Impact of Operating Systems on Edge Device: Benchmarking Performance, Reliability, and Post-Quantum Readiness

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Cyber Security
May 2025
Fabien Guihard

Supervisors:
Tahir Mohammad
Jouni Isoaho
Nicolae Paladi

UNIVERSITY OF TURKU
Department of Computing

FABIEN GUIHARD: Impact of Operating Systems on Edge Device: Benchmarking Performance, Reliability, and Post-Quantum Readiness

Master of Science (Tech) Thesis, 100 p., 5 app. p.
Cyber Security
May 2025

As edge computing advances, the growing threat of quantum computers is driving public-key cryptographic standards toward deprecation by 2030. Many studies focus on post-quantum algorithms for constrained devices, and others tackle scalable edge management, yet very few examine the edge operating system as the point where performance, reliability, and post-quantum security converge.

This thesis begins by outlining a framework of key requirements for reliable and scalable edge operating systems, then addresses the research gap through an empirical comparison of three OS architectures (mutable, immutable, and container-based) using six Linux distributions deployed on a Raspberry Pi 4B edge node. A reproducible testbed was used to conduct three evaluations. The first measured system-level performance, including CPU, RAM, and disk I/O. The second assessed the operating system's resilience during system updates interrupted by power loss. The third evaluated the latency of two NIST post-quantum cryptography standards: ML-KEM and ML-DSA.

Statistical analysis revealed that the immutable design of NixOS provides the best overall balance, offering top-tier throughput with atomic and declarative updates that withstood every fault-injection scenario. The container-based system openSUSE MicroOS matched NixOS in automated rollback capability and passed every power-failure test, yet introduced higher PQC latencies and disk-I/O overhead. Mutable systems (Ultramarine Linux, Manjaro ARM, DietPi, Raspberry Pi OS Lite) showed varied performance strengths but shared a critical weakness. Ultramarine delivered the fastest post-quantum operations, Manjaro offered a well-rounded balance, DietPi excelled in RAM throughput thanks to its minimal footprint, and Raspberry Pi OS Lite provided a stable baseline. Yet every one of these mutable distributions failed to reboot cleanly after power-cut interruptions and required manual repair.

These results confirm that OS architecture directly affects system performance, operational resilience, and the runtime cost of quantum-safe cryptography. OS choice must be treated as a primary design decision for future edge deployments rather than a question of convenience or familiarity.

Keywords: Edge Computing, Operating System, Linux, Performance, Security, Scalability, Reliability, Post-Quantum Cryptography, Resource Efficiency, Raspberry Pi

Acknowledgments

This master's thesis was conducted within the framework of *Horizon Europe*, the European Union's key funding program for research and innovation, which supports groundbreaking projects aimed at tackling climate change, boosting competitiveness, and fostering sustainable growth [1]. Within this program, the thesis is part of *GLocalFlex*, a collaborative project focused on improving the flexibility, efficiency, and resilience of European energy systems [2]. Contributing to this initiative from a cybersecurity perspective has been a highly rewarding experience, allowing me to engage in meaningful research, deepen my knowledge of the field, and connect with experts across Europe.

I cannot overstate the fact that none of this would have been possible without my primary advisor, Nicolae Paladi, to whom I am profoundly grateful. His support, mentorship, and trust in me have been essential not only for the completion of this thesis but also throughout my master's degree journey. His guidance has left a lasting impact on my growth, for which I will always be deeply thankful.

I would also like to warmly thank Riccardo Silvestri and Pascal Chaussumier, whose dedication to the project has been a key factor in the success of this thesis. Their involvement made this thesis both possible and genuinely enjoyable to be part of.

Finally, I appreciate my supervisors at the University of Turku, Tahir Mohammad and Jouni Isoaho, for their crucial guidance in shaping this thesis.

Contents

List of Figures

List of Tables

List of acronyms

ANSSI French National Agency for the Security of Information Systems

COS Container-based Operating System

CoW copy-on-write

CPU Central Processing Unit

CV Coefficient of Variation

DH Diffie-Hellman

ECC Elliptic curve cryptography

ENISA The European Union Agency for Cybersecurity

FIO Flexible I/O Tester

I/O Input/Output

IOS Immutable Operating System

IoT Internet of Things

IQR Interquartile Range

LBC Lattice-based Cryptography

MB Megabyte

MBW Memory Bandwidth

ML-DSA Module-Lattice-Based Digital Signature Algorithm

ML-KEM Module-Lattice-Based Key-Encapsulation Mechanism

MOS Mutable Operating System

NIST National Institute of Standards and Technology

OS Operating System

OTA Over-the-Air

PQC Post-Quantum Cryptography

RAM Random-access Memory

RPi OS Lite Raspberry Pi OS Lite

RSA Rivest-Shamir-Adleman

RTOS Real-time Operating System

SBC Single-board Computer

SD Standard Deviation

SIMD Single instruction, multiple data

SLH-DSA Stateless Hash-Based Digital Signature Algorithm

SOMs System-on-Modules

SSH Secure Shell Protocol

THP Transparent Huge Pages

TLB Translation Lookaside Buffer

1 Introduction

1.1 Motivation

In recent years, the computing landscape has undergone a notable transformation, moving away from centralized cloud infrastructures toward more distributed edge computing models [3]. Research indicates that this trend is expected to continue, with **edge computing playing an increasingly prominent role in the future** [4]. According to a recent report by Grand View Research, the edge computing market was valued at $16.45 billion in 2023 and is projected to grow at a compound annual growth rate of 36.9% through 2030 [5]. This ever-growing paradigm is gaining popularity, as it enables data processing to occur closer to where it is generated (at the "edge" of the network), reducing latency, improving responsiveness, and lowering bandwidth demands. As a result, edge computing has gained traction in sectors such as energy, manufacturing, transportation, and smart cities, where real-time data processing and local autonomy are essential [6].

The proliferation of embedded devices has led to the deployment of millions of edge nodes in increasingly complex and large-scale infrastructures [7]. These devices are often physically remote, resource-constrained, and deployed in environments where manual maintenance is difficult or costly [8][9]. Ensuring long-term reliability, maintainability, and security for such distributed systems places significant responsibility on the underlying software stack, particularly the **operating system**.

At the same time, the security landscape is undergoing a profound transformation. Ad-

vances in **quantum computing** pose a serious threat to classical cryptographic algorithms. In November 2024, the U.S. National Institute of Standards and Technology (NIST) announced a definitive timeline for phasing out traditional encryption methods. Widely adopted schemes such as RSA, DH, ECDSA, EdDSA, and ECDH are set to be officially **deprecated by 2030** and disallowed by 2035 [10]. This accelerated timeline is driven not only by the expected emergence of practical quantum capabilities in the 2030s but also by the growing concern over **"harvest now, decrypt later" attacks**, in which encrypted data is collected today for future decryption once quantum technology matures [11].

This growing urgency is mirrored by a sharp rise in global interest in post-quantum cryptography. As illustrated in Figure 1.1, Google Trends data shows a dramatic increase in worldwide search popularity for the term “Post-Quantum Cryptography” (PQC) over the past few years, with a particularly steep climb starting around 2022. This surge reflects both academic and industrial attention converging on the topic, underscoring its critical relevance in securing the future of digital infrastructure.

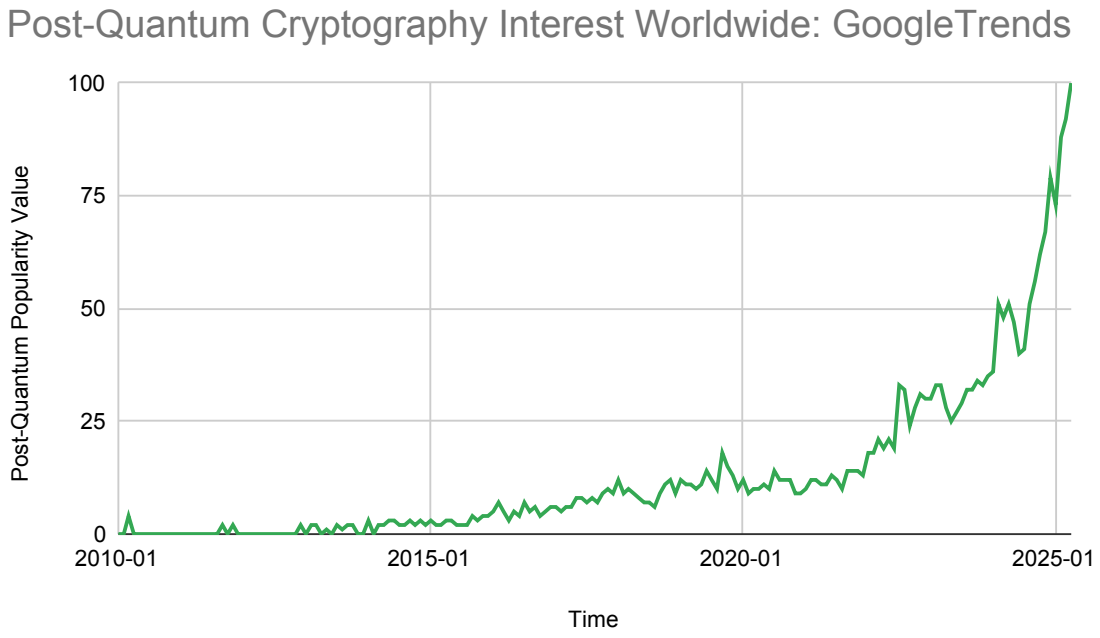

Figure 1.1: Post-Quantum Interest Worldwide by Google Trends

1.2 Problem Statement

As edge devices spread across critical infrastructure, their security and maintainability become harder to ensure. Distributed and often unattended, these systems require operating systems that can manage critical workloads, support reliable updates, and maintain integrity over long lifespans.
In large-scale deployments involving thousands or even millions of devices, small inefficiencies or vulnerabilities can lead to significant operational risks. Failures during over-the-air (OTA) updates, configuration drift, or the absence of rollback mechanisms may result in inconsistent device states or service interruptions. Since edge devices are often expected to operate for a decade or more, the OS must provide a stable and maintainable environment that can adapt to evolving security requirements.

A major shift on the horizon is the transition to post-quantum cryptography. With classical algorithms projected to become obsolete by the 2030, devices deployed today must be ready to adopt quantum-resistant standards. Migrating to PQC is a long and complex process with significant implications for operating system design and hardware compatibility. Furthermore, past transitions have been slow; for example, it took more than a decade for many industries to move from SHA-1 to SHA-2 [12]. Given the accelerating timeline of quantum threats, proactive preparation is now essential.

Despite the growing diversity of Linux-based operating systems, there is a striking lack of systematic evaluation tailored to the needs of secure large-scale edge environments. Existing literature often focuses on general-purpose computing or security in isolation, without accounting for the specific operational constraints of edge deployments. These constraints include limited computing resources, unreliable connectivity, unattended updates, and increasing cryptographic complexity. As a result, system architects are left without clear and evidence-based guidance to compare operating system architectures in terms of security, resilience, maintainability, and cryptographic flexibility. This absence of actionable knowledge presents a critical blind spot. Making the wrong OS choice is not a theoretical concern; it can result in unpatchable vulnerabilities, broken update

mechanisms, operational downtime, or an inability to meet future cryptographic standards. In an era where millions of edge devices may remain in the field for a decade or more, the long-term impact of this foundational decision is substantial and must not be underestimated.

1.3 Research Questions

This thesis aims to address three key research questions:

- **RQ1**: To what extent has current research explored the interplay between edge computing, operating system design, and readiness for the post-quantum cryptographic transition anticipated by 2030?
- **RQ2**: What are the critical operational factors to consider when selecting an operating system for scalable edge deployments?
- **RQ3**: How do different operating system architectures compare in real-world edge environments in terms of performance, update robustness, and PQC workload handling?

The three research questions are designed to complement each other by addressing the topic from different angles. RQ1 is conceptual and literature-driven, aiming to define the core challenges and evaluate the existing design space. RQ2 is strategic and comparative, focusing on the development of a decision-making framework. RQ3 is experimental, involving hands-on evaluation.

1.4 Research Objectives

To address the research questions outlined above, this thesis pursues the following objectives, structured across analytical, evaluative, and experimental dimensions:

The first objective is to **Map the Current Research Landscape.** The idea is to conduct a structured literature review to assess how current research addresses the intersection of edge computing, operating system design, and PQC readiness, identifying key

themes and gaps.

The second objective is to **define OS selection criteria** by establishing operational requirements and evaluation factors relevant to edge scenarios, including resource constraints, update mechanisms, immutability, and cryptographic agility.

The third objective is to **evaluate OS architectures** by benchmarking selected Linux-based systems on a Raspberry Pi 4B, assessing CPU, RAM, disk I/O, update robustness, and PQC workload handling through reproducible tests.

The fourth objective is to **provide deployment insights** by synthesizing results into practical guidance for selecting or designing OSs suited for secure, scalable, and PQC-ready edge environments.

1.5 Thesis Organization

The thesis is structured as follows. **Chapter 2** introduces edge computing concepts, architectures, and hardware platforms. **Chapter 3** reviews the state of research on PQC, edge computing, and operating systems, identifying key gaps and challenges. **Chapter 4** defines edge-specific OS requirements, classifies OS architectures, and proposes a selection framework. **Chapter 5** outlines the research methodology and benchmarking framework. It describes the OS candidates, the testbed setup, and the execution environment. **Chapter 6** presents system-level benchmarks (CPU, RAM, disk I/O) across OSs. **Chapter 7** evaluates OS update resilience under power failure scenarios. **Chapter 8** assesses the performance of PQC primitives across different OSs under load. **Chapter 9** synthesizes the findings and discusses their implications in the broader research landscape. It revisits the research questions and outlines the contribution to the field. **Chapter 10** concludes the thesis and presents limitations and directions for future work.

2 Overview of Edge Computing

2.1 Fundamentals

Edge computing is a computing paradigm that brings processing, storage, and network resources closer to the source of data, typically at the edge of the network [13]. Although definitions vary across the literature, the core idea remains consistent: edge computing shifts cloud-like capabilities such as computation, storage, and intelligence closer to end devices like sensors or mobile units. Researchers emphasize the importance of proximity in both geographic and network terms [14]. Industry definitions, such as that from China's Edge Computing Industry Alliance, also highlight its ability to support real-time processing, data optimization, and privacy-sensitive applications [15].

Edge computing is commonly described using a three-layer architecture: the cloud layer, the edge layer, and the terminal (or perception) layer, as illustrated in Figure 2.1 [16][17]. The **terminal layer** includes end devices such as sensors, smart plugs, smartphones. These devices act as both data producers and consumers, focusing on sensing rather than computation. They collect raw data and forward it to upper layers for processing [14]. The upper layer is the **edge layer**, which acts as the intermediary between terminals and the cloud. It includes infrastructure like gateways, and is responsible for local data processing and real-time analysis. Its proximity to end devices enables lower latency. The **cloud layer** provides centralized, large-scale computing and storage. It handles tasks requiring intensive computation, long-term storage, and global data integration. It also supports the edge layer by updating policies and algorithms as needed. While

this layered architecture offers many advantages, it also introduces a number of technical and operational challenges that must be addressed to ensure effective and secure edge deployments.

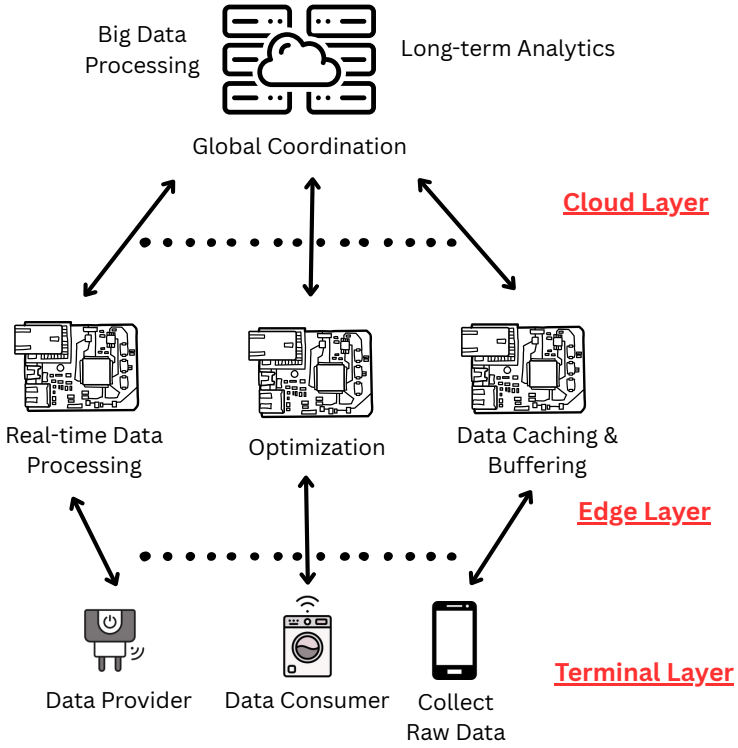

Figure 2.1: Edge Computing Architecture Overview

2.2 Challenges

A defining characteristic of edge environments is their inherent constraints. Edge devices are typically built on low-power, low-cost hardware platforms, such as the Raspberry Pi, which provide **limited computational resources**, memory, and storage compared to centralized servers [18]. For example, the Raspberry Pi 4 Model B offers up to 8 GB of RAM, a quad-core ARM Cortex-A72 processor running at 1.5 GHz, and microSD-based storage. While these specifications are adequate for lightweight tasks and general-purpose workloads, they quickly become a limiting factor when dealing with modern compute-intensive applications such as advanced deep learning inference and large language models, which have been increasingly leveraged by both industry and research in recent years [19]. As a result, significant efforts are being made to optimize these kind of workloads

to fit within the tight resource constraints of edge computing environments.

These devices are also frequently deployed in **harsh or remote environments** where physical access is limited or entirely impractical [20]. In such cases, making software robustness and reliable update mechanisms is more than essential.

Connectivity at the edge is also frequently intermittent or **unreliable** [21]. Devices may depend on cellular networks, local mesh networks, or other communication methods that can introduce variable bandwidth, latency, or disconnections. In smart home environments, for example, it is common for users to disconnect power and network connections entirely when leaving for vacations. As a result, edge systems must be capable of tolerating connectivity loss and continuing to perform critical functions autonomously, without relying on constant external access.

In addition, edge deployments are growing significantly in scale. In modern energy systems or smart infrastructure, it is increasingly common to manage thousands or even millions of edge nodes. At this scale, **small inefficiencies can quickly escalate** [22]. To remain manageable and secure over time, systems must support automated updates, rollback capabilities, and scalable configuration management [23].

These constraints and deployment realities place specific demands on the software stack, especially the operating system. The OS must enable secure and reliable operation in decentralized, resource-limited, and intermittently connected environments. These requirements frame the focus of this thesis, which investigates how operating systems can address the long-term challenges of secure, scalable, and future-ready edge computing, particularly in the context of the approaching transition to PQC. Addressing these challenges also depends on the underlying hardware, as different platforms offer varying capabilities and constraints. As such, hardware selection plays a critical role in the design and performance of edge systems.

2.3 Typical Edge Hardware

A wide variety of single-board computers (SBCs) and system-on-modules (SoMs) are used in edge environments, ranging from consumer-grade devices to industrial-grade embedded systems. Notable examples include the Raspberry Pi, NVIDIA Jetson series, BeagleBone, and Intel NUCs [24]. These platforms differ in processing power, energy efficiency, hardware acceleration support (e.g., for AI workloads), and I/O options, enabling developers to tailor solutions to a wide range of edge scenarios, including home automation and smart grid control systems.

Among these, the Raspberry Pi stands out as one of the most widely adopted edge computing platforms [25]. Its popularity stems from several factors: low cost, compact size, energy efficiency, and strong community support. Extensive documentation, prebuilt software images, and an active open-source ecosystem make the Raspberry Pi particularly suitable for prototyping and deployment in both academic research and commercial applications [26]. As such, it plays a foundational role in edge computing development and experimentation across diverse domains. **In this thesis, we will use the Raspberry Pi 4 Model B as the reference edge platform for experimentation and evaluation**.

3 Landscape of Edge OS and Post-Quantum Cryptography

The convergence of **PQC**, **edge computing**, and **operating system** design is an emerging research area. Each of these domains is well-studied individually, but literature combining all three remains sparse. In fact, a recent survey from February 2024 describes itself as "the first review study" dedicated entirely to edge computing security in the post-quantum era [27]. This suggests that discussions involving PQC, edge computing, and operating systems together are rare, with most research focusing on two-way intersections such as PQC and edge, or edge and OS, rather than addressing all three simultaneously. Below, we survey how these three areas intersect, the extent of post-quantum readiness in edge OS architectures, leading research efforts, and remaining gaps in the literature.

3.1 Current State of PQC

3.1.1 Overview of PQC

The emergence of quantum computing poses a significant threat to current cryptographic systems, particularly those based on asymmetric cryptography (public-key). While quantum computing promises to revolutionize fields such as drug discovery, climate modeling, and complex optimization, it also introduces serious challenges to cybersecurity. A key concern is **Shor's algorithm**, introduced in the mid-1990s, which allows quantum computers to efficiently solve the integer factorization and discrete logarithm problems [28].

These problems form the mathematical foundation of nearly all widely deployed public-key cryptosystems, including RSA and Elliptic Curve Cryptography (ECC). If large-scale quantum computers become a reality, they could render these systems insecure, compromising the confidentiality and authenticity of communications and digital assets across the globe. Many forms of sensitive data such as classified documents, medical records, or infrastructure control messages must remain secure for decades. Moreover, cryptographic transitions are complex and slow-moving processes, often requiring years of standardization, testing, and implementation. Therefore, the quantum threat must be addressed proactively [29]. In response to this looming threat, the field of PQC has emerged, focusing on developing cryptographic algorithms that remain secure against both quantum and classical attack [30].

It is important to note, however, that not all cryptographic techniques are equally vulnerable. Symmetric cryptography, such as AES or ChaCha20, is only mildly affected by quantum algorithms like Grover's algorithm, which offers a quadratic speedup. This can be mitigated by simply doubling key lengths (e.g., using AES-256 instead of AES-128), preserving their long-term viability [29]. Given the heightened threat to public-key cryptography, NIST began standardizing post-quantum algorithms.

3.1.2 PQC Standardization

In August 2024, NIST released the final versions of three PQC standards [31], which are summarized in Figure 3.1:

- **FIPS 203**: Based on the CRYSTALS-Kyber algorithm, now referred to as ML-KEM (Module-Lattice-Based Key Encapsulation Mechanism), intended for general encryption purposes [32].
- **FIPS 204**: Utilizing the CRYSTALS-Dilithium algorithm, renamed ML-DSA (Module-Lattice-Based Digital Signature Algorithm), designed for digital signatures [32].
- **FIPS 205**: Employing the Sphincs+ algorithm, now called SLH-DSA (Stateless Hash-Based Digital Signature Algorithm), also aimed at digital signatures and serving as a backup method in case vulnerabilities are discovered in ML-DSA [33].

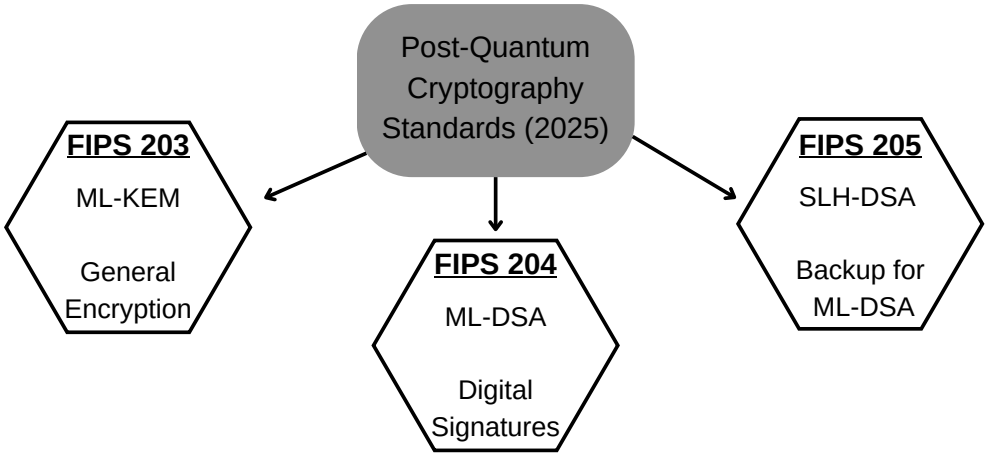

Figure 3.1: Post-Quantum Cryptography Standards in April 2025

These standards provide a foundation for organizations to begin transitioning their cryptographic infrastructures to quantum-resistant algorithms.

However, NIST has also recognized the need for cryptographic diversity, particularly in the event that vulnerabilities are discovered in the current frontrunners. For that reason, several additional algorithms are still under consideration in the fourth round of the NIST PQC standardization process, either as alternate candidates or to diversify algorithmic assumptions.

Other Leading PQC Algorithms under Consideration

- **FALCON**: The next algorithm likely to be standardized [34]. It is a lattice-based digital signature scheme offering smaller signature sizes than Dilithium. Although it was not selected as the primary digital signature standard, it is considered a strong alternative and is expected to be standardized in the near future to support use cases requiring compact signatures [35].
- **BIKE** (Bit Flipping Key Encapsulation): A code-based key encapsulation mechanism, part of NIST's alternate candidates. BIKE is valued for its simplicity and different underlying assumptions compared to lattice-based schemes, but remains under evaluation as of April 2025 [36].
- **Classic McEliece**: Another code-based KEM based on the hardness of decoding a random linear code. It is notable for its long public keys but fast decryption, making it suitable for certain specialized use cases [37]. Classic McEliece is currently under

consideration for standardization but has not yet been finalized.

These and other algorithms are still in the pipeline for potential standardization. Their inclusion aims to ensure long-term security by reducing reliance on any single class of mathematical problems.

The four main categories of PQC schemes

PQC algorithms are broadly categorized into four main families, each defined by the mathematical problems they are based on [27]. Each category presents unique trade-offs in terms of performance, key and signature sizes, and implementation complexity [38].

The first and most prominent category is **Lattice-Based Cryptography**, which relies on the hardness of problems such as Learning With Errors (LWE), Ring-LWE, Module-LWE, and Short Integer Solutions (SIS) [39]. This family has gained popularity due to its strong balance between security and efficiency, its relatively simple and secure implementation, and well-understood theoretical foundations [40]. Among the most notable algorithms in this group are Kyber (ML-KEM), selected by NIST in 2024 for key encapsulation, and Dilithium (ML-DSA) for digital signatures. These algorithms are well-suited for general-purpose encryption and signing, including in embedded and high-performance systems [41].

The second category, **Code-Based Cryptography**, is grounded in the difficulty of decoding random linear error-correcting codes, such as the Syndrome Decoding problem [42]. Known for its resilience against cryptanalysis over decades, this category is especially attractive for systems requiring small public keys [43]. Two notable examples include BIKE (Bit Flipping Key Encapsulation), a NIST finalist, and Classic McEliece, which, despite its large public keys, offers extremely fast decryption. Code-based schemes are particularly applicable to long-term secure messaging and devices with ample flash memory.

Hash-Based Cryptography forms the third category and is built entirely on the security of cryptographic hash functions. It is considered highly conservative and well-understood, as it avoids number-theoretic assumptions altogether [44]. Stateless variants

of these schemes are especially secure, as they eliminate the risk of private key reuse. A key example is SPHINCS+ (SLH-DSA), a NIST-standardized signature algorithm. However, hash-based approaches typically involve larger signature sizes (ranging from 8 to 17 KB) and slower performance compared to lattice-based signatures [45]. These characteristics make them ideal for software and firmware signing, where verification is more critical than signature size.

The final category is based on **Multivariate Quadratic** (MQ) Equations, which involves solving systems of quadratic equations over finite fields [46]. These schemes were historically valued for their efficiency in digital signatures. However, interest in this category has declined due to serious security and performance issues [47]. Algorithms such as Rainbow, once a finalist, have since been broken, and GeMSS was withdrawn during the standardization process. As a result, no MQ-based algorithms remain in the current set of NIST PQC candidates.

In parallel to NIST's efforts, and more specifically within Europe, the French National Agency for the Security of Information Systems (ANSSI) has outlined **a phased approach** to the PQC transition. According to ANSSI's 2023 position paper, the agency plans to issue the first security visas for products implementing hybrid post-quantum cryptography around 2024-2025 [48]. This initial phase involves integrating PQC alongside classical algorithms to ensure a smooth and secure transition.

The migration to PQC is a complex and resource-intensive process. NIST's draft report, IR 8547, emphasizes the need for organizations to begin planning their transition strategies promptly. The report suggests a target date of **2035** for completing the migration to post-quantum cryptography, aligning with the anticipated timeline for the emergence of quantum computers capable of breaking current cryptographic systems [10].

3.1.3 Cryptographic Agility

Cryptographic agility, or crypto-agility, refers to the capability of a security system to swiftly adapt its cryptographic algorithms and protocols in response to emerging threats or advancements in technology [49]. This adaptability is crucial in the context of PQC,

as it enables organizations to replace vulnerable algorithms with quantum-resistant alternatives without necessitating extensive modifications to existing infrastructures [50]. A crypto-agile system allows for the seamless integration of new cryptographic standards, ensuring sustained security and compliance [51].

The importance of cryptographic agility is underscored by the dynamic nature of the cybersecurity landscape. As new vulnerabilities are discovered and technological advancements occur, the ability to rapidly update cryptographic mechanisms becomes essential. Implementing crypto-agility involves designing systems with modular architectures, maintaining an up-to-date inventory of cryptographic assets, and establishing processes for the timely deployment of updates [52].

3.1.4 Challenges for Edge Devices

Transitioning to PQC presents significant challenges, particularly for resource-constrained devices commonly found in edge computing environments [53]. While high-powered computers can handle these algorithms efficiently, many PQC algorithms involve larger key sizes and more complex computations than classical ones, resulting in higher demands on processing power, memory, and energy consumption [54]. Edge devices may struggle to meet these heightened requirements, potentially impacting their performance. One important example of these constraints is the size of digital signatures generated by post-quantum schemes. As shown in Figure 3.2, ML-DSA signature schemes produce significantly larger signatures compared to classical algorithms. For instance, ML-DSA-87 produces a signature of 4627 bytes, whereas EdDSA only requires 64 bytes and RSA-2048 uses 256 bytes. Even the smallest ML-DSA variant, ML-DSA-44, generates a signature size of 2420 bytes, which is nearly 10 times larger than RSA-2048 and more than 37 times larger than EdDSA.

