Isolating Mobile Operating System Performance Through Hardware-Controlled Benchmarking

Naser AlDuaij

***Abstract*—Existing mobile benchmarking tools conflate hardware capabilities with software efficiency, complicating fair architectural comparisons. This paper introduces MobiOSBench, a standardized and flexible evaluation framework designed to isolate mobile operating system performance through hardware-controlled benchmarking. By executing workloads natively and holding physical hardware constant, MobiOSBench controls for silicon disparities to measure true software overhead. The framework evaluates systems across four contextual pillars: Privacy, Resources, Performance, and Efficiency, using configurable weighted user profiles to translate raw throughput, memory consumption overhead, and ambient telemetry into real-world performance indices. MobiOSBench is validated through extensive evaluations of four diverse mobile operating systems deployed systematically across three distinct physical devices. Results confirm the framework reliably exposes intrinsic architectural trade-offs, distinguishing systems optimized for power management from those prioritizing privacy or raw computational throughput. Extended case studies on iOS and a Raspberry Pi edge device further demonstrate its cross-architecture adaptability. A sensitivity analysis additionally confirms that these rankings remain largely stable under varying weight assignments, and a systematic comparison against existing benchmarking tools establishes the framework's methodological novelty. Ultimately, MobiOSBench provides a mathematically rigorous methodology for transparent, context-aware mobile systems evaluation.**

***Index Terms*—Operating Systems, Mobile Systems, Android, iOS, Linux, Internet of Things, Edge Computing, Performance Evaluation, Benchmarking**

I. Introduction

THE rapid evolution of mobile operating systems (OS) has significantly reshaped the technological landscape, driving innovation in device capabilities, application ecosystems, and user experience. From the dominance of Android and iOS to emerging Linux-based mobile architectures such as Ubuntu Touch and Sailfish OS, the diversity in mobile platforms reflects the unique architectural philosophies of their developers. This architectural divergence is increasingly evident among major original equipment manufacturers. For instance, recent industry shifts, such as Motorola partnering with the privacy-centric GrapheneOS Foundation, demonstrate a growing commercial demand for highly specialized alternative software stacks [1]. Consequently, each operating system presents distinct advantages in performance, security, or usability tailored to specific hardware configurations. However, this diversity also introduces a critical challenge: how to fairly and accurately compare these varied architectures while ensuring that the underlying hardware remains constant, thereby isolating the intrinsic efficiency of the software stack from physical silicon disparities.

Desktop and server benchmarking benefits from straightforward comparisons through dual-booting setups that natively run Windows or Linux on the same hardware. This approach ensures that performance differences are solely attributed to the operating systems being tested. In the mobile domain, however, such an approach is rarely feasible due to hardware dependencies and software compatibility issues. Existing mobile operating system benchmarking tools primarily evaluate mobile devices as holistic systems, encompassing hardware and software interactions. These tools, while valuable for consumer decision-making, often conflate hardware capabilities with operating system performance, obscuring the true performance of the operating system itself. For instance, comparing an iPhone to an Android device often highlights hardware-specific optimizations rather than isolating the impact of software and the operating system. This conflation presents significant limitations for operating system developers and researchers seeking to evaluate the standalone performance of operating systems.

A key example of this discrepancy is shown in Cider [2]. Cider is a system built to natively execute iOS applications on Android, alongside Android applications, on the same hardware. Cider evaluated the performance difference by running benchmarking tools. The Android version and the iOS version of the same benchmarking tools showed significant differences due to the way the Android and iOS operating systems ran these applications. This highlighted a clear performance difference attributable solely to operating system and software differences, despite using identical hardware.

Mobile operating systems vary significantly in their design, spanning a wide spectrum of architectural paradigms. On one end, modern mobile operating systems such as Android and iOS embrace highly optimized and vertically integrated architectures tailored for specific hardware ecosystems [2, 3]. On the other end, Linux-based mobile operating systems often retain modular, open-source architectures that prioritize flexibility and user customization [4]. These architectural differences, whether in kernel design, memory management, or

Naser AlDuaij is with the Department of Computer Science, Kuwait University, Kuwait. E-mail: naser.alduaij@ku.edu.kw. ORCID: 0000-0002-0028-5902

Manuscript received 22 April 2026; revised 14 July 2026; accepted 3 September 2026.

This is the author's accepted version of the article. The final version of record is published in *IEEE Internet of Things Journal*, vol. 13, no. 22, pp. 54908–54928, Nov. 2026, doi: 10.1109/JIOT.2026.3731540.

© 2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

application sandboxing, further complicate direct performance comparisons across platforms.

The diversity in mobile operating system architecture also mirrors the broader debates in computing between specialized, tightly integrated systems and general-purpose, modular designs. Android and iOS, for example, have been developed to maximize performance and power efficiency on specific hardware, often relying on proprietary solutions. In contrast, Linux-based mobile operating systems draw from the ethos of openness and adaptability, enabling broader hardware support but sometimes at the cost of optimization for any single device. This dichotomy underscores the importance of a benchmarking framework that accounts for such trade-offs while ensuring fair comparisons on identical hardware.

The need for a fair benchmarking framework for mobile operating systems, one that isolates operating system performance while keeping hardware constant, has never been greater. Such a framework would provide consistent metrics to evaluate operating system performance, enabling developers to identify bottlenecks, optimize resource allocation, and enhance user experience. Additionally, it would empower researchers to investigate fundamental questions about the trade-offs between architectural philosophies, such as modern mobile architectures versus traditional Linux-based designs. Furthermore, such a framework would be invaluable for end-users seeking objective information to guide their choice of devices and software ecosystems.

This paper addresses this critical gap by introducing an evaluation framework, MobiOSBench, tailored specifically for mobile operating systems. Unlike existing tools that merely measure raw silicon limits, MobiOSBench provides a standardized methodology to evaluate the full native software stack. By establishing uniform testing parameters, the framework effectively isolates operating system performance, proving particularly advantageous when hardware variables can be held constant. The proposed framework moves beyond basic computational tests by simultaneously tracking systemic costs, evaluating the operating system across four distinct pillars: Privacy, Resources, Performance, and Efficiency. By standardizing the benchmarking process across these contextual pillars, MobiOSBench enables meaningful comparisons across a wide range of operating systems.

To validate the utility of MobiOSBench, the author conducted comprehensive evaluations of four distinct mobile operating systems systematically deployed across three different physical devices. The results demonstrate that MobiOSBench consistently identifies relative performance differences by applying mathematically weighted and dynamically configurable user profiles to the aggregated telemetry. For example, an operating system that excels in raw compute may be heavily penalized in a privacy-conscious user profile if it exhibits excessive ambient network telemetry. This highly contextual approach underscores the robustness of the proposed framework and its potential to serve as a standardized tool for the mobile computing community, especially when evaluating open-source mobile operating systems.

Furthermore, the MobiOSBench framework demonstrates architectural agnosticism through targeted case studies, proving its capacity to dynamically adapt to heavily sandboxed proprietary ecosystems and resource-constrained edge devices without compromising the underlying mathematical methodology.

Beyond its immediate contributions, the introduction of MobiOSBench paves the way for future advancements in mobile operating system research. By providing a reliable means to isolate and evaluate operating system performance, this framework can facilitate deeper explorations into topics such as energy efficiency, security optimizations, and user-centric design improvements. For instance, such evaluations can reveal how transitioning older devices to more efficient operating systems might significantly reduce battery consumption and improve overall performance, offering a quantifiable pathway for extending device lifespans and mitigating electronic waste. Moreover, MobiOSBench lays a foundation for integrating benchmarking insights into the development life cycle of mobile operating systems, ensuring that performance metrics inform iterative enhancements.

By addressing the limitations of current benchmarking approaches and introducing a novel framework for fixed-hardware testing, this work aims to advance the field of mobile operating system research and foster innovation in performance evaluation methodologies.

