

   ```
   [uid=0 gid=0 perms=0777] /etc/wpa_supplicant.conf = {
   ctrl_interface=/var/run/wpa_supplicant
   update_config=1
   network=\{
           ssid=""
           psk=""
           key_mgmt=WPA-PSK
           proto=WPA2
           pairwise=CCMP
           group=CCMP
   \}
   }
   ```

3. Using the QNX configured shell that was setup, run:

   ```
   cd bsp/images
   make clean
   make
   ```

   You should see a `ifs-rpi4.bin` generated in the `bsp/images/` directory. This is your QNX IFS image containing Wi-Fi, SSH, and the OTA script.

4. Copy `ifs-rpi4.bin` to your micro SD card.
5. Do the same process for QNX 8.0 using the default `rpi4.build` included with the 8.0 BSP. This will be the firmware we will download OTA. Place this `ifs-rpi4.bin` on your Windows server.

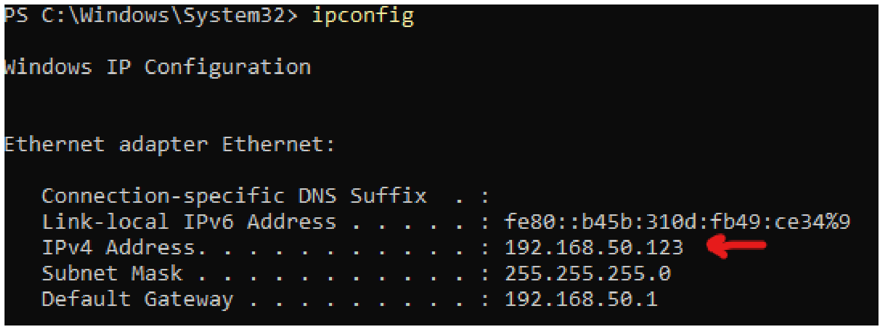
4.4 Windows 11 SSH Setup

1. **Install OpenSSH Server on Windows 11:** Go to *Settings* → *System* → *Optional Features* → *Add an optional feature* → install *OpenSSH Server.*

2. **Enable and Start Services:** In PowerShell (as Administrator), run the following commands:

```
Get-Service ssh-agent, sshd | Set-Service -StartupType Automatic
Start-Service ssh-agent
Start-Service sshd
```

3. **Verify Connection:** Find your server's IP by running `ipconfig` and looking for IPv4 address. Then check the connection by running `ssh yourWindowsUser@YOUR_PC_IP`



```
PS C:\Windows\System32> ipconfig

Windows IP Configuration


Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::b45b:310d:fb49:ce34%9
   IPv4 Address. . . . . . . . . . . : 192.168.50.123
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.50.1
```

4.5 Compile U-Boot

At this point, move over to your WSL Ubuntu environment. To enable advanced boot flow (including fallback logic and signature checks), you need a custom U-Boot build (original U-Boot [6]). We provide a convenient script `build_uboot.sh` on Github to facilitate this process.