Moreover, larger signature sizes imply greater consumption of network bandwidth, which can be a limiting factor in edge deployments where connectivity is constrained or intermittent. This further underscores the importance of evaluating the trade-offs between security strength and practical system constraints when adopting post-quantum solutions

for edge devices.

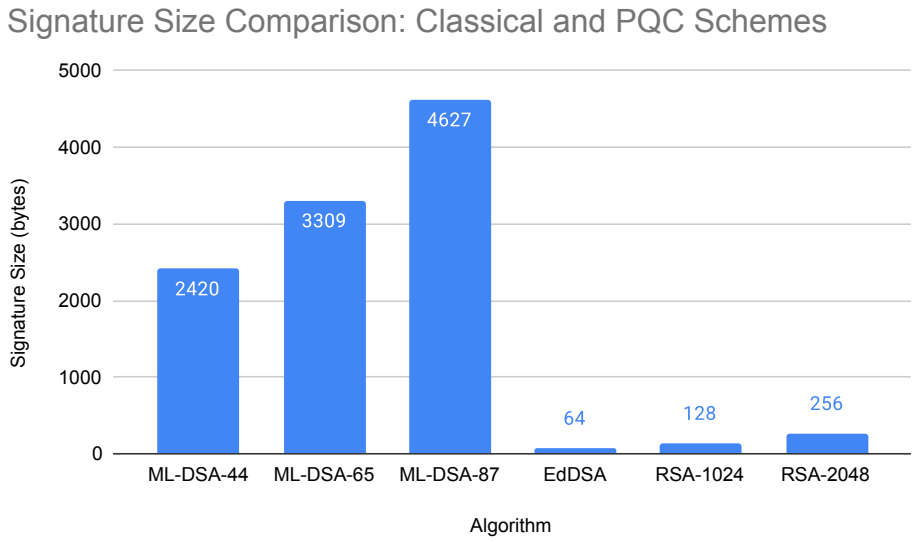

Figure 3.2: Evaluating Signature Size Growth in Post-Quantum Cryptography

3.2 PQC in Edge Computing

Researchers have started exploring quantum-resistant cryptography for edge devices. A consistent finding is that resource constraints on edge devices (low CPU speed, limited memory and power) pose a major challenge for adopting standard PQC algorithms. Most use cases require cryptographic operations that can execute quickly enough to support real-time responsiveness, without taking several seconds [55].

To address these challenges, the number of relevant studies has grown significantly since the first paper on the topic in 2018, reflecting a growing interest and urgency in this emerging research area.

The first paper combining the area of PQC and edge in 2018 proposed using lattice-based post-quantum cryptography to secure resource-constrained edge devices, highlighting the need for long-term data protection and demonstrating the feasibility of implementing PQC on 8 and 32-bit microcontrollers [56].

Hines et al. show that PQC is power-feasible even on constrained devices like Raspberry Pis [57]. The paper examines the power consumption of the 2022 NIST Round 3 PQC

candidate algorithms. Despite being a well-scoped power analysis paper, it leaves important OS-level variables unexplored, which limits the broader applicability to real-world edge deployments. In the study, only Raspberry Pi OS was used across all Raspberry Pi devices, with no comparison to alternative operating systems. This is a limitation, as other OSs may offer different levels of process scheduling efficiency, idle power optimization, and overall resource management. Some lightweight operating systems, for example, are specifically designed to reduce idle power consumption and manage memory more efficiently. Running the same cryptographic tasks on a variety of operating systems could reveal meaningful differences in performance and power usage. This paper supports the feasibility of implementing PQC on edge-class devices such as the Raspberry Pi. However, this thesis aims to take the analysis further by considering the OS as a central component of the cryptographic environment. It examines not only the performance of PQC itself but also how different OS design choices impact the integration and overall effectiveness of PQC.

A recent evaluation by Patterson et al. (May 2025) introduced a dedicated framework for measuring the energy consumption of post-quantum key generation on Raspberry Pi devices, focusing primarily on the ML-KEM algorithm from NIST's 2024 standardization. The study demonstrated that ML-KEM not only provides quantum-resistant security but also achieves significantly higher energy efficiency compared to RSA. At elevated NIST security levels, energy savings reached up to 1,500 times relative to RSA-4096 [58]. These findings offer compelling empirical evidence that post-quantum cryptography, particularly lattice-based schemes, is well-suited for deployment on edge-class devices. This balance of strong security and low power consumption reinforces the viability of quantum-safe key exchange in energy-constrained environments.

The thorough survey from February 2024 of over twenty-four significant contributions reveals a wide variety of approaches to PQC, with lattice-based cryptography (LBC) standing out as the most widely adopted paradigm [27]. Several studies have demonstrated practical implementations and optimizations of lattice-based primitives tailored for edge computing.

Zhao et al. proposed in 2022 a RISC-V processor optimized for Module-LWE operations, achieving up to 3.5× better performance compared to Kyber and 3.3× over Dilithium, highlighting the importance of dedicated hardware acceleration [59]. Similarly, Xu et al. addressed in 2023 performance bottlenecks in encrypted edge-to-cloud data transfers by introducing a lightweight LWE-based protocol that not only enhanced key derivation speeds by up to 238× but also drastically reduced secret key storage requirements from 547 MB to just 1.7 MB [60].

El Kassem et al. introduced in 2019 a lattice-based Direct Anonymous Attestation (DAA) scheme to improve privacy and authentication in edge systems, demonstrating halved TPM storage needs and significantly faster signature operations [61]. At the protocol level, Dharminder et al. proposed in 2021 a post-quantum secure, conditionally privacy-preserving authentication scheme for edge-based vehicular communications [62].

Further notable contributions include performance tuning of key PQC candidates. Kim et al. achieved in 2022 remarkable improvements in the Falcon and Dilithium algorithms by leveraging ARMv8's NEON engine, resulting in up to 113% faster signing and 69% faster verification. These optimizations make such schemes more viable for resource-constrained devices common in edge systems [63][64].

Beyond LBC, other post-quantum methods have also been explored. Kumari et al. proposed in 2022 a hash-based signature scheme enhanced with a Bernoulli-Karatsuba multiplication algorithm, demonstrating improved power efficiency and reduced delay [65]. Meanwhile, Akleylek et al. presented in 2020 a multivariate polynomial-based signature scheme that showed promising results in GPU-based edge authentication scenarios [66].

Innovative and hybrid schemes were also introduced to support niche applications. For instance, El-Latif et al. utilized in 2022 quantum walks to develop a lightweight image encryption technique for IoT systems. In the realm of smart energy, Li et al. proposed in 2022 a privacy-preserving signcryption scheme for Smart Grids, leveraging mobile fog computing and a quantum key pool to support secure multi-level data aggregation [67].

On the infrastructure level, the integration of Quantum Key Distribution (QKD) was explored in industrial contexts. Zhu et al. addressed in 2023 resource allocation chal-

lenges in QKD-secured data center networks [68], while Corrias et al. demonstrated in 2023 a real-world implementation of QKD between a factory and an edge server in Italy's Industry 4.0 ecosystem [69].

In 2024, researchers started investigating offloading heavy cryptography to more powerful nodes. For example, one proposed architecture introduces a Post-Quantum Edge Server (PQES) that acts as a cryptographic proxy [70]. The PQES sits between local devices and the wider network, translating between PQC algorithms and traditional cryptography so that constrained devices can remain lightweight. By handling key generation, encryption, and certificate management centrally, this approach preserves device performance while still providing quantum-resistant security at the network's edge.

3.2.1 Key Gaps

Together, these contributions highlight the research activity around post-quantum cryptography in edge environments, ranging from algorithmic optimizations and hardware acceleration to novel use cases and deployment strategies. However, research at the intersection of edge computing and PQC only began in 2018, with a noticeable increase in publications between 2022 and 2023. This indicates that the field is still emerging and remains relatively underexplored [71]. While many studies focus on algorithmic efficiency and cryptographic protocol design, there is much less information about how operating systems actually support or limit the deployment of these post-quantum mechanisms in practice, particularly in large-scale and heterogeneous edge environments.

3.3 OSs for Scalable and Reliable Edge Computing

As edge computing continues to scale across heterogeneous and resource-constrained environments, traditional general-purpose operating systems are increasingly misaligned with the specific requirements of distributed, large-scale deployments. These environments demand platforms that are not only lightweight and efficient, but also secure, resilient, and maintainable over long lifecycles with minimal intervention.

To meet these demands, researchers and practitioners are rethinking OS architecture by prioritizing modularity, immutability, and reproducibility. A key shift has been the move away from monolithic, general-purpose systems toward specialized, minimal, and declaratively configured operating systems that are easier to maintain and scale across a diverse edge fleet.

One important direction in this evolution is the adoption of minimal and modular system compositions, often assembled using build systems like **Yocto or Buildroot**. These frameworks allow developers to create highly customized Linux distributions that include only the essential components needed for the target hardware and application domain. This not only reduces the attack surface but also optimizes performance and resource usage. Unlike traditional package-based systems, these build tools support fine-grained control over included libraries, services, and drivers, making them ideal for edge scenarios where storage, compute, and energy budgets are tight [72].

Another architectural trend is the emergence of **container-optimized** operating systems such as Bottlerocket or Fedora CoreOS [73]. These systems treat the OS as a thin, stable foundation for running containerized workloads. They typically come with a minimal read-only root filesystem, integrated container runtimes, and robust update mechanisms. By offloading most user-space functionality to containers, the host OS can remain simple, stable, and consistent across devices. This separation of responsibilities enhances fault isolation and facilitates agile software delivery models suited for modern DevOps and edge CI/CD workflows.

In parallel, the **immutability paradigm** has emerged as a powerful approach to managing the lifecycle of distributed systems at scale [74]. Immutable operating systems offer several benefits that make them particularly well-suited for edge deployments. First, they support **atomic and reliable updates**, where system updates are applied as complete image replacements. This approach reduces the risk of partial updates or broken dependencies, which is especially important in unattended or remote environments with limited recovery options. Second, they help **reduce the attack surface** by preventing

runtime modifications to the base system. This limitation makes persistent compromises less likely and helps detect unauthorized changes more easily. Third, immutable systems **improve debugging and rollback capabilities** by maintaining versioned and consistent system images, allowing operators to revert to known-good states with minimal effort. Finally, they promote **operational consistency across large fleets** of devices by ensuring identical system states, which enhances reproducibility and simplifies validation processes.

These characteristics make **immutable OSs a strong fit for edge environments**, which require high availability, strong resilience, and streamlined lifecycle management.

A recent paper published in 2023 by Niklas Gollenstede, titled **reUpNix**, presents an innovative extension of NixOS with capabilities tailored for embedded and edge computing environments [75]. reUpNix introduces a methodology for building reproducible, updateable, and reconfigurable embedded Linux stacks, addressing key shortcomings of traditional NixOS in constrained environments. Notably, it reduces the NixOS base system size by up to 86% through minimization of unnecessary dependencies (e.g., Perl, glibc-locales), elimination of runtime localization, and tailored kernel builds.

Combining Nix's merged-tree update model with A/B boot partitioning, reUpNix supports failure-atomic and reproducible system updates. These updates are built off-device, transferred unidirectionally, and applied without manual intervention, which is crucial for bandwidth-limited or physically inaccessible edge nodes.

reUpNix supports both native Nix services and third-party OCI containers using systemd-nspawn for container isolation. Its shared, deduplicated Nix store enables fine-grained reuse of files across services and containers, significantly reducing storage and update overhead. Moreover, reUpNix introduces layered system profiles and hardware watchdog-based recovery, enabling multi-mission support and robust reconfiguration without risking total system failure.

The update mechanism is further optimized through techniques such as file deduplication, block-level reuse, and binary patching (e.g., BSDiff), achieving transfer size reductions of up to 99.88% compared to naive methods. This makes it especially attractive for deploy-

ment scenarios with strict bandwidth constraints, such as spaceborne systems or rural IoT infrastructures.
In addition to its technical strengths, reUpNix anticipates future security needs by supporting the integration of post-quantum cryptographic components directly into system images and update pipelines. This positions it as a forward-compatible solution for security-sensitive and long-lifecycle edge deployments.

In summary, the convergence of modular design principles and immutability-centric lifecycle management is reshaping OS design for the edge. Solutions like reUpNix demonstrate how these ideas can be practically integrated into production-ready systems, offering a pathway toward more secure, maintainable, and scalable edge infrastructures.

3.3.1 Key Gaps

Overall, the literature on edge OS design is fairly mature in addressing scalability (through modularity and orchestration support) and reliability (through isolation, safety, and self-healing mechanisms). However, until recently, most of this research assumed classical security primitives; the question now is how to make such edge operating systems quantum-safe without compromising their lightweight and reliable nature.
Important questions remain regarding how cryptographic operations are handled at the OS level and how resource constraints affect the real-time use of PQC. These system-level concerns are essential for long-term security and reliability, but are rarely addressed in current research. The next section explores how operating systems are being developed or adapted to address these challenges.

3.4 PQC Readiness in Edge OS Architecture

The integration of PQC into edge computing platforms is still at an early stage. Most existing edge OSes and middleware do not yet include PQC by default, but there is growing awareness of the need for quantum readiness. Some mainstream platforms are preparing for PQC through crypto-agile frameworks. For instance, Red Hat has outlined a strategic

roadmap to adopt cryptographic methods resistant to both classical and quantum attacks. The company is actively involved in collaborations with industry groups and monitors the progress of standardization bodies such as NIST and the Internet Engineering Task Force (IETF) to assess the security, performance, and scalability of quantum-resistant algorithms. Red Hat's strategy is built around several key phases. The "classical" phase represents the current state where products contain only traditional cryptographic functions. In the "PQ-Capable" phase, post-quantum functions are included but are not yet the default option. The goal is to progress to a "PQ-Ready" phase, where quantum-resistant algorithms become the default for most applications. Classical cryptography will still be available for compatibility when needed. This phase also supports hybrid cryptographic schemes that combine classical and post-quantum algorithms, offering a layered defense in case one algorithm is compromised. Red Hat is also aligning its development roadmap with the cryptographic transition guidelines issued by governments including the United States, France, and Germany [76].

In October 2024, Red Hat engineer Daiki Ueno gave a presentation on migrating operating systems toward post-quantum cryptography, highlighting the need to update core components such as OpenSSL, GnuTLS, and IPSec that an OS relies on [77]. Similarly, the IETF industry group is updating protocols such as TLS 1.3 [78] and SSH [79] to support hybrid post-quantum key exchange, which will eventually be integrated into OS networking stacks.

3.4.1 Key Gaps

Despite these steps, full post-quantum readiness in edge computing is not yet reality – most current deployments remain on classical ECC/RSA-based security, with PQC seen in pilot projects or research testbeds. The extent of PQC-aware OS architectures today is limited primarily to experimental systems and roadmap planning. The consensus in recent literature is that much work remains to adapt OS components (device identity management, secure boot, update signing, etc.) to use quantum-resistant algorithms, all while keeping the edge system scalable and stable.

3.5 Open Challenges and Next Steps

Despite recent progress, important gaps remain at the intersection of PQC, edge computing, and OS design (see Figure 3.3). Much of the existing work focuses on individual components, such as cryptographic algorithms or OS kernels, without offering integrated frameworks. There is a need for lightweight edge platforms that combine PQC support with proven scalability. The performance impact of PQC in edge environments, particularly for key exchange and signatures, remains unclear. Further research is needed to measure it and explore OS-level optimizations for maintaining responsiveness. Although crypto-agility is frequently mentioned, it is rarely implemented at scale. Managing secure updates across large fleets of edge devices will require dual-algorithm support and resilient update mechanisms built into the OS. Current standards from NIST, ENISA, ANSSI, and others offer general guidance on PQC but do not provide detailed recommendations for edge-specific OS design. Practical design guidelines and minimum cryptographic support requirements are still missing. Finally, more real-world experiments are needed. Many solutions exist in theory, but few have been validated in long-term, large-scale edge deployments. Bridging this maturity gap is essential for building trust in PQC-ready edge infrastructure. As adoption advances, deeper integration into edge platforms will require coordination between cryptographers, OS developers, and industry to move from isolated components to unified, quantum-secure systems.

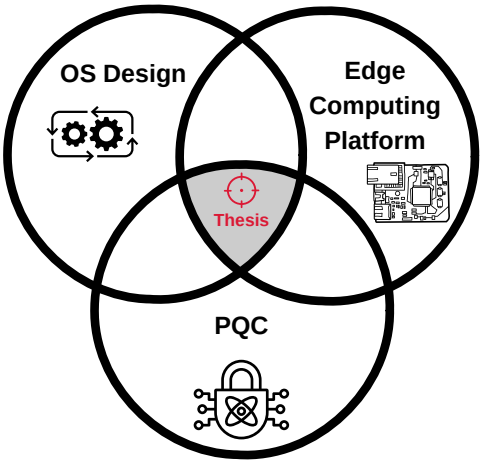

Figure 3.3: Intersection of three distinct domains: Post-Quantum Cryptography, Edge Computing, and Operating System Architecture

4 OS Selection Criteria for Secure and Efficient Edge Deployments

4.1 Key Traits of Post-Quantum Edge OSs

Operating systems designed for edge environments must meet a distinct set of operational requirements [80]. These arise not only from the decentralized and resource-constrained nature of edge infrastructure, but also from the increasing cryptographic complexity introduced by the post-quantum era. Unlike traditional systems, edge nodes are often expected to operate autonomously in physically inaccessible locations, sometimes for years at a time, all while maintaining a strong security posture against both classical and quantum-era threats.

To begin with, **autonomy and resilience** are fundamental. An edge OS must be capable of reliable operation without constant reliance on cloud connectivity. This includes the ability to verify software update integrity locally, ideally using post-quantum algorithms, and to recover from system failures in a controlled manner, especially when physical intervention is not feasible.

Equally important is the ability to **manage systems remotely and securely**. Telemetry and software updates must be transmitted and verified with strong cryptographic guarantees. The OS should support crypto-agile authentication mechanisms and remote attestation frameworks that can adapt to future post-quantum standards. Crucially, these upgrades should not necessitate a full OS reinstallation.

A **robust and future-proof update mechanism** is also essential. Updates need to be atomic, verifiable, and capable of being rolled back in case of failure. The update system should be extensible to support digital signatures based on post-quantum cryptography, including hybrid schemes that combine classical and post-quantum algorithms during transitional phases.

Security must go beyond basic hardening techniques such as secure boot and surface minimization. The OS must be designed with **cryptographic agility** in mind. It should allow the integration of post-quantum libraries, enable dual-stack cryptography where needed, and make it possible to replace cryptographic primitives as standards evolve, all without requiring significant architectural changes [81].

Consistency and reproducibility also become increasingly important in this context. As post-quantum algorithms may impose greater computational or memory demands, reproducibility ensures that configurations can be tested and validated across different hardware nodes. An edge OS must guarantee that all deployed instances maintain identical cryptographic and system-level configurations to reduce complexity and increase reliability at scale.

Finally, **longevity and maintainability are critical** [82]. Many edge deployments are expected to remain in operation for over a decade. This long lifecycle must align with the evolving timeline of post-quantum cryptographic standardization, which is projected to continue through 2030 and beyond. To remain secure and functional, the OS must offer predictable patching mechanisms and support crypto-upgradable toolchains that can evolve without disrupting deployed systems.

4.2 Comparing Edge OS Architecture Classes and Their Update Models

The OS is a core component of any edge device, providing the foundation for workload execution, security enforcement, update management, and system recovery. The OS on edge devices must be lightweight, resilient, secure, and capable of long-term operation

without manual intervention.

Operating systems deployed at the edge generally fall into four broad categories, as shown in Figure 4.1: traditional general-purpose distributions that are mutable (1), immutable operating systems (2), container-based operating systems (3), and custom-built minimal systems via Yocto or Buildroot (4). Each category offers different trade-offs in terms of flexibility, security, reliability, and resource usage.

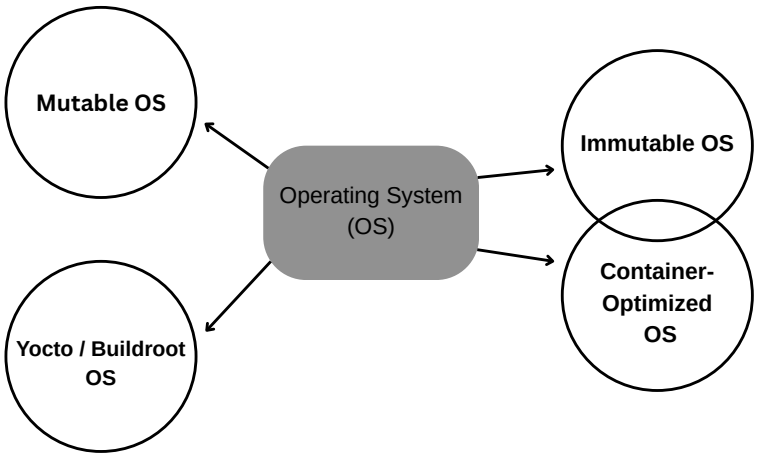

Figure 4.1: Classification of Operating Systems for Edge Devices.

4.2.1 Mutable OSs: Flexible but Fragile

Traditional operating systems, that can be referred to as mutable OSs (MOS), form the foundation of most computing environments today [83]. These systems allow direct, in-place modification of the root filesystem, meaning that system files, configurations, and installed applications can be changed at runtime [84]. Examples include general-purpose Linux distributions like Ubuntu, Debian, and CentOS.

This flexibility has long been considered a strength, as it enables administrators to tailor systems to specific use cases. However, it also introduces significant challenges, especially in large-scale or security-critical deployments. Over time, systems are prone to configuration drift, where individual machines deviate slightly from their intended or baseline state [85]. This can occur due to differences in how and when updates are applied, or through manual interventions during troubleshooting. Such drift makes it increasingly difficult to reproduce system behavior, track changes, or maintain consistent security postures.

Approaches to Updates

The primary mechanism for managing software and updates on mutable OSs is through package managers like APT, DNF or YUM, and Zypper. These tools automate the installation, upgrade, and removal of software by resolving dependencies and fetching packages from remote repositories [84].

During a typical update process, the system first checks for newer versions of installed packages. If updates are available, it downloads them along with any required dependencies, then proceeds to overwrite the relevant system files directly in the live environment. Depending on the nature of the updates, the process may also restart services or prompt the user for a system reboot to apply changes fully.

This update model is imperative and stateful (non-atomic), modifying the live system during runtime. If an update fails midway due to power loss, package conflicts, or broken dependencies, the system can end up in an inconsistent state that requires manual intervention. Small differences in configuration or update timing can result in inconsistent behavior across nodes. Package conflicts or dependency mismatches often prompt user involvement, which poses significant challenges when managing systems at scale [86]. Faulty or partial updates may leave systems in a non-functional state. Moreover, security baselines are hard to verify and maintain.

To mitigate this complexity, tools like Ansible and Puppet are often used to automate provisioning and updates. These tools help enforce consistency by defining configuration states or execution workflows [87]. However, they introduce additional layers of complexity and cannot fully prevent drift or inconsistencies across large fleets of devices. They still rely on the mutable nature of the system and are affected by the same fragility in update mechanisms [88].

The need for consistency, reproducibility, and fail-safe updates has led to the emergence of a different paradigm: the Immutable Operating System. These systems take a fundamentally different approach [74]. The next section introduces this model and explains how it addresses the shortcomings of traditional systems.

4.2.2 Immutable OSs: Built for Resilience

Immutable Operating Systems (IOS) represent a major advancement in OS design, particularly well-suited for modern cloud-native, edge, and distributed environments. Unlike traditional systems that permit in-place modifications of system files and configurations, IOSs enforce a clear separation between **immutable** and **mutable** components. This section draws primarily from the comprehensive survey by Sebastian Böhm and Guido Wirtz, who define an IOS as follows:

> "An Immutable Operating System is a special type of operating system, primarily a minimal Linux distribution that introduces read-only file systems, automatic atomic updates, rollbacks, declarative configuration, and workload isolation to achieve better reliability, scalability, and security, especially to serve as a container host." [89]

At the core of an IOS is a read-only root filesystem, typically mounted at */usr*, which cannot be modified during runtime. Directories such as */etc*, */var*, */home*, and */root* remain writable to allow for configuration files, log storage, and user data. However, it is important to emphasize that all core OS files remain immutable. This design prevents configuration drift and unauthorized changes, ensuring that the OS remains consistent, reproducible, and tamper-resistant across reboots.

By enforcing immutability, IOSs guarantee that all instances of a system run the **exact same software state** [90]. This greatly simplifies debugging and maintenance: a bug observed on one node is reproducible across all others, enabling systematic fixes. Moreover, this homogeneity eliminates the common "works on some, fails on others" issue often encountered in large-scale deployments.

To support system evolution, IOSs rely on update mechanisms that do not alter the running system. Instead, updates are applied to a copy or snapshot of the system state. If an update fails or results in an unbootable system, a **rollback** to the last known working version is automatically or manually triggered. These systems are typically designed for unattended operation, with features such as scheduled updates, automated reboots, and boot health checks to ensure seamless integration into production environments.

Configuration in IOSs is also handled **declaratively**, meaning that instead of executing a series of imperative commands to configure the system step by step, the user specifies the desired end state of the system, such as which users should exist, what services should be enabled, and how the network should be configured. The system then automatically interprets and applies this configuration during the early boot process. Tools like Ignition or cloud-init are responsible for interpreting these declarative specifications and performing the necessary actions to bring the system into the defined state. This approach greatly simplifies system initialization and enables consistent, repeatable deployments across heterogeneous environments, making it especially well-suited for mass provisioning in large-scale cloud or edge infrastructures.

A GitHub repository named "Awesome-Atomic" maintains a list of various immutable Linux distributions [91]. The table 4.1 presents those compatible with the Raspberry Pi, with additional details focused on their suitability for large-scale edge deployments. Each distribution is evaluated based on key criteria relevant to operating systems in such environments.

Criteria	**NixOS**	**Ubuntu Core**	**Vanilla OS**	**Fedora Silverblue**
Base Architecture	Nix	Debian / Snap	Debian Sid	Fedora
Package Management	Nix	Snap	APX, Flatpak	OSTree + RPM (rpm-ostree)
Update Mechanism	Atomic, Declarative	Transactional (Snap-based)	Atomic (ABRoot)	Atomic (OSTree)
Resource Usage	Moderate	Moderate	Moderate	Moderate to High
Flexibility	High	Low	Moderate	Low
Ease of Use	Advanced	Beginner-friendly	Moderate	Moderate
Enterprise Support	Consulting companies	Yes	In development	No

Table 4.1: Overview of Key Immutable Operating Systems

So far, the immutability model has been shown to offer several key benefits:

- **Reliability**, by preventing unintended runtime modifications;
- **Security**, by reducing the attack surface and preventing persistent threats;
- **Scalability**, by making large-scale, uniform deployments easier to manage.

However, this model also introduces specific challenges related to system updates. Since live modification is not allowed, IOSs must adopt specialized update mechanisms that preserve system integrity while allowing for evolution over time. These mechanisms are typically **atomic**, **reversible**, and often **automated**, minimizing the risk of failed updates and downtime.

Approaches to Updates

In the section, we present the three predominant update models employed in IOSs: *transactional updates*, *OSTree-based image management*, and the *A/B partition schema* [89].

The **transactional (snapshot-based) update model**, as depicted in Figure 4.2, leverages copy-on-write (CoW) filesystems, such as *btrfs*, to implement atomic, versioned updates through the use of filesystem snapshots [92]. It is prominently used by openSUSE MicroOS, which employs the *transactional-update* utility to manage system modifications in a safe and consistent manner [93].

When a system update is initiated, a new snapshot of the root filesystem is created using *btrfs*'s native snapshotting capabilities. Instead of applying updates directly to the running system, all changes such as package installations, removals, or configuration modifications are written to the newly created snapshot. This approach ensures that the live environment remains unaffected during the update process.

The system is then configured to boot from the updated snapshot on the next reboot. If the boot process completes successfully, the new snapshot becomes the active system state. However, if the boot fails or the system becomes unstable, the bootloader can be configured to automatically fall back to the previously known working snapshot, thereby ensuring a fail-safe rollback mechanism.

This update model is also tightly integrated with traditional Linux package managers. For instance, *zypper*, the default package manager on openSUSE, can be invoked through transactional-update to install or remove RPM packages in a transactional fashion, meaning that changes are only committed if the update and reboot sequence succeeds.

Internally, the *transactional-update* tool follows a series of steps to apply changes safely.

It first mounts a new overlay snapshot of the root filesystem, then applies updates (such as package installations) to this snapshot using package manager hooks. Once complete, it marks the snapshot as the next boot target via GRUB or another bootloader. Upon a successful reboot, the system switches to the new snapshot. If the update fails, the system automatically reverts to the previous snapshot without requiring manual intervention. This mechanism ensures atomicity, consistency, and reversibility, aligning well with the reliability principles required in critical infrastructure environments.