The remainder of this paper is organized as follows: Section II reviews related work, including existing benchmarking tools and cross-platform performance studies. Section III provides background on the available mobile operating systems and their differences. Section IV details the methodology underlying the MobiOSBench framework, including its design principles and benchmarking metrics. Section V presents the evaluation results, showcasing the effectiveness of the framework in delivering fixed-hardware comparisons alongside exploratory case studies that demonstrate its dynamic adaptability across divergent architectures. Section V also provides a sensitivity analysis and a comparison against existing benchmarking frameworks and tools. Section VI highlights the key findings, future work, and threats to validity. Finally, Section VII concludes the paper.

II. Related Work

The diversity of mobile operating systems and their architectural paradigms has led to various comparative studies. However, existing literature predominantly focuses on application-level performance, hardware-software interactions, or qualitative user interface comparisons, rather than empirically isolating underlying operating system differences.

Hardware and Traditional Benchmarking: Microbenchmark suites and traditional tools like lmbench [5], SPEC [6], Sysbench [7], Phoronix Test Suite [8], Geekbench [9], AnTuTu [10], and PassMark [11] are frequently utilized for cross-platform evaluation. However, these tools are fundamentally engineered to compare physical silicon across devices. Consequently, they output singular, opaque scores or execution times in isolation. Such one-dimensional metrics fail to capture critical system-level behaviors, such as boot latency, background memory consumption, or ambient networking overhead, and

offer little translatable value regarding software efficiency or user experience.

Execution Boundaries and OS Adaptations: Other works have explored the boundaries of cross-platform execution and kernel adaptations. Cider [2], whose primary goal is enabling native execution of iOS applications on Android rather than establishing a benchmarking methodology, achieves this by compiling portions of the iOS kernel (XNU) and system daemons within the Android environment. Its evaluation nonetheless exposes significant performance disparities between the two environments on identical hardware. However, Cider's reliance on operating system modifications and interception of function calls introduces inherent overhead that would limit its applicability as a standardized operating system benchmarking methodology. Similarly, studies examining the Android Linux-based kernel [12] note that while Android reuses 99% of mainline Linux functionality, a small fraction of modified files dictate its mobile-specific capabilities. These adaptations illustrate the fundamental challenge of using traditional Linux benchmarks to evaluate divergent mobile environments.

Mobile-Specific Evaluation Frameworks: Recent efforts have attempted to isolate software performance on identical hardware to provide more accurate comparative metrics. The AnROM framework [13] introduced a methodology to empirically evaluate the performance, privacy, and resource trade-offs of custom Android-based operating systems. While AnROM successfully isolated software impact, its scope was strictly confined to the Android ecosystem. Furthermore, as highlighted by recent memory management studies [14], existing mobile benchmarking suites often fail to isolate real-world operating system performance from the complexities of heterogeneous, multi-tiered asymmetrical core architectures. MLPerf Inference [15] provides a standardized methodology for evaluating machine learning inference across diverse hardware and software configurations, and MLPerf Mobile Inference [16] extends this methodology specifically to mobile devices. While focused exclusively on artificial intelligence workloads rather than general operating system performance, these works demonstrate the value of reproducible, task-specific benchmarking frameworks.

MobiOSBench bridges these methodological gaps by holding the physical hardware constant and explicitly capturing holistic system behaviors alongside raw compute performance. By expanding the evaluation paradigm beyond a single ecosystem, MobiOSBench enables standardized, cross-architectural comparisons of fundamentally divergent operating systems, ranging from open-source Linux variants to heavily sandboxed proprietary environments.

III. Background

Mobile operating systems provide essential functionalities and define the user experience. To ensure a rigorous comparative analysis, MobiOSBench evaluates operating systems based on two strict criteria: (a) they are explicitly designed for mobile devices, and (b) they are structurally distinct, rather than mere downstream derivatives of another system. For example, MobiOSBench evaluates Android rather than custom Android derivatives such as LineageOS. To the best of the author's knowledge, MobiOSBench is the first system to enumerate, classify, and benchmark this diverse landscape of active mobile operating systems and environments.

OS	Kernel	Source	Target Devices
iOS [17]	Hybrid (XNU)	Closed	Apple
HarmonyOS NEXT [18]	Micro. (Harmony)	Closed	Huawei
Android [19]	Monolithic (Linux)	Open	Various
Fedora Mobility [20]	Monolithic (Linux)	Open	Various
KaiOS [21]	Monolithic (Linux)	Open	Keypads
LuneOS [22]	Monolithic (Linux)	Open	Various
Maemo Leste [23]	Monolithic (Linux)	Open	Various
Manjaro ARM [24]	Monolithic (Linux)	Open	Pine64
Mobian [25]	Monolithic (Linux)	Open	Various
Nemo Mobile [26]	Mono. (Linux/Mer)	Open	Pine64, Volla
postmarketOS [27]	Monolithic (Linux)	Open	Various
PureOS [28]	Monolithic (Linux)	Open	Purism
Sailfish OS [29]	Mono. (Linux/Mer)	Open	Jolla, Xperia
Ubuntu Touch [30]	Monolithic (Linux)	Open	Various

TABLE I: Active mobile operating systems

Modern consumer systems like Android and iOS are characterized by highly optimized, vertically integrated architectures targeting specific hardware. Android builds upon the Linux kernel, modifying it with mobile-specific additions like the Binder IPC mechanism, together with user-space components such as the Android Runtime (ART) [12]. iOS employs the XNU hybrid kernel [31], balancing microkernel modularity with monolithic performance [32].

In contrast, alternative mobile operating systems, shown in Table I, prioritize modularity, privacy, and open-source principles over strict hardware optimization [33]. Systems such as Ubuntu Touch, postmarketOS, and PureOS leverage the Linux kernel. To bridge the gap between these standard Linux environments and proprietary mobile hardware, projects like `Halium` [34], which uses the `libhybris` [35] compatibility layer, are often utilized to interact with Android device drivers.

This architectural diversity, ranging from tightly coupled proprietary kernels to community-driven Linux environments, highlights the need for a unified evaluation framework. By holding the physical hardware constant, MobiOSBench isolates the specific contributions of the operating system itself, evaluating CPU utilization, memory management, and responsiveness to reveal the inherent trade-offs of these divergent designs.

IV. Methodology

This section outlines the MobiOSBench methodology used to evaluate and compare mobile operating systems. The proposed MobiOSBench framework is designed to provide a comprehensive analysis of mobile operating systems, focusing on general system information, security and privacy features, resource usage, performance, and efficiency. The MobiOSBench methodology ensures that the analysis is not limited to hardware metrics; it also evaluates the mobile operating system software itself. The methodology goes beyond traditional benchmarks by incorporating tests that address important mobile requirements, such as power consumption and connectivity services. This comprehensive approach provides a nuanced understanding of how mobile operating systems

perform under real-world conditions. This section provides information on the framework tests, why they are relevant, and how the information can generally be retrieved.

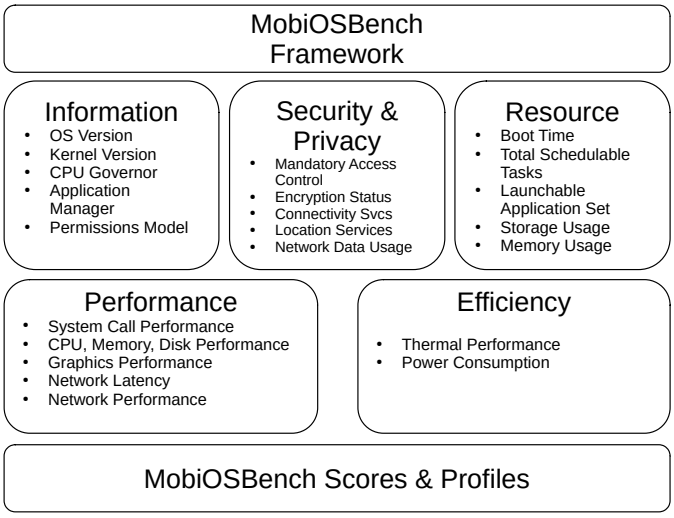


Fig. 1: MobiOSBench overview

Figure 1 provides an overview of the MobiOSBench framework, illustrating its components used for evaluating a mobile operating system.