1. Make sure you have installed necessary packages:

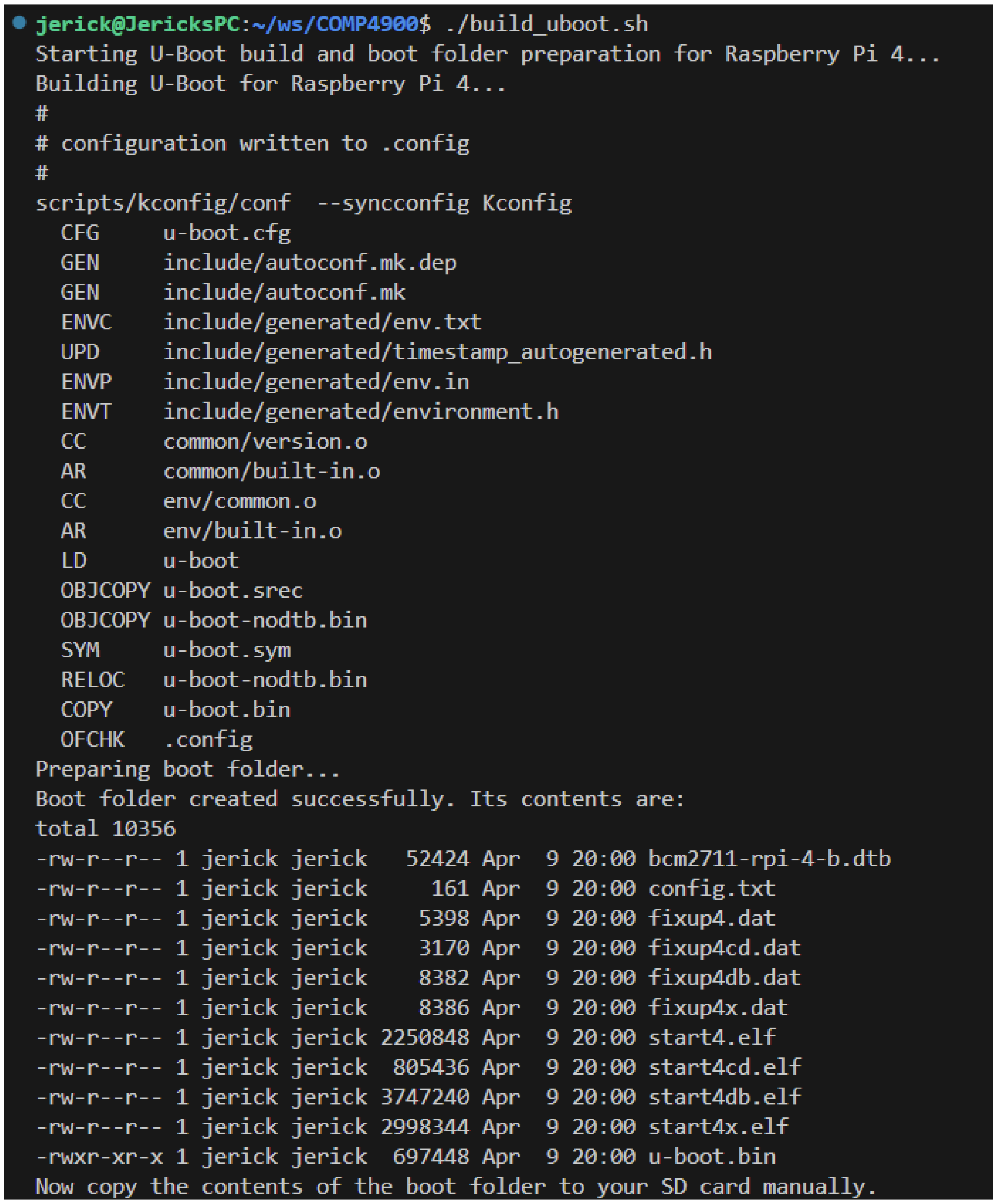
```
sudo apt update
sudo apt install -y git dosfstools gcc-aarch64-linux-gnu make \
  flex bison bc libssl-dev libgnutls28-dev
```

2. **Execute the Script:**

```
./build_uboot.sh
```

This script will clone and build a custom `u-boot` branch at version v2025.01 for the Raspberry Pi 4. It also clones the official Raspberry Pi firmware repository, preparing a `boot/` folder with:

- `u-boot.bin`
- `start4.elf`, `fixup4.dat`, and device tree blobs
- `config.txt` that instructs the Pi to load `u-boot.bin` as the "kernel"



```
jerick@JericksPC:~/ws/COMP4900$ ./build_uboot.sh
Starting U-Boot build and boot folder preparation for Raspberry Pi 4...
Building U-Boot for Raspberry Pi 4...
#
# configuration written to .config
#
scripts/kconfig/conf  --syncconfig Kconfig
  CFG     u-boot.cfg
  GEN     include/autoconf.mk.dep
  GEN     include/autoconf.mk
  ENVC    include/generated/env.txt
  UPD     include/generated/timestamp_autogenerated.h
  ENVP    include/generated/env.in
  ENVT    include/generated/environment.h
  CC      common/version.o
  AR      common/built-in.o
  CC      env/common.o
  AR      env/built-in.o
  LD      u-boot
  OBJCOPY u-boot.srec
  OBJCOPY u-boot-nodtb.bin
  SYM     u-boot.sym
  RELOC   u-boot-nodtb.bin
  COPY    u-boot.bin
  OFCHK   .config
Preparing boot folder...
Boot folder created successfully. Its contents are:
total 10356
-rw-r--r-- 1 jerick jerick   52424 Apr  9 20:00 bcm2711-rpi-4-b.dtb
-rw-r--r-- 1 jerick jerick     161 Apr  9 20:00 config.txt
-rw-r--r-- 1 jerick jerick    5398 Apr  9 20:00 fixup4.dat
-rw-r--r-- 1 jerick jerick    3170 Apr  9 20:00 fixup4cd.dat
-rw-r--r-- 1 jerick jerick    8382 Apr  9 20:00 fixup4db.dat
-rw-r--r-- 1 jerick jerick    8386 Apr  9 20:00 fixup4x.dat
-rw-r--r-- 1 jerick jerick 2250848 Apr  9 20:00 start4.elf
-rw-r--r-- 1 jerick jerick  805436 Apr  9 20:00 start4cd.elf
-rw-r--r-- 1 jerick jerick 3747240 Apr  9 20:00 start4db.elf
-rw-r--r-- 1 jerick jerick 2998344 Apr  9 20:00 start4x.elf
-rwxr-xr-x 1 jerick jerick  697448 Apr  9 20:00 u-boot.bin
Now copy the contents of the boot folder to your SD card manually.
```