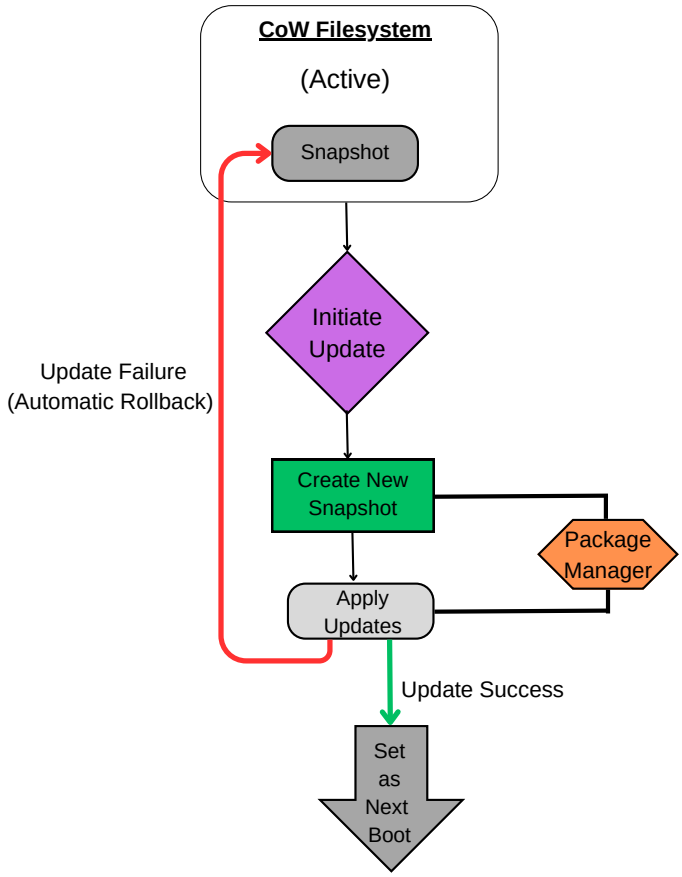

Figure 4.2: Fundamental Concepts of the Transactional Update Model

The **image-based update model** is centered around the use of *OSTree* (also known as *libostree*), a versioned filesystem and deployment mechanism that treats the entire root filesystem of the OS as a content-addressed object store, conceptually similar to Git, but designed for binary files and system images [94]. This model is employed by distributions such as Fedora CoreOS, Fedora Silverblue, and Fedora Kinoite, which rely on rpm-ostree, a hybrid system that layers RPM packages on top of a base OSTree commit.

OSTree constructs the root filesystem as a tree of hard-linked, immutable files stored in

/ostree/repo [95]. Each update is treated as a new commit in the OSTree repository, representing a full filesystem state. These commits are content-addressed and cryptographically hashed, allowing for robust integrity verification and tamper detection [96]. In OSTree-based systems, updates begin by fetching a new commit that represents the updated filesystem tree from a remote OSTree repository. This commit is then stored locally and added to the system's deployment history. A new deployment is created in a separate bootable directory (e.g., /ostree/deploy/$STATEROOT/$CHECKSUM), which contains symbolic and hard links to the appropriate files. The bootloader, such as *GRUB* or *bootupd*, is updated to set this new deployment as the next boot target. Upon reboot, the system seamlessly switches to the new deployment without altering the previous version. If any issues occur, the user can easily roll back to the previous deployment using the *rpm-ostree rollback* command or by selecting the earlier version from the boot menu. OSTree ensures that the system is only modified between reboots and in a fully atomic fashion. Application installations must be layered as additional rpm-ostree layers or containerized. Every system update is a full state snapshot, with no partial or incremental state changes applied live. Switching between deployments is as simple as selecting a different OSTree commit. OSTree also supports *static delta updates*, where only the changed portions of the filesystem tree are transmitted and applied, reducing bandwidth and disk usage [97].

Figure 4.3 illustrates the core concepts of the OSTree update model [98]. At the center of this model are **repositories**, which serve as central storage for all OS versions, making it easier to distribute updates across large-scale deployments. **Branches** represent different OS variants, such as stable or development versions, allowing for parallel development and testing. OSTree ensures that **system upgrades** are atomic and consistent, minimizing the risk of partial or unstable updates. In the event of an issue, users can perform a **rollback** to a previous version with ease, enhancing overall system reliability. Each **commit** is an immutable snapshot of the full filesystem, providing strong traceability and integrity. **Signing and verification** mechanisms further secure the update process by ensuring that only trusted and authenticated commits are deployed. During

a **deployment**, a selected commit is activated as the system's boot target. **Checkouts** create the live environment from that specific commit, defining what the system boots into. Together, these components form a modern, version-controlled framework for managing the OS lifecycle. By combining Git-like structures with robust update mechanisms, OSTree offers a reliable and secure solution for maintaining Linux-based systems at scale.

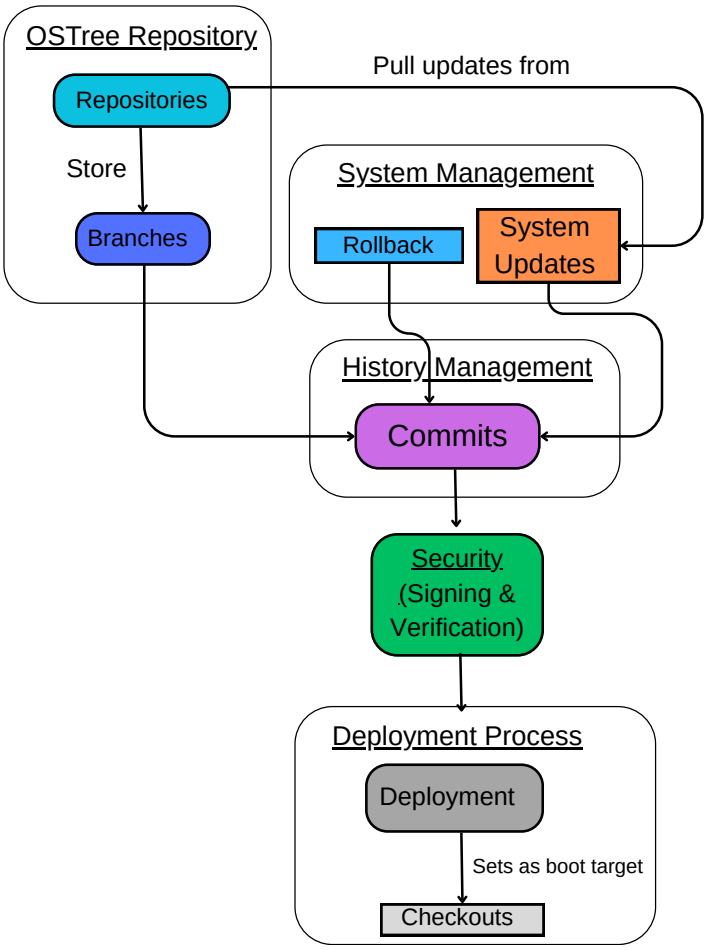

Figure 4.3: Fundamental concepts of the OSTree update model

The **A/B partition update model**, as depicted in Figure 4.4, is a robust and deterministic update mechanism commonly used in immutable operating systems designed for reliability-critical environments such as edge computing, embedded systems, and Kubernetes clusters. This model is implemented by OSs like Flatcar Container Linux, AWS Bottlerocket, and Talos Linux [89]. The system is provisioned with two root partitions, conventionally labeled A and B, each containing a complete bootable image of the OS. At any point, the system actively runs from one of these partitions (e.g., A), while the other (e.g., B) remains inactive and available for updates [99]. In this update workflow,

the system initially boots from the active partition (A), which remains read-only during runtime to preserve immutability and prevent configuration drift. When an update is triggered, either manually or by an orchestrated controller, the new OS image is written to the inactive partition (B). This image is typically a signed and pre-verified system snapshot, such as a SquashFS, ext4 image, or a container-like filesystem. The bootloader, such as GRUB or a system-specific alternative, is then configured to boot from partition B on the next reboot. After rebooting, the system transitions into the updated partition (B). If the boot is successful and the system reaches a defined healthy state, verified through boot-time checks, systemd targets, or readiness probes, the update is marked as successful and partition B becomes the new active partition. If the system fails to boot or does not reach the required operational state within a set timeout, the bootloader automatically rolls back to the last known good state on partition A. This rollback mechanism helps maintain system stability, even when updates are faulty or incomplete. This approach effectively eliminates the risk of deploying broken updates to production environments, especially important in unattended, autonomous systems such as IoT devices or remote edge nodes. Updates are fully atomic: either the entire updated image is booted successfully, or the previous known-good system is restored without intermediate or partial states. A/B systems often include cryptographic verification (e.g., image signature verification via dm-verity [100]) to ensure the integrity and authenticity of updates before boot.

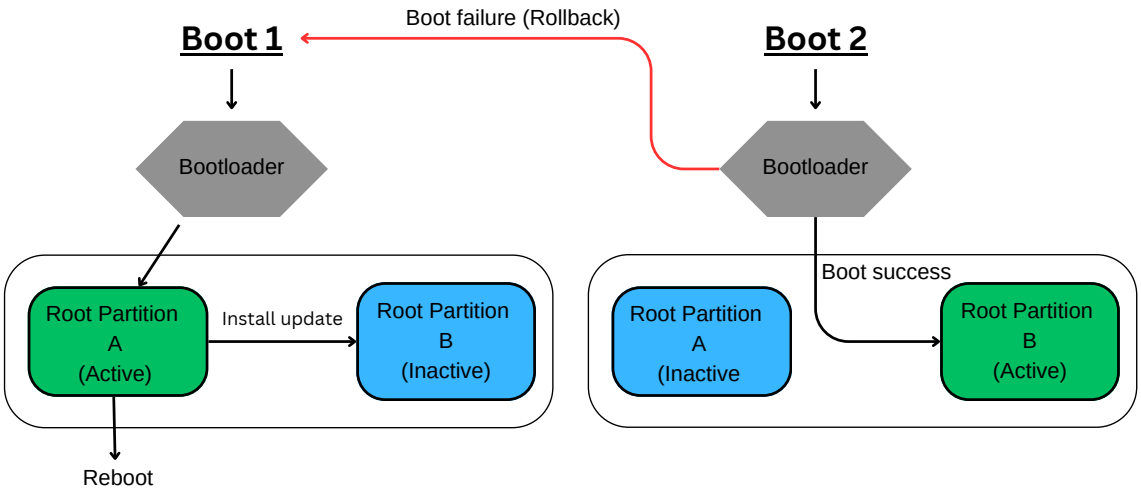

Figure 4.4: A/B partition update model

NixOS as a Leading Immutable OS

NixOS exemplifies the immutable OS model and has **gained popularity** for its declarative configuration, atomic upgrades, and reproducible builds, offering a consistent and reliable foundation for modern infrastructure [74].

It is built around the **Nix package manager**, a purely **functional tool** that enables its unique approach to system configuration and software deployment [101]. Unlike traditional mutable Linux distributions, NixOS is designed around immutability, declarative system management, and full reproducibility [102]. System configuration is treated as code, defined in a single version-controlled file (configuration.nix) that describes the entire system state (packages, services, users, and kernel version). This allows systems to be built, rebuilt, or rolled back deterministically. NixOS's immutability is enforced through a **generation-based update model**. Instead of modifying the live system, it builds new system generations in isolation and activates them by updating a symlink (/run/current-system) [103]. The root filesystem is composed of read-only, content-addressed paths in the Nix store (/nix/store), ensuring consistency and preventing unintended changes [75]. While conceptually similar to transactional and OSTree-based update models, the generation-based approach in NixOS is unique in that it builds system generations from declarative configurations and switches between them atomically via symlinks, without relying on filesystem snapshots or partition duplication. These atomic upgrades allow safe testing of system changes. If an update fails, users can simply reboot and select a previous working generation from the boot menu [104]. This rollback mechanism is especially valuable in edge environments where physical access is limited. NixOS also avoids dependency conflicts by computing cryptographic hashes of all build inputs, allowing multiple versions of packages to coexist without interference. NixOS **supports isolated workloads** via tools like nix-shell and integration with container runtimes [105]. It also enables multiple system profiles, making it ideal for edge use cases such as industrial IoT or satellites, where evolving requirements must not compromise the base platform. While NixOS enforces immutability, it remains flexible. Changes are introduced through source-controlled configuration updates and rebuilds, ensuring all modifications are in-

tentional, traceable, and reversible.

As demand grows for secure, reliable systems, NixOS stands out as a modern OS that brings immutability, declarativity, and reproducibility into practical use. It serves both developers who need precision and production environments that require resilience and automation

4.2.3 Container-based OSs: Optimized for Isolation

Container-based Operating Systems (COS) represent a specialized category of Linux-based systems explicitly designed to run containerized workloads. They are very lightweight operating systems designed specifically to run containers — and nothing else [73].

COSs overlap with immutable operating systems but are not the same. Although these two models share several traits, especially in how they handle system state and updates, they emerge from different needs and embody distinct philosophies.

While COSs inherit the immutability and atomic update properties, they go further by assuming that all workloads will run in containers. These systems are built from the ground up to host containers, often eliminating traditional system features such as package managers, SSH access, or even a shell. Examples like Talos Linux, Bottlerocket, and Flatcar Container Linux operate under the assumption that the host OS should be as minimal and secure as possible. Everything else, including update mechanisms and administrative interfaces, runs in containers or via APIs.

This shift has major implications. Whereas immutable OSs still allow an administrator to SSH in and tweak configuration files or install additional software through controlled layering, a container-based OS does not. It offloads all custom logic into containers and often integrates tightly with Kubernetes or GitOps tools for orchestration and management. Some, like Talos Linux, are completely API-driven and administrators never "log in" to the device in the traditional sense.

Immutable OSs strike a powerful balance by locking down the core system while still offering some administrator control and update flexibility. Container-based OSs take it one step further by assuming developers won't touch the OS at all. They'll build every-

thing as containerized applications. This model fits in cloud-native ecosystems but can be overly rigid if you have legacy needs or require more hands-on control.

So, Immutable and Container-based Operating Systems share core principles such as a read-only root filesystem, atomic updates, rollback support, and a focus on reliability and security. However, their design goals differ. Immutable OSs represent a *system-level model*, emphasizing stability and reproducibility while still allowing a range of workloads. In contrast, Container-based OSs follow an *application-centric model*, assuming all workloads run as containers and often removing traditional system interfaces in favor of remote, declarative control.

The tables 4.2 and 4.3 present a comparison of leading container-based OSs, outlining their essential attributes and design choices.

Criteria	**Talox Linux**	**Bottlerocket**	**Flatcar Container Linux**	**Kairos**
Base Architecture	Custom	Custom	Gentoo	Custom
Container Runtime	Containerd	Containerd	Docker/ K8s	None
Package Manager	No	No	Ignition config	Bundles
Update Mechanism	A/B, API-driven	A/B, API-driven	A/B, Rolling	A/B
SSH	No	No	Yes	Yes
Enterprise Support	Consulting companies	Yes	No	Consulting companies
Use Case	K8s node only	AWS EKS	K8s Docker Swarm	K8s

Table 4.2: Comparison of Leading Container-based Operating Systems - First Part

openSUSE microOS as a Leading Container-based OS

openSUSE MicroOS is a modern, lightweight Linux distribution designed specifically for hosting containerized workloads in a secure and reliable manner. As part of the broader openSUSE family, MicroOS distinguishes itself with an immutable root filesystem, automated updates, and minimal surface area, making it particularly suitable for edge devices.

A central feature of MicroOS is its use of the `transactional-update` wrapper tool, which manages system updates in a safe and atomic way. Unlike traditional package

Criteria	**openSUSE MicroOS**	**Fedora CoreOS**	**Photon OS**	**RancherOS**
Base Architecture	openSUSE	Fedora	VMware	Custom
Container Runtime	Podman	Docker/ podman	Docker	Docker
Package Manager	'transactional-update' (wrapper tool)	rpm-ostree	tdnf	No
Update Mechanism	Atomic, Transactional	Atomic (OSTree)	RPM (granular updates)	A/B
SSH	Yes	No	No	Yes
Enterprise Support	No	No	No	Yes
Use Case	General-purpose container hosting	OpenShift, K8s nodes	vSphere Azure, EC2 GCE	Docker-only Environments

Table 4.3: Comparison of Leading Container-based Operating Systems - Second Part

managers that apply changes directly to the live system, `transactional-update` performs all modifications in a separate Btrfs snapshot. These updates only take effect after a reboot, ensuring that the running system remains stable throughout the process. This approach prevents issues caused by interrupted or failed updates, which are common in mutable systems. In case something goes wrong, users can easily roll back to the last known good state using the built-in snapshot and rollback capabilities.

The utility of `transactional-update` becomes even more apparent in unattended or production environments, where consistency and reliability are critical. By isolating changes from the live system and enabling controlled activation on reboot, it greatly reduces the risk of downtime or misconfiguration. Combined with its container-first philosophy and support for Podman as a rootless container runtime, openSUSE MicroOS offers a compelling solution for running secure, self-maintaining container hosts at scale.

4.2.4 OS images built with Yocto or Buildroot

Custom minimal systems are Linux-based operating systems tailored from the ground up for a specific hardware platform and use case. Rather than starting with a general-purpose distribution and stripping it down, these systems are constructed by selecting only the essential components required for the target environment. Two of the most

widely adopted tools for building such systems are the Yocto Project and Buildroot.

Yocto is known for its power and robustness, making it a strong choice for complex, large-scale projects. However, this capability comes with trade-offs: a steep learning curve and slower build performance. Buildroot, in contrast, offers a simpler and more lightweight build environment that enables fast iteration, making it better suited for quick prototyping and smaller-scale deployments, albeit with less flexibility for highly customized setups [106].

A major advantage of custom minimal systems is the high level of control they provide during the design phase. Developers can fine-tune kernel configurations, include only essential libraries and drivers, and tailor the user-space environment to match the exact needs of the application [107]. This flexibility allows for highly optimized systems in terms of both performance and hardware compatibility. However, this advantage typically ends at deployment: once a system is compiled and deployed, it becomes much harder to change or extend. Without a dedicated update strategy designed upfront, post-deployment modifications can be extremely challenging.

In addition to flexibility, these OS images are extremely lightweight, often just a few megabytes in size. For instance, Yocto-based images can be as small as 600 megabytes, significantly smaller than minimal Ubuntu images, which typically start at around 3 gigabytes [108]. This minimal footprint is a key benefit in edge computing, where hardware is frequently constrained in CPU, memory, and storage. Custom OS images built via Yocto or Buildroot can boot quickly, consume less power, and make more efficient use of limited resources. This makes them especially well-suited for single-board computers and energy-sensitive applications, where performance-per-watt and memory optimization are critical.

It is also worth noting that the Yocto Project supports building images that are compatible with a variety of popular Linux distribution environments through its use of **build targets** and **SDK integration**. While Yocto is not a full Linux distribution itself, it provides the tools and metadata layers necessary to construct custom distributions tailored to specific hardware. This differs from conventional distributions like Ubuntu or

Fedora, which are prebuilt, general-purpose systems. Instead, Yocto provides a flexible framework for assembling a Linux system from source, component by component, enabling exact reproducibility and fine-grained customization.

As indicated in the *poky.conf* file within the official Yocto Project documentation [109][110][111], the supported host environments and reference platforms are included in Table 4.4.

Table 4.4: Supported Host Environments and Reference Platforms for the Yocto Project

Poky 5.1	Ubuntu 20.04	Fedora 39
Poky 5.2	Ubuntu 22.04	Fedora 40
Debian 11	Ubuntu 24.04	Fedora 41
Debian 12	Ubuntu 24.10	CentOS Stream 9
rocky-8	OpenSUSE Leap 15.5	AlmaLinux 8.8
rocky-9	OpenSUSE Leap 15.6	AlmaLinux 9.2

Approaches to Updates

The minimal and static nature of these systems presents major challenges for implementing software updates. Unlike most Linux distributions, Yocto or Buildroot-based systems do not come with a built-in update mechanism. As a result, one of the most basic approaches is manual re-flashing, where a new image is generated and the entire SD card is reflashed. While this method can be useful during development, it is neither scalable nor safe for remote devices, making it unsuitable for production use. Therefore, update strategies must be carefully designed and integrated during the system development phase.

The most commonly used mechanisms in the **Yocto ecosystem** are *RAUC*, *SWUpdate*, *Mender*, and *OSTree*, each offering a different trade-off between complexity, flexibility, atomicity, and robustness [112]. Below is a short overview of these different mechanisms.

RAUC (Robust Auto-Update Controller) is one of the most widely used tools for system updates with Yocto and offers a versatile approach, supporting both file-based and block-level update mechanisms. It handles full system bundles and supports secure, X.509-signed OTA updates. RAUC integrates with bootloaders for A/B setups and roll-

back, and is ideal for embedded systems requiring high assurance and flexibility. Yocto integration is available through *meta-rauc*.

Mender is another block-based A/B update system with a focus on robustness and remote management. It supports full rootfs and kernel updates, along with custom modules for other components. Mender provides automatic rollback on failure, secure artifact signing, and a hosted or self-managed backend. Yocto integration is available via *meta-mender*.

SWUpdate is another well known tool and is a flexible update framework supporting both block and file-based updates. It can update bootloaders, kernels, rootfs, and external components. It supports A/B and rescue setups, rollback (via bootloader integration), and a wide variety of backends like Hawkbit. It is highly customizable and well integrated with Yocto via *meta-swupdate*.

Swupd is a file-based update mechanism initially developed for Clear Linux. It updates files directly within a single root filesystem and supports incremental and delta updates. It's lightweight and minimizes downtime, but lacks built-in rollback and recovery features. Best for systems where high update speed and minimal space usage are priorities.

Lastly, **OSTree** is also supported as an update mechanism in Yocto. As seen in the Immutable OS section 4.2.2, it is a file-based, Git-like mechanism for managing root filesystem trees. It enables atomic updates with efficient delta delivery and supports rollback via deployment switching. It requires specific filesystem layout conventions and bootloader compatibility. Good for use cases where fine-grained file versioning is useful. Yocto integration is in progress via *meta-ostree*.

For **Buildroot**, update strategies are similarly not provided out of the box and must be integrated manually. Given Buildroot's focus on simplicity and generating complete root filesystems from scratch, its default workflow is not geared toward incremental updates. The most straightforward approach is also full image replacement, where the entire root filesystem is rebuilt and reflashed onto the device. However, for production environments, more advanced solutions such as **SWUpdate** [113] or **RAUC** [114] can be manually integrated. Both tools are compatible with Buildroot and supported through

dedicated community-maintained packages or external layers. These allow implementing A/B partition schemes, secure OTA updates, and rollback mechanisms, similar to Yocto-based setups. Although integration may require more manual configuration compared to Yocto's metadata-driven layers, the result can be equally robust if planned early in the system design phase.

4.3 Understanding PQC Workload Demands on Edge OSs

Resource limitations in edge devices become even more significant when integrating post-quantum cryptography.

For Key Encapsulation Mechanisms (**KEMs**), such as ML-KEM (Kyber), the key operations to focus on are **key generation, encapsulation, and decapsulation** [115]. These operations play a central role in establishing secure communications. Key generation affects how devices are provisioned and how certificates are issued. Encapsulation and decapsulation are critical for securely exchanging keys between devices.

In the case of **digital signature schemes**, including ML-DSA (Dilithium) and SLH-DSA (SPHINCS+), the most meaningful operations to measure are **key generation, signing, and verification** [116]. Key generation again impacts provisioning and certificate issuance. Signing is particularly relevant for tasks like code signing, OTA updates, and maintaining audit logs. Verification, on the other hand, is essential for validating updates, securing the boot process, and supporting protocols like TLS and authentication mechanisms. In edge environments, verification tends to be performed more frequently than signing, such as when checking updates or remote commands. As a result, its efficiency can have a greater impact on overall system performance.

4.3.1 Operational Implications for OS Selection

Integrating PQC into edge environments introduces operational constraints that influence OS selection:

- **CPU and RAM Efficiency**: Lattice-based schemes like ML-KEM and ML-DSA require more CPU cycles and memory than classical algorithms. An edge-compatible OS must minimize overhead, support SIMD optimizations (e.g., NEON), and limit background services to free resources.
- **Deterministic Performance**: In time-sensitive edge scenarios, consistent response times matter more than peak throughput.
- **Robust Filesystem and Updates**: Larger PQC keys and signatures affect update verification and boot-time checks. OSs with atomic updates (e.g., OSTree) or reproducible builds (e.g., NixOS) handle this more reliably.
- **Boot and Recovery**: PQC in boot flows increases the cost of delays or failures. OSs with fast, staged initialization and resilient recovery mechanisms maintain better uptime.

4.3.2 OS-Level Factors in PQC Deployment

Operating systems are not neutral platforms in the transition to post-quantum security [117]. Their internal design, including kernel version, scheduling behavior, background services, power management, and cryptographic library integration, directly influences the performance and practicality of deploying PQC on edge devices. Even when the same cryptographic library is used, results can vary based on the underlying C standard library, compiler optimizations, CPU frequency scaling, and support for instruction sets such as NEON on ARM. These implementation details determine how efficiently an OS can manage the additional computational and memory demands introduced by PQC algorithms.

Selecting an OS for secure, scalable edge deployments requires more than traditional performance metrics. Factors like cryptographic readiness, consistent performance under load, and reliability in constrained conditions are essential. The framework developed in this thesis provides a more realistic basis for evaluating OS suitability in the post-quantum era.

5 Research Methodology

5.1 Framework for Evaluating Edge OS Suitability

The goal of this study is to evaluate Linux-based operating systems suitable for deployment in edge computing environments, with a focus on performance, reliability, and security. The selection of OS candidates was guided by the following criteria:

- **Architectural Diversity**: Inclusion of both mutable and immutable OS models to explore trade-offs between flexibility and stability.
- **Lightweight Footprint**: Preference for distributions with minimal resource overhead, suitable for constrained edge hardware.
- **Mainline Kernel, Community, and Enterprise Support**: Preference for distributions with active development, strong community or enterprise backing, comprehensive documentation, and solid hardware compatibility with ARM-based platforms.

Operating systems built using `Yocto` or `Buildroot` were intentionally excluded from the performance benchmarking in this chapter. While these meta-distributions are commonly used in embedded and edge product development, they represent highly customized environments where system performance is tightly coupled to specific application requirements and build configurations. Their inclusion would limit comparability due to the lack of a standardized base image and the overhead required to ensure a consistent software stack across all tested systems.

That said, if a custom-built OS had been included, this study would have prioritized

Yocto over Buildroot. Yocto offers a more scalable and maintainable build system, better suited for complex or large-scale edge deployments. Unlike Buildroot, which targets rapid prototyping with limited extensibility, Yocto supports layered builds, version-controlled recipes, and integration with advanced update frameworks such as RAUC, SWUpdate, and Mender. These capabilities are critical for secure and reliable edge software management. Among the many distributions that Yocto can build or target, **openSUSE Leap** would have been a strong candidate for benchmarking. As a mature distribution, it offers several capabilities aligned with the goals of edge system reliability. Notably, when configured with Snapper and the Btrfs filesystem, openSUSE Leap supports **filesystem-level rollback** for system updates. This mechanism automatically creates Btrfs snapshots before and after critical operations such as package upgrades or system configuration changes. In the event of failure or instability, users can revert the system to a previous working state without manual recovery, significantly improving update safety and resilience [118].

5.2 OS Candidates Selected for Evaluation

The six selected operating systems reflect a cross-section of architectural philosophies and optimization goals within the Linux ecosystem.

5.2.1 Immutable and Container-based OSs

- **NixOS**: A declarative, immutable OS that leverages the Nix package manager. NixOS was selected as a representative of modern immutable infrastructure models and purely functional system configuration. Its emphasis on reproducibility, rollback capability, and minimal global state aligns well with the goals of secure and reliable edge deployments.
- **openSUSE MicroOS**: An immutable and container-centric Linux distribution designed for automated and transactional system updates. Unlike NixOS, MicroOS adopts a more conventional base system (rpm + btrfs + zypper) while offering a mini-

mal environment optimized for container workloads (e.g., with `podman` and `Kubernetes`). Its inclusion highlights the trade-offs between container-native design and traditional package management models in edge systems.

5.2.2 Mutable OSs

- **Raspberry Pi OS Lite**: A lightweight, mutable Debian-based distribution specifically optimized for Raspberry Pi hardware. It serves as a well-established baseline OS with excellent hardware support, low memory footprint, and widespread use in academic and hobbyist edge deployments.
- **Ultramarine Linux (ARM Edition)**: A Fedora-based distribution tailored for ARM SBCs. It offers a mutable systems that aim to combine usability and performance.
- **DietPi**: An extremely minimal Debian derivative focused on resource efficiency. It offers aggressive memory optimizations, lightweight default services, and a simplified software installation process. DietPi was chosen for its relevance in ultra-lightweight edge scenarios where minimizing overhead is critical.
- **Manjaro ARM**: A mutable, Arch-based rolling-release distribution with a growing ARM community. It combines up-to-date kernels and packages with access to the Arch User Repository (AUR), making it suitable for developers who need the latest features. Its performance characteristics offer a contrast to more conservative, stable systems like Debian derivatives.

This diverse selection enables comparative analysis of key trade-offs in edge system design, between mutability and reproducibility, size and functionality, and innovation versus stability. The benchmarking results in the following sections aim to provide actionable insight into which OS models best align with the operational demands of scalable, secure edge deployments.

5.3 Testbed Setup

To ensure consistency and fairness across all performance measurements, a uniform hardware testbed was used throughout the benchmarking process. The experiments were conducted on a Raspberry Pi 4 Model B, a widely adopted single-board computer in edge computing research due to its balance between performance, power efficiency, and broad OS support.

Table 5.1 summarizes the hardware configuration of the edge device used in this study. The system features a quad-core ARMv8 Cortex-A72 processor clocked at 1.80GHz, with dynamic frequency scaling managed by the `ondemand` governor via the `cpufreq-dt` driver. The device is equipped with 8GB of RAM and a 128GB solid-state drive formatted with the `ext4` filesystem using a block size of 4096 bytes. To minimize disk write overhead and improve performance consistency, the filesystem was mounted with the `noatime` and `rw` options.

Table 5.1: Hardware Configuration of the Edge Device

Component	**Details**
Processor	ARMv8 Cortex-A72 @ 1.80GHz, 4 cores
CPU Scaling	`cpufreq-dt`, governor: `ondemand`
Motherboard	Raspberry Pi 4 Model B Rev 1.5
Chipset	Broadcom BCM2711
Memory	8GB
Disk	128GB ED2S5, ext4, block size: 4096
Mount Options	`noatime`, `rw`

5.3.1 Benchmark Execution Environment

To ensure fairness and reproducibility across all tested operating systems, a standardized software environment was established prior to executing any benchmarks. All tests were performed on a common base configuration to eliminate variability caused by missing dependencies, divergent runtime environments, or distribution-specific package versions.

Importantly, to maintain consistency at the kernel level, which could potentially influence performance results, all operating systems were configured to run the **same Linux kernel version (6.12)**.

A minimal yet sufficient set of packages was manually installed on each system to support benchmark compilation, container management, and low-level system testing. The common software stack included:

- `docker`
- `git`, `wget`, `unzip`, `file`, `which` – for retrieving, extracting, and identifying benchmark dependencies and files.
- `gcc`, `glibc`, `cmake` – standard development tools required to compile and run source-based benchmarks (e.g., CoreMark, RAMspeed).
- `php` – used by the Phoronix Test Suite for scripting and automation.
- `tmux` – for terminal session management during test automation.