A. General Information

The first part of the methodology involves gathering general information about the operating system. This part defines what the system is and what environment it is currently running in. This general information provides insights into the architecture, design, and intended use cases of the mobile operating system. The following information is retrieved:

Operating System Version: The operating system version used is programmatically extracted by querying standard system property interfaces. Specifically, this is achieved by reading standard release files such as `/etc/os-release` or interfacing with the native property service on Android. This information establishes the specific software environment being benchmarked, ensuring reproducibility.

Kernel Version: The kernel version used by the operating system is determined by interfacing with the native `uname` utility. This information highlights the underlying operating system architecture and its update cycle. Understanding the kernel version also provides insights into the level of optimization and security patches integrated into the system.

CPU Governor: The CPU governor [36] manages the CPU frequency scaling. It adjusts the CPU clock speed to balance power consumption and performance. To retrieve the active governor algorithm, the framework programmatically queries the virtual filesystem at standard hardware abstraction paths such as `/sys/devices/system/cpu/`. The choice of the CPU governor algorithm directly affects performance on mobile systems, making this automated extraction critical for establishing a valid benchmarking baseline.

Application Manager: To identify the active package management infrastructure, the framework employs a universal heuristic probing mechanism. It systematically queries the execution environment for signature package manager binaries and services such as `pm`, `click`, `apk`, or `apt`. By evaluating the success of these probes, the framework dynamically fingerprints the application ecosystem. This unified approach ensures the benchmarking suite remains highly extensible and architecture-agnostic across diverse mobile platforms. Knowing the application management infrastructure highlights architectural differences in application provisioning across platforms.

Permissions Model: The user-space permissions model is logically inferred from the active application packaging format and ecosystem. This model is critical for understanding how the operating system manages user and application privileges. It dictates user authority over sensitive data, the specific mechanisms through which hardware and software permissions are requested and granted, and the granularity of access control, such as per-app versus system-wide architectures.

B. Security and Privacy Benchmarks

The second part of the methodology focuses on the default security and privacy mechanisms of the mobile operating system. This part defines how the system protects itself out of the box. This information reveals how the architecture protects user data and isolates sensitive hardware interfaces. The following security and privacy metrics are retrieved:

Mandatory Access Control: The active kernel security module is programmatically detected by querying the virtual filesystems of the operating system. Rather than relying on user-space configuration files, the framework probes standard kernel interfaces such as `/sys/kernel/security/` and `/sys/fs/selinux/`. By evaluating the presence and enforcement states within these paths, the framework definitively identifies active isolation mechanisms like SELinux or AppArmor. This automated inspection reveals how strictly the underlying kernel enforces access control policies across different mobile architectures.

Encryption Status: The default encryption state is programmatically verified by querying abstract system properties and block device mappings. The framework standardizes on detecting underlying cryptographic states regardless of the high-level operating system. For example, on Android environments, the framework interfaces with the native property service to detect file-based encryption. Similarly, on Linux-based operating systems, it probes the kernel device mapper and virtual block interfaces such as `/sys/class/block/` for signatures tied to the Linux Unified Key Setup standard, commonly known as LUKS, while also checking for filesystem-level encryption mechanisms like `fscrypt`. This automated check evaluates whether the operating system encrypts user data by default, which is essential for ensuring privacy and protecting against unauthorized access if a device is lost or stolen.

Connectivity Services: The default states of interfaces like Wi-Fi, Bluetooth, and NFC are programmatically inspected using a flexible combination of low-level network system calls and high-level framework queries. The framework abstracts the

hardware discovery process, prioritizing native cross-platform execution where possible. For Wi-Fi, the framework achieves this by issuing the `SIOCGIFFLAGS ioctl` request against standard wireless network interfaces. Where direct access is restricted, the framework adapts to the underlying security constraints. On Linux systems, it probes hardware state abstractions such as `/sys/class/rfkill/` or interfaces with the `D-Bus` message daemon. On Android, it leverages the Java Native Interface (JNI) to query the application framework directly. This automated evaluation identifies potential security and privacy risks from automatically enabled connectivity features, which may expose the device to tracking or unauthorized connections. Wi-Fi is assumed to be always on for the purpose of benchmarking.

Location Services: The status of location services is programmatically verified by querying high-level system APIs and inter-process communication buses. The framework employs an adaptable query mechanism that respects the architectural security boundaries of the host system. For instance, because Android employs strict sandboxing and Hardware Abstraction Layer policies that restrict direct native access, the framework utilizes the JNI to invoke location manager methods directly. Conversely, on standard Linux systems, the framework interfaces with configuration backends and message buses such as `gsettings` or `D-Bus` to read the active location service state. This automated check determines whether location tracking is enabled by default, assessing the approach taken by the operating system to manage location data, user prompts, and overall transparency.

Network Data Usage: Once this test is reached, the system is left idle for five minutes. The total volume of network data transmitted and received is quantified in KB. To ensure precise measurement across divergent architectures, the framework employs standard network interface queries utilizing `getifaddrs` where applicable, such as on Linux environments. Conversely, due to stricter virtual filesystem access controls on Android, the framework utilizes JNI to invoke the higher-level `TrafficStats` Android API.

C. Resource Benchmarks

The third part of the methodology measures the efficiency of the mobile operating system in utilizing hardware resources. This part defines what the system costs to exist. These tests are crucial for understanding the impact of the operating system on device performance and user experience. The following information is retrieved:

Boot Time: The system initialization duration is measured from the device starting to the exact moment the operating system achieves software readiness. To maintain cross-platform portability, the framework employs an abstract, event-driven hook architecture rather than relying on hardware-dependent visual cues. It actively listens for late-stage initialization signals broadcast by the underlying service manager, capturing the absolute kernel time the instant user-space transitions into a fully interactive software state. For instance, on standard Linux-based systems, the framework implements this concept via `systemd` service profiling by injecting a user-service drop-in with an `ExecStartPost` hook to record the `CLOCK_BOOTTIME`. Similarly, within the Android environment, the framework applies this same methodology by deploying a System Broadcast Receiver to intercept the `LOCKED_BOOT_COMPLETED` intent, recording the elapsed real time the moment the system completes booting. Because these kernel-anchored clocks begin only once the bootloader transfers control, a fixed, per-device and per-operating-system timing constant covering the preceding bootloader stage is added to each recorded value, so the reported duration reflects the full interval from power-on. As this stage is firmware-driven and highly repeatable, the constant is obtained through repeated external timing measurements. The physical display hardware requires a brief calibration margin to render the final lock screen and physically accept touch inputs, which varies by operating system and device configuration. By measuring the pure framework initialization before this hardware delay, the methodology successfully isolates the software stack. This unified approach eliminates hardware-specific display rendering variables and provides a scientifically reproducible metric for architectural boot efficiency across diverse mobile operating systems.

Total Schedulable Tasks: The active execution overhead is quantified by extracting the total number of scheduled kernel tasks. Because modern sandboxed environments aggressively restrict file-level access to the global process tree, the framework bypasses virtual filesystem traversal entirely. To maintain cross-platform automation, the framework invokes native kernel interfaces, such as `sysinfo`, to query system information. By directly querying the core kernel data structures, the framework extracts the exact global count of active scheduling entities, encompassing both primary processes and their associated concurrent threads. This architectural approach guarantees a highly accurate measurement of system overhead without requiring elevated debugging privileges. Furthermore, by capturing the total thread count, the methodology exposes the true scheduling burden placed on the CPU. This successfully highlights the heavy multithreading overhead inherent to operating systems reliant on inter-process communication mechanisms.

Launchable Application Set: The default software payload is evaluated by quantifying the exact number of pre-installed graphical applications exposed directly to the user. To avoid conflating hidden system daemons with actual interactive software, the framework standardizes its detection methodology to isolate purely launchable applications. Rather than enumerating all installed system packages, the framework specifically filters for entities designated to appear within the primary graphical environment. Within the Android architecture, the framework achieves this by utilizing the JNI to query the system package manager for activities explicitly bound to a launcher category intent. Similarly, on Linux-based mobile operating systems, the framework leverages native desktop bus interfaces and application registry queries to actively filter out background services and extract the true count of launchable graphical applications. This unified approach accurately gauges the default user-space application footprint across different mobile operating systems.