4.5.1 Boot Fallback Script (`boot.cmd`)

We also provide a U-Boot boot script (`boot.cmd`) on Github to allow fallback behaviour if the primary firmware fails.

```
if test -z "${boot_img}"; then
    setenv boot_img ifs-rpi4.bin
```

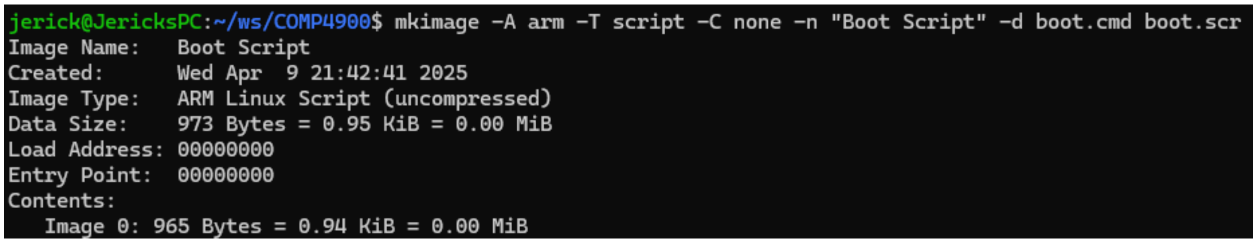
```
fi

echo "DEBUG: boot_img is set to ${boot_img}"
echo "DEBUG: Attempting to load ${boot_img} from MMC..."

if fatload mmc 0 0x80000 ${boot_img}; then
    echo "DEBUG: Successfully loaded ${boot_img}"
    sleep 5
    go 0x80000
else
    echo "DEBUG: Failed to load ${boot_img}"
    if test "${boot_img}" = "ifs-rpi4.bin"; then
        echo "DEBUG: Primary image failed, trying backup..."
        if fatload mmc 0 0x80000 ifs-rpi4.bin.bak; then
            echo "DEBUG: Successfully loaded backup image ifs-rpi4.bin.bak"
            sleep 5
            go 0x80000
        else
            echo "DEBUG: No valid image found (backup failed)"
        fi
    else
        echo "DEBUG: boot_img is not the default and load failed: ${boot_img}"
    fi
fi
```

Compile this `boot.cmd` into a U-Boot script `boot.scr` using the following command:

```
mkimage -A arm -T script -C none -n "Boot Script" -d boot.cmd boot.scr
```



```
jerick@JericksPC:~/ws/COMP4900$ mkimage -A arm -T script -C none -n "Boot Script" -d boot.cmd boot.scr
Image Name:   Boot Script
Created:      Wed Apr  9 21:42:41 2025
Image Type:   ARM Linux Script (uncompressed)
Data Size:    973 Bytes = 0.95 KiB = 0.00 MiB
Load Address: 00000000
Entry Point:  00000000
Contents:
   Image 0: 965 Bytes = 0.94 KiB = 0.00 MiB
```

Copy `boot.scr` to your SD card's FAT partition.

4.6 Generating LMS Keys and Signing Firmware (Optional PQ Security)

If you want to enforce firmware integrity, you can generate LMS or other post-quantum signatures. We provide two C programs to run on WSL Ubuntu:

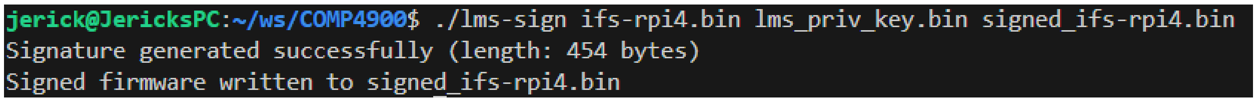
`generate_lms_keys.c` Uses the OQS library [7] to create a public/secret LMS key pair. Compilation example:

```
gcc generate_lms_keys.c -loqs -lcrypto -o generate_lms_keys
./generate_lms_keys
# Outputs lms_pub_key.bin and lms_priv_key.bin
```

`lms-sign.c` Signs the firmware image and appends the signature:

```
gcc lms-sign.c -loqs -lcrypto -o lms-sign
./lms-sign ifs-rpi4.bin lms_priv_key.bin signed_ifs-rpi4.bin
```

Then you can deploy `signed_ifs-rpi4.bin` as the final firmware.



```
jerick@JericksPC:~/ws/COMP4900$ ./lms-sign ifs-rpi4.bin lms_priv_key.bin signed_ifs-rpi4.bin
Signature generated successfully (length: 454 bytes)
Signed firmware written to signed_ifs-rpi4.bin
```