In addition, a lightweight `BusyBox` Docker container was run during each benchmark. This was not intended to simulate a realistic edge workload, but rather to enforce consistency across all test environments. By introducing an identical and controlled background process on each operating system, the benchmarking conditions remained uniform. This ensured that all systems experienced the same minimal runtime activity during testing, eliminating discrepancies caused by any idle-state optimizations or varying default background services.

Overall, this standardized testing environment allows for accurate and fair performance comparisons by focusing on the behavior of the operating system kernel and its interaction with system resources, rather than on differences in package management or user space utilities

6 Evaluating System Performance Across OS Architectures

6.1 Objective

The objective of this evaluation is to analyze and compare the system-level performance characteristics of the selected operating systems designed for edge environments.

In edge computing scenarios, system resources are often limited, workloads are highly diverse, and performance predictability is critical. Therefore, this chapter focuses on benchmarking three foundational dimensions of system performance:

- **CPU Throughput** – Reflects the processing capability of the OS and is crucial for compute-bound workloads such as data filtering, encryption, or inference tasks.
- **Memory Bandwidth (RAM)** – Measures how efficiently an OS manages data transfers within main memory, which directly impacts applications involving buffering, caching, or real-time data streaming.
- **Disk I/O Performance** – Captures how well the OS handles storage operations, which is essential for logging, local persistence, container image unpacking, and filesystem responsiveness.

These three metrics form **the backbone of system performance** on constrained edge devices. By conducting reproducible benchmarks in a uniform hardware environment described in Table 5.1, this analysis aims to provide both a **quantitative and qualitative**

foundation for identifying operating systems that are optimized not just for peak performance, but for consistent and resource-efficient behavior under realistic edge workloads.

6.2 Specific Methodology

All system-level performance evaluations in this study were conducted using the **Phoronix Test Suite (PTS)**, a widely adopted and open-source benchmarking platform [119]. PTS has been extensively used in academic and industry research for cross-platform performance analysis. Notable examples include the work by Marko Boras et al. on the performance evaluation of desktop Linux operating systems [83], and the study by Christos Liambas et al. comparing the performance of digital forensic tools across various operating systems [120]. Its ability to provide consistent test execution, automated result collection, and standardized procedures makes it a reliable and reproducible benchmarking framework for comparative OS analysis.

PTS operates by executing a set of predefined or custom benchmark test profiles, capturing detailed performance metrics, and storing results in a structured format for comparison. Each test is run in a controlled environment, with system information logged alongside the results to ensure transparency and reproducibility.

6.2.1 CPU Benchmark

The first CPU benchmark used in this study is **CoreMark**, a lightweight yet effective synthetic benchmark developed by EEMBC to evaluate processor core performance. Designed as a modern alternative to the outdated Dhrystone benchmark, CoreMark provides a single numerical score that enables quick and consistent performance comparisons across platforms. It simulates realistic embedded workloads, including list processing, matrix manipulation, state machine evaluation, and cyclic redundancy checks (CRC). The CRC serves a dual purpose by both contributing to the computational workload and acting as a built-in validation mechanism to ensure result correctness. To prevent compiler optimizations from skewing the results, all values are generated dynamically at runtime, and

no external library calls are used during the timed execution.

CoreMark reports performance in **iterations per second**, indicating how many complete benchmark iterations the processor can execute within one second. Each iteration consists of a fixed sequence of operations that reflect typical control logic and arithmetic workloads in embedded systems. This unit of measurement captures the sustained computational throughput of the processor while remaining independent of memory capacity or data volume, making it particularly useful for cross-platform benchmarking in embedded and edge computing environments.

In this study, the benchmark was configured with a size parameter of 666. In CoreMark, this parameter specifies the number of elements used in the internal data structures, such as the linked list and matrix. A size of 666 means the benchmark processes a linked list with 666 nodes, and the matrix and state machine components are proportionally scaled. This configuration is intentionally small to ensure the working set fits entirely within the L1 or L2 cache of most CPUs. As a result, the benchmark focuses on stressing CPU core logic, branch prediction, and instruction throughput, rather than being constrained by memory latency or bandwidth limitations.

To complement the CoreMark benchmark, a series of cryptographic performance tests was conducted using the Phoronix Test Suite's Botan-based profiles. These benchmarks evaluate the encryption throughput of five widely used symmetric algorithms: **Blowfish, KASUMI, CAST-256, AES-256, and ChaCha20-Poly1305**. Only the encryption phase was measured.

The results are reported in **MiB/s** (mebibytes per second), indicating how fast the CPU encrypts data. Each benchmark encrypts a fixed data volume and measures the time taken. Throughput is calculated as:

$$\text{Throughput (MiB/s)} = \frac{\text{Total Data Encrypted (MiB)}}{\text{Time Taken (s)}}$$

This metric provides a more practical and interpretable measure of cryptographic performance than instruction count or cycle-based metrics, as it directly reflects the CPU's

ability to process secure data in real time. Each algorithm represents a distinct cryptographic design and application domain. AES-256 and ChaCha20-Poly1305 are widely used in modern security protocols such as TLS and VPNs, while Blowfish, CAST-256, and KASUMI are older or domain-specific ciphers still relevant in constrained or embedded systems.

Benchmarking encryption performance is particularly relevant for edge computing, where devices frequently handle secure data transmission, local encryption, and privacy-preserving computations. Unlike general-purpose workloads, cryptographic operations place stress on specific CPU subsystems, including instruction-level parallelism, memory hierarchies, and, where available, hardware acceleration. Measuring encryption throughput thus provides a realistic and security-aware perspective on processor performance in resource-constrained environments.

6.2.2 RAM Benchmark

To evaluate memory performance, this study used the **RAMspeed SMP** benchmark from the Phoronix Test Suite, a tool specifically designed to assess sustained memory bandwidth under parallel workloads. RAMspeed operates by executing low-level memory operations in rapid succession, bypassing higher-level abstractions to isolate the raw throughput of the memory subsystem. The results are reported in **megabytes per second (MB/s)**, indicating the effective rate at which data is moved or manipulated in memory. The benchmark consists of several tests that simulate common memory access patterns for both integer and floating-point data types. Each test is multithreaded and designed to reflect real-world computational tasks frequently encountered in embedded systems, edge computing, and high-performance workloads.

- **Add Integer (MB/s)**: Measures the speed at which the system can read two integer arrays from memory, perform element-wise addition, and store the result in a third array. This tests memory read/write throughput and integer arithmetic combined.
- **Copy Integer (MB/s)**: Assesses the bandwidth for copying data from one integer array to another. It reflects pure memory transfer speed without arithmetic overhead,

making it useful for identifying memory bandwidth limits.

- **Scale Integer (MB/s)**: Evaluates performance when scaling an integer array by a constant factor. This test combines memory access with basic arithmetic and simulates workloads such as signal scaling or normalization.
- **Triad Integer (MB/s)**: Measures the system's ability to compute and store the result of a fused multiply-add operation, such as `C = A + B * scalar`. It is often considered a good indicator of memory bandwidth under mixed compute and transfer loads.
- **Add Float (MB/s)**: Similar to Add Integer, but operates on 32-bit or 64-bit floating-point values. This test reflects performance in scientific, multimedia, or machine learning workloads that rely heavily on floating-point math.
- **Copy Float (MB/s)**: Tests raw memory copy speed for floating-point arrays. It is useful for detecting how efficiently floating-point data structures can be moved across the memory hierarchy.
- **Scale Float (MB/s)**: Measures the throughput for multiplying every element of a floating-point array by a constant value. This is common in numerical simulations, graphics pipelines, and data preprocessing.

To complement the RAMspeed tests and provide additional insight into memory transfer performance, the **MBW** (Memory BandWidth) benchmark was also executed. MBW is a lightweight tool designed to measure the throughput of basic memory copy operations in user space, using standard C library functions such as `memcpy()`. Unlike RAMspeed, which evaluates a variety of memory operations with arithmetic logic, MBW focuses purely on memory copy performance. This makes it particularly useful for isolating the raw memory bandwidth of a system without interference from computational overhead. Two specific test modes were used:

- **MBW Copy128MiB (MiB/s)**: This test measures the memory bandwidth while copying 128 MiB of data in bulk. The size of the memory block remains consistent during the test, allowing for high-throughput sustained copy operations that stress the memory controller and data paths. It is representative of real-world scenarios such as

data buffering, stream processing, and inter-process communication.

- **MBW Copy128MiB-FixedSize (MiB/s)**: In this variant, the test also copies 128 MiB of data, but with strict control over memory allocation and access patterns. It avoids dynamic memory management optimizations and ensures that each run uses a fixed-size, page-aligned buffer. This helps eliminate variability due to malloc behavior or memory alignment, resulting in more deterministic and reproducible measurements of peak memory throughput.

These MBW results complement the RAMspeed benchmarks by providing a focused evaluation of memory copy efficiency, which is critical in edge environments where fast data movement between components (e.g., sensors, buffers, processing units) is often required. Together, RAMspeed and MBW offer a well-rounded view of the memory subsystem's performance under both compute-intensive and pure-transfer workloads.

6.2.3 Disk I/O Benchmark

Disk I/O performance was evaluated using **FIO** (Flexible I/O Tester), a powerful and widely used benchmarking tool originally developed by Jens Axboe for testing and validating the Linux I/O subsystem and various I/O schedulers. FIO provides fine-grained control over test parameters, allowing for precise simulation of real-world disk access patterns. In this study, FIO was used to benchmark both **sequential read** and **sequential write** performance. The following configuration was applied for both tests:

- **I/O Type**: Sequential Read / Sequential Write
- **I/O Engine**: `io_uring` – the modern Linux asynchronous I/O interface introduced in kernel 5.1, offering lower overhead and higher throughput compared to legacy interfaces like `libaio`.
- **Direct I/O**: `Yes` – bypasses the OS page cache to ensure that measured performance reflects actual disk access latency and throughput, rather than cache-assisted speeds.
- **Block Size**: 4KB – a standard block size that reflects typical filesystem and hardware-level I/O granularity, allowing for accurate benchmarking of small, aligned I/O opera-

tions.

- **Job Count**: 1 – a single-threaded workload was used to ensure consistency across all OSs and to better reflect the I/O performance of single-task edge workloads.
- **Target**: Default test directory – FIO writes and reads from a temporary file located on the main mounted filesystem of the containerized environment, ensuring a consistent disk path for all tests.

The benchmark measures throughput in **megabytes per second (MB/s)**, indicating the sustained read or write rate under the given configuration. Sequential read tests simulate workloads such as log analysis, media streaming, or loading of large models, while sequential write tests reflect scenarios like sensor data buffering, logging, or configuration saves. By using `io_uring` with direct I/O and a controlled block size, this benchmark provides an accurate representation of raw disk subsystem performance, independent of filesystem caching or read-ahead mechanisms. The use of consistent parameters across all operating systems ensures fair and reproducible comparisons in the context of edge system deployment.

To complement block-level disk measurements from FIO, this study also used **CacheBench** from the LLCbench suite. It offers valuable insight into processor's cache and memory hierarchy performance, which significantly affects real-world I/O, especially in buffered or hybrid I/O scenarios. CacheBench is designed to measure the effective **bandwidth between different levels of memory**, including L1/L2/L3 caches and main memory, under high-intensity read and write operations. It performs a series of memory accesses with controlled strides and working set sizes, simulating how data is moved through the cache hierarchy under load. In this study, the following test modes were executed:

- **Cache Read (MB/s)**: Measures the bandwidth for sequential memory read operations from cache-aligned regions. This test provides insight into how quickly the CPU can fetch data from the L1/L2/L3 caches during I/O operations.
- **Cache Write (MB/s)**: Measures the speed at which the CPU can write to cache-aligned memory regions, reflecting the upper limit of data output rates during buffered

or write-behind disk operations.

The results, reported in **megabytes per second (MB/s)**, quantify the maximum cache-level data transfer rates the system can sustain. High performance in these tests suggests efficient memory hierarchy behavior, which is particularly relevant for edge workloads that rely heavily on memory-mapped I/O, fast temporary file handling, or caching layers in local storage stacks.

Including CacheBench alongside FIO enables this study to assess both disk throughput and memory/cache bandwidth, providing a more complete view of the data path from storage to processor in edge environments.

To evaluate filesystem-level performance, this study included the **FS-Mark** benchmark, a utility specifically designed to stress and measure the performance of file creation and metadata operations in a realistic workload scenario. While tools like FIO focus on raw block-level I/O throughput, FS-Mark provides insight into the behavior and efficiency of the file system under metadata-heavy operations. In this study, the benchmark was executed with the following configuration: the test involved creating and writing 1,000 files, each 1 MB in size. This test simulates a common real-world use case where a system must create and write many medium-sized files in quick succession. It stresses several components of the I/O stack simultaneously:

- **Filesystem metadata operations**: Each file creation involves updates to inodes, directory entries, and allocation maps.
- **Write throughput**: Writing 1,000 MB of data exercises sustained sequential write capabilities.
- **File allocation and journaling**: Depending on the filesystem type and mount options, journaling, delayed allocation, and write barriers may impact performance.

The result, reported in **files per second (files/s)**, represents the rate at which the system can complete the full cycle of creating, writing, and closing files. A higher value indicates better performance in handling file-intensive workloads, such as logging systems, temporary storage in pipelines, container image unpacking, or edge devices that buffer

sensor data locally before upload. FS-Mark complements block-level and cache/memory-level tests by offering a practical measure of how well each operating system handles real-world file operations under pressure. It also helps reveal performance bottlenecks in mount configurations, journal policies, or background I/O handling mechanisms that are critical in edge scenarios.

Finally, this Disk I/O benchmark also included the **"Unpacking the Linux Kernel"** test, identified as `pts/unpack-linux-1.2.0` in the Phoronix Test Suite. This benchmark evaluates real-world file system and I/O performance by measuring how long it takes to extract a compressed Linux kernel source archive.

Specifically, the test involves extracting a `.tar.xz` archive of the Linux 5.19 kernel source tree, simulating a common workload where a large number of files and directories must be created and written to disk in rapid succession. The archive contains thousands of small files organized in a deeply nested directory structure, making the operation both I/O- and metadata-intensive.

The result is reported in **seconds**, representing the total time required to complete the unpacking process. A lower time indicates better overall I/O performance, as the system must efficiently manage both:

- **Sequential and random writes**: Writing many small files and directory metadata to disk.
- **Decompression overhead**: Processing the `xz`-compressed archive in memory and streaming the data to the filesystem.
- **File system handling efficiency**: Interactions with journaling, allocation strategies, and inode management.

This benchmark complements synthetic I/O tests (such as FIO and FS-Mark) by providing a practical, application-level workload that mirrors real-world usage scenarios. It is particularly relevant in edge systems that rely on containerized application deployment, software updates, or log file extraction, where filesystem responsiveness and disk throughput have a direct impact on performance and reliability.

6.2.4 Analysis Approach

Rationale for Using Average Normalized Scores

To fairly compare the performance of multiple operating systems across a diverse set of metrics, this study employs a two-step aggregation approach: **per-metric normalization**, followed by **average score computation using the geometric mean**.

Normalization Strategy. Each raw benchmark score is first normalized on a per-metric basis to allow fair comparisons across metrics of different scales and units. Since most metrics in this study represent performance scores where **higher is better**, normalization is performed by dividing each score by the maximum score observed for that metric:

$$\text{Normalized}_{i,m} = \frac{x_{i,m}}{\max\limits_{j}(x_{j,m})} \tag{6.1}$$

Here, $x_{i,m}$ is the raw benchmark value of operating system i for metric m, and the index j iterates over all operating systems in the benchmark set to identify the maximum value for metric m. This yields a normalized score in the range $[0, 1]$, where a score of 1.0 indicates the best-performing system for that metric.

However, for metrics where **lower values indicate better performance**, such as the "Unpack Kernel" benchmark which measures the time (in seconds) required to extract a large compressed archive, this normalization strategy is inverted. In these cases, the minimum value (that is, the shortest execution time) corresponds to the best performance, and normalization is performed as follows:

$$\text{Normalized}_{i,m} = \frac{\min\limits_{j}(x_{j,m})}{x_{i,m}} \tag{6.2}$$

This ensures that, regardless of whether a metric is "higher is better" or "lower is better," a normalized value of 1.0 always represents the best performance, and all scores remain within the $[0, 1]$ interval. This consistency is essential for computing aggregated performance scores using the geometric mean, and for producing intuitive comparative visualizations.

Geometric Mean Aggregation. To aggregate normalized scores across all metrics in a category (e.g., CPU), the **geometric mean** is used instead of the arithmetic mean. The geometric mean is particularly suitable for performance analysis, as it provides a balanced average of ratios and avoids the distortion caused by outliers or skewed distributions. For an operating system i with n normalized scores $\{s_{i,1}, s_{i,2}, ..., s_{i,n}\}$, the average normalized score is computed as:

$$\text{GeometricMean}_i = \left(\prod_{k=1}^{n} s_{i,k} \right)^{1/n} \tag{6.3}$$

This methodology is widely recommended in the systems performance community, including by Philip J. Fleming and John J. Wallace, for its mathematical suitability in summarizing relative performance across multiple benchmarks [121].

Visualization. The computed geometric mean scores are used to generate comparative bar plots for each resource category (CPU, RAM, Disk I/O), enabling quick interpretation of relative performance. This approach distills complex multidimensional benchmark data into a single interpretable score per OS per category, while still preserving metric-level detail in accompanying tables.

This method ensures consistency, fairness, and statistical soundness in the cross-platform evaluation of operating systems for edge environments.

Statistical Dispersion of Normalized Performance

To evaluate the overall consistency of each operating system's performance, the statistical dispersion analysis was applied across the **entire set of normalized performance scores**, combining all metrics from the CPU, RAM, and Disk I/O benchmark categories. Rather than analyzing variability within each domain separately, this unified approach captures how uniformly each OS performs across the full spectrum of system-level tasks. This is particularly relevant for edge environments, where general-purpose consistency across subsystems is often more critical than isolated peak performance.

Five classical dispersion measures were used:

1. **Standard Deviation (SD)**: Measures the average amount by which individual nor-

malized scores deviate from the geometric mean. A lower standard deviation indicates more consistent performance. It is computed as:

$$\sigma_i = \sqrt{\frac{1}{n}\sum_{k=1}^{n}\left(s_{i,k} - \mu_i\right)^2} \tag{6.4}$$

where $s_{i,k}$ is the normalized score for metric k on operating system i, and μ_i is the geometric mean for that OS.

2. **Variance**: The square of the standard deviation. It provides a non-rooted version of dispersion, more sensitive to large outliers. Defined as:

$$\mathrm{Var}_i = \frac{1}{n}\sum_{k=1}^{n}\left(s_{i,k} - \mu_i\right)^2 \tag{6.5}$$

3. **Range**: The simplest dispersion metric, computed as the difference between the highest and lowest normalized values for a given OS:

$$\mathrm{Range}_i = \max_k(s_{i,k}) - \min_k(s_{i,k}) \tag{6.6}$$

4. **Interquartile Range (IQR)**: The IQR quantifies the spread of the middle 50% of the data, defined as the difference between the third quartile (Q_3) and the first quartile (Q_1):

$$\mathrm{IQR}_i = Q_{3,i} - Q_{1,i} \tag{6.7}$$

For each operating system i, the IQR highlights the internal consistency of normalized scores by measuring the variability in the central portion of its performance profile. Unlike variance or standard deviation, the IQR is robust to extreme values, making it valuable when analyzing performance distributions that may contain outliers.

5. **Coefficient of Variation (CV)**: The CV expresses the ratio of the standard deviation to the mean of normalized scores, offering a unitless measure of relative dispersion:

$$\mathrm{CV}_i = \frac{\sigma_i}{\bar{s}_i}$$

where σ_i is the standard deviation and $\bar{s}_i$ is the arithmetic mean of the normalized scores for OS i. The CV is useful for comparing variability between OSs with different

mean performance levels. A lower CV indicates a more balanced and predictable performance profile, which is desirable for real-world edge deployments where workload consistency is critical.

These measures were calculated for each operating system across all normalized metrics from the CPU, RAM, and Disk I/O datasets. Prior to aggregation, all benchmark values were normalized to the range $[0, 1]$, with appropriate inversion applied for metrics where lower values indicate better performance (e.g., Unpack Kernel). This ensures consistency in interpretation across all tests.

Visualization via Parallel Coordinates. To visually explore and compare the dispersion characteristics across operating systems, the computed five dispersion metrics were plotted using a **Parallel Coordinates Plot**. This technique enables multidimensional comparisons by representing each operating system as a polyline that intersects a series of vertical axes, with each axis corresponding to a specific statistical metric. All metrics are normalized to the $[0, 1]$ interval to ensure visual comparability, even if their original scales differ substantially.

Parallel coordinates effectively reveal consistency, outliers, and performance trade-offs across multiple dispersion dimensions. Unlike bar plots, which are typically limited to single-metric comparisons, parallel coordinates allow for compact and expressive visual inspection of how each OS behaves across the entire spectrum of variability indicators. In general, **lower dispersion values are preferable**, as it reflects consistently strong performance across workloads rather than isolated peaks.

6.3 Results and Analysis

CPU Results

CPU Performance Summary. As shown by the CPU throughput measurements in Table 6.1 and Figure 6.1, **NixOS** achieves the highest geometric mean score (0.992), closely followed by **Ultramarine Linux** and **RPi OS Lite**. Ultramarine delivers the best CoreMark result and strong performance across all ciphers. NixOS leads in four of seven

Table 6.1: CPU Performance Comparison of Edge OSs on a Common Software Stack

Metric	RPi OS Lite	openSUSE MicroOS	Ultramarine Linux	DietPi	Manjaro ARM	NixOS
Coremark-Size666 (iterations/s)	37971.68	27365.67	**38170.17**	31677.34	31891.76	37521.61
Blowfish (MiB/s)	120.19	82.83	121.25	91.09	100.75	**121.55**
KASUMI (MiB/s)	33.04	26.19	35.25	25.19	27.53	**35.36**
Twofish (MiB/s)	**107.41**	70.97	104.38	82.17	89.93	104.96
CAST-256 (MiB/s)	**58.06**	38.97	56.92	43.18	48.49	57.17
AES-256 (MiB/s)	54.90	37.39	54.79	41.39	45.83	**54.91**
ChaCha20Poly1305 (MiB/s)	126.91	95.30	130.18	101.12	106.48	**130.763**

cryptographic benchmarks: Blowfish, KASUMI, AES-256, and ChaCha20-Poly1305. RPi OS Lite outperforms others in CAST-256 and Twofish, maintaining competitive performance across the board.

These three distributions form a high-performing group with minimal variation in average scores. **Manjaro ARM** ranks lower, with a geometric mean of 0.823. It performs consistently but never leads in any test.

Consistency and Dispersion. While geometric mean scores reveal average performance, statistical dispersion highlights how consistently each OS performs across the various CPU benchmarks (see Figure 6.2). **Ultramarine Linux** has the lowest dispersion across all metrics, indicating highly stable CPU behavior. **NixOS**, despite its top average, shows greater variability, particularly in IQR and CV. **RPi OS Lite** and **Manjaro ARM** demonstrate moderate dispersion with predictable, if not leading, performance. **DietPi** and **openSUSE MicroOS** exhibit the highest variability and the weakest CPU scores overall.

Interpretation. Most of the performance spread might be attributed to build and kernel configuration choices rather than inherent hardware limitations. For example, openSUSE MicroOS uses the `musl` C library and a conservative implementation of `memcpy`, which avoids aggressive vectorization. This means memory operations are handled in smaller chunks using general-purpose instructions. As a result, memory-intensive algorithms like Twofish, which rely heavily on fast and repeated memory access patterns, are

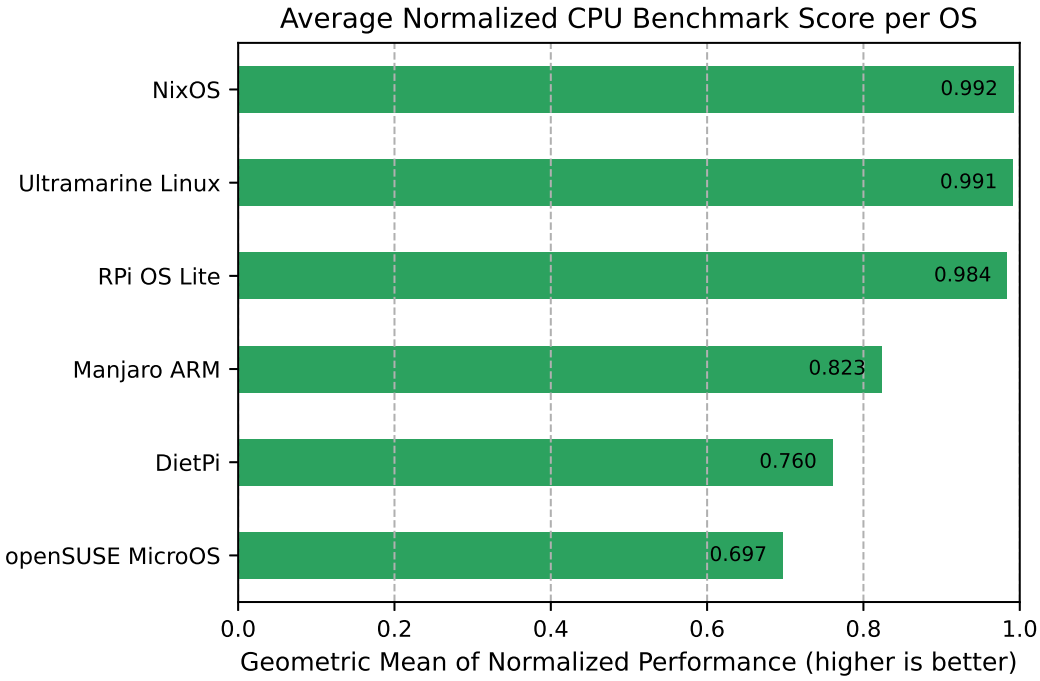

Figure 6.1: Average Normalized CPU Benchmark Score per OS

more likely to exceed the capacity of the L1 cache and spill into the L2 cache. The L2 cache serves as a larger backup to the faster but smaller L1 cache. While it helps reduce access to main memory, it has higher latency, which can slow down performance if used frequently.

In contrast, distributions like Ultramarine and NixOS use `glibc` or `llvm-libc` with SIMD-optimized routines, particularly through NEON instructions on ARM architectures. SIMD (Single Instruction, Multiple Data) is a type of parallel processing that allows the processor to perform the same operation on multiple data points at once. In the context of memory functions like `memcpy`, SIMD enables copying large blocks of memory in parallel, which significantly increases throughput. This reduces the number of memory accesses and helps keep more data within the faster L1 cache, improving performance in workloads that are sensitive to memory bandwidth and latency. Compiler choice also plays a role: CoreMark runs 39% slower on openSUSE compared to Ultramarine, likely due to openSUSE using GCC 12. In contrast, Ultramarine and NixOS employ newer compilers like GCC 14 or Clang, with builds optimized for `-mcpu=cortex-a72`, resulting in more efficient code generation for the target architecture.

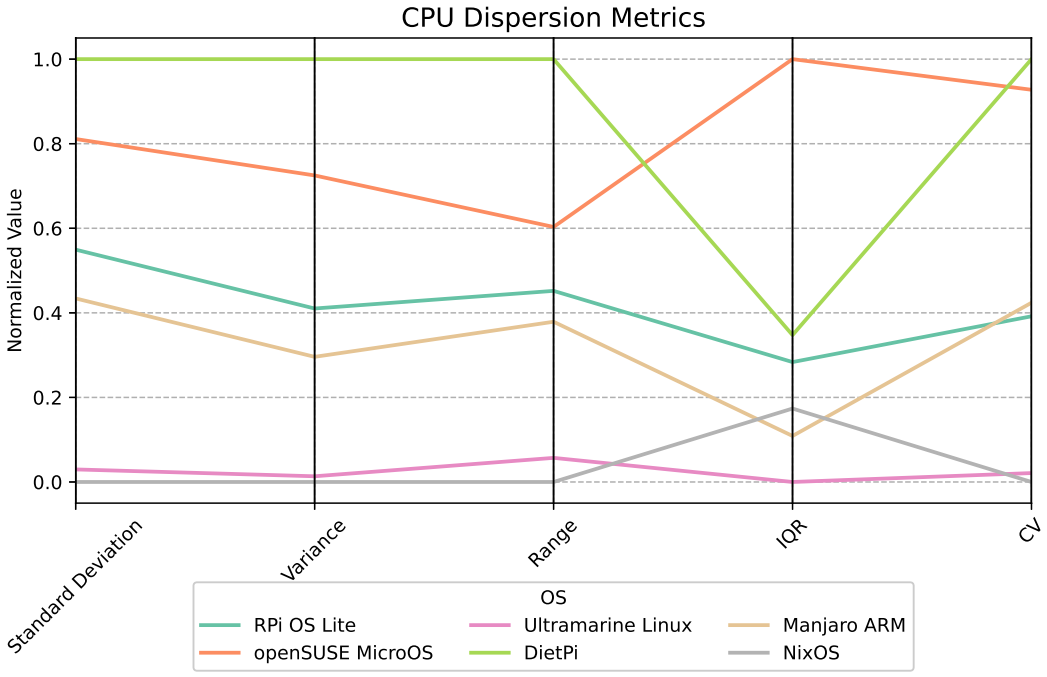

Figure 6.2: CPU Dispersion Metrics (Normalized)

Conclusion. Among the tested OSs, NixOS, Ultramarine, and RPi OS Lite deliver the best balance of throughput and consistency. In short, kernel configuration, and user-space libraries, rather than the underlying silicon could explain the clear separation between the high and mid-performance tiers.