Storage Usage: To comprehensively capture the storage usage, the framework utilizes the `getmntent` and `statvfs` standard interfaces to query the block device mount statistics across all physical partitions. By calculating the exact block utilization across the read-only system mount points and the mutable data environment, the methodology is able to accurately extract the total storage usage. This approach successfully mirrors the real-world user experience by accounting for both the immutable core framework binaries and the resident applications. Under a fresh install, this would reflect the true storage footprint of the operating system. This information is particularly important given that mobile architectures are fundamentally constrained by limited storage capacity.

Memory Usage: The idle memory consumption is quantified by directly parsing the `meminfo` file within `procfs`. This direct approach extracts the highly granular physical memory footprint of the operating system. This test highlights the core memory efficiency of the architecture and its ability to minimize background resource allocation. To account for dynamic background fluctuations, the memory usage is continuously sampled over a defined execution window and automatically averaged by MobiOSBench.

D. Performance Benchmarks

The fourth part of the methodology evaluates the performance of the mobile operating system. A series of performance benchmarks is conducted with the brightness programmatically set to the maximum for all tests. The following tests measure system performance:

System Call Performance: To evaluate both baseline kernel efficiency and architectural process overhead, two distinct system calls are measured. First, a lightweight system call, `gettid()`, is executed ten million times to measure raw user-to-kernel context switching latency. Second, a heavyweight process creation call, `fork()`, is executed one thousand times to measure the latency of memory management, page table duplication, and user-space process instantiation. This dual approach isolates the fundamental performance of the underlying kernel from the structural penalties imposed by the specific mobile operating system architecture, such as the Zygote model in Android. Execution times are measured in seconds, and `syscall()` is used to call these system calls to avoid any wrappers that might cache values.

CPU, Memory, and Disk Performance: These benchmarks, measured in seconds, are conducted by running CPU, memory, and disk operations. CPU is benchmarked with a billion arithmetic operations, memory is tested by storing and loading 2 GB of data, and disk is tested by writing and then reading 2 GB of page-aligned data. To avoid skewed results due to caching and compiler optimization, the `O_DIRECT` or `F_NOCACHE` file modes, `fsync`, and the `volatile` keyword are used. These metrics provide a comprehensive view of the hardware utilization of the mobile operating system.

Graphics Performance: 2D and 3D graphics tests, measured in seconds, are conducted using native OpenGL by rendering a thousand 2D frames and then a thousand 3D frames to evaluate the graphical capabilities of the mobile operating system, which are critical for gaming and multimedia applications.

Network Latency: The network latency, measured in milliseconds, is quantified using a TCP-based connection method. This exercises the same connection-establishment path used by real applications, whereas ICMP echo requests bypass the TCP stack entirely and, in production networks, are frequently throttled or blocked by providers [37]. To isolate the performance of the native TCP/IP stack in the operating system from external hardware variables such as wireless interference, the benchmark is executed entirely over the local loopback interface utilizing an automated, concurrently spawned receiver thread. Specifically, the framework measures the elapsed time required to successfully complete a TCP three-way handshake via the `connect()` system call. The automated test executes 105 consecutive iterations, discarding the first five to allow the system to warm up, before calculating the definitive average. A strictly enforced 300 ms idle period is inserted between iterations to isolate measurements. All sockets are configured with the `TCP_NODELAY` socket option.

Network Performance: A TCP throughput test is conducted to measure sustained data transfer rates in MB per second, a crucial metric for evaluating the efficiency of modern mobile applications. To isolate the network stack of the operating system from physical hardware constraints and wireless interference, the benchmark transmits 1 GB of data over the local loopback interface to an automated, concurrently spawned receiver thread. The framework utilizes a page-aligned 1 MB buffer. An initial 1 MB payload is transmitted unmeasured to warm up the connection. Crucially, to ensure the recorded time reflects true end-to-end traversal rather than asynchronous user-to-kernel memory copying, the framework employs a graceful TCP teardown. By invoking `shutdown()` and blocking on a `read()` call until the receiver closes the connection, the timer strictly captures the moment the kernel has successfully flushed the entire payload. All sockets are configured with the `TCP_NODELAY` socket option.

The MobiOSBench performance benchmarks are deliberately restricted to native system interface tests rather than real-world application workloads. This is a principled design decision that follows directly from the framework's core goal of isolating operating system behavior from application-layer variability. Mobile applications targeting different platforms are not functionally equivalent implementations. A web browser on Android, Ubuntu Touch, and Mobian represents three independently developed codebases with different rendering engines, memory management strategies, and platform integration approaches. Even a shared application codebase compiled across platforms does not resolve this confound, as identical source code necessarily invokes different platform-specific middleware layers on each target. Prior work on mobile application programming interface evolution demonstrates that functionally equivalent operations implemented through different interface generations can produce performance differences exceeding 50% [38] within an identical operating system environment, confirming that application-layer tests reflect implementation choices rather than operating system

behavior.

Real-world application workloads simultaneously invoke multiple operating system subsystems including memory allocators, graphics compositors, storage drivers, and networking stacks in ways that vary unpredictably across platforms, making it difficult to attribute measured performance differences to any specific operating system component. MobiOSBench addresses this by evaluating workloads that execute through identical or very similar native system interfaces on all platforms, ensuring that measured differences reflect operating system architectural properties rather than application or middleware variability. The graphics benchmark illustrates this principle directly. Rendering performance under a controlled synthetic workload is dominated by display server compositing behavior, a specific and reproducible operating system-level characteristic, rather than raw hardware graphics capability. This confirms that even seemingly hardware-bound workloads expose meaningful operating system-level differences when properly isolated, validating the synthetic benchmark approach.

E. Efficiency Benchmarks

The fifth and final part of the methodology evaluates the performance of the mobile operating system with respect to how the entire system behaves over a sustained period. When the test is started, the benchmark remains idle for five minutes to let the system reach thermal equilibrium. The user is instructed to disconnect power from the system. After the initial five-minute idle period, a specific set of performance benchmark tests is run for the next ten minutes. During that period, the thermal performance and power consumption are measured.

To evaluate the sustained performance and holistic resource management of the operating system, the efficiency test phase employs a fixed-time continuous stress methodology. The five-minute idle period guarantees the hardware reaches thermal equilibrium before sustained active testing commences. The framework executes a strict ten-minute loop of the core performance tests, specifically CPU arithmetic operations, memory loading and storing, and disk reading and writing, while tracking the total time spent during the test.

This specific workload is intentionally selected to isolate the core operating system functionality. Network testing is explicitly excluded from this loop, as socket blocking states introduce artificial idle periods that cool the processor. Similarly, graphics rendering is omitted because mobile GPUs rely heavily on proprietary, closed-source driver blobs that enforce their own hidden hardware management rules, thereby masking the behavior of the operating system regarding thermal management and power consumption. By pinning only the subsystems managed directly by the kernel, and maintaining this load for ten minutes to guarantee thermal saturation, the framework naturally reveals how the operating system behaves under stress. The tests are as follows:

Thermal Performance: The system temperature, measured in degrees Celsius, is continuously monitored immediately following the five-minute idle period used to allow the device to reach thermal equilibrium. At this stage, the display brightness is maximized, and the framework executes a continuous ten-minute loop of the core performance benchmarks. To ensure a strictly equivalent evaluation across divergent operating systems and bypass restricted hardware abstraction layers, the architecture standardizes on tracking the physical battery temperature by dynamically querying the `BatteryManager` API via JNI on Android and parsing the `/sys/class/power_supply/battery/temp` virtual filesystem file on Linux. Unlike localized system-on-chip CPU sensors, which exhibit extreme volatility dictated by instantaneous scheduler frequency spikes, the battery acts as a unified thermal mass [39, 40]. This provides a highly stable and holistic measure of chassis heat saturation, accurately assessing the thermal management capabilities of the operating system under sustained hardware stress.