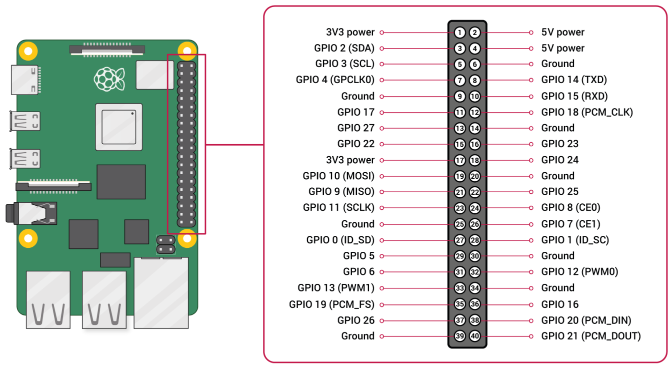
4.7 Prepare Raspberry Pi

1. **Prepare the SD Card:**
   - Copy the `boot/` contents from your U-Boot build (i.e., `u-boot.bin`, firmware blobs, `config.txt`) onto the SD card's FAT partition.
   - Remove any old or temporary boot files.
2. **Review `config.txt`:** This configures the Raspberry Pi. Ensure the details match:

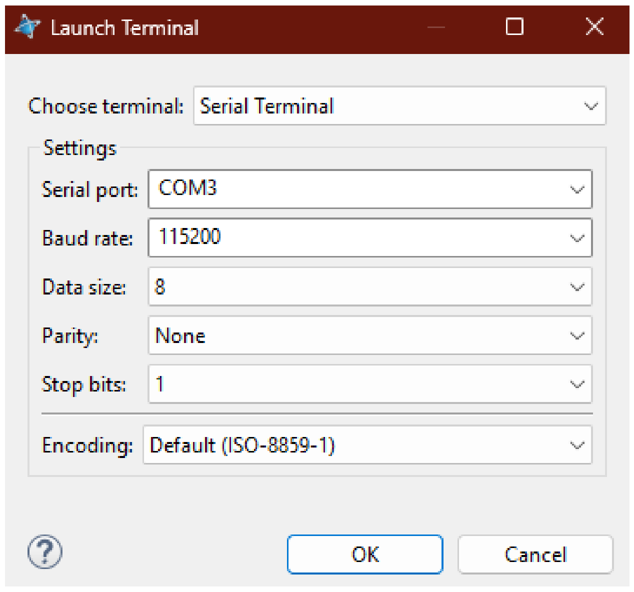
```
arm_64bit=1
cmdline=startup.txt
device_tree=bcm2711-rpi-4-b.dtb
enable_uart=1
force_turbo=1
gpu_mem=16
max_framebuffers=2
kernel=u-boot.bin
```

4.8 Serial Connection and Testing

1. **Physical Wiring:** Connect the TTL-USB cable to the Pi's UART pins (Black to Ground and TX/RX to GPIO 14/15 respectively) and the USB side to your Windows PC [8].



2. **Serial Terminal Settings:** You can use QNX Momentix's serial terminal or a tool like PuTTY. Configure the Baud rate to `115200`, 8 data bits, no parity, 1 stop bit.



Choose terminal: Serial Terminal

Settings

Serial port: COM3

Baud rate: 115200

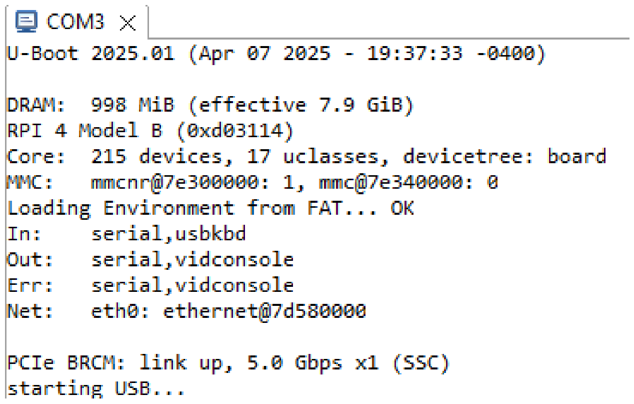
Data size: 8

Parity: None

Stop bits: 1

Encoding: Default (ISO-8859-1)

3. **Observe Boot:** Power on the Pi. You should see U-Boot messages that will immediately load QNX 7.1, thanks to the `boot.scr` script.



```
U-Boot 2025.01 (Apr 07 2025 - 19:37:33 -0400)

DRAM:  998 MiB (effective 7.9 GiB)
RPI 4 Model B (0xd03114)
Core:  215 devices, 17 uclasses, devicetree: board
MMC:   mmcnr@7e300000: 1, mmc@7e340000: 0
Loading Environment from FAT... OK
In:    serial,usbkbd
Out:   serial,vidconsole
Err:   serial,vidconsole
Net:   eth0: ethernet@7d580000

PCIe BRCM: link up, 5.0 Gbps x1 (SSC)
starting USB...
```

4.9 Booting with Auto-Fallback Using `boot.scr`

The U-Boot autoboot and fallback behaviour is fully handled through a script-based approach. We provide a custom U-Boot script `boot.cmd`, which is compiled into a U-Boot-compatible script binary `boot.scr`.