RAM Results

Table 6.2: RAM Performance Comparison of Edge OSs on a Common Software Stack

Metric	RPi OS Lite	openSUSE MicroOS	Ultramarine Linux	DietPi	Manjaro ARM	NixOS
Add Integer (MBs/s)	4086.71	3592.43	3529.95	**4282.88**	3933.36	3711.25
Copy Integer (MB/s)	3478.46	4077.03	3831.50	4357.96	3429.69	**4441.79**
Scale Integer (MB/s)	3354.21	4026.05	3843.28	4520.45	3673.79	**4592.11**
Triad Integer (MB/s)	2639.11	**4163.78**	3601.72	2872.03	2430.06	3971.51
Add Float (MB/s)	3960.02	3915.32	3467.42	**4325.34**	3990.87	3825.25
Copy Float (MB/s)	3537.16	3983.38	3873.12	**4895.8**	3610.61	4419.92
Scale Float (MB/s)	3677.05	4046.20	3878.05	**4938.11**	4038.5	4137.79
Copy128MiB (MiB/s)	**2459.777**	2283.713	2239.169	2204.937	2349.814	2172.112
Copy128MiB-FixedSize (MiB/s)	**2409.189**	2360.534	2223.789	1993.410	2329.071	2165.964

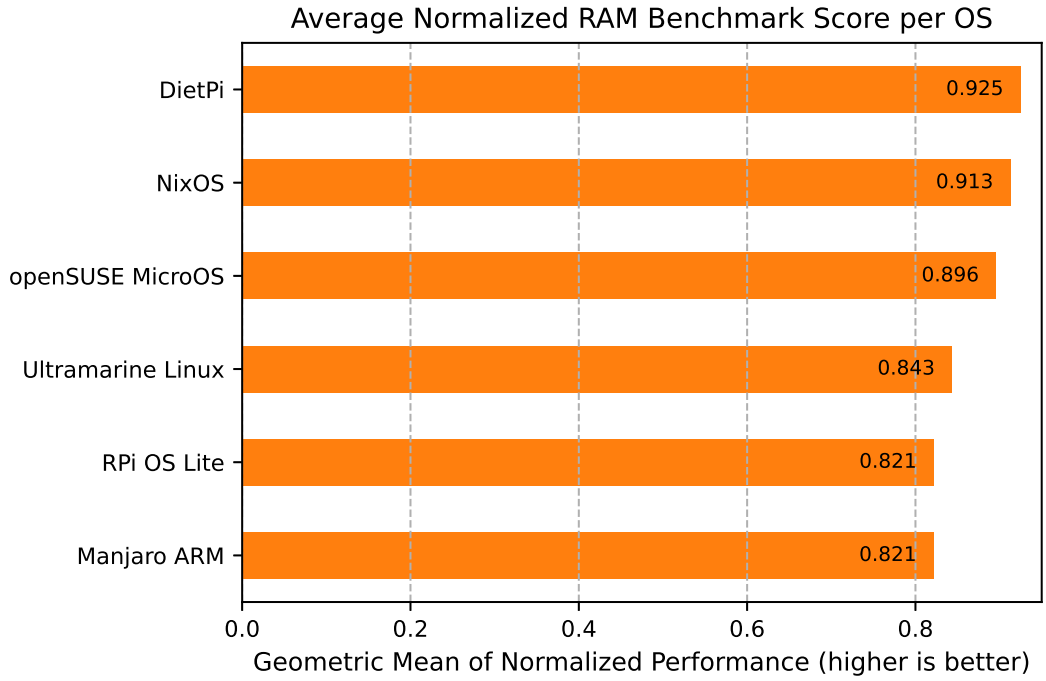

Figure 6.3: Average Normalized RAM Benchmark Score per OS

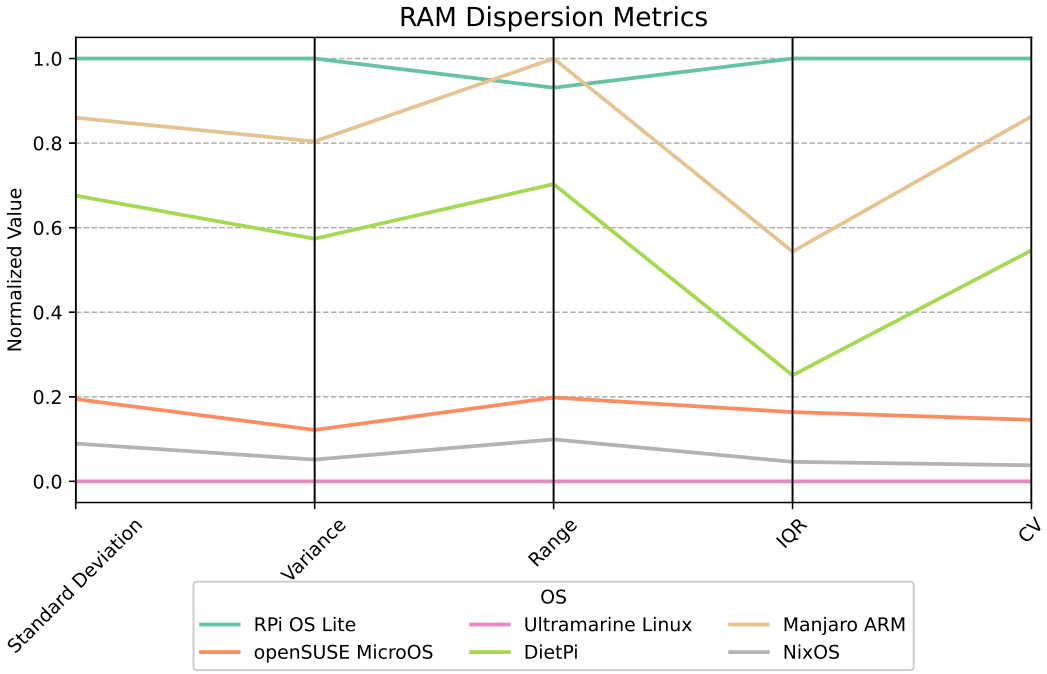

Figure 6.4: RAM Dispersion Metrics (Normalized)

RAM Performance Summary. For RAM performance, as presented in Table 6.2 and Figure 6.3, **DietPi ranks first** with a normalized average score of 0.925, demonstrating consistent dominance in floating-point operations and competitive results in integer-based tests. It led in four metrics: Add Integer, Add Float, Copy Float, and Scale Float. This is

particularly notable considering DietPi's minimalist design, as it combines a lightweight footprint with high memory throughput. To minimize disk writes and enhance performance, DietPi stores temporary files and logs in a tmpfs RAM drive by default. This strategy not only speeds up operations involving these files but also reduces wear on storage devices, which is particularly beneficial for systems using SD cards [122].

NixOS follows closely with a score of 0.913, leading in Copy Integer and Scale Integer, and showing strong performance across the remaining benchmarks. It also achieved high results in both MBW tests, indicating efficient memory copy behavior.

openSUSE MicroOS exceeded expectations with a score of 0.896, especially when compared to its weaker CPU results. It achieved the top score in the Triad Integer benchmark and remained consistently near the top in other metrics. Notably, openSUSE MicroOS is one of the few ARM64 distros still built for a 64 kB kernel page size [123] [124]. Recent snapshots of openSUSE MicroOS for ARM64 include a `kernel-64kb` flavor, indicating ongoing support for this configuration [125]. This larger page size reduces the number of required TLB (Translation Lookaside Buffer) entries by a factor of 16, since 64 kB pages are sixteen times larger than the standard 4 kB pages. As a result, the processor can cover sixteen times more memory with the same number of TLB entries. This is particularly beneficial during memory-intensive workloads like the Triad kernel, which involves three large arrays (`A + B × scalar → C`) and is especially prone to TLB misses. Additionally, openSUSE patches `binutils` to make 64 kB the default common-page size [123].

By contrast, **Ultramarine**, despite showing stronger CPU results, fall behind in memory-intensive workloads like Triad. One contributing factor could be its kernel configuration: it ships with `CONFIG_TRANSPARENT_HUGEPAGE_MADVISE=y` rather than `ALWAYS` [126]. *Transparent Huge Pages (THP)* are a Linux feature that allows the system to use larger memory pages, typically 2 MB instead of the standard 4 kB, to reduce TLB pressure and improve memory access efficiency. However, when THP is set to `MADVISE`, the kernel only promotes standard pages to huge pages if the application explicitly requests it using `madvise(MADV_HUGEPAGE)`. The `Triad` benchmark does not issue such calls. As a result, its arrays remain allocated as 4 kB pages, and no promotion to 2 MB huge pages occurs.

Without huge pages, the TLB must manage many more entries to cover the same memory region, increasing the likelihood of TLB misses and reducing overall performance. This illustrates how kernel configuration and application behavior jointly determine whether advanced memory features like THP are effectively utilized.

Consistency and Dispersion. Beyond throughput, stability matters. Dispersion metrics reveal key insights (see Figure 6.4).

While **DietPi** ranks first in geometric mean RAM performance (0.925), its dispersion metrics reveal some notable variability. It exhibits a mid-to-high range on both **range** and **variance**, indicating that although it excels in specific benchmarks (e.g., floating-point operations), its performance fluctuates more across metrics compared to top contenders like NixOS or Ultramarine Linux. Interestingly, its **IQR** is among the lowest, suggesting that the middle 50% of its scores are tightly grouped, but outliers (such as lower scores in the MBW tests) drive up overall variability. The **CV** also remains moderate, indicating a decent balance between high mean and spread.

NixOS again demonstrates its well-rounded nature, showing not only strong performance across the board but also consistently low dispersion metrics. It maintains one of the lowest values for **standard deviation**, **variance**, and **range**, meaning its RAM performance is both high and stable across all test types. This is a strong indicator of a mature memory stack configuration and optimized memory subsystem. NixOS also achieves a low **IQR** and **CV**, reinforcing its robustness and reliability, especially important for edge deployments where predictability is critical.

openSUSE MicroOS also performs well in terms of consistency, maintaining relatively low values for **variance**, **CV**, and **IQR**. Its standard deviation is slightly higher than NixOS, but still well within the range of a consistent performer. This matches its position as a runner-up in RAM throughput, and its statistical stability strengthens its case as a reliable system under memory-intensive conditions.

Despite ranking fourth in mean RAM performance (0.843), **Ultramarine Linux** exhibits the lowest dispersion across all metrics. Its near-zero values in **standard deviation**, **variance**, **range**, and **CV** suggest extremely stable behavior, even if it does not achieve

the absolute highest throughput. For applications where predictable memory performance is prioritized over peak bandwidth, Ultramarine might be an excellent candidate. Both **RPi OS Lite** and **Manjaro ARM** show higher variability, especially in terms of **range** and **IQR**. This indicates fluctuating performance between specific tests, which might be due to differences in memory management strategies or lower optimization for RAM-bound operations. Their lower geometric mean scores (both at 0.821) combined with high dispersion reinforce that their performance is not only less competitive on average but also less consistent.

Conclusion. Considering both average performance and consistency, NixOS stands out as the most balanced RAM performer. DietPi is ideal for raw memory throughput on constrained devices, while Ultramarine Linux offers unmatched stability. openSUSE MicroOS strikes a strong middle ground. RPi OS Lite and Manjaro ARM may suffice for general use but are less optimized for memory-intensive or latency-sensitive edge workloads.

Disk I/O Results

Table 6.3: Disk I/O Performance Comparison of Edge OSs on a Common Software Stack

Metric	RPi OS Lite	openSUSE MicroOS	Ultramarine Linux	DietPi	Manjaro ARM	NixOS
DiskSeqRead (MB/s)	39.6	**40.3**	40.2	39.6	33.1	37.2
DiskSeqWrite (MB/s)	**15.8**	2.921	15.0	15.0	15.3	15.6
CacheRead (MB/s)	**4571.61**	3709.73	4551.72	3758.39	3806.50	4570.75
CacheWrite (MB/s)	13038.11	10600.30	13038.19	10805.54	10841.84	**13048.30**
Unpack Kernel (s)	23.31	65.63	22.35	34.74	**13.27**	14.36
1000 files-1MB (files/s)	25.9	18.5	23.1	24.1	24.8	**26.0**

Disk I/O Performance Summary. In terms of disk I/O performance, as shown in Table 6.3 and Figure 6.5, **NixOS clearly leads** in overall Disk I/O performance, achieving top results in Cache Write (13,048.30 MB/s) and FS-Mark (26.0 files/sec), and scoring near the best in Cache Read (4570.75 MB/s) and Unpack Kernel (14.36 s). Its consistently strong performance across all six benchmarks reflects a balanced and efficient I/O

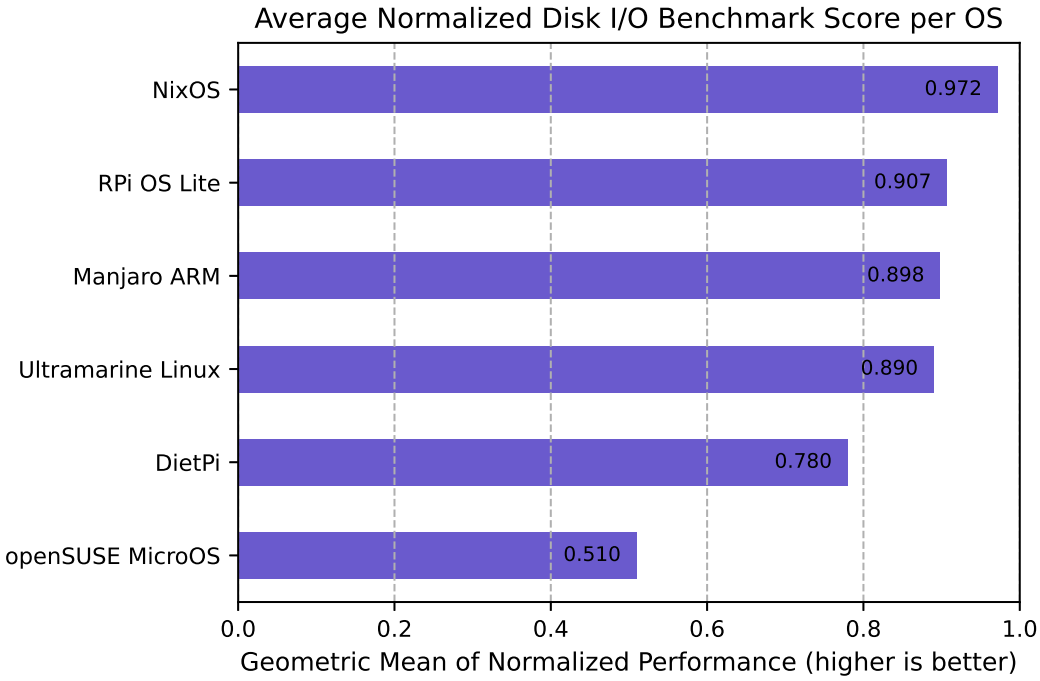

Figure 6.5: Average Normalized Disk IO Benchmark Score per OS

profile.

This behavior can be attributed to the efficient filesystem layout used by NixOS. All binaries and dependencies are stored in the `/nix/store`, which follows a content-addressable and read-only structure. This approach helps minimize file duplication, enhances cache locality, and reduces the number of in-place file modifications. These characteristics lower I/O overhead and improve performance in tasks involving directory traversal, large file reads, or unpacking compressed archives, such as those measured by filesystem benchmarks. Moreover, NixOS features minimal background I/O activity. Since services are only enabled explicitly, it tends to run fewer I/O-heavy daemons, such as journald in persistent mode, log rotation tools, or background software updaters. It also maintains a stateless and clean system environment. Unlike traditional systems, NixOS does not mutate system state during package installations or upgrades. This helps reduce file fragmentation, metadata overhead, and leftover temporary files. As a result, disk usage remains clean and predictable, which is particularly advantageous in benchmarks like FS-Mark and kernel unpacking.

RPi OS Lite follows closely with a normalized score of 0.907. It leads in Cache Read

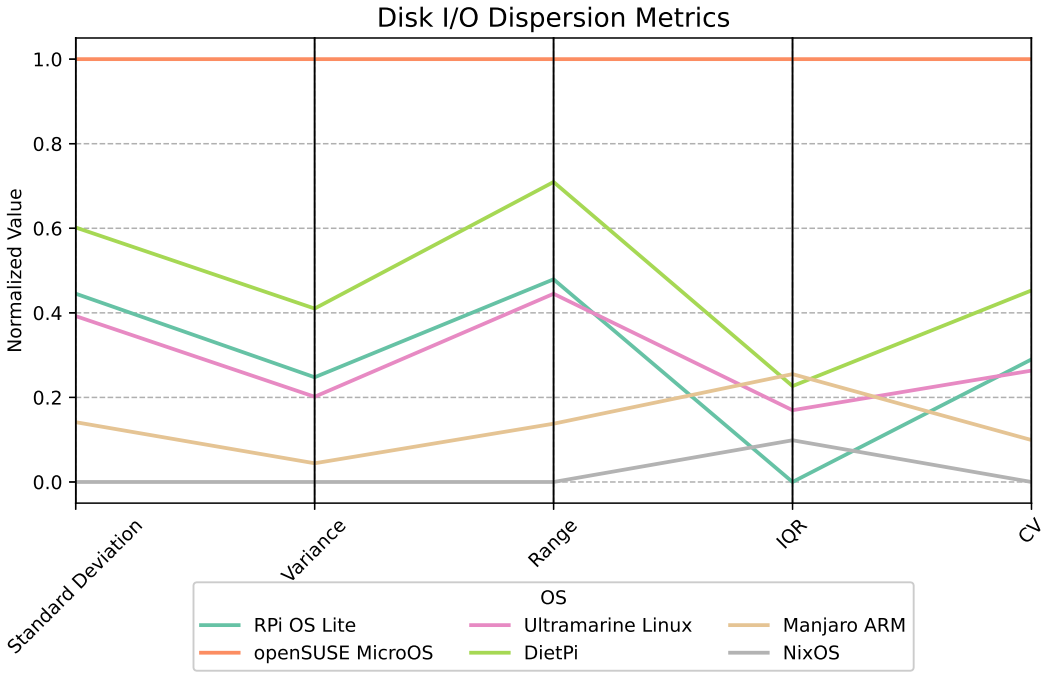

Figure 6.6: Disk I/O Dispersion Metrics (Normalized)

(4571.61 MB/s) and Sequential Write (15.8 MB/s), with solid performance in FS-Mark and Unpack Kernel. Its lightweight kernel and minimal services likely contribute to efficient, low-latency disk transactions, which is especially useful for logging and edge storage workloads on Raspberry Pi devices.

Consistency and Dispersion. While NixOS demonstrates the highest average normalized Disk I/O performance, statistical dispersion metrics provide a deeper understanding of how consistently that performance is maintained across different tests (see Figure 6.6). **NixOS** not only performs best on average but is also the most consistent. It shows minimal variance, range, and interquartile spread, indicating tightly clustered results and reliable behavior across all tests. Its low CV confirms stability relative to its strong mean.

Manjaro ARM shows moderate consistency, with slightly more spread in IQR and CV but no extreme outliers. It's a solid general-purpose option.

openSUSE MicroOS displays the highest variability, caused by strong read performance (40.3 MB/s) but weak write performance (2.921 MB/s). This imbalance leads to high standard deviation and CV, making it less predictable and potentially unsuitable

for critical I/O-bound workloads.

DietPi performs well in some tests (Cache Read/Write) but lags in others like Unpack Kernel, resulting in one of the widest performance ranges.

RPi OS Lite has an almost negligible IQR, suggesting that most of its performance values are concentrated within the middle 50% of its dataset. This reveals that while it does have strong performance peaks (e.g., Cache Read), they are not isolated outliers but representative of its broader performance distribution.

Ultramarine Linux presents a balanced picture, with moderate dispersion across all statistical measures. While it does not lead any metric, it maintains competitive results in key tests like Cache Read (4551.72 MB/s) and Unpack Kernel (22.35 s), making it a dependable choice for scenarios prioritizing predictable, middle-ground performance.

Conclusion. For disk-intensive edge workloads, NixOS is the most robust choice, offering top-tier performance with exceptional consistency. RPi OS Lite is highly efficient and stable, making it ideal for Raspberry Pi-based deployments. Ultramarine provides steady general-purpose performance. DietPi is viable when performance variability is acceptable. openSUSE MicroOS requires caution due to its inconsistent write behavior.

6.4 Discussion and Recommendations

The comparative performance evaluation across CPU, RAM, and Disk I/O dimensions reveals a diverse landscape of strengths and trade-offs among the tested operating systems. The analysis integrates both average normalized performance and statistical dispersion to offer a holistic perspective suitable for decision-making in edge environments

Key Observations

NixOS delivers top-tier performance across CPU, RAM, and disk I/O with consistently low dispersion. Its stateless, declarative architecture and optimized toolchain make it well-suited for scalable and secure edge deployments.

Ultramarine Linux is the most stable across benchmarks, excelling in performance con-

sistency rather than peak throughput. Its predictable RAM behavior and minimal variability make it a strong choice for real-time or stability-sensitive applications.scenarios.

openSUSE MicroOS performs well in RAM workloads, likely due to its 64 kB page size support. While it shows weak CPU and erratic disk results, its consistent RAM behavior makes it appealing for container-first and memory-focused deployments.

DietPi offers the best RAM throughput and competitive CPU performance, making it an attractive option for resource-constrained systems. However, it exhibits higher dispersion in several categories, particularly Disk I/O. While ideal for lightweight memory-bound workloads, DietPi may require additional tuning to ensure predictability across heterogeneous tasks.

RPi OS Lite provides good average performance and consistent behavior across all benchmarks. It ranks second in Disk I/O and shows minimal dispersion in CPU and memory tests. As a distribution tailored for the Raspberry Pi hardware, its efficiency and predictability make it a safe and practical choice for Raspberry Pi-based edge deployments.

Manjaro ARM performs reasonably across the board but lacks standout strengths and shows higher performance variation, making it less ideal for critical or highly predictable edge applications.

Recommendations

Based on the combined benchmark results and statistical consistency analysis, the following recommendations can be made:

- **For high-performance and stability-critical deployments:** *NixOS* is the most suitable choice. It delivers industry-leading results across all categories with low variability, making it a strong candidate for complex edge environments. Beyond performance, its immutable design adds a critical operational advantage: system state is defined declaratively, upgrades are atomic, and rollback is seamless. This ensures not only high throughput and consistency, but also resilience, reproducibility, and security, which are core requirements for scalable and autonomous edge deployments. As such,

NixOS uniquely aligns performance excellence with modern infrastructure principles.

- **For predictability-first or real-time workloads:** *Ultramarine Linux* is ideal due to its exceptionally low dispersion, ensuring consistent behavior even if it does not lead in raw speed.
- **For memory-sensitive or RAM-optimized systems:** *DietPi* excels in RAM throughput and is suitable where lightweight footprints are necessary, provided variability in disk performance is acceptable.
- **For experimental or container-first scenarios with strong memory handling:** *openSUSE MicroOS* is a solid option if disk I/O is not the primary bottleneck.
- **For general-purpose edge workloads:** *RPi OS Lite* is a natural fit, balancing performance and consistency in a package tailored to the hardware. *Manjaro ARM* is also usable, but its higher dispersion indicates the need for further tuning or consideration of more consistent alternatives.

7 Testing the Update Resilience of OS Architectures to Power Failures

7.1 Objective

The second set of experiments evaluates and compares the resilience of different edge-compatible operating systems when a system update process is unexpectedly interrupted by a power failure. This scenario is highly relevant in edge computing environments, where devices are often deployed in unreliable or remote locations and may be exposed to power instability. As a concrete example, in residential environments where edge devices are under the direct control of the user, it is possible that the user leaves for vacation and shuts off the electricity in the house. If a system update is in progress at that time or has been scheduled shortly before the power is cut, the update process may be interrupted. This can lead to an incomplete or corrupted system state, potentially making the device unusable without anyone available to perform a recovery.

Moreover, even in urban or well-managed settings, power grids are not immune to disruption. A recent large-scale blackout affecting Spain, Portugal, and parts of France in April 2025 serves as a reminder that electricity providers cannot guarantee uninterrupted service at all times. During this event, millions were affected by a prolonged power outage that disrupted both residential and industrial infrastructure [127]. Events like this highlight the need for robust update mechanisms that can tolerate unexpected shutdowns without compromising system integrity.

Ensuring that an operating system can gracefully handle such failures without compromising system integrity is critical for maintaining the reliability and maintainability of edge infrastructures. Indeed, operating system updates often involve changes to critical system components such as the kernel, libraries, bootloader, or configuration files. An unexpected power cut during this process can leave the system in a partially-updated or corrupted state, potentially rendering it unbootable.

To explore how different system architectures influence resilience in such scenarios, this experiment includes a comparative analysis of both mutable and immutable operating systems. Mutable systems perform in-place modifications during updates. In contrast immutable systems employ mechanisms like atomic updates, snapshotting, or generational rollbacks.

By testing a representative set of operating systems across this architectural spectrum, the experiment assesses how update mechanisms and filesystem designs affect recovery behavior, user intervention needs, and downtime. These insights are particularly valuable for guiding OS selection in scalable, unattended edge deployments where robustness against power loss or environmental instability is critical.

The OS set and hardware platform used in this experiment remain consistent with earlier performance tests. Immutable systems (openSUSE MicroOS, NixOS) are compared with mutable ones (Raspberry Pi OS Lite, DietPi, Manjaro ARM, Ultramarine Linux) to evaluate fault tolerance under identical failure conditions. The testbed is based on a Raspberry Pi 4B, and its specifications are provided in Table 5.1, ensuring fairness and reproducibility across all test cases.

7.2 Test Procedure

The test procedure consists of **four main phases**, designed to simulate and evaluate the impact of an unexpected power failure during a system update. Each step is conducted in a controlled and repeatable manner across all tested operating systems.

1. **Preparation:** To ensure that meaningful updates are available during testing, each

operating system is flashed using an official release image that is at least 4 to 8 weeks old at the time of the experiment. This approach increases the likelihood of system-level updates, such as kernel or core library changes, and realistically reflects the state of edge devices that have been deployed for some time without maintenance. Following the image installation, each system undergoes an initial boot and baseline configuration, including network setup and SSH access where applicable.

2. **Update Triggering:** Once the system is operational, the standard update mechanism for each OS is invoked. Update progress is monitored using system logs and, where possible, through a serial console connection. The specific commands used to initiate updates are as follows:
   - Debian-based systems (Raspberry Pi OS, DietPi): `apt full-upgrade`
   - Manjaro ARM: `pacman -Syu`
   - Ultramarine Linux: `dnf upgrade`
   - NixOS: `nixos-rebuild switch -upgrade`
   - openSUSE MicroOS: `transactional-update dup`
3. **Power Failure Injection:** During the update process, at approximately **30% to 60%** completion (estimated based on console output and observed update duration), a simulated power failure is introduced by physically disconnecting the power supply from the Raspberry Pi. This step aims to interrupt the update at a critical point, increasing the chances of exposing update-related vulnerabilities or inconsistencies.
4. **Post-Reboot Observation:** After power is restored, the device is rebooted and its behavior is closely observed. The evaluation criteria include whether the system successfully boots, the presence of any broken packages or corrupted filesystems, and whether manual recovery steps are required. If the system boots successfully, service availability is also assessed to determine whether critical services are operational and stable.

To ensure that the results are robust and not dependent on timing artifacts or one-off behaviors, each test scenario was repeated multiple times. This included varying the

precise moment of the power interruption slightly in each trial, while keeping the overall window between 30% and 60% of the update progress. Repeating the experiments helped verify the consistency of observed outcomes and minimize the impact of random variation. This approach enhances the reliability and fairness of the comparison across operating systems and ensures that the conclusions drawn reflect typical rather than exceptional system behavior.

7.3 Results and Analysis

7.3.1 Immutable Operating Systems

openSUSE MicroOS. For openSUSE MicroOS, the update was initiated using the command `transactional-update dup`, which performs a full system upgrade in a transactional and atomic manner. During testing, a power failure was introduced partway through the update process. Upon restoring power, the system booted normally on the first attempt, without any noticeable delay or service degradation.

Upon logging in and re-running the update command, the system simply restarted the transactional update process from the beginning. No signs of corruption, partial changes, or inconsistent system state were observed. This behavior confirms that the update had not yet been activated, since MicroOS applies updates to a separate Btrfs snapshot that is only switched into use after a successful reboot. Because the power failure occurred before the reboot phase, the new snapshot was never activated, and the system continued running the previous stable version without impact.

This result highlights the robustness of openSUSE MicroOS's update mechanism. Its ability to discard incomplete update states and cleanly retry the process without manual intervention makes it particularly well-suited for edge environments. The separation between the running system and the update target ensures high resilience to interruptions, aligning with the goals of reliable, unattended operation in distributed infrastructures.

NixOS. NixOS follows a declarative, immutable system design and uses the `nix` package manager along with `nixos-rebuild` to perform system updates. When the command

`sudo nixos-rebuild switch -upgrade` is executed, the system builds a new configuration generation in the Nix store and only activates it upon successful completion. This approach ensures that the current running system remains untouched during the update process.

In this experiment, a power failure was simulated while the system was performing the update. Despite the interruption, the system booted successfully on the first attempt after power was restored. This is consistent with the expected behavior of NixOS, as the existing system generation remains fully functional and no active changes are made to the live system until the new configuration is built and switched.

Upon re-running the update command after reboot, the process resumed without error. Since the interrupted build process had not completed, no changes had been applied, and the system simply restarted the update operation. No corruption, degraded state, or manual recovery was necessary.

This result highlights the resilience of NixOS to failures during system updates. Its functional and atomic update model ensures that incomplete or failed updates have no effect on the running system. This design makes NixOS particularly well-suited for deployment in edge environments where stability, recoverability, and unattended operation are essential.

7.3.2 Mutable Operating Systems

Raspberry Pi OS Lite. During testing with **RPi OS Lite**, a power failure was intentionally introduced while the `sudo apt update` command was executing. Although this command typically updates local package metadata and does not directly install or upgrade software packages, it modifies critical files such as those in `/var/lib/apt/lists/` and manages lock files related to the package database. If interrupted during this process, especially when other system services or automated maintenance tasks are accessing the package manager at the same time, the system may be left in an inconsistent or partially modified state.

After the simulated power loss, the first reboot attempt failed. This suggests that the

system was temporarily unable to initialize certain components correctly. However, the second boot attempt was successful, indicating that some transient issues had been resolved or bypassed during the boot process. Upon accessing the system and running `sudo apt full-upgrade`, the following error was returned:

```
E: dpkg was interrupted, you must manually run
'sudo dpkg -configure -a' to correct the problem
```

This message indicates that a package operation managed by `dpkg` had been interrupted before completion, leaving the system's package database in an incomplete configuration state. In this situation, the user must manually restore the integrity of the package manager by running the `sudo dpkg -configure -a` command. This reconfigures any partially installed packages and ensures that the package management system can resume normal operation.