Power Consumption: To quantify the absolute energy demanded by the operating system under sustained stress, the framework records the total battery capacity drawdown concurrently with the ten-minute thermal loop. Because evaluating true operating system power drain requires isolating the hardware from external charging, the framework explicitly mandates the physical disconnection of the device from any power source prior to execution. This guarantees the power management integrated circuit routes current exclusively from the internal battery. To capture the precise percentage delta, the architecture records the battery capacity immediately before and after the maximized-brightness performance benchmark phase. To adapt to strict user-space security limitations that mask raw voltage and current data, the framework queries the native `BatteryManager` API on Android to retrieve the relative integer capacity, while on Linux, it directly parses the `/sys/class/power_supply/battery/capacity` virtual filesystem file. Provided the target testing hardware remains identical, this percentage delta serves as a reliable heuristic for the power consumption required by the kernel to sustain maximum hardware throughput.

To ensure the reproducibility and validity of the benchmarking suite, all experiments are conducted under a standardized hardware environment. The display is programmatically forced to a persistent active state with the brightness set to 100% to prevent adaptive dimming or low-power state transitions during execution. Furthermore, all background applications are terminated, and the device is physically disconnected from external power sources to ensure the operating system relies exclusively on internal battery management. The device is rebooted after each benchmark suite iteration and then left idle for a few minutes to clear any late-stage boot processes. These controls are maintained consistently across the benchmark to eliminate environmental variables.

Mobile operating systems such as Ubuntu Touch aggressively throttle out-of-focus applications to conserve battery life. To prevent the native terminal from suspending the benchmark during execution, all tests on systemd-based platforms are executed within a transient foreground control group managed by the user-level service manager, using `systemd-run`. This explicit isolation guarantees the threads maintain continu-

ous access to the CPU scheduler and prevents artificial latency spikes or deadlocks.

The benchmarking suite is implemented in C, compiled with a C++ toolchain, with JNI extensions for Android and any platform relying on a Java-based runtime. The operating systems are never modified, and the Android version only requires the `INTERNET`, `BLUETOOTH`, and `RECEIVE_BOOT_COMPLETED` permissions. The benchmarking tool averages multiple runs and reports any case where there is a significant variation across runs.

The MobiOSBench framework and methodology provide a structured and innovative approach for evaluating mobile operating systems, addressing both traditional performance metrics and mobile-specific requirements. By relying on programmatically retrievable system interfaces, MobiOSBench ensures a comprehensive and reproducible analysis.

The inclusion of tests for security and privacy features, resource usage, and performance highlights the holistic nature of the evaluation. Additionally, the integration of real-world scenarios, such as power consumption and thermal performance under load, ensures that the results are practical and relevant.

By focusing on the performance of the operating system itself and minimizing the influence of hardware variability, the MobiOSBench methodology offers unique insights into the trade-offs and optimizations inherent in different operating systems. These insights provide valuable guidance for developers, researchers, and end-users in selecting and optimizing mobile operating systems for diverse use cases.

F. The Unified Quantitative Index (MobiOSBench Score)

To provide a comprehensive and universally comparable evaluation of each mobile operating system, the MobiOSBench Score is introduced. This index aggregates the different benchmark results into a single standardized metric. Because the collected data encompasses fundamentally different units, an Absolute Baseline Scaling methodology based on Unit Scaled Multipliers is employed. This approach converts every recorded metric into a unitless performance ratio relative to a fixed theoretical baseline of one, completely eliminating unit heterogeneity and localized scaling biases.

For metrics where a lower measured value indicates superior performance, such as execution time or battery drain, the performance ratio M_i is calculated by dividing the theoretical baseline of one by the measured value V_i ($M_i = \frac{1}{V_i}$). Consequently, a smaller measured footprint yields a mathematically larger reward ratio. Conversely, for metrics where a higher measured value is optimal, such as network throughput, the raw measured value directly becomes the multiplier ($M_i = V_i$), appropriately rewarding high performance with a larger scaling factor.

To accurately reflect specific user priorities without mathematically diluting critical data points, these unitless ratios are categorized into four distinct functional pillars:

The `Privacy Pillar` aggregates the raw network transmission and reception metrics collected during the idle phase alongside a calculated `Security Risk Index`. To ensure that an increased attack surface mathematically reduces the final score, this index is calculated as the inverse of one plus accumulated demerit points ($\frac{1}{1+\text{demerits}}$) for insecure default configurations regarding mandatory access control, Encryption Status, Connectivity Services, and Location Services. Under Connectivity Services, each connectivity medium counts as an independent attack surface area. For example, if both NFC and Bluetooth are enabled, the demerit points are two rather than simply one. Demerits follow the default configuration policy of the operating system rather than device-specific driver availability, so a medium enabled by default incurs its demerit even where driver support is incomplete. Since Wi-Fi is assumed to be enabled by default, it is exempt from the index.

The `Resources Pillar` aggregates the system footprint, encompassing Boot Time, Total Schedulable Tasks, Launchable Application Set, Storage Usage, and Memory Usage.

The `Performance Pillar` evaluates active computational throughput by aggregating System Calls, CPU arithmetic operations, Memory Read and Write operations, Disk Read and Write operations, 2D and 3D Graphics rendering, Network Latency, and Network Performance.

Finally, the `Efficiency Pillar` aggregates the hardware strain metrics collected during the intensive benchmarking phase, specifically the Thermal Performance and Power Consumption.

Within each of these four pillars, an unweighted geometric mean is calculated to synthesize the disparate performance ratios into a single pillar score. The geometric mean is the established standard for aggregating benchmark ratios, as arithmetic means are known to produce misleading results when metrics differ significantly in magnitude [41, 42]. For example, Total Schedulable Tasks ranges from a few hundred to several thousand across evaluated systems, while Network Latency ratios operate at a fundamentally different scale; an arithmetic mean would be dominated by the former regardless of relative performance significance. If n is defined as the total number of metrics within a specific pillar, the pillar score P is calculated using (1).

$$P = \left[\prod_{i=1}^{n} M_i \right]^{\frac{1}{n}} \tag{1}$$

Because operating system preference is highly subjective, different weight distributions are applied to these four calculated pillars to create four distinct Use Case Profiles: a) The `Daily Driver` profile represents a standard user requiring a balanced experience, assigning fractional weights w of 0.4 to Performance, 0.3 to Efficiency, 0.2 to Resources, and 0.1 to Privacy. b) The `Power User` profile prioritizes raw computational throughput and multitasking ability, heavily favoring Performance at 0.6, alongside Resources at 0.2, Efficiency at 0.1, and Privacy at 0.1. c) The `Privacy Advocate` profile strictly demands secure defaults and low telemetry above all else, placing a substantial 0.5 weight on Privacy, with Efficiency at 0.2, Performance at 0.2, and Resources at 0.1. d) The `Neutral Baseline` profile assigns equal (0.25)

weights to all pillars. The specific weight distributions for the Use Case Profiles are intended as representative heuristics rather than definitive industry standards or empirically derived universal constants.

The MobiOSBench architecture allows researchers and developers to dynamically reconfigure priorities or cleanly isolate specific pillars to accommodate unique hardware topologies. By gracefully masking obfuscated telemetry in closed ecosystems or disengaging unsupported metrics such as internal battery consumption and thermals on tethered devices, MobiOSBench ensures that cross-architecture evaluations remain statistically valid.

The final MobiOSBench Score for a specific profile is calculated using a weighted geometric mean of the four-pillar scores. By raising each pillar score to the power of its assigned weight, the privacy and performance ratios dynamically scale to match the exact priorities of the end user. To convert the resulting fractional ratio into a legible, human-readable integer consistent with industry benchmarking practices, a constant Index Scaling Factor (β) of 1000 is applied. The sum of the weights always equals exactly one. The final equation is given by (2).

$$\mathrm{MobiOSBenchScore}_{\mathrm{Profile}} = \beta \times \left(P_{\mathrm{priv}}^{w_1} \times P_{\mathrm{res}}^{w_2} \times P_{\mathrm{perf}}^{w_3} \times P_{\mathrm{eff}}^{w_4} \right) \tag{2}$$

Because the Absolute Baseline Scaling methodology inherently treats every normalized variable as a performance ratio, the final MobiOSBench Score functions as a composite speedup index. Therefore, a higher final numerical value strictly indicates a superior operating system configuration.