This script ensures the Pi attempts to boot the default firmware image `ifs-rpi4.bin`, and falls back to `ifs-rpi4.bin.bak` if the primary image fails to load.

Once `boot.scr` is on the SD card, U-Boot will automatically execute it on boot. This eliminates the need to manually interrupt the boot process or enter ‘setenv‘ commands.

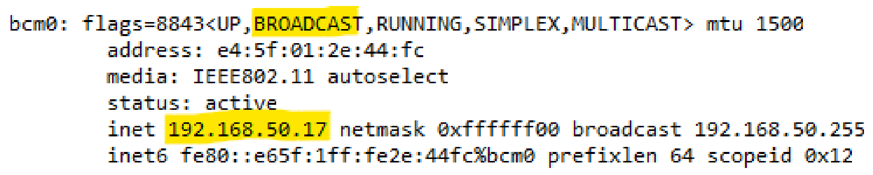
If you want to load a different firmware, interrupt the autoboot by hitting any key, and enter the following commands:

```
setenv boot_img '{FILENAME OF FIRMWARE}'
saveenv
run bootcmd
```

This setup provides resilience in the face of OTA update failures, ensuring the Pi can recover using the backup firmware image if needed.

4.10 Performing the OTA Update

1. **Confirm Pi is on the Network:** The Pi must have an IP address. Run `ifconfig` on the Pi running QNX and locate a broadcast IP address beginning with `192.168.xxx.xxx`



```
bcm0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        address: e4:5f:01:2e:44:fc
        media: IEEE802.11 autoselect
        status: active
        inet 192.168.50.17 netmask 0xffffff00 broadcast 192.168.50.255
        inet6 fe80::e65f:1ff:fe2e:44fc%bcm0 prefixlen 64 scopeid 0x12
```

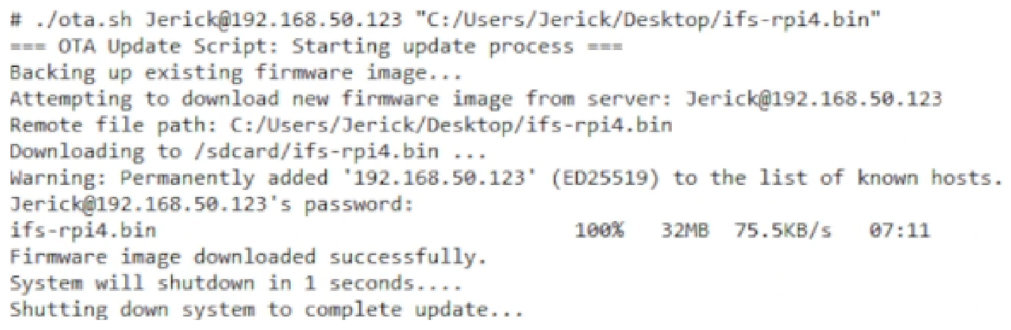
2. **Check the Windows Host:** Ensure `sshd` is running so the Pi can `scp` new images.

3. **Run the OTA Script:** On the Pi's shell:

   ```
   cd /scripts
   ./ota.sh yourWindowsUser@YOUR_HOST_IP C:/path/to/ifs-rpi4.bin
   ```

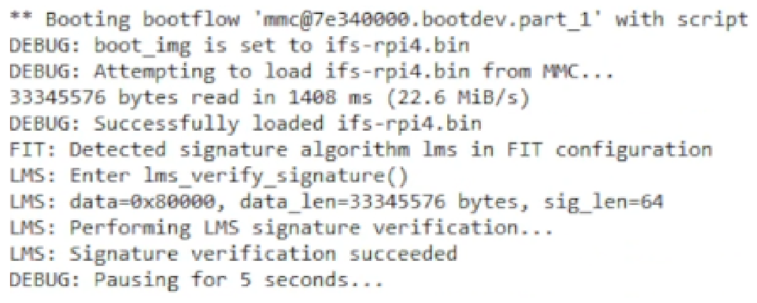
   This will:

   - Backup the old `ifs-rpi4.bin` to `ifs-rpi4.bin.bak`
   - `scp` the new firmware from the host
   - Reboot the Pi if download is successful
   - Fallback to the backup if the download fails