This result highlights a critical limitation of mutable operating systems in the context of edge computing. Even if no packages are being actively installed, a power loss during routine maintenance operations such as `apt update` can result in system instability and require manual recovery steps. In unattended edge deployments, this behavior poses a significant reliability risk. It emphasizes the importance of update mechanisms that can tolerate abrupt interruptions, such as those employed in immutable or transactional operating systems.

DietPi. DietPi, a minimal Debian-based distribution optimized for low-resource environments, uses the same package management system as Raspberry Pi OS Lite, namely `apt` and `dpkg`. During testing, a simulated power failure was introduced while the system was executing `sudo apt full-upgrade`. Despite DietPi's smaller system footprint and shorter update duration, the effects of the interruption closely resembled those observed on Raspberry Pi OS.

Following the power cut, the first reboot attempt failed, indicating that the system had entered an unstable or partially configured state. On the second boot, the system managed to reach a login shell. However, when attempting to re-run the upgrade command,

the following error was displayed:

```
E: dpkg was interrupted, you must manually run
‘sudo dpkg -configure -a’ to correct the problem
```

This confirms that the interruption had left `dpkg` in an inconsistent state, with at least one package partially configured. Manual intervention was required to recover the system by running `sudo dpkg -configure -a`, after which the package manager resumed normal operation.

This outcome demonstrates that, like Raspberry Pi OS Lite, DietPi does not include built-in resilience mechanisms such as transactional updates or atomic rollbacks. Although its minimalism may reduce the update surface, it does not prevent corruption or instability when updates are interrupted. As a result, DietPi shares the same limitations as other mutable Debian-based distributions in scenarios involving unexpected power loss during critical system maintenance.

Manjaro ARM. In the Manjaro ARM test case, a power failure was introduced while the system was running `sudo pacman -Syu`, which is the standard command for performing a full system upgrade on Arch-based systems. Unlike some other mutable systems tested, the system successfully booted on the first attempt after power was restored, although the boot time was slightly longer than usual. This suggests that while the update process was interrupted, critical system components were not in the middle of being modified, or the package manager was able to recover to a minimally stable state.

Upon logging in and re-running `sudo pacman -Syu`, the following error was encountered:

```
error: failed to synchronize all databases (unable to lock database)
```

This error indicates that the package database was left in a locked state, most likely due to the abrupt shutdown. Pacman creates a lock file (typically `/var/lib/pacman/db.lck`) during update operations to prevent multiple processes from modifying the package database simultaneously. Since the update was interrupted, the lock file persisted and prevented further package operations from proceeding.

This issue was resolved by manually removing the lock file using the command:

```
sudo rm /var/lib/pacman/db.lck
```

After this manual step, the package manager resumed normal operation and the system was able to complete the update process without further error.
The results from this test highlight that although Manjaro ARM remained bootable after an interrupted update, its package manager was left in an unusable state until manual intervention was performed. This reinforces the broader conclusion that mutable rolling-release systems are particularly sensitive to update-time interruptions. Even if they do not immediately fail to boot, they may enter an inconsistent state that prevents automated or unattended recovery. This behavior makes them less suitable for deployment in edge environments where resilience and autonomy are critical.

Ultramarine Linux. Ultramarine Linux, in its standard edition, is a Fedora-based mutable operating system that uses the `dnf` package manager for system updates. During testing, a power failure was simulated while the system was executing `sudo dnf upgrade`. Since `dnf` performs in-place updates that modify the live filesystem, this architecture does not provide transactional guarantees or rollback mechanisms during updates.
Following the interruption, the system failed to boot on the first attempt. This suggests that one or more critical components, such as the kernel, system libraries, or the init system, may have been in the process of being modified at the time of the power loss. As a result, the system entered an inconsistent or partially updated state. On the second boot attempt, the system was able to reach the login shell, although some warnings appeared during startup, indicating degraded system integrity.
Upon running the update command again, `dnf` produced errors indicating that the RPM database was in an inconsistent state, which required manual recovery. This was resolved by rebuilding the RPM database using the following commands:

```
sudo rm -f /var/lib/rpm/__db* && sudo rpm --rebuilddb
```

After these recovery steps, the update process could be resumed normally. This outcome highlights the fragility of mutable systems such as Ultramarine Linux when exposed to power interruptions during update operations. The lack of transactional protection and

the need for manual recovery make such systems less suited to unattended or remote edge environments, where availability and fault tolerance are critical operational requirements.

Tables 7.1 and 7.2 summarize the results of the power failure experiments, comparing the resilience of each tested operating system in terms of boot success, need for recovery, and post-reboot service availability.

Table 7.1: Resilience to Power Failure – RPi OS Lite, DietPi, Manjaro ARM

Category	**RPi OS Lite**	**DietPi**	**Manjaro ARM**
OS Type	Mutable (Debian/APT)	Mutable (Debian/APT)	Mutable (Arch/Pacman)
Boot Success	On 2nd boot only	On 2nd boot only	Yes
Recovery Needed	Yes	Yes	Yes
Service Availability	Partial	Partial	Partial
Notes	Manual `dpkg -configure -a` required	Manual `dpkg -configure -a` required	Lock file issue, removed `db.lck`

Table 7.2: Resilience to Power Failure – Ultramarine Linux, NixOS, openSUSE MicroOS

Category	**Ultramarine Linux**	**NixOS**	**openSUSE MicroOS**
OS Type	Mutable (Fedora/DNF)	Immutable (Nix)	Immutable ('transactional-update')
Boot Success	No	Yes	Yes
Recovery Needed	Yes	No	No
Service Availability	Partial	Full	Full
Notes	Required `rpm -rebuilddb` to fix RPM DB	Booted into last active generation; update not applied	Booted from last valid snapshot; update not applied

8 Evaluating PQC Performance Across OS Architectures

8.1 Specific Methodology

This section outlines the methodology used to evaluate the performance of PQC schemes across different OS architectures. The goal is to assess the computational impact of PQC operations in realistic, resource-constrained edge environments.

8.1.1 Selected PQC Algorithms

This analysis covers two of the three algorithms standardized by NIST in 2024, as detailed in Section 3.1.1:

- **FIPS 203 (ML-KEM)**: This key encapsulation mechanism supports three parameter sets: **512, 768, and 1024**. These values correspond to the module dimension in the lattice-based construction and directly influence both the computational cost and the security level. Specifically, ML-KEM-512, 768, and 1024 target NIST security levels 1, 3, and 5, respectively. These levels represent increasing resistance to attacks, with Level 1 comparable to AES-128 and Level 5 to AES-256 [128]. As the parameter set increases, the algorithm provides stronger cryptographic guarantees but also incurs higher demands in terms of computation time, memory usage, and energy consumption during operations such as key generation, encapsulation, and decapsulation.
- **FIPS 204 (ML-DSA)**: This digital signature scheme includes three parameter sets:

44, 65, and 87. These parameter sets define the configuration for key generation, signing, and verification, with higher values offering stronger security guarantees. Specifically, ML-DSA-44 targets NIST security strength category 2 (equivalent to SHA3-256 [128]), ML-DSA-65 targets category 3, and ML-DSA-87 targets category 5 [129]. As the parameter set increases, so does the expected security level, but at the cost of increased signature size, longer execution time, and higher resource consumption. Additionally, each parameter set includes algorithm-specific variables such as the expected number of signing loop repetitions, which also impact performance. The evaluated operations for this scheme include key generation, signing, and signature verification, all of which scale in cost with the selected parameter set and its cryptographic complexity.

FIPS 205 (SLH-DSA) was not included in this study, as no stable and optimized implementation was available at the time of testing.

The selection of ML-KEM and ML-DSA was guided by their standardization status, practical relevance, and readiness for deployment in real-world systems. Both algorithms have been officially validated and standardized by NIST in 2024, making them trustworthy candidates for post-quantum adoption. These two algorithms also cover distinct cryptographic functions: ML-KEM provides key encapsulation, while ML-DSA offers digital signatures. Including both in the evaluation enables a comprehensive analysis of PQC performance across the two core primitives required for secure communication and authentication.

All implementations were sourced from the **liboqs library**, maintained by the Open Quantum Safe (OQS) project [130]. This library provides a unified API and optimized C implementations of multiple PQC algorithms, making it well-suited for benchmarking and integration testing.

8.1.2 Stress-Based Load Simulation

In addition to the conditions described in the research methodology, cryptographic operations were executed while the system was under moderate load to better approximate real-world edge deployment scenarios. The stress was introduced using the `stress-ng`

tool, with the configuration shown in Table 8.1. This setup ensured that both CPU and memory were actively utilized during benchmarking, mimicking common edge workloads such as local data processing, monitoring tasks, or lightweight inference.

`stress-ng` is a versatile Linux-based workload generator designed to impose a configurable load on various system components, including CPU, memory, I/O, and more. It supports over 300 different stress tests and provides detailed control over how resources are exercised. For this study, it was used to simulate typical multi-tasking behavior by spawning dedicated CPU workers and memory allocators during cryptographic benchmarking.

Using `stress-ng` offers several advantages. It ensures consistent and reproducible stress levels across all tested operating systems, which is critical for fair comparison. Its compatibility with a wide range of Linux distributions, including embedded and minimal OS configurations, makes it especially suitable for this multi-platform evaluation. Furthermore, by introducing controlled background activity, `stress-ng` helps assess how well post-quantum algorithms perform in realistic scenarios where edge devices must juggle cryptographic tasks alongside concurrent workloads.

Table 8.1: System Load Configuration Applied During PQC Benchmarking

Stress-ng Option	**Value**	**Description**
`-cpu`	4	Launches 4 CPU stress workers to consume CPU cycles
`-vm`	2	Starts 2 memory stress workers (virtual memory allocators)
`-vm-bytes`	75%	Each memory stressor allocates 75% of system RAM

Hardware and OS Platforms. All experiments were conducted on the same hardware configuration used in the previous tests, as detailed in Table 5.1. To ensure consistency across all tests, each of the six operating systems was configured with the same set of packages listed in 5.3.1, with the addition of `stress-ng`. The same version of the liboqs library, compiled from source, was used on every system.

8.1.3 Iteration Volume and Statistical Significance

To ensure statistically meaningful and stable measurements, all benchmarked cryptographic operations were executed over a large number of iterations. Each test scenario involved performing between **10,000 and 500,000 repetitions** of the key generation, encapsulation, or signing process, depending on the specific algorithm and its computational footprint. A key objective of the high iteration count is effective **noise reduction**. Running each cryptographic primitive many times minimizes the impact of transient system fluctuations, such as CPU frequency scaling or background service interruptions. Averaging results over a large sample ensures that the reported performance metrics accurately reflect consistent system behavior rather than outliers. Each benchmark was preceded by a calibration phase to determine the ideal iteration count that balances accuracy with test duration, given the device's thermal and operational constraints. This methodology aligns with best practices in empirical cryptographic benchmarking and enables fair comparison across OS architectures and PQC implementations, as demonstrated in related studies [58].

8.1.4 Analysis Approach

To complement the raw execution time measurements, a normalized statistical analysis was conducted to provide a fair and interpretable comparison of ML-KEM and ML-DSA performance across operating systems. The approach is based on geometric mean aggregation of normalized performance metrics, allowing consistent cross-platform evaluation despite differences in absolute timing values.

Normalization of Execution Times. Each OS was evaluated across nine benchmark cases: three ML-KEM operations (Key Generation, Encapsulation, Decapsulation) for each of the three parameter sets (512, 768, and 1024). The same applies for the three ML-DSA operations (Key Generation, Signing, Signature Verification). For each metric i (e.g., ML-KEM-768 Encaps), the normalized performance score $N_{i,j}$ for operating system j is defined as:

$$N_{i,j} = \frac{\min_k(T_{i,k})}{T_{i,j}}$$

where $T_{i,j}$ denotes the raw execution time of operation i on operating system j, and $\min_k(T_{i,k})$ represents the best (i.e., lowest) execution time observed for operation i across all operating systems. A normalized score of 1.0 indicates the best performance for that specific operation, while values below 1.0 indicate slower relative performance. This normalization produces dimensionless ratios that allow fair comparisons across heterogeneous workloads.

Geometric Mean Aggregation. To summarize the overall post-quantum performance of each OS, the geometric mean G_j of its normalized scores is computed:

$$G_j = \left(\prod_{i=1}^{9} N_{i,j} \right)^{\frac{1}{9}}$$

Ranking and Interpretation. Operating systems are ranked based on their geometric mean score G_j, with higher values indicating better overall performance relative to the best observed system. This approach highlights systems that perform well across all parameter sets and operations, rather than excelling in only a subset.

8.2 Results and Analysis

8.2.1 ML-KEM Results

The ML-KEM benchmark results highlight stark differences in how efficiently each OS handles PQC operations under identical hardware and software conditions.

Overall Trends and Leaders. At all three parameter sets (512, 768, and 1024), **Ultramarine Linux emerges as the clear leader**, consistently delivering the lowest latencies across key generation, encapsulation, and decapsulation. At the ML-KEM-512 level, as shown in Figure 8.1, Ultramarine Linux achieves exceptional performance (Keygen: 46 µs, Encaps: 51 µs, Decaps: 53 µs), outperforming all competitors and indicating a highly optimized toolchain and build environment. **Manjaro ARM closely follows**,

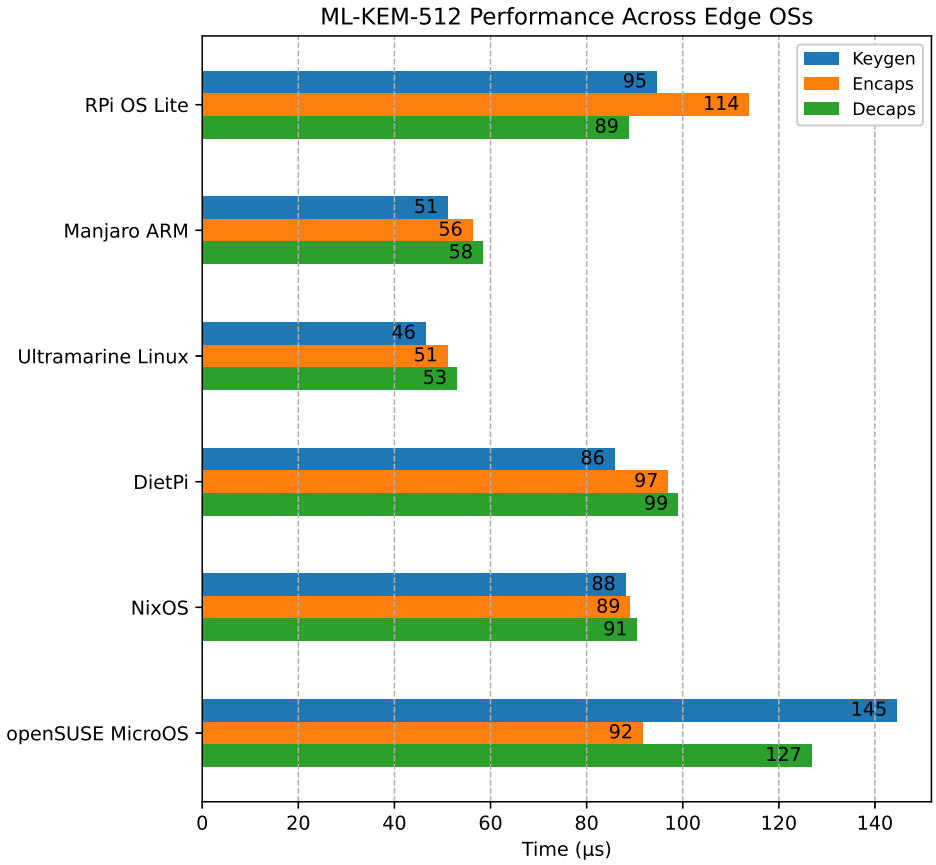

Figure 8.1: Comparing ML-KEM-512 Performance Across Edge OSs Under Identical Workloads

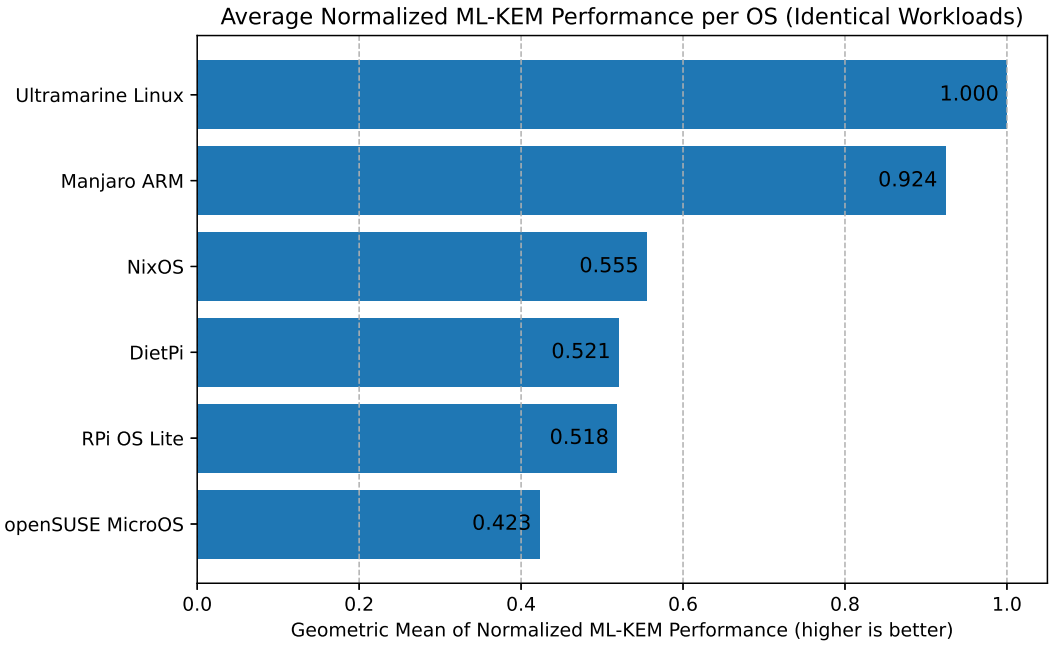

Figure 8.2: Overall ML-KEM Performance Across OSs Under Identical Workloads

maintaining sub-60 µs times across all operations, likely benefiting from architecture-specific optimizations and support for AArch64-accelerated code paths in liboqs.

Lowest Performing OS. On the opposite end, **openSUSE MicroOS** demonstrates the weakest performance, with Keygen and Decaps times reaching 145 µs and 127 µs, respectively. These figures are approximately 2.5–3× slower than the leaders, indicating systemic inefficiencies.

Middle-Tier Performance. **NixOS, DietPi, and RPi OS Lite form a middle tier**, showing moderate and nearly equivalent performance. Their latencies for ML-KEM-512 key generation range from 86–95 µs, with encapsulation and decapsulation remaining within a consistent margin. These results suggest the use of generic, portable builds of liboqs and conservative compiler settings, which prioritize compatibility and stability over maximum speed.

Consistency Across Security Levels. The performance trends remain consistent at higher security levels. In ML-KEM-768 (see Appendix A), Ultramarine Linux maintains its lead with tightly clustered execution times around 78–84 µs. Manjaro ARM again performs strongly, while NixOS, DietPi, and RPi OS Lite remain in the middle range. openSUSE MicroOS continues to lag behind with significantly elevated values (up to 182 µs for encapsulation), further reinforcing its outlier status in PQC workloads.
As expected, ML-KEM-1024, shown in Appendix A, introduces greater computational cost across all OSs, amplifying the relative differences. Ultramarine Linux is again the most efficient, remaining below 125 µs for all operations. Manjaro ARM maintains comparable performance, while the middle-tier operating systems scale in a predictable manner. In contrast, openSUSE MicroOS exhibits a significant performance degradation, with key generation reaching 349 µs and decapsulation taking 288 µs. These values place it well outside the practical latency thresholds required for edge environments where low response times are critical.

Normalized Performance. The normalized performance chart in Figure 8.2 provides a geometric mean-based overview of overall ML-KEM efficiency per OS across all three

parameter sets. This reveals that **Ultramarine Linux leads consistently**, achieving the best average performance across all operations and sizes.

Manjaro ARM performs nearly as well, with a normalized score of 0.924, making it a strong contender for edge deployments requiring efficient PQC operations.

NixOS, DietPi, and RPi OS Lite form a middle tier, with scores between 0.51 and 0.56. These systems may still be viable choices depending on non-cryptographic constraints (e.g., update strategy, package ecosystem).

openSUSE MicroOS ranks lowest with a normalized score of 0.423. Its high execution time variance and significantly slower keygen performance reduce its suitability for PQC-heavy edge workloads. These systemic inefficiencies may be attributed to the use of the `musl` C library, hardened compilation settings, and the reliance on portable C implementations resulting from disabled SIMD optimizations. These results reinforce that toolchain optimization, libc choice, and SIMD support significantly affect PQC efficiency.

Implications for Edge Deployment. For latency-sensitive edge workloads, systems like Ultramarine and Manjaro offer tangible benefits. Others like NixOS may still be viable depending on priorities like update strategy, ecosystem familiarity, or immutability features.

8.2.2 ML-DSA Results

Overall Trends and Leaders. The ML-DSA benchmarks reveal consistent and significant performance differences across operating systems, underscoring the impact of system-level optimizations on lattice-based signature schemes. As with the ML-KEM results, **Ultramarine Linux and Manjaro ARM lead across all three ML-DSA security levels** (44, 65, and 87), delivering the lowest average latencies in key generation, signing, and verification.

At ML-DSA-44 (see Figure 8.3), Ultramarine Linux sets the performance baseline with remarkably low operation times (Keypair: 221 µs, Sign: 969 µs, Verify: 230 µs), followed closely by Manjaro ARM. Their fast and consistent results suggest the presence

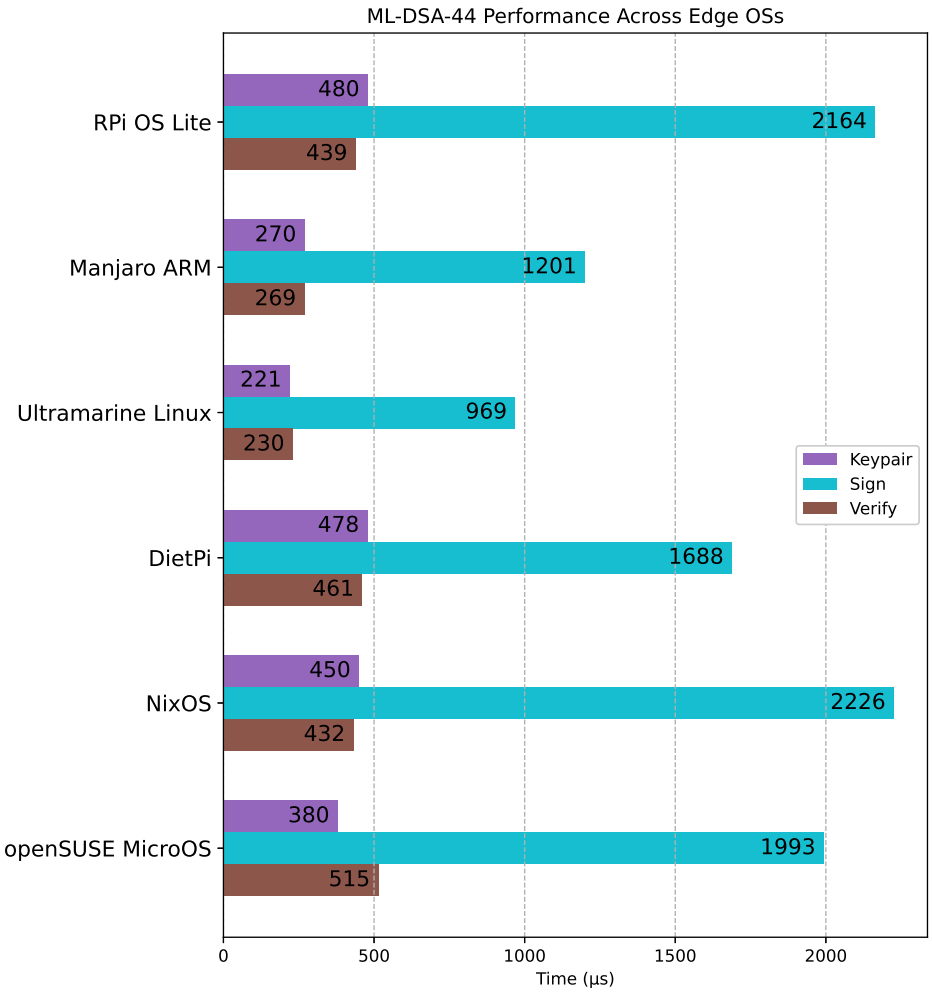

Figure 8.3: Comparing ML-DSA-44 Performance Across Edge OSs Under Identical Workloads

of optimized builds that leverage NEON-accelerated SHAKE implementations and carefully tuned memory allocation strategies, both of which are critical for ML-DSA's hash-intensive workflows. In contrast, openSUSE MicroOS lags significantly, requiring over 2 ms to generate a signature and more than twice as long for verification compared to the leading systems.

As expected, the computational cost increases with higher parameter sets. ML-DSA-65 and ML-DSA-87, shown in Appendix B, involve larger keys and signatures, additional matrix multiplications, and deeper SHAKE hash chains. Signing performance is particularly impacted, with execution times reaching 5515 µs on DietPi, 4511 µs on NixOS, and 3832 µs on RPi OS Lite at the ML-DSA-87 level. These results are approximately two to three times slower than those recorded on Ultramarine or Manjaro, further highlighting

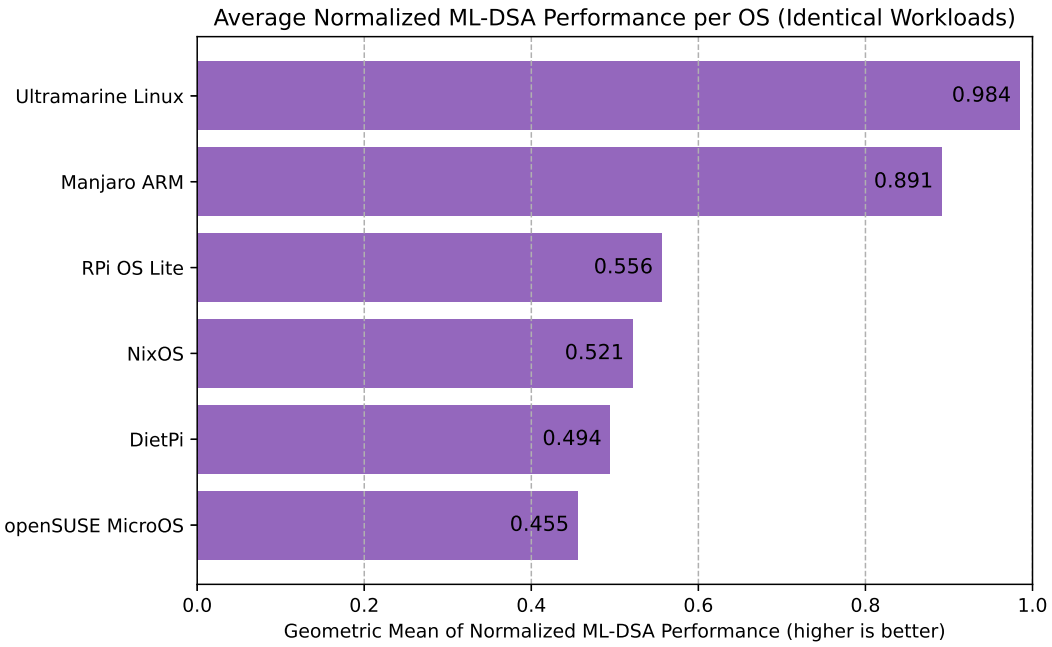

Figure 8.4: Overall ML-DSA Performance Across OSs Under Identical Workloads

the efficiency gap between systems.

Notably, signature verification remains relatively uniform across most OSs, especially at the ML-DSA-44 and ML-DSA-65 levels. This pattern is consistent with ML-DSA's algorithm design, where the verifier performs fewer polynomial operations than the signer. For edge nodes primarily tasked with verification, such as validating firmware updates or secure messages, this makes even mid-tier systems like NixOS and RPi OS Lite viable, despite their longer keypair and signing durations.

Implications for Edge Deployment. For signature-heavy tasks like certificate issuance, device authentication, or consensus protocols, signing performance is critical. Ultramarine Linux and Manjaro ARM stand out by maintaining sub-2500 µs signing times even with ML-DSA-87, making them more suitable for these roles.

This assessment is further supported by the geometric mean summary presented in Figure 8.4. Ultramarine Linux achieves the highest normalized score at 0.984, indicating strong and consistent performance across all workloads. Manjaro ARM follows closely with a score of 0.891, while openSUSE MicroOS ranks the lowest at 0.455. This lower score reflects the consistently high operation latencies observed across the evaluated tasks.

9 Discussion

9.1 Bridging Gaps in Existing Work

This thesis examined how OS design influences secure, scalable, and efficient edge computing in the context of PQC, a critical but often overlooked intersection in the current research landscape. Although PQC and edge computing have each advanced significantly, prior work has largely concentrated on algorithm design, protocol integration, or hardware feasibility, often treating these areas in isolation. The OS, which is essential for managing scheduling, updates, and cryptographic workloads, has received limited attention in this context.

By performing a structured evaluation of Linux-based operating systems with different architectural models, including mutable, immutable, and container-centric designs, this work addresses that gap. It evaluates how practical OS-level design choices affect PQC cryptographic performance, system reliability, and long-term maintainability in realistic edge environments.

The remainder of this chapter expands on these findings. The next sections present comparative results across three critical dimensions: performance trade-offs, resilience during interrupted updates, and the system-level impact on PQC workloads. Together, these insights highlight how OS design directly shapes the readiness of edge platforms for secure operation in a post-quantum world.