The Use Case Profile definitions are informed by documented behavioral patterns in mobile user research: empirical studies demonstrate that smartphone users exhibit measurably distinct behavioral segments with divergent privacy and performance priorities [43], and that performance-related trade-off preferences vary significantly across user personas and contexts [44]. These findings confirm that the segmentation approach underlying MobiOSBench's profiles reflects genuine behavioral diversity rather than arbitrary categorization. The primary scientific contribution of the MobiOSBench scoring architecture lies in its mathematical flexibility, and researchers, developers, and administrators are explicitly encouraged to recalibrate these weights to reflect the precise priorities of their specific deployment context. Raw pillar scores are reported alongside all weighted composite scores throughout this paper, ensuring that weight-adjusted rankings remain fully verifiable against the underlying benchmark telemetry regardless of the profile configuration chosen.

It is important to clarify that the MobiOSBench Score does not claim to guarantee universal fairness through its weight assignments; rather, it guarantees mathematical consistency. Once a researcher defines priorities via weights, the weighted geometric mean produces consistent and reproducible scores across any hardware configuration. Fairness is enforced by the underlying evaluation methodology itself through identical physical hardware, identical workloads, and absolute baseline scaling, independent of the specific weight values chosen.

V. Evaluation

For evaluation, the MobiOSBench benchmark suite is executed ten times per configuration, with results averaged and reported to two decimal places. Measured values are reported with 95% confidence intervals for the mean, computed using Student's t-distribution; an interval displayed as ± 0.00 indicates a half-width below 0.005 in the units of the corresponding metric. This section presents the selection of mobile devices and operating systems and the corresponding execution and analysis of the MobiOSBench evaluation.

A. Mobile Operating System and Mobile Device Selection

Mobile Operating Systems
Android
Fedora Mobility
LuneOS
Maemo Leste
Mobian
postmarketOS
Ubuntu Touch

TABLE II: Candidate list of active mobile operating systems

The selection of mobile devices and operating systems plays a crucial role in evaluating the MobiOSBench framework. Testing multiple devices with different operating systems allows for: a) A thorough comparison of how the same operating system performs on varying hardware. b) A thorough comparison of how various operating systems perform on the same hardware. c) Validation of MobiOSBench being able to isolate operating system performance from hardware variability. d) Demonstration of the versatility of the framework in evaluating diverse operating systems.

To evaluate the MobiOSBench framework, mobile operating systems that support a variety of devices must be chosen, as shown in Table II. This ensures the ability to compare the same operating system across different hardware configurations and compare different operating systems on the same hardware. Mobile operating systems like iOS are excluded due to their restrictive hardware support. For example, iOS is only compatible with Apple devices, which limits comparative analysis unless the operating system is ported to non-Apple hardware or Apple devices support alternative operating systems. Android devices, on the other hand, can support multiple operating systems. LuneOS and Maemo Leste are still active projects but support only a small set of severely outdated devices. Similarly, Fedora Mobility supports an extremely limited number of devices.

Figure 2 shows a Venn diagram of chosen modern mobile devices that are heavily supported by multiple operating systems and the mobile operating systems that support them. The Venn diagram shows only three devices, as few smartphones receive sustained support from multiple operating systems. For example, some devices that run Android are only supported by Ubuntu Touch as an alternative operating system. The OnePlus 6 device is supported by all of the chosen mobile operating systems, enabling direct comparisons with multiple mobile devices, such as the Google Pixel 3a and the Fairphone 4. MobiOSBench thus uses the three devices and the common

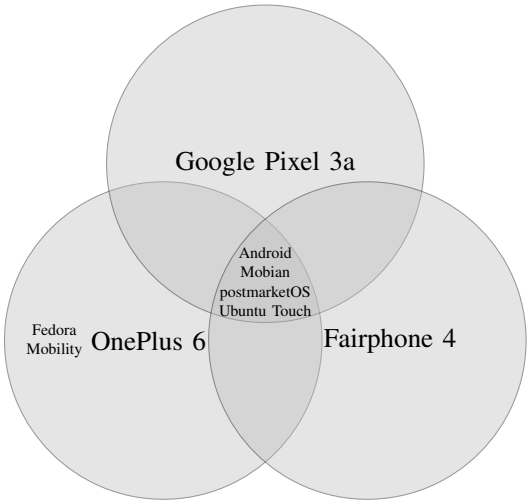


Fig. 2: Operating system overlap across evaluated devices

operating systems supported, namely Android, Mobian, postmarketOS, and Ubuntu Touch.

The OnePlus 6 uses a Qualcomm Snapdragon 845 Octa-core (4x2.8 GHz, 4x1.7 GHz) CPU, Adreno 630 GPU, 6 GB RAM, 64 GB of storage. The Google Pixel 3a uses a Qualcomm Snapdragon 670 Octa-core (2x2.0 GHz, 6x1.7 GHz) CPU, Adreno 615 GPU, 4 GB RAM, 64 GB of storage. Finally, the Fairphone 4 uses a Qualcomm Snapdragon 750G 5G Octa-core (2x2.2 GHz, 6x1.8 GHz) CPU, Adreno 619 GPU, 6 GB RAM, 128 GB of storage. The Fairphone 4 has the newest and best-performing CPU, followed by the OnePlus 6. The Google Pixel 3a has the lowest amount of RAM.

B. MobiOSBench Framework Testing

1) General Information: All optional telemetry and analytics prompts were declined during system installation. The device was connected to the local Wi-Fi and internet. Once the MobiOSBench benchmark was installed, the system was rebooted. Rebooting is not required by the suite, but is needed for a valid boot-time measurement. Before the entirely automated MobiOSBench benchmark was initiated, the system was left idle for a few minutes to ensure all startup activity had completed.

Table III lists the general system information for each operating system. Android has different releases for each device; therefore, the operating system version and the kernel version differ depending on the release date. For postmarketOS, Ubuntu Touch, and Mobian, the operating system versions are identical across the devices since each of these operating systems is responsible for maintaining and supporting the devices. As such, the releases are coordinated. However, some kernel major versions differ across operating systems. For application managers, Android uses the Google Play Store; postmarketOS and Mobian use GNOME Software; and Ubuntu Touch uses OpenStore. Finally, all operating systems on all devices support a per-application and per-service permissions model. This model allows approving specific services, such as location or camera access, on a per-application basis.

postmarketOS and Mobian use the most recent kernel versions among the evaluated systems. Ubuntu Touch lags behind Android in kernel version on the OnePlus 6 and Pixel 3a, though it carries a more recent patch level on the Fairphone 4. These kernel version differences may affect the security features or performance on these devices. Android 11 is the only release officially available on all three devices and was used to hold the Android baseline constant. Fairphone ships only the final Android 11 build (January 2023), which accounts for its newer kernel.

While application managers might differ, a per-application, per-service permissions model appears to be the most prevalent across mobile operating systems. All operating systems on all devices use the `schedutil` CPU governor. Thus, there is no variation attributable to governor selection.

2) Security and Privacy Benchmarks: Table IV lists the privacy and security-related features of each operating system across the three devices. For each operating system, the reported information is generally consistent across devices. Android uses SELinux and enables encryption, location services, NFC, and Bluetooth by default. postmarketOS does not use any mandatory access control (MAC) and does not enable encryption or location services by default. While NFC and Bluetooth are enabled by default for the OnePlus 6 and the Pixel 3a, the Fairphone 4 does not currently have the NFC feature working in postmarketOS [45]. Ubuntu Touch uses AppArmor for its mandatory access control but does not enable encryption. Location services, NFC, and Bluetooth are enabled by default. Mobian is similar to postmarketOS but with a working NFC feature for Fairphone 4. The absence of encryption in these cases reflects ongoing platform development rather than a deliberate architectural decision. A similar situation applies to NFC on the Fairphone 4 under postmarketOS; since postmarketOS enables NFC by default, the demerit still applies, while Table IV reports the observed state.