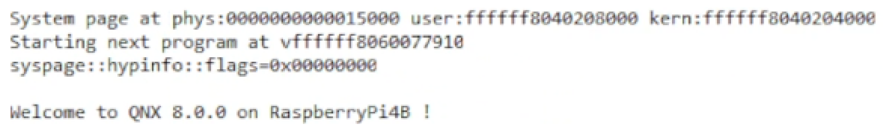


```
# ./ota.sh Jerick@192.168.50.123 "C:/Users/Jerick/Desktop/ifs-rpi4.bin"
=== OTA Update Script: Starting update process ===
Backing up existing firmware image...
Attempting to download new firmware image from server: Jerick@192.168.50.123
Remote file path: C:/Users/Jerick/Desktop/ifs-rpi4.bin
Downloading to /sdcard/ifs-rpi4.bin ...
Warning: Permanently added '192.168.50.123' (ED25519) to the list of known hosts.
Jerick@192.168.50.123's password:
ifs-rpi4.bin                                  100%   32MB  75.5KB/s   07:11
Firmware image downloaded successfully.
System will shutdown in 1 seconds....
Shutting down system to complete update...
```

4. **Observe Reboot and Validate:** After the update, U-Boot will attempt to load `ifs-rpi4.bin` and perform a secure boot check if applicable. If it fails, it loads `ifs-rpi4.bin.bak`.



```
** Booting bootflow 'mmc@7e340000.bootdev.part_1' with script
DEBUG: boot_img is set to ifs-rpi4.bin
DEBUG: Attempting to load ifs-rpi4.bin from MMC...
33345576 bytes read in 1408 ms (22.6 MiB/s)
DEBUG: Successfully loaded ifs-rpi4.bin
FIT: Detected signature algorithm lms in FIT configuration
LMS: Enter lms_verify_signature()
LMS: data=0x80000, data_len=33345576 bytes, sig_len=64
LMS: Performing LMS signature verification...
LMS: Signature verification succeeded
DEBUG: Pausing for 5 seconds...
```



```
System page at phys:0000000000015000 user:ffffff8040208000 kern:ffffff8040204000
Starting next program at vffffff8060077910
syspage::hypinfo::flags=0x00000000

Welcome to QNX 8.0.0 on RaspberryPi4B !
```

The fallback logic in `boot.cmd` and the `ota.sh` script ensures reliability in case of failed updates. This demonstrates a complete end-to-end pipeline for building, signing, and deploying QNX images on the Raspberry Pi.

5 Analysis of the OTA Process and PQC Signing

In this section, we present quantitative and qualitative insights into our OTA workflow on the Raspberry Pi 4 and the impact of incorporating PQC signing. We first summarize the key performance indicators measured on our testbed, then discuss the overhead costs and benefits of secure boot verification, and finally examine the code changes made to U-Boot to enable this functionality.

5.1 Key Performance Indicators (KPI)

Below are the primary metrics we collected during our experiments:

- **Boot Time and Initialization**
  - *Baseline Boot (No Secure Verification):* ~ 3 seconds (from U-Boot start to OS handoff).
  - *With Secure Boot (Verification Enabled):* ~ 13 seconds total.
  - **Boot Time Overhead:** $\sim 330\%$ increase (secure boot adds about 10 seconds).
- **OTA Update Process**
  - *Firmware Download Time:* $\sim$ 7 minutes 13 seconds for a 32 MB image, using `scp` over Wi-Fi.
  - *Overall OTA Duration:* $\sim$ 7–8 minutes. This includes `scp` transfer overhead, slow SD card writes, and any additional time for signature verification.
- **System Resource Utilization**
  - *Code Size Increase:* +50–100 KB in the U-Boot binary (due to LMS/OQS integration).
  - *RAM Overhead:* An additional 20–30 KB used during signature verification.
  - *CPU Utilization (Peak):* 15–20% on the Cortex-A72 cores.
  - *Energy Efficiency Impact:* Slightly higher overall power consumption during boot, driven by extended boot times and cryptographic operations.
- **Performance & Rollback**
  - *Secure Boot Time (Including verification):* $\sim$ 13 seconds from U-Boot start to OS.
  - *Rollback Scenario:* If verification fails, the system immediately attempts to load `ifs-rpi4.bin.bak` (fallback image). This ensures a quick recovery and preserves system responsiveness.

Overall, the largest performance penalty arises from cryptographic operations at boot (an additional 10 seconds beyond the 3-second baseline), and from the `scp`-based OTA image transfer over a relatively slow wireless link. Despite these overheads, the system remains responsive, and the fallback mechanism ensures reliability if verification fails.