9.2 System-level Performance Trade-offs in Edge OSs

Selecting an OS for edge computing requires balancing performance, consistency, and architectural fit. This study shows that some systems offer raw speed, while others excel through stability, both of which are critical traits for real-world edge deployments. **NixOS emerges as the most well-rounded option**, delivering top-tier CPU, RAM, and Disk I/O performance with low statistical variability. Its immutable, declarative design adds operational resilience by enabling atomic upgrades, clean rollbacks, and minimized configuration drift. This makes NixOS not just fast, but also robust and scalable. **Ultramarine Linux** offers unmatched consistency across benchmarks, making it ideal for workloads requiring predictable behavior. **DietPi** stands out for its RAM throughput and minimalism, particularly suited for constrained, memory-bound environments like SD-card-based systems. **Raspberry Pi OS Lite** remains a reliable, low-overhead baseline for Raspberry Pi devices, with balanced and predictable performance. **openSUSE MicroOS and Manjaro ARM** show specific strengths, with MicroOS excelling in immutability and rollback support, and Manjaro offering strong general-purpose usability. However, both exhibit variability that may affect their suitability in critical edge contexts.

9.3 OS Update Resilience in Edge

This study demonstrates that **immutable operating systems clearly outperform mutable ones** in their ability to withstand power failures during system updates. NixOS and openSUSE MicroOS both demonstrated reliable boot behavior following interruptions, with no need for manual recovery and seamless continuation of updates. This reliability is attributed to their atomic and transactional update mechanisms. In contrast, all **mutable OSs** (Raspberry Pi OS Lite, DietPi, Manjaro ARM, Ultramarine Linux) experienced boot issues or required manual recovery. These failures highlight the risks of in-place updates and underscore the need for more robust system design in edge environments. For deployments where autonomy, recoverability, and operational integrity are essential, immutable systems are the clear recommendation.

9.4 OS Influence on PQC

This evaluation reveals that OS architecture significantly affects the performance of PQC operations on edge devices.

Across both the ML-KEM and ML-DSA benchmarks, **Ultramarine Linux** consistently delivered the best results, with the lowest latencies and highest normalized scores. Its optimized toolchain and efficient handling of cryptographic workloads make it especially suitable for PQC-intensive edge applications. **Manjaro ARM** closely followed, maintaining strong performance with competitive signing and encapsulation times, making it a viable alternative for performance-critical deployments. **NixOS, DietPi, and Raspberry Pi OS Lite** formed a middle tier, showing adequate and predictable performance but lacking the level of optimization needed for high-throughput PQC tasks. However, their reliability and broader ecosystem support may make them appealing for general-purpose or verification-heavy roles. **openSUSE MicroOS** ranked last, suffering from high operation latencies and limited SIMD usage. Its secure-by-default build settings may be beneficial from a hardening perspective but impose performance penalties that hinder its PQC suitability in constrained environments.

Overall, for edge systems expected to perform frequent key exchanges or digital signatures under load, selecting a **well-optimized OS like Ultramarine Linux or Manjaro ARM offers significant benefits**. For systems where maintainability or immutability is more critical, **NixOS remains a strong middle-ground choice** with consistent performance and robust operational characteristics.

Collectively, the contributions of this thesis are **threefold**. First, it provides one of the few end-to-end evaluations of post-quantum algorithms (ML-KEM, ML-DSA) across diverse edge-focused operating systems. Second, it demonstrates that immutable systems like NixOS and openSUSE MicroOS offer stronger resilience to update interruptions, which is critical in remote or unreliable edge environments. Third, it proposes a reproducible benchmarking approach and provides system-level insights to guide operating system selection for quantum-secure edge deployments.

10 Conclusion and Future Work

Conclusion for RQ1. Since 2018, research into PQC for edge computing has progressed, especially in algorithm optimization and feasibility on constrained devices. However, most studies address PQC, edge computing, and OS design in isolation, without exploring their intersection. This is a critical gap, given the OS's central role in resource management, scheduling, and update mechanisms.

Although key foundations like the three NIST PQC standards have been established [31], current research still falls short in addressing the **system-level challenges of deploying PQC in large-scale edge environments**. A deeper understanding of OS-level impacts and more integrated solutions are needed to achieve true quantum readiness by 2030. Bridging this gap requires not only algorithmic advances but also a careful re-evaluation of the operating systems that underpin edge deployments, as their design choices critically influence both cryptographic performance and system resilience.

Conclusion for RQ2. Selecting an OS for scalable edge deployments requires balancing operational reliability, long-term maintainability, and resource efficiency. Critical factors include support for atomic and verifiable update mechanisms, remote management capabilities, cryptographic agility, and reproducibility across devices. Edge operating systems must also operate securely and autonomously in constrained environments, often without physical access. **Immutable and container-based systems** (e.g., NixOS, openSUSE MicroOS) offer strong guarantees in consistency, rollback safety, and lifecycle automation, making them increasingly suitable for edge-scale deployments. In contrast, **mutable systems** provide flexibility but are prone to drift and operational fragility at

scale. Importantly, OS design directly impacts the performance of cryptographic operations, including PQC workloads. Kernel behavior, scheduling, and background processes influence how efficiently these operations run under real-world edge conditions, a dimension that is often overlooked in traditional OS selection processes.
In short, OS choice is not just about footprint or feature set; it fundamentally shapes the scalability, security, and cryptographic resilience of edge infrastructure.

Three key experiments were conducted to evaluate different aspects of OS behavior on edge devices. The first experiment involved **system-level benchmarking** of CPU, memory, and I/O performance across six OSs running on a Raspberry Pi 4B. The goal was to assess baseline performance differences between various OS architectures. The second experiment tested update resilience by **simulating power failures during critical OS update phases**. This was done to observe how different systems handle recovery in scenarios where physical intervention is not possible. The third experiment focused on PQC. It measured the **performance of ML-KEM and ML-DSA primitives** under controlled conditions to assess the cryptographic readiness of each OS.

A consistent and reproducible testbed was set up using a **Raspberry Pi 4B** board, with each OS configured to run the same Linux kernel version (6.12). A uniform software environment was installed to avoid dependency-induced variability. For PQC benchmarking, systems were tested under moderate background load to mimic real-world edge conditions. The evaluation included a diverse set of OSs representing different architectural approaches. The immutable systems selected were **NixOS** and **openSUSE MicroOS**, both designed for reliability and minimal system drift. The mutable systems included **Raspberry Pi OS Lite, DietPi, Manjaro ARM, and Ultramarine Linux**, which offer more traditional, stateful configurations with varying levels of optimization for resource-constrained hardware. This selection enabled a comprehensive comparison of how different OS architectures influence system behavior, update robustness, and PQC performance in edge environments.

Conclusion for RQ3. NixOS demonstrated the most balanced overall performance across CPU, memory, and disk I/O benchmarks. Ultramarine Linux and Manjaro ARM stood out in PQC tasks, delivering the lowest latencies in key exchange and signature operations. Immutable systems outperformed their mutable counterparts in handling interrupted updates. Both NixOS and openSUSE MicroOS were able to boot successfully after simulated power failures. In contrast, all mutable systems required manual intervention to recover. Ultramarine Linux achieved the highest efficiency in PQC workloads, with Manjaro ARM performing closely behind. NixOS combined strong cryptographic performance with reliable operational behavior. openSUSE MicroOS showed lower performance in PQC tasks, likely due to its security-hardened build configuration.

The design of the OS is a critical factor in building secure, efficient, scalable, and post-quantum-ready edge infrastructures. Although cryptographic algorithms often take center stage, this study shows that their practical effectiveness and the resilience of the systems that implement them are deeply influenced by the underlying OS architecture. Immutable and declarative systems such as NixOS are well suited for long-term edge deployments, particularly in scenarios that demand consistency and minimal manual intervention. In contrast, performance-oriented systems like Ultramarine Linux and Manjaro ARM show strong potential for handling cryptographic workloads efficiently, especially when high throughput or low-latency responses are needed.
Selecting an OS for post-quantum deployments requires a careful balance between immutability, update reliability, cryptographic performance, and resource constraints. These trade-offs must be considered early in the system design process to ensure secure and sustainable edge computing environments. While this thesis provides a structured evaluation of OS-level impacts on edge system performance and PQC readiness, several aspects remain beyond its scope and present valuable directions for future exploration.

10.1 Limitations and Future Work

This study focused on a representative subset of Linux-based OSs compatible with the Raspberry Pi. Future work could expand to include real-time operating systems (RTOSs),

unikernels, or Yocto/Buildroot-based systems to capture **a broader range of OS design** trade-offs and operational traits.

All experiments used a single hardware platform: the Raspberry Pi 4B with 8GB RAM. While common in edge scenarios, OS behavior could vary across platforms with different storage (e.g., eMMC, NVMe), bootloaders, or **hardware components** like watchdog timers. Future studies could explore these factors to assess cross-platform performance and reliability.

The performance evaluation in this thesis relies on synthetic benchmarks such as CoreMark, RAMspeed. While effective and recognized in research for standardized comparisons, these benchmarks may not fully reflect real-world workload dynamics. Future work could incorporate **application-level tests** such as TensorFlow Lite inference to provide a more comprehensive view of system suitability in practical edge deployments.

In addition, **measuring boot and reboot times** could offer valuable insight, as fast startup is crucial in edge environments with frequent power cycles or OS updates. Evaluating startup latency helps assess system readiness and responsiveness in time-sensitive applications.

Further research could explore deeper aspects of resilience by performing **low-level filesystem integrity checks and analyzing kernel logs**. This would provide insight into system behavior beyond basic recovery from visible failures.

Finally, while this work assessed PQC primitives such as ML-KEM and ML-DSA in isolation, future studies could explore their **integration into end-to-end protocols** like TLS 1.3. Evaluating handshake speed, certificate handling, and hybrid schemes across OSs would yield more practical insights into their edge deployment readiness.

References

[1] European Commission, *Horizon Europe - Funding Programmes and Open Calls* (2025), [Online]. Available: https://research-and-innovation.ec.europa.eu/funding/funding-opportunities/funding-programmes-and-open-calls/horizon-europe_en (visited on 02/05/2025).

[2] European Commission, *A Global as well as Local Flexibility Marketplace to Demonstrate Grid Balancing Mechanisms through Cross-sectoral Interconnected and Integrated Energy Ecosystems enabling Automatic Flexibility Trading* (2025), [Online]. Available: https://cordis.europa.eu/project/id/101096399 (visited on 02/05/2025).

[3] M. Satyanarayanan, "The emergence of edge computing", *Computer*, vol. 50, no. 1, pp. 30–39, 2017, IEEE Computer Society. DOI: 10.1109/MC.2017.9.

[4] S. George, H. George, and T. Baskar, "Edge computing and the future of cloud computing: A survey of industry perspectives and predictions", *Partners Universal International Research Journal (PUIRJ)*, vol. 02, pp. 19–44, Jun. 2023. DOI: 10.5281/zenodo.8020101.

[5] Grand View Research, *Edge Computing Market Size, Share & Trends Analysis Report By Component, By Application, By Industry Vertical, By Organization Size, By Region, And Segment Forecasts, 2024 - 2030* (2024), [Online]. Available: https://www.grandviewresearch.com/industry-analysis/edge-computing-market (visited on 03/30/2025).

[6] J. B. Awotunde, K. Muduli, and B. Brahma, Eds., *Computational Intelligence for Analysis of Trends in Industry 4.0 and 5.0*, 1st ed. Auerbach Publications, New York, NY, USA, 2025. DOI: 10.1201/9781003533023.

[7] W. Shi, G. Pallis, and Z. Xu, "Edge computing [scanning the issue]", *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1474–1481, 2019. DOI: 10.1109/JPROC.2019.2928287.

[8] S. Bagchi, M.-B. Siddiqui, P. Wood, and H. Zhang, "Dependability in edge computing", *Communications of the Association for Computing Machinery*, vol. 63, no. 1, pp. 58–66, Dec. 2019, New York, NY, USA, ISSN: 0001-0782. DOI: 10.1145/3362068.

[9] P. Souza, T. Ferreto, and R. Calheiros, "Maintenance operations on cloud, edge, and iot environments: Taxonomy, survey, and research challenges", *Association for Computing Machinery Computing Surveys*, vol. 56, no. 10, Jun. 2024, New York, NY, USA, ISSN: 0360-0300. DOI: 10.1145/3659097.

[10] D. Moody, R. Perlner, A. Regenscheid, A. Robinson, D. Cooper, National Institute of Standards and Technology, Gaithersburg, MD, USA, *Transition to Post-Quantum Cryptography Standards, NIST Internal Report 8547, Initial Public Draft, Information Technology Laboratory Computer Security Resource Center (CSRC), U.S. Department of Commerce* (2024), DOI: 10.6028/NIST.IR.8547.ipd.

[11] M. Barenkamp, "Steal now, decrypt later - post-quantum-kryptografie und ki", *Informatik Spektrum*, vol. 45, pp. 349–355, Jul. 2022. DOI: 10.1007/s00287-022-01474-z.

[12] Tomas Gustavsson, *NIST Drops New Deadline for PQC Transition, Keyfactor* (2024), [Online]. Available: https://www.keyfactor.com/blog/nist-drops-new-deadline-for-pqc-transition/ (visited on 03/30/2025).

[13] S. P. Singh, A. Nayyar, R. Kumar, and A. Sharma, "Fog computing: From architecture to edge computing and big data processing", *The Journal of Supercomputing*,

vol. 75, no. 4, pp. 2070–2105, Apr. 2019, ISSN: 0920-8542. DOI: 10.1007/s11227-018-2701-2.

[14] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research", *IEEE Access*, vol. 8, pp. 85 714–85 728, 2020. DOI: 10.1109/ACCESS.2020.2991734.

[15] Y. W. Xuehai Hong, "Edge computing technology: Development and countermeasures", *Strategic Study of Chinese Academy of Engineering*, vol. 20, no. 2, 20, p. 20, 2018. DOI: 10.15302/J-SSCAE-2018.02.004.

[16] G. Nain, K. Pattanaik, and G. Sharma, "Towards edge computing in intelligent manufacturing: Past, present and future", *Journal of Manufacturing Systems*, vol. 62, pp. 588–611, 2022, ISSN: 0278-6125. DOI: 10.1016/j.jmsy.2022.01.010.

[17] H. Zhang, Q. Wang, J. Feng, Z. Yang, and H. Wang, "Design of environmental perception system based on edge computing", *Computer Journal*, vol. 32, no. 2, pp. 233–239, Apr. 2021, Computer Society of the Republic of China, ISSN: 1991-1599. DOI: 10.3966/199115992021043202020.

[18] A. M. Sheikh, M. R. Islam, M. H. Habaebi, S. A. Zabidi, A. R. Bin Najeeb, and A. Kabbani, "A survey on edge computing (ec) security challenges: Classification, threats, and mitigation strategies", *Future Internet*, vol. 17, no. 4, 2025, ISSN: 1999-5903. DOI: 10.3390/fi17040175.

[19] M. Merenda, C. Porcaro, and D. Iero, "Edge machine learning for ai-enabled iot devices: A review", *Sensors*, vol. 20, no. 9, 2020, ISSN: 1424-8220. DOI: 10.3390/s20092533.

[20] A. Vega, P. Bose, and A. Buyuktosunoglu, *Rugged Embedded Systems: Computing in Harsh Environments*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016, ISBN: 0128024593. DOI: https://dl.acm.org/doi/abs/10.5555/3086897.

[21] Luke Hoffer, VMware, *Overcoming Networking Challenges to Deliver Edge-Native Applications* (2024), [Online]. Available: https://blogs.vmware.com/sase/2024/08/20/overcoming-networking-challenges-to-deliver-edge-native-applications/ (visited on 04/04/2025).

[22] P. Cong, J. Zhou, L. Li, K. Cao, T. Wei, and K. Li, "A survey of hierarchical energy optimization for mobile edge computing: A perspective from end devices to the cloud", *Association for Computing Machinery, ACM Computing Surveys (CSUR), New York, NY, USA*, vol. 53, no. 2, Apr. 2020, ISSN: 0360-0300. DOI: 10.1145/3378935.

[23] B. M. John Zao Chuck Byers, "The industrial internet of things distributed computing in the edge", *Industrial Internet Consortium*, p. 36, 2020, Industrial Internet Consortium. [Online]. Available: https://www.iiconsortium.org/pdf/IIoT-Distributed-Computing-in-the-Edge.pdf.

[24] U. Iqbal, T. Davies, and P. Perez, "A review of recent hardware and software advances in gpu-accelerated edge-computing single-board computers (sbcs) for computer vision", *Sensors*, vol. 24, no. 15, 2024, ISSN: 1424-8220. DOI: 10.3390/s24154830.

[25] S. E. Mathe, H. K. Kondaveeti, S. Vappangi, S. D. Vanambathina, and N. K. Kumaravelu, "A comprehensive review on applications of raspberry pi", *Computer Science Review*, vol. 52, p. 100 636, 2024, ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2024.100636.

[26] A. Hassan, H. Nahar, W. M. Shah, and A. Abd-Aziz, "Performance evaluation of raspberry pi as an iot edge signal processing device for a real-time flash flood forecasting system", *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 10, 2022. DOI: 10.14569/IJACSA.2022.01310100.

[27] A. Karakaya and A. Ulu, "A survey on post-quantum based approaches for edge computing security", *WIREs Computational Statistics*, vol. 16, no. 1, e1644, 2024. DOI: 10.1002/wics.1644.

[28] S. A. Käppler and B. Schneider, "Post-quantum cryptography: An introductory overview and implementation challenges of quantum-resistant algorithms", in *Proceedings of the Society 5.0 Conference 2022 - Integrating Digital World and Real World to Resolve Challenges in Business and Society, Brugg-Windisch, Switzerland*, K. Hinkelmann and A. Gerber, Eds., ser. EPiC Series in Computing, vol. 84, EasyChair, 2022, pp. 61–71. DOI: 10.29007/2tpw.

[29] Kimmo Järvinen, *The future of public key cryptography will be post-quantum cryptography* (2022), [Online]. Available: https://xiphera.com/the-future-of-public-key-cryptography-will-be-post-quantum-cryptography/ (visited on 05/28/2025).

[30] Ganesh Gopalan, *Post-Quantum Cryptography: The Future of Secure Communications and the Role of Standards* (2024), [Online]. Available: https://www.appviewx.com/blogs/post-quantum-cryptography-the-future-of-secure-communications-and-the-role-of-standards/ (visited on 04/10/2025).

[31] Chad Boutin, National Institute of Standards and Technology, *NIST Releases First 3 Finalized Post-Quantum Encryption Standards, Cryptographic Technology Group, U.S. Department of Commerce* (2024), [Online]. Available: https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards (visited on 04/10/2025).

[32] Q. D. Truong and H. Lee, "Efficient polynomial arithmetic and hash modules for ml-dsa and ml-kem standards", in *2024 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), Taipei, Taiwan*, 2024, pp. 776–780. DOI: 10.1109/APCCAS62602.2024.10808736.

[33] D. Cooper, *Stateless Hash-Based Digital Signature Standard* (2024), Federal Inf. Process. Stds., NIST FIPS, Gaithersburg, Maryland, USA, DOI: 10.6028/NIST.FIPS.205.

[34] GQI, *NIST Has Finalized the First Three PQC Algorithms; 45 More Are Still in the Pipeline* (2024), [Online]. Available: https://quantumcomputingreport.

com/nist-has-finalized-the-first-three-pqc-algorithms-45-more-are-still-in-the-pipeline/ (visited on 04/19/2025).

[35] F. Opiłka, M. Niemiec, M. Gagliardi, and M. A. Kourtis, "Performance analysis of post-quantum cryptography algorithms for digital signature", *Applied Sciences*, vol. 14, no. 12, 2024, ISSN: 2076-3417. DOI: 10.3390/app14124994.

[36] M. R. Nosouhi, S. W. A. Shah, L. Pan, and R. Doss, "Bit flipping key encapsulation for the post-quantum era", *IEEE Access*, vol. 11, pp. 56 181–56 195, 2023. DOI: 10.1109/ACCESS.2023.3282928.

[37] M.-S. Chen and T. Chou, "Classic mceliece on the arm cortex-m4", *International Association for Cryptologic Research (IACR), Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 3, pp. 125–148, Jul. 2021. DOI: 10.46586/tches.v2021.i3.125-148.

[38] N. Sood, "Cryptography in post quantum computing era", *The Social Science Research Network (SSRN)*, Jan. 2024, Independent Researcher. DOI: 10.2139/ssrn.4705470.

[39] K. de Boer and W. van Woerden, *Lattice-based Cryptography: A survey on the security of the lattice-based NIST finalists* (2025), International Association for Cryptologic Research (IACR), Cryptology ePrint Archive, Paper 2025/304, [Online]. Available: https://eprint.iacr.org/2025/304 (visited on 04/15/2025).

[40] X. Wang, G. Xu, and Y. Yu, "Lattice-based cryptography: A survey", *Chinese Annals of Mathematics, Series B*, vol. 44, pp. 945–960, Nov. 2023. DOI: 10.1007/s11401-023-0053-6.

[41] T. Güneysu, V. Lyubashevsky, and T. Pöppelmann, "Practical lattice-based cryptography: A signature scheme for embedded systems", in *Cryptographic Hardware and Embedded Systems – CHES 2012*, E. Prouff and P. Schaumont, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 530–547, ISBN: 978-3-642-33027-8. DOI: 10.1007/978-3-642-33027-8_31.

[42] E. Persichetti, "Improving the efficiency of code-based cryptography", Ph.D. dissertation, The University of Auckland, Jan. 2013. DOI: 10.13140/2.1.1957.8881.

[43] R. Overbeck and N. Sendrier, "Code-based cryptography", in *Post-Quantum Cryptography*, D. J. Bernstein, J. Buchmann, and E. Dahmen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 95–145, ISBN: 978-3-540-88702-7. DOI: 10.1007/978-3-540-88702-7_4.

[44] V. Srivastava, A. Baksi, and S. K. Debnath, "An overview of hash based signatures", *International Association for Cryptologic Research (IACR), Cryptology ePrint Archive*, vol. 2023, p. 411, 2023. [Online]. Available: https://eprint.iacr.org/2023/411 (visited on 04/15/2025).

[45] E. Fathalla and M. Azab, "Beyond classical cryptography: A systematic review of post-quantum hash-based signature schemes, security, and optimizations", *IEEE Access*, vol. 12, pp. 175 969–175 987, 2024. DOI: 10.1109/ACCESS.2024.3485602.

[46] L.-C. Wang, T.-j. Wei, J.-M. Shih, Y.-H. Hu, and C.-C. Hsieh, "An algorithm for solving over-determined multivariate quadratic systems over finite fields", *Advances in Mathematics of Communications, American Institute of Mathematical Sciences*, vol. 18, no. 1, pp. 55–90, 2024, ISSN: 1930-5346. DOI: 10.3934/amc.2022001.

[47] L. Ran and M. Trimoska, "Shifting our knowledge of mq-sign security", in *Post-Quantum Cryptography*, R. Niederhagen and M.-J. O. Saarinen, Eds., Cham: Springer Nature Switzerland, 2025, pp. 232–252, ISBN: 978-3-031-86599-2. DOI: 10.1007/978-3-031-86599-2_8.

[48] ANSSI, *ANSSI views on the Post-Quantum Cryptography transition (2023 follow up)* (2023), [Online]. Available: https://cyber.gouv.fr/sites/default/files/document/follow_up_position_paper_on_post_quantum_cryptography.pdf.

[49] Marin Ivezic, *Introduction to Crypto-Agility* (2022), [Online]. Available: https://postquantum.com/post-quantum/introduction-crypto-agili (visited on 04/08/2025).

[50] N. Alnahawi, N. Schmitt, A. Wiesmaier, A. Heinemann, and T. Grasmeyer, "On the state of crypto-agility", *The International Association for Cryptologic Research (IACR) ePrint Arch.*, vol. 2023, p. 487, 2023. [Online]. Available: https://eprint.iacr.org/2023/487.pdf.

[51] Tim Callan, Sectigo, *What is crypto-agility and how can organizations achieve it?* (2023), [Online]. Available: https://www.sectigo.com/resource-library/what-is-crypto-agility-and-how-to-achieve-it? (visited on 04/10/2025).

[52] E. Barker, L. Chen, D. Cooper, D. Moody, A. Regenscheid, M. Souppaya, National Institute of Standards and Technology, Gaithersburg, MD, USA, *Considerations for Achieving Crypto Agility: Strategies and Practices, NIST Cybersecurity White Paper 39, Initial Public Draft, U.S. Department of Commerce* (2025), DOI: 10.6028/NIST.CSWP.39.ipd.

[53] S. P C, K. Jain, and P. Krishnan, "Analysis of post-quantum cryptography for internet of things", in *2022 6th International Conference on Intelligent Computing and Control Systems (ICICCS), Erode, India*, 2022, pp. 387–394. DOI: 10.1109/ICICCS53718.2022.9787987.

[54] G. Fitzgibbon and C. Ottaviani, "Constrained device performance benchmarking with the implementation of post-quantum cryptography", *Cryptography*, vol. 8, no. 2, 2024, ISSN: 2410-387X. DOI: 10.3390/cryptography8020021.

[55] Derek Atkins, *Requirements for Post-Quantum Cryptography on Embedded Devices in the IoT, Shelton, Connecticut, USA* (2021), [Online]. Available: https://csrc.nist.gov/CSRC/media/Events/third-pqc-standardization-conference/documents/accepted-papers/atkins-requirements-pqc-iot-pqc2021.pdf (visited on 04/13/2025).

[56] Z. Liu, K.-K. R. Choo, and J. Grossschadl, "Securing edge devices in the post-quantum internet of things using lattice-based cryptography", *IEEE Communications Magazine*, vol. 56, pp. 158–162, Feb. 2018. DOI: 10.1109/MCOM.2018.1700330.

[57] K. Hines, M. Raavi, and Villeneuve, "Post-quantum cipher power analysis in lightweight devices", in *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '22, San Antonio, TX, USA: Association for Computing Machinery, 2022, pp. 282–284, ISBN: 9781450392167. DOI: 10.1145/3507657.3529652.

[58] J. C. Patterson, W. J. Buchanan, and C. Turino, *Energy Consumption Framework and Analysis of Post-Quantum Key-Generation on Embedded Devices* (2025), Cryptology ePrint Archive, Paper 2025/909, DOI: 10.20944/preprints202505.1815.v1.

[59] Y. Zhao, R. Xie, G. Xin, and J. Han, "A high-performance domain-specific processor with matrix extension of risc-v for module-lwe applications", *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, pp. 1–14, Jul. 2022. DOI: 10.1109/TCSI.2022.3162593.

[60] L. Xu, X. Yuan, Z. Zhou, C. Wang, and C. Xu, "Towards efficient cryptographic data validation service in edge computing", *IEEE Transactions on Services Computing*, vol. 16, pp. 656–669, Sep. 2021. DOI: 10.1109/TSC.2021.3111208.

[61] N. El Kassem, "Lattice-based direct anonymous attestation", English, Ph.D. dissertation, University of Surrey, 2020. DOI: 10.15126/thesis.00855402.

[62] D. Dharminder, S. Kumari, and U. Kumar, "Post quantum secure conditional privacy preserving authentication for edge based vehicular communication", *Transactions on Emerging Telecommunications Technologies*, vol. 32, Nov. 2021. DOI: 10.1002/ett.4346.

[63] Y. Kim, J. Song, and S. Seo, "Accelerating falcon on armv8", *IEEE Access*, vol. 10, pp. 44 446–44 460, Jan. 2022. DOI: 10.1109/ACCESS.2022.3169784.

[64] Y. Kim, J. Song, T.-Y. Youn, and S. Seo, "Crystals-dilithium on armv8", *Security and Communication Networks*, vol. 2022, pp. 1–12, Feb. 2022. DOI: 10.1155/2022/5226390.

[65] S. Kumari, M. Singh, R. Singh, and H. Tewari, "Signature based merkle hash multiplication algorithm to secure the communication in iot devices", *Knowledge-Based Systems*, vol. 253, p. 109 543, Jul. 2022. DOI: 10.1016/j.knosys.2022.109543.

[66] S. Akleylek, M. Soysaldi, W. K. Lee, S. Hwang, and D. Wong, "Novel postquantum mq-based signature scheme for internet of things with parallel implementation", *IEEE Internet of Things Journal*, vol. 8, pp. 6983–6994, Apr. 2021. DOI: 10.1109/JIOT.2020.3038388.

[67] K. Li, R.-h. Shi, M. Wu, Y. Li, and X. Zhang, "A novel privacy-preserving multi-level aggregate signcryption and query scheme for smart grid via mobile fog computing", *Journal of Information Security and Applications*, vol. 67, p. 103 214, Jun. 2022. DOI: 10.1016/j.jisa.2022.103214.

[68] Z. Qingcheng, X. Yu, Y. Zhao, A. Nag, and J. Zhang, "Resource allocation in quantum-key-distribution- secured datacenter networks with cloud–edge collaboration", *IEEE Internet of Things Journal*, vol. PP, pp. 10 916–10 932, Jun. 2023. DOI: 10.1109/JIOT.2023.3242725.

[69] N. Corrias, I. Vagniluca, and Francesconi, "Implementation of italian industry 4.0 quantum testbed in turin", *IET Quantum Communication*, vol. 5, Sep. 2023. DOI: 10.1049/qtc2.12074.

[70] N. N. Minhas and K. Mansoor, "Edge-computing-based scheme for post-quantum iot security for e-health", *IEEE Internet of Things Journal*, vol. 11, no. 19, pp. 31 331–31 337, 2024. DOI: 10.1109/JIOT.2024.3418959.

[71] A. Ulu and A. Karakaya, "A survey on post-quantum based approaches for edge computing security", *WIREs Computational Statistics*, vol. 16, p. 37, Feb. 2024. DOI: 10.1002/wics.1644.

[72] F. Vasquez and C. Simmonds, *Mastering Embedded Linux Programming: Create fast and reliable embedded solutions with Linux 5.4 and the Yocto Project*

3.1 (Dunfell). Packt Publishing, 2021, ISBN: 9781789535112. [Online]. Available: https://books.google.fr/books?id=K4ApEAAAQBAJ (visited on 05/04/2025).

[73] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors", *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 275–287, Mar. 2007, Association for Computing Machinery, New York, NY, USA, ISSN: 0163-5980. DOI: 10.1145/1272998.1273025.

[74] Z. Akhtar, "Operating systems (os): An insight investigative research analysis and future directions", *Journal of Technology and Informatics (JoTI)*, vol. 6, pp. 58–69, Oct. 2024, Universitas Dinamika. DOI: 10.37802/joti.v6i1.637.