After reporting the location services status, the device is left idle for five minutes to collect background network data usage. All other applications are closed to ensure they do not affect the results. The data reveals a contrast in background network activity between the operating systems. Android consistently exhibits the highest background data transmission across all devices, and the highest reception on the OnePlus 6 and Fairphone 4, transmitting over 22 KB and receiving over 111 KB on the latter while completely idle. Conversely, the alternative operating systems transmit substantially less in every configuration. Mobian transmits less than 1 KB across all devices, and Ubuntu Touch maintains the lowest reception footprint at roughly 11 KB. This result quantitatively shows that Android possesses a more active background network footprint, whereas the native Linux ports generally conserve user bandwidth when idle, potentially reducing privacy exposure.

3) Resource Benchmarks: Table V lists the general resource usage by the operating systems for each device. The boot times recorded do not reveal a consistent trend across devices as they depend heavily on the operating system along with the drivers. For example, the Fairphone 4 has a known

OS	Device	OS Version	Kernel	CPU Governor	App Manager	Permissions Model
Android	OnePlus 6	OxygenOS 11 (11/2021)	4.9.227	schedutil	Google Play Store	Per-app, per-service
	Pixel 3a	Android 11 (10/2021)	4.9.248	schedutil	Google Play Store	Per-app, per-service
	Fairphone 4	Android 11 (01/2023)	4.19.157	schedutil	Google Play Store	Per-app, per-service
postmarketOS	OnePlus 6	25.12	6.16.7	schedutil	GNOME Software	Per-app, per-service
	Pixel 3a	25.12	6.18.5	schedutil	GNOME Software	Per-app, per-service
	Fairphone 4	25.12	6.17.6	schedutil	GNOME Software	Per-app, per-service
Ubuntu Touch	OnePlus 6	20.04.6 LTS	4.9.112	schedutil	OpenStore	Per-app, per-service
	Pixel 3a	20.04.6 LTS	4.9.124	schedutil	OpenStore	Per-app, per-service
	Fairphone 4	20.04.6 LTS	4.19.265	schedutil	OpenStore	Per-app, per-service
Mobian	OnePlus 6	Debian 13 Trixie	6.12	schedutil	GNOME Software	Per-app, per-service
	Pixel 3a	Debian 13 Trixie	6.12	schedutil	GNOME Software	Per-app, per-service
	Fairphone 4	Debian 13 Trixie	6.12	schedutil	GNOME Software	Per-app, per-service

TABLE III: General system information for each operating system across devices

OS	Device	MAC	Encryption	Connectivity	Location	Net TX (KB)	Net RX (KB)
Android	OnePlus 6	SELinux	Enabled	NFC, Bluetooth	Enabled	14.28 ± 4.62	100.81 ± 28.32
	Pixel 3a	SELinux	Enabled	NFC, Bluetooth	Enabled	20.14 ± 9.35	25.84 ± 10.24
	Fairphone 4	SELinux	Enabled	NFC, Bluetooth	Enabled	22.80 ± 6.28	111.65 ± 14.36
postmarketOS	OnePlus 6	No	Disabled	NFC, Bluetooth	Disabled	1.49 ± 0.21	50.56 ± 10.97
	Pixel 3a	No	Disabled	NFC, Bluetooth	Disabled	1.68 ± 0.12	48.17 ± 9.91
	Fairphone 4	No	Disabled	Bluetooth	Disabled	1.46 ± 0.22	47.09 ± 9.55
Ubuntu Touch	OnePlus 6	AppArmor	Disabled	NFC, Bluetooth	Enabled	4.51 ± 0.25	10.92 ± 0.99
	Pixel 3a	AppArmor	Disabled	NFC, Bluetooth	Enabled	4.51 ± 0.33	11.57 ± 1.91
	Fairphone 4	AppArmor	Disabled	NFC, Bluetooth	Enabled	5.07 ± 0.15	11.29 ± 0.97
Mobian	OnePlus 6	No	Disabled	NFC, Bluetooth	Disabled	0.72 ± 0.32	47.81 ± 12.10
	Pixel 3a	No	Disabled	NFC, Bluetooth	Disabled	0.89 ± 0.52	48.49 ± 11.42
	Fairphone 4	No	Disabled	NFC, Bluetooth	Disabled	0.81 ± 0.35	45.95 ± 10.75

TABLE IV: Security and Privacy features for each operating system across devices

OS	Device	Boot Time (s)	Sched. Tasks	Apps	Storage (GB)	Memory (GB)
Android	OnePlus 6	25.59 ± 0.47	3645 ± 110	30	12.31 ± 0.01	2.25 ± 0.05
	Pixel 3a	41.72 ± 2.56	2783 ± 60	21	6.68 ± 0.01	1.97 ± 0.02
	Fairphone 4	43.92 ± 0.59	3485 ± 120	23	7.98 ± 0.45	2.28 ± 0.04
postmarketOS	OnePlus 6	33.77 ± 3.25	700 ± 7	22	2.47 ± 0.04	1.01 ± 0.06
	Pixel 3a	60.95 ± 4.57	680 ± 5	22	2.41 ± 0.03	0.97 ± 0.05
	Fairphone 4	37.02 ± 6.61	713 ± 22	22	2.21 ± 0.01	0.98 ± 0.05
Ubuntu Touch	OnePlus 6	30.68 ± 0.75	1495 ± 54	19	4.82 ± 0.09	1.37 ± 0.06
	Pixel 3a	46.66 ± 0.66	1337 ± 4	19	4.50 ± 0.07	1.21 ± 0.02
	Fairphone 4	26.09 ± 0.49	1700 ± 6	19	4.62 ± 0.08	1.58 ± 0.02
Mobian	OnePlus 6	40.20 ± 8.68	650 ± 30	28	4.23 ± 0.01	1.12 ± 0.04
	Pixel 3a	30.73 ± 5.16	629 ± 27	28	3.94 ± 0.00	1.05 ± 0.02
	Fairphone 4	117.50 ± 10.81	658 ± 21	28	4.10 ± 0.45	1.11 ± 0.02

TABLE V: Resource benchmarks for each operating system across devices

driver timeout in Mobian that significantly delays the boot-up process [46]. On average, Ubuntu Touch shows the fastest boot time, followed by Android and postmarketOS. On these devices, the bootloader stage includes the unlocked-bootloader warning, which is therefore reflected in the reported boot times. Ignoring the hardware-specific Mobian timeout on the Fairphone 4, Mobian's boot times on other devices remain highly variable compared to the rest.

Android shows the highest number of tasks across all devices, followed by Ubuntu Touch with almost half the number of tasks. Android has the highest number of tasks due to additional services provided by the operating system. Mobian and postmarketOS have the lowest footprint owing to their lighter-weight operating systems. On average, the number of launchable applications under Mobian is the highest, followed by Android, then postmarketOS, and finally Ubuntu Touch. Mobian and Android add additional applications and services for users, while Ubuntu Touch focuses on a minimal core application set.

As a larger operating system and due to the higher number of applications and processes, Android uses much more storage, over 6 GB per device. Ubuntu Touch and Mobian use around 4 GB per device since Ubuntu Touch includes Android container layers and Mobian has more applications and background services running. postmarketOS, a lighter-weight operating system, uses the least amount of storage at around 2 GB. For memory usage, Android uses the highest amount of memory due to the larger number of processes

running. Ubuntu Touch uses, on average, more memory than postmarketOS and Mobian.

The resource results reveal a specific footprint for each operating system. Android shows heavier resource usage due to additional applications and services provided to the user. MobiOSBench also identifies postmarketOS as a lightweight operating system alternative.