The following output shows a snapshot of the system during the OTA script.

```
# top
24 processes; 90 threads;
CPU states: 11.3% user, 1.2% kernel
CPU  0 Idle: 76.5%
CPU  1 Idle: 85.0%
CPU  2 Idle: 99.9%
CPU  3 Idle: 100.0%
Memory: 8128M total, 7720M avail, page size 4K

             Min       Max       Average
CPU 0 idle:   71%       85%       76%
CPU 1 idle:   80%       90%       85%
CPU 2 idle:   99%      100%       99%
CPU 3 idle:   99%      100%      100%
Mem Avail:   7720MB     7720MB    7720MB
Processes:    24        24        24
Threads:      90        90        90
```

5.2 Overhead Analysis of Secure Boot

The measured 330% boot time increase is not unexpected for an embedded system performing on-board post-quantum cryptography. These algorithm operations, while faster than many alternate PQC schemes, do incur additional checks compared to classical schemes like RSA. In our setup, the entire 32 byte hash of the firmware payload is verified against a 256 byte LMS signature using the OQS library. This step takes place within U-Boot before transitioning to QNX. From a functional safety perspective, the trade-off of 10 seconds additional latency is often acceptable, given the significantly strengthened security posture.

5.3 OTA Transfer Times

Transferring the updated firmware image via `scp` took around 7 minutes for a 32 MB binary. This is primarily due to:

- **Wi-Fi Throughput:** The Pi 4's Wi-Fi speed can be modest under certain conditions, particularly if multiple devices share the same network channel.
- **`scp` Encryption Overhead:** `scp` encrypts data in transit using an SSH tunnel, adding CPU overhead on both the Pi and the host.
- **Slow SD Card Writes:** The Pi's SD card interface may throttle writes when the card's cache is flushed frequently.

Despite the extended time, the update remained resilient thanks to the fallback logic in `boot.src` and the `ota.sh` script. If verification fails or the download times out, the system reverts to `ifs-rpi4.bin.bak`.

5.4 Code Changes in U-Boot for LMS Verification

5.4.1 Secure Boot Command: `secureboot`

The main entry point for LMS verification is a new U-Boot command, defined in `cmd/secureboot.c`:

Listing 1: Excerpts of `cmd/secureboot.c`

```
/* secureboot <data_addr> <total_size> */
static int do_secureboot(struct cmd_tbl *cmdtp, int flag, int argc, char *
    const argv[])
{
    /* Parse arguments */
    data_addr = simple_strtoul(argv[1], NULL, 16);
    total_size = simple_strtoul(argv[2], NULL, 16);

    /* Separate the 256-byte signature from the firmware payload */
    payload_size = total_size - SIG_LEN;
    uint8_t *signature = (uint8_t *)(data_addr + payload_size);

    /* Compute the SHA-256 over the main payload */
    sha256_csum_wd((const void *)data_addr, payload_size, hash, 0);

    /* Verify the signature with our newly added lms_verify_signature() */
    ret = lms_verify_signature(hash, sizeof(hash), &pubkey, signature,
       SIG_LEN);
    ...
}
```

This command expects two arguments:

1. `data_addr` - The memory address where `ifs-rpi4.bin` was loaded.
2. `total_size` - The total size of the image, including the appended 256 byte LMS signature.

The code then hashes everything except the signature region and invokes LMS verification.

5.4.2 LMS Library Integration: `lms.c/h`

We introduce a lightweight library in `u-boot/lib/lms` that wraps the OQS library calls for LMS. The `lms_verify_signature()` function in `lms.c` is responsible for creating an OQS signature object and verifying the data against the public key:

Listing 2: Excerpts from `lms.c` using OQS for verification

```
OQS_SIG *lms = OQS_SIG_new("LMS_SHA256_H10_W");
OQS_SIG_free(lms);
...
```

The real cryptographic heavy lifting is handled by the **liboqs** library. We compile `u-boot` with these references, thus increasing the U-Boot binary size by up to 100 KB.

5.4.3 FIT Image Support

We also added a new crypto registration for LMS in `image.c` with the macro `U_BOOT_CRYPTO_ALGO(lms)`, enabling usage of LMS within U-Boot's FIT image signing infrastructure. This is not strictly necessary for the demonstration, but it paves the way for adopting standard U-Boot flows if desired.

5.5 Discussion and Conclusions on PQC Integration

Security vs. Performance Trade-offs Our measurements confirm that integrating secure boot verification, while providing post-quantum security, introduces a noticeable overhead in boot times. For many embedded or safety-critical applications—particularly in automotive or industrial control—an additional 10 seconds at boot may be an acceptable cost if it mitigates the risk of compromised firmware. Systems with more stringent boot-time constraints will need to weigh alternative PQC signature schemes or hardware accelerators.

OTA Reliability The fallback mechanism is crucial to preserving reliability. If the new firmware fails signature validation or is otherwise corrupted during download, the Pi reverts to the backup image. In practice, this approach vastly reduces the risk of bricking the device when updating it remotely. Our demonstration also shows that `scp`-based distribution can be slow. Depending on application needs, a more efficient or resilient transfer protocol could be used.