[75] N. Gollenstede, U. Kulau, and C. Dietrich, "Reupnix: Reconfigurable and updateable embedded systems", in *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2023, Orlando, FL, USA: Association for Computing Machinery, 2023, pp. 40–51, ISBN: 9798400701740. DOI: 10.1145/3589610.3596273.

[76] Emily Fox, Simo Sorce, *Red Hat's path to post-quantum cryptography* (2024), [Online]. Available: https://www.redhat.com/en/blog/red-hats-path-post-quantum-cryptography (visited on 04/14/2025).

[77] Atom Ramirez, Marie Ramirez, *An Introduction to Post-Quantum Cryptography for Linux System Administrators, 2025 Southern California Linux Expo, Pasadena Convention Center, Pasadena, CA, USA* (2025), [Online]. Available: SCaLE22x_AnIntroToPost-QuantumCryptographyforLinuxSAs.pdf.

[78] T. Reddy.K, T. Hollebeek, J. Gray, and S. Fluhrer, "Use of SLH-DSA in TLS 1.3", Internet Engineering Task Force, Internet-Draft draft-reddy-tls-slhdsa-01, Apr. 2025, 9 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-reddy-tls-slhdsa/01/ (visited on 04/27/2025).

[79] P. Kampanakis, D. Stebila, and T. Hansen, "PQ/T Hybrid Key Exchange in SSH", Internet Engineering Task Force, Internet-Draft draft-ietf-sshm-mlkem-hybrid-kex-

00, Jan. 2025, 14 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-sshm-mlkem-hybrid-kex/00/ (visited on 03/01/2025).

[80] P. Sinha, *Distributed Operating Systems: Concepts and Design*. PHI Learning, 1998, ISBN: 9788120313804. [Online]. Available: https://books.google.fr/books?id=SewHKWac2I4C (visited on 05/01/2025).

[81] R. Harishankar, M. Osborne, J. S. Arun, J. Buselli, and J. Janechek, *Crypto-agility and quantum-safe readiness* (2024), IBM, [Online]. Available: https://www.ibm.com/quantum/blog/crypto-agility? (visited on 05/14/2025).

[82] R. Carbone, *Long-term Operating System Maintenance: A Linux Case Study* (Technical memorandum). Defence R&D Canada - Valcartier, 2013. [Online]. Available: https://books.google.fr/books?id=tLOX0AEACAAJ (visited on 04/24/2025).

[83] M. Boras, J. Balen, and K. Vdovjak, "Performance evaluation of linux operating systems", in *2020 International Conference on Smart Systems and Technologies (SST)*, Osijek, Croatia, 2020, pp. 115–120. DOI: 10.1109/SST49455.2020.9264055.

[84] K. Hitchcock, "Package installation", in *The Enterprise Linux Administrator: Journey to a New Linux Career*. Berkeley, CA: Apress, 2023, pp. 193–236, ISBN: 978-1-4842-8801-6. DOI: 10.1007/978-1-4842-8801-6_7.

[85] F. J. Meng, M. N. Wegman, J. M. Xu, X. Zhang, P. Chen, and G. Chafle, "It troubleshooting with drift analysis in the devops era", *IBM Journal of Research and Development*, vol. 61, no. 1, 6:62–6:73, 2017. DOI: 10.1147/JRD.2016.2630478.

[86] M. Bano and D. Zowghi, "A systematic review on the relationship between user involvement and system success", *Information and Software Technology*, vol. 58, pp. 148–169, 2015, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2014.06.011.

[87] S. Kwon, S. Lee, T. Kim, D. Ryu, and J. Baik, "Exploring llm-based automated repairing of ansible script in edge-cloud infrastructures", *Journal of Web Engineering*, vol. 22, no. 6, pp. 889–912, 2023. DOI: 10.13052/jwe1540-9589.2263.

[88] Ettore Di Giacinto, *Understanding Immutable Linux OS: Benefits, Architecture, and Challenges* (2023), [Online]. Available: https://kairos.io/blog/2023/03/22/understanding-immutable-linux-os-benefits-architecture-and-challenges/ (visited on 04/04/2025).

[89] S. Böhm and G. Wirtz, "Immutable operating systems: A survey", in *Proceedings of the 15th Central European Workshop on Services and their Composition (ZEUS 2023)*, vol. 3386, Hannover, Germany: CEUR-WS, May 2023, p. 10. [Online]. Available: https://ceur-ws.org/Vol-3386/paper9.pdf (visited on 02/20/2025).

[90] Axel Quack, *The Rise of Immutable Linux Distributions: Part 1 – Vanilla OS* (2024), [Online]. Available: https://www.capturetheflag.today/the-rise-of-immutable-linux-distributions/ (visited on 04/04/2025).

[91] Malix-Labs, *Awesome Atomic - An awesome curated knowledge-base about atomic systems* (2025), [Online]. Available: https://github.com/Malix-Labs/Awesome-Atomic (visited on 04/14/2025).

[92] S. S. Bisht, *Understand Btrfs File System (Copy On Write, Sub-Volumes, Snapshots, Quota Group)* (2023), [Online]. Available: https://surajsinghbisht054.medium.com/understand-btrfs-file-system-copy-on-write-sub-volumes-snapshots-quota-group-part-1-c6305f87df9b (visited on 04/05/2025).

[93] SUSE, *Administering SUSE Linux Micro Using transactional-update* (2025), [Online]. Available: https://documentation.suse.com/smart/systems-management/html/Micro-transactional-updates/index.html (visited on 04/05/2025).

[94] OSTree documentation, *libostree* (2024), [Online]. Available: https://github.com/ostreedev/ostree (visited on 04/05/2025).

[95] CentOS documentation, *OSTree* (2025), [Online]. Available: https://docs.centos.org/automotive-sig-documentation/features-and-concepts/con_ostree/ (visited on 04/06/2025).

[96] CentOS documentation, *Creating and maintaining OS images with OSTree* (2025), [Online]. Available: https://sigs.centos.org/automotive/building/updating_ostree/#creating-an-ostree-based-image (visited on 04/05/2025).

[97] CentOS documentation, *Creating static deltas* (2025), [Online]. Available: https://sigs.centos.org/automotive/building/creating_static_deltas/ (visited on 04/05/2025).

[98] ByteWaveNetwork, *Understanding OSTree: A Modern Approach to Version Control for Operating Systems (Part1)* (2024), [Online]. Available: https://medium.com/@ByteWaveNetwork/understanding-ostree-a-modern-approach-to-version-control-for-operating-systems-642ad5f2aeb5 (visited on 04/24/2025).

[99] David Byrne, *Updating Linux using A/B Partitions* (2018), [Online]. Available: https://blog.davidbyrne.dev/2018/08/16/linux-ab-partitions (visited on 04/06/2025).

[100] Android documentation, *Implement dm-verity* (2025), [Online]. Available: https://source.android.com/docs/security/features/verifiedboot/dm-verity (visited on 04/06/2025).

[101] B. Bzeznik, O. Henriot, V. Reis, O. Richard, and L. Tavard, "Nix as hpc package management system", in *Proceedings of the Fourth International Workshop on HPC User Support Tools*, ser. HUST'17, Denver, CO, USA: Association for Computing Machinery, 2017, ISBN: 9781450351300. DOI: 10.1145/3152493.3152556.

[102] Q. Guilloteau, J. Bleuzen, M. Poquet, and O. Richard, "Painless transposition of reproducible distributed environments with nixos compose", in *2022 IEEE International Conference on Cluster Computing (CLUSTER), Heidelberg, Germany*, 2022, pp. 1–12. DOI: 10.1109/CLUSTER51413.2022.00051.

[103] E. Dolstra, A. Löh, and N. Pierron, "Nixos: A purely functional linux distribution", *Journal of Functional Programming*, vol. 20, no. 5–6, pp. 577–615, 2010. DOI: 10.1017/S0956796810000195.

[104] N. Singh, A. S. Baghel, and P. Mishara, "Modularity and lazy evaluation in nixos: A functional approach to operating system configuration", in *2025 International Conference on Cognitive Computing in Engineering, Communications, Sciences and Biomedical Health Informatics (IC3ECSBHI), Delhi, India*, 2025, pp. 415–419. DOI: 10.1109/IC3ECSBHI63591.2025.10991248.

[105] R. Goswami, R. S., A. Goswami, S. Goswami, and D. Goswami, "Reproducible high performance computing without redundancy with nix", in *2022 Seventh International Conference on Parallel, Distributed and Grid Computing (PDGC), Waknaghat, India*, 2022, pp. 238–242. DOI: 10.1109/PDGC56933.2022.10053342.

[106] Tuomo Perä, "Comparison of custom embedded Linux build systems: Yocto and Buildroot", *Master's Thesis, Aalto University, Finland*, 2022. [Online]. Available: https://aaltodoc.aalto.fi/items/5707a0af-8b04-4447-9b85-391f6b147cd2.

[107] Igal Zeifman, Sternum, *What Is the Yocto Project?* (2023), [Online]. Available: https://sternumiot.com/iot-blog/yocto-project-components-use-cases-and-a-quick-tutorial/ (visited on 04/04/2025).

[108] Kristian Amlie, *Yocto and OTA software updates in an IoT project* (2021), [Online]. Available: https://mender.io/blog/yocto-and-ota-software-updates (visited on 04/04/2025).

[109] Yocto Project Documentation, *System Requirements* (2025), [Online]. Available: https://docs.yoctoproject.org/ref-manual/system-requirements.html#supported-linux-distributions (visited on 04/17/2025).

[110] Yocto Project Documentation, *Distribution Support* (2018), [Online]. Available: https://wiki.yoctoproject.org/wiki/Distribution_Support (visited on 04/17/2025).

[111] Yocto Project Documentation, *Poky Build Tool and Metadata (poky.conf file)* (2025), [Online]. Available: https://git.yoctoproject.org/poky/tree/meta-poky/conf/distro/poky.conf (visited on 04/17/2025).

[112] Yocto Project Documentation, *System Update* (2024), [Online]. Available: https://wiki.yoctoproject.org/wiki/System_Update#babic's_SWUpdate (visited on 04/08/2025).

[113] Angelo Compagnucci, "Upgrading buildroot based devices with swupdate", *Amarula Solutions BV, Linux Lab, Florence, Italy*, 2019. [Online]. Available: https://www.slideshare.net/slideshow/angelo-compagnucci-upgrading-buildroot-based-devices-with-swupdate/125724101 (visited on 04/17/2025).

[114] Charles Steinkuehler, *Buildroot + RAUC* (2024), [Online]. Available: https://github.com/cdsteinkuehler/br2rauc (visited on 04/17/2025).

[115] H. Kim, H. Jung, A. Satriawan, and H. Lee, "A configurable ml-kem/kyber key-encapsulation hardware accelerator architecture", *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 71, no. 11, pp. 4678–4682, 2024. DOI: 10.1109/TCSII.2024.3442228.

[116] H. Kosuge and K. Xagawa, *The Security of ML-DSA against Fault-Injection Attacks* (2025), International Association for Cryptologic Research (IACR), Cryptology ePrint Archive, Paper 2025/904, [Online]. Available: https://eprint.iacr.org/2025/904 (visited on 05/01/2025).

[117] R. Varma, C. Melville, C. Pinello, and T. Sahai, "Post quantum secure command and control of mobile agents inserting quantum-resistant encryption schemes in the secure robot operating system", *International Journal of Semantic Computing*, vol. 15, no. 03, pp. 359–379, 2021. DOI: 10.1142/S1793351X21400092.

[118] openSUSE, *System Recovery and Snapshot Management with Snapper* (2025), [Online]. Available: https://doc.opensuse.org/documentation/leap/archive/15.0/reference/html/book.opensuse.reference/cha.snapper.html (visited on 05/19/2025).

[119] Phoronix Test Suite, *Phoronix Test Suite 10.8.5* (2025), [Online]. Available: https://github.com/phoronix-test-suite/phoronix-test-suite (visited on 05/16/2025).

[120] C. Liambas and A. Manios, "Performance comparison analysis of digital forensic tools in various operating systems", in *2024 12th International Symposium on Digital Forensics and Security (ISDFS)*, San Antonio, Texas, USA, 2024, pp. 1–5. DOI: 10.1109/ISDFS60797.2024.10527289.

[121] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: The correct way to summarize benchmark results", *Communications of the Association for Computing Machinery*, vol. 29, no. 3, pp. 218–221, Mar. 1986, New York, NY, USA, ISSN: 0001-0782. DOI: 10.1145/5666.5673.

[122] StephanStS, *Comparison to other Debian based distributions – Why should you use DietPi?* (2021), [Online]. Available: https://dietpi.com/blog/?p=888 (visited on 05/18/2025).

[123] openSUSE, *File aarch64-common-pagesize.patch of Package binutils.36797* (2025), [Online]. Available: https://build.opensuse.org/projects/SUSE%3ASLE-15-SP3%3AUpdate/packages/binutils.36797/files/aarch64-common-pagesize.patch?expand=0 (visited on 05/21/2025).

[124] Red Hat, *Chapter 2. The 64k page size kernel* (2025), [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/managing_monitoring_and_updating_the_kernel/what-is-kernel-64k_managing-monitoring-and-updating-the-kernel (visited on 05/21/2025).

[125] Guillaume Gardet, *New ARM MicroOS snapshot 20250503 released!* (2025), [Online]. Available: https://lists.opensuse.org/archives/list/microos@lists.opensuse.org/thread/7B2USPTJRDNYT56E5RUZUW3WE75C36WJ/ (visited on 05/21/2025).

[126] GitHub - Open Source, *kernel-x86_64 – fedora.config* (2025), [Online]. Available: https://raw.githubusercontent.com/projg2/fedora-kernel-config-for-gentoo/6.6.12-gentoo/kernel-x86_64-fedora.config (visited on 05/21/2025).

[127] Aleksandar Brezar, Clea Skopeliti, *Spain, Portugal and parts of France hit by massive power outage* (2025), [Online]. Available: `https://www.euronews.com/my-europe/2025/04/28/spain-portugal-and-parts-of-france-hit-by-massive-power-outage` (visited on 05/15/2025).

[128] Alanna Titterington, *The first post-quantum encryption standards* (2024), [Online]. Available: `https://www.kaspersky.com/blog/post-quantum-cryptography-standards/52066/` (visited on 05/18/2025).

[129] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler and Damien Stehlé, *CRYSTALS-Dilithium - Algorithm Specifications and Supporting Documentation (Version 3.1)* (2021), [Online]. Available: `https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf` (visited on 05/18/2025).

[130] Open Quantum Safe, *C library for prototyping and experimenting with quantum-resistant cryptography* (2025), [Online]. Available: `https://github.com/open-quantum-safe/liboqs` (visited on 05/17/2025).

Appendix A ML-KEM-768 & ML-KEM-1024 Performance Results

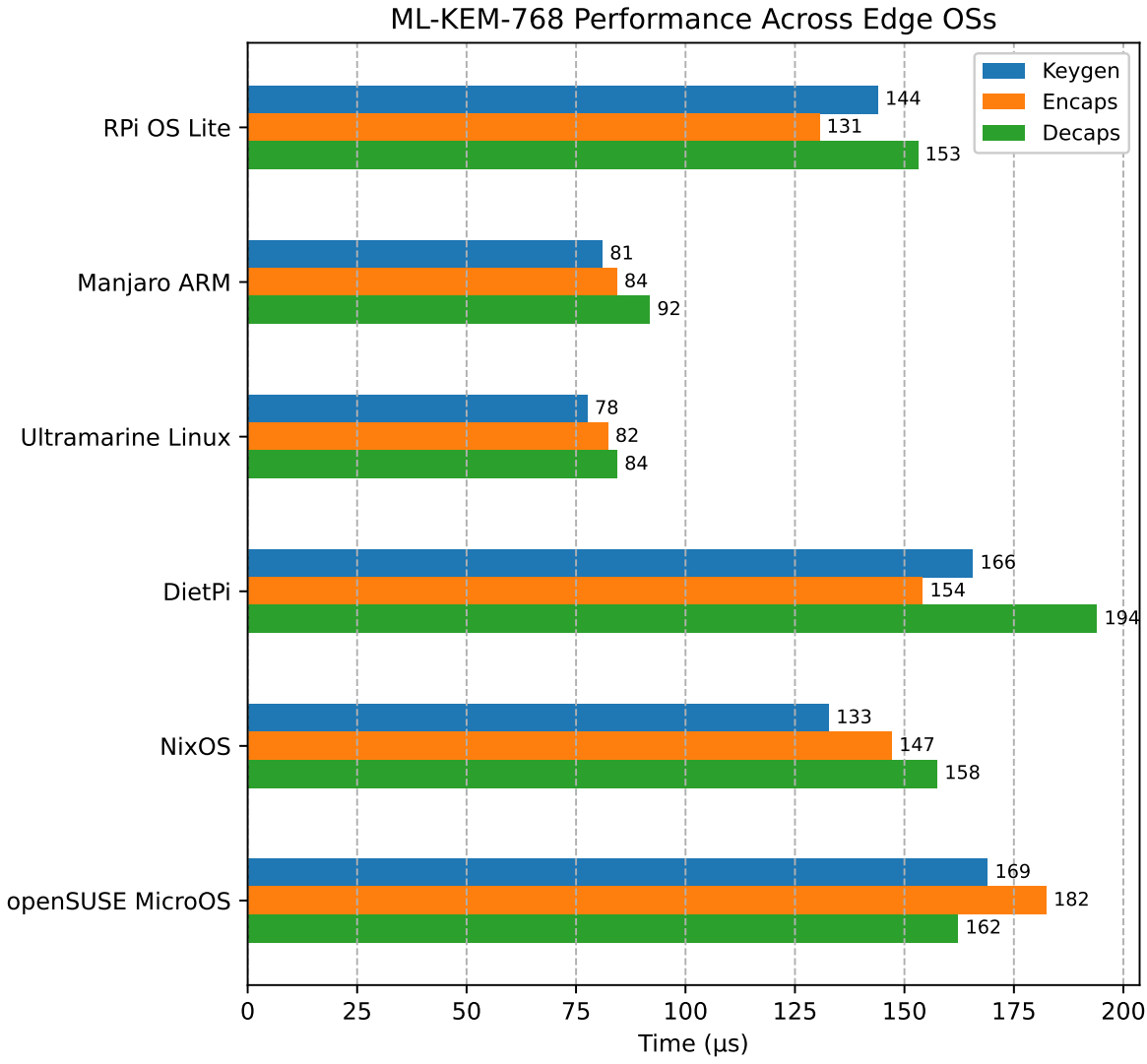

Figure A.1: Comparing ML-KEM-768 Performance Across Edge OSs Under Identical Workloads

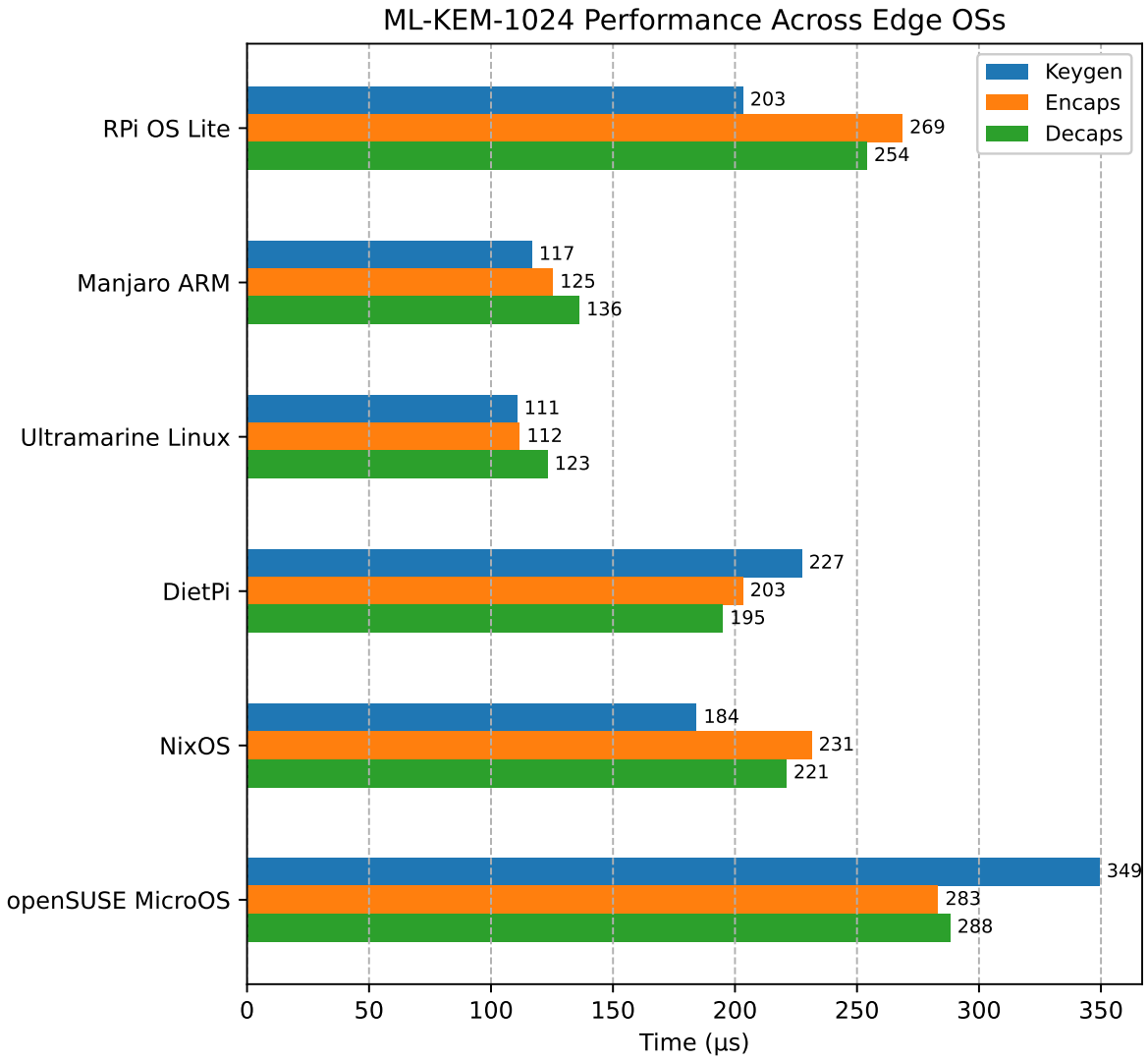

Figure A.2: Comparing ML-KEM-1024 Performance Across Edge OSs Under Identical Workloads

Appendix B ML-DSA-65 & ML-DSA-87 Performance Results

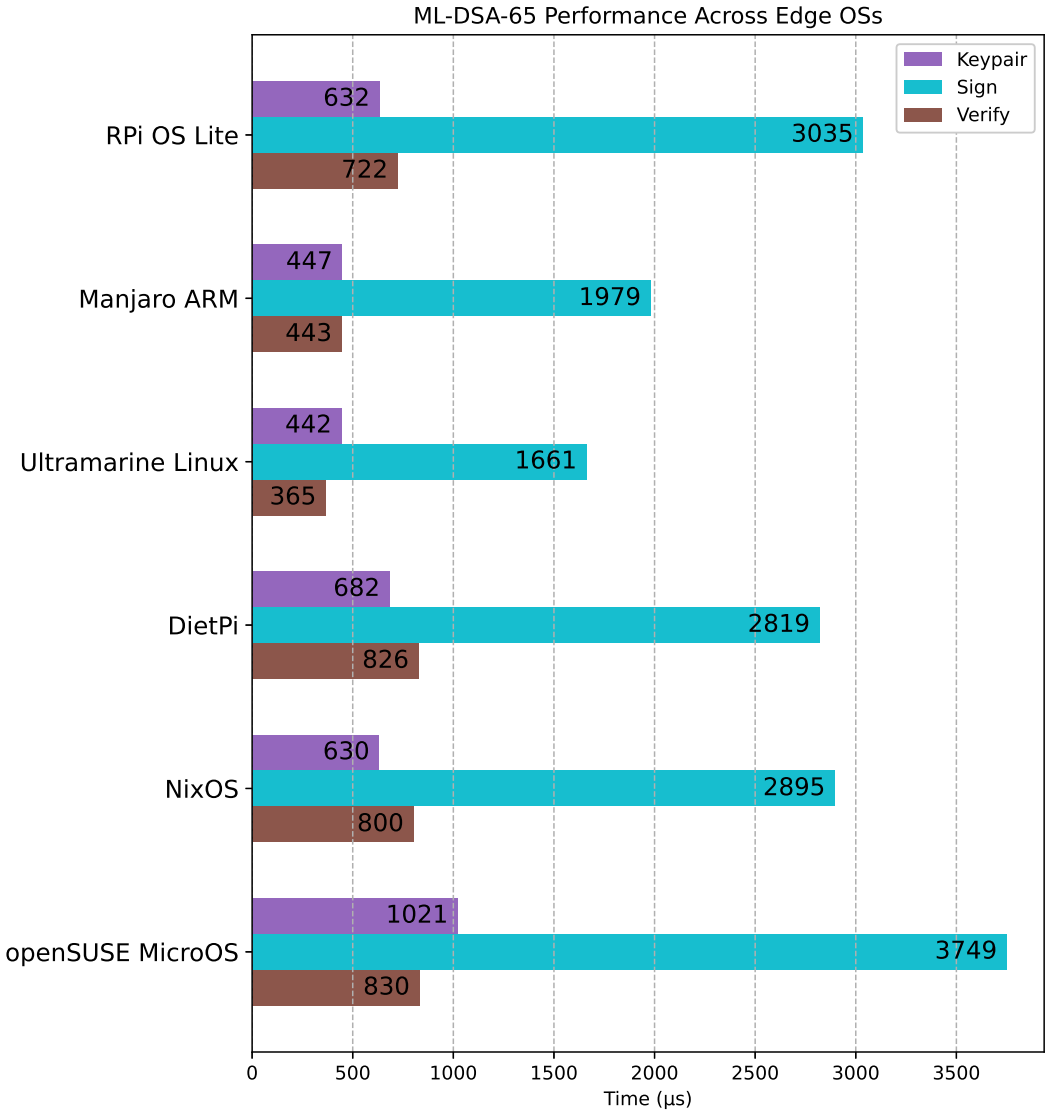

Figure B.1: Comparing ML-DSA-65 Performance Across Edge OSs Under Identical Workloads

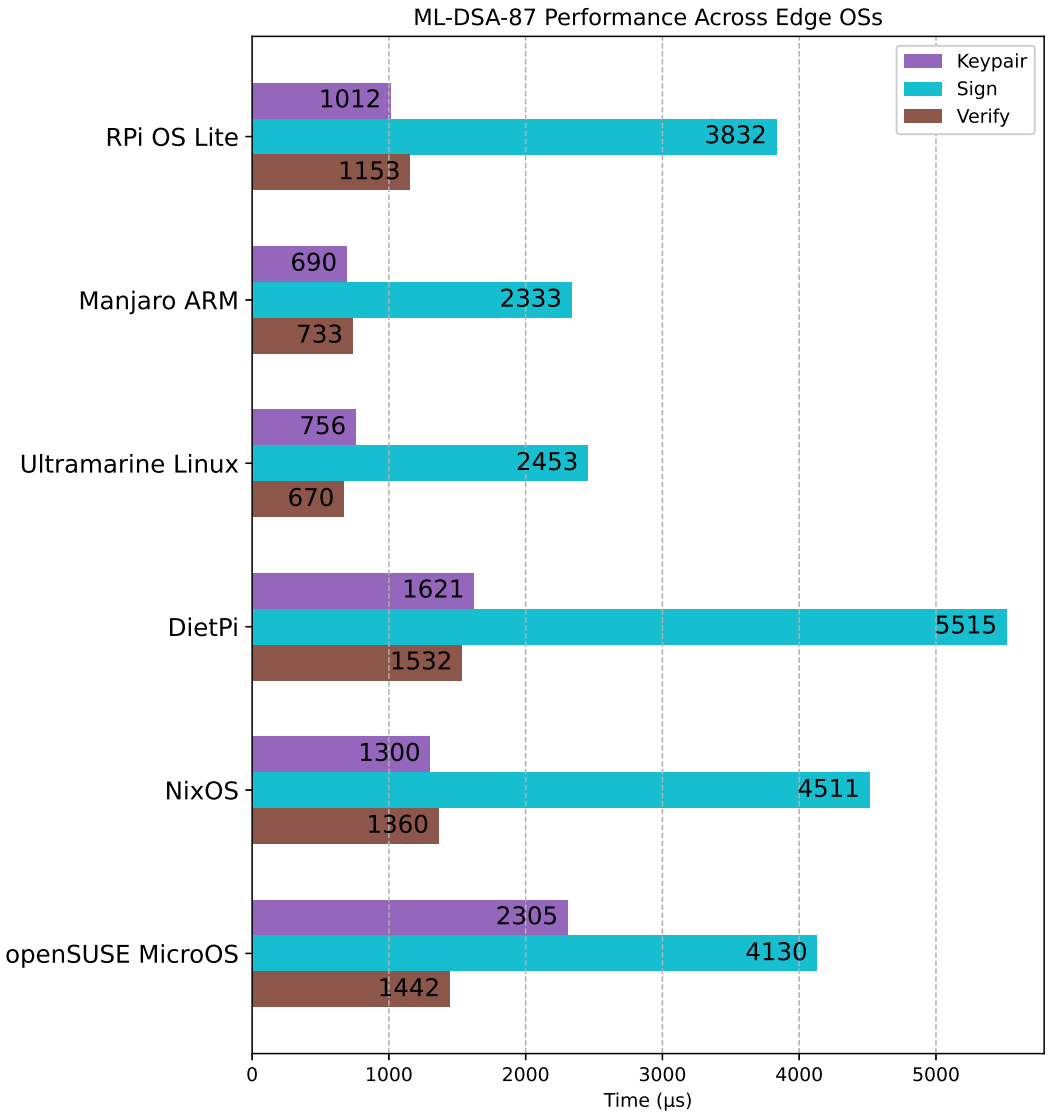

Figure B.2: Comparing ML-DSA-87 Performance Across Edge OSs Under Identical Workloads

Use of AI in this Master's Thesis

NB: I used AI (GPT-4o LLM) to help refine and improve the flow of some of the longer sentences in this report. The prompts I used were focused on improving clarity, and rephrasing technical content for better readability. For example, I provided sentences such as "Refine the sentence to enhance clarity and flow: [MY TEXT]".