Resource Footprint and Scalability With an overhead of 20–30 KB of RAM during verification, the system remains quite feasible for the Raspberry Pi 4's hardware. On lower-end devices (with sub-megabyte RAM or older ARM cores), the overhead might become more problematic. Nonetheless, LMS is among the more efficient PQC solutions, and future hardware with dedicated cryptographic engines may further reduce the bottleneck.

Final Remarks Overall, our measurements show that integrating quantum-resistant LMS code signing into a QNX-based OTA pipeline is both viable and effective. Despite the moderate performance costs, the enhanced security posture, resilience against both classical and quantum adversaries, makes it an attractive option for next-generation embedded systems requiring long product lifetimes and robust supply chain security.

6 Future Work

While the current implementation provides a strong foundation for secure, quantum-resistant OTA updates on embedded systems, several promising directions exist for future development and refinement. One of the most impactful improvements would be the integration of this framework into a fully automated CI/CD (Continuous Integration/Continuous Deployment) pipeline. Such a pipeline could facilitate end-to-end automation, from building and signing update packages to securely deploying them across multiple devices, enabling scalable management of embedded systems in production environments. This would also improve reliability and reduce the potential for human error during the update process.

Another area for exploration is support for multiple PQC signature schemes. While LMS is currently the most efficient and practical for our use case, other NIST-approved candidates, such as XMSS and SPHINCS+, offer alternative security models and performance characteristics [3]. Adding modular support for different PQC schemes would allow developers to tailor the security and performance trade-offs to their specific application needs. This flexibility would be particularly useful in environments with diverse hardware constraints or varying threat models.

In addition to expanding cryptographic options, enhancing the functionality of the rollback mechanism presents an opportunity for future improvement. Our current implementation includes a fallback strategy using a backup firmware image (`ifs-rpi4.bin.bak`) that is automatically restored if signature verification or image loading fails. This ensures that the system remains bootable even in cases of corrupted downloads or verification failure. However, the rollback is file-based and limited to a single fallback image. To further strengthen reliability, especially under harsh or failure-prone conditions (e.g., power loss during flashing), a more advanced rollback system could be introduced. One potential enhancement would be a dual-partition scheme where updates are written to an inactive partition, only switching to it after successful verification and testing. Another option would be to implement a file system snapshot mechanism, allowing the system to revert to a known-good state in case of mid-update failures. These additions would elevate the system's resilience, making it more suitable for mission-critical deployments.

Finally, the system could benefit from enhanced monitoring and telemetry features. These would provide visibility into update status, failure rates, performance metrics, and possible intrusion attempts. Logging and reporting tools could be developed to support forensic analysis and compliance audits, especially in regulated industries like automotive and healthcare. Additionally, expanding testing to a wider variety of hardware platforms would help assess the portability and scalability of the framework. This would confirm its applicability in more resource-constrained or specialized embedded systems, increasing its value across a broader set of use cases.

References

[1] A. Truskovsky, “Do you know about quantum’s threat to ota software updates?” *ISARA Corporation Blog*, Oct. 2017, Accessed: April 2, 2025. [Online]. Available: https://www.isara.com/blog-posts/quantums-threat-ota-software-updates.html.

[2] S. Mahmood, H. N. Nguyen, and S. A. Shaikh, “Systematic threat assessment and security testing of automotive over-the-air (ota) updates,” *Vehicular Communications*, vol. 35, p. 100 468, 2022, ISSN: 2214-2096. DOI: https://doi.org/10.1016/j.vehcom.2022.100468. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214209622000158.

[3] D. A. Cooper, D. C. Apon, Q. H. Dang, M. S. Davidson, M. J. Dworkin, and C. A. Miller, “Recommendation for Stateful Hash-Based Signature Schemes,” National Institute of Standards and Technology, Tech. Rep. NIST Special Publication 800-208, Oct. 2020, Accessed: April 2, 2025. DOI: 10.6028/NIST.SP.800-208. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-208.pdf.

[4] T. Gagliardoni, “Quantum attack resource estimate: Using shor’s algorithm to break rsa vs dh/dsa vs ecc,” *Kudelski Security Research*, Aug. 2021, Accessed: April 2, 2025. [Online]. Available: https://research.kudelskisecurity.com/2021/08/24/quantum-attack-resource-estimate-using-shors-algorithm-to-break-rsa-vs-dh-dsa-vs-ecc/.

[5] QNX Software Systems, *QNX Software Center*, https://www.qnx.com/download/group.html?programid=29178, Accessed: April 2, 2025.

[6] U-Boot Development Team, *U-Boot Source Repository*, https://source.denx.de/u-boot/u-boot, Accessed: April 2, 2025.

[7] Open Quantum Safe Project, *Open Quantum Safe*, https://openquantumsafe.org/, Accessed: April 2, 2025.

[8] Raspberry Pi Ltd, *Raspberry Pi Documentation: Raspberry Pi Hardware*, Accessed: April 2, 2025, 2024. [Online]. Available: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html.