4) Performance Benchmarks: Table VI presents the execution times for the `gettid` and `fork` system calls across all tested devices and operating systems. The `gettid` benchmark serves as a critical control metric for baseline kernel efficiency. Across all platforms, `gettid` execution times remained generally consistent, ranging from 1.15 s to 2.83 s, demonstrating that the underlying Linux kernels handle fundamental user-to-kernel context switching with comparable efficiency, establishing a fair hardware baseline irrespective of the overlying operating system.

In contrast, the `fork` benchmark exposes massive architectural disparities in process creation overhead. While the traditional GNU/Linux environments, such as postmarketOS, Ubuntu Touch, and Mobian, executed the workload rapidly and generally under 2.5 s, Android exhibited a severe latency penalty, taking up to 35.43 s on the OnePlus 6, nearly 45 times slower than Mobian on the same device. This degradation can be attributed to a heavyweight user-space architecture in Android, where process creation is heavily burdened by the Android Runtime process address space, aggressive Binder IPC registration, and complex SELinux enforcement, unlike the lightweight, native execution models of its alternatives. Fairphone 4 shows significantly better performance compared to the OnePlus 6 due to its more modern silicon and newer kernel.

Table VI also details the execution times for CPU, memory, and disk workloads across the tested devices. For the CPU benchmark, postmarketOS and Mobian consistently demonstrate the lowest or statistically tied execution times across all hardware. Android performs competitively but lags slightly behind these lightweight operating systems. Ubuntu Touch struggles significantly on older hardware like the OnePlus 6 and Pixel 3a, taking almost twice as long as the other operating systems to complete the workload. However, on the modern Fairphone 4, all operating systems converge to approximately 7-9 s. This convergence indicates that modern silicon can overcome software-level CPU overhead.

The memory read and write benchmarks reveal a different performance landscape. Android shows highly optimized memory management on legacy hardware, largely outpacing the Linux alternatives for memory operations, particularly on the OnePlus 6. Ubuntu Touch again presents severe performance degradation on older devices. It requires nearly 18 s for memory writes on the OnePlus 6, whereas Android completes the same task in under 8 s. This disparity suggests that the legacy kernel paired with Ubuntu Touch lacks the memory allocation optimizations present in Android and the newer kernels utilized by postmarketOS and Mobian.

Disk storage performance is heavily dictated by the underlying flash hardware, yet software handling remains a crucial factor. Read speeds are broadly consistent across Android, Ubuntu Touch, and Mobian per device, though postmarketOS exhibits notably higher disk read times on the OnePlus 6 and Fairphone 4. Disk write speeds expose Ubuntu Touch as a massive outlier once more, taking over 26 s on the Pixel 3a. Meanwhile, postmarketOS and Mobian slightly outperform Android in disk write speeds across the board. The data confirms that the alternative mobile operating systems can interface with mobile flash storage just as efficiently as commercial Android releases, provided the kernel ports are properly optimized.

Table VI presents the execution times for 2D and 3D graphics rendering benchmarks. A prominent trend across all operating systems and devices is the near-identical execution time between the 2D and 3D workloads. This parity indicates that execution is dominated by display synchronization and compositing rather than raw GPU compute [47]. postmarketOS and Mobian sit at approximately 16.7 s, corresponding to one frame per 60 Hz refresh interval, while Android and Ubuntu Touch complete below it; with hardware held constant, these differences reflect the presentation behavior of each display stack rather than rendering capability.

Ubuntu Touch generally records the lowest rendering times of the evaluated systems. It significantly outpaces Android on older hardware, completing 3D workloads on the Pixel 3a in just over 11 s compared to Android requiring over 15 s. Android maintains highly consistent performance across all devices but fundamentally trails Ubuntu Touch in this workload. Conversely, postmarketOS and Mobian exhibit the highest execution times, generally requiring 16-17 s regardless of the underlying hardware.

To isolate raw hardware performance from compositor overhead, postmarketOS uniquely allows users to disable vertical synchronization (VSync) [47, 48]. While these uncapped results are omitted from the main table to maintain a standardized baseline, disabling vertical synchronization significantly reduces rendering times to an average of 0.4 s for the OnePlus 6, 0.7 s for the Pixel 3a, and 0.5 s for the Fairphone 4. These uncapped results accurately align with the true computational capabilities of the underlying mobile GPUs. This hardware-level data confirms that the primary graphics bottleneck on these operating systems lies within software synchronization protocols rather than actual hardware rendering limitations.

Table VI details the network latency and throughput performance across the mobile operating systems. The latency benchmark exposes a significant performance gap between Android and the other operating systems. Android consistently exhibits the highest network latency across all three devices, from 3.14 to 6.30 ms. In contrast, postmarketOS, Ubuntu Touch, and Mobian operate with near or sub-millisecond latency. Mobian achieves the lowest latency overall, recording 0.21 ms on the Fairphone 4. This disparity highlights the substantial overhead introduced by the complex network routing daemons and hardware abstraction layers native to the Android ecosystem compared to the direct network stack access available in traditional Linux environments.

The network throughput results further validate the efficiency of the native Linux network stack. Mobian dominates this category, achieving the highest data transfer rates

System Calls, CPU, and Memory Performance						
OS	**Device**	**`gettid` (s)**	**`fork` (s)**	**CPU (s)**	**Mem RD (s)**	**Mem WR (s)**
Android	OnePlus 6	2.32 ± 0.01	35.43 ± 0.71	11.47 ± 0.00	5.84 ± 0.01	7.76 ± 0.02
	Pixel 3a	2.48 ± 0.00	21.13 ± 1.47	16.13 ± 0.01	8.30 ± 0.01	10.99 ± 0.01
	Fairphone 4	1.15 ± 0.01	8.23 ± 0.82	8.99 ± 0.01	6.05 ± 0.00	8.14 ± 0.05
postmarketOS	OnePlus 6	2.05 ± 0.18	1.59 ± 0.23	10.85 ± 0.82	6.45 ± 0.50	9.16 ± 0.70
	Pixel 3a	2.66 ± 0.18	2.19 ± 0.23	13.75 ± 0.83	8.23 ± 0.50	11.63 ± 0.70
	Fairphone 4	1.74 ± 0.01	1.49 ± 0.01	7.13 ± 0.02	5.93 ± 0.01	9.79 ± 0.04
Ubuntu Touch	OnePlus 6	2.83 ± 0.40	1.16 ± 0.27	20.54 ± 3.38	12.53 ± 1.66	17.83 ± 1.83
	Pixel 3a	1.98 ± 0.08	0.31 ± 0.03	24.98 ± 2.11	19.65 ± 0.37	17.00 ± 0.31
	Fairphone 4	1.24 ± 0.00	0.18 ± 0.01	7.11 ± 0.01	5.93 ± 0.00	10.58 ± 0.04
Mobian	OnePlus 6	1.86 ± 0.01	0.79 ± 0.05	10.52 ± 0.04	6.24 ± 0.01	8.85 ± 0.02
	Pixel 3a	2.51 ± 0.04	0.91 ± 0.04	14.46 ± 0.30	8.47 ± 0.05	12.58 ± 0.16
	Fairphone 4	1.71 ± 0.02	0.85 ± 0.02	7.14 ± 0.03	5.95 ± 0.02	9.80 ± 0.03

Disk I/O, Graphics, and Network Performance							
OS	**Device**	**Disk RD (s)**	**Disk WR (s)**	**Graphics 2D (s)**	**Graphics 3D (s)**	**Net Latency (ms)**	**Net Perf (MB/s)**
Android	OnePlus 6	2.47 ± 0.01	9.98 ± 0.05	15.40 ± 0.01	15.39 ± 0.02	6.30 ± 0.37	1839.80 ± 133.01
	Pixel 3a	6.28 ± 0.02	12.20 ± 0.26	15.64 ± 0.06	15.56 ± 0.03	3.14 ± 0.16	822.25 ± 51.60
	Fairphone 4	1.98 ± 0.01	5.18 ± 0.03	14.35 ± 0.03	14.32 ± 0.03	4.52 ± 0.18	1997.15 ± 74.35
postmarketOS	OnePlus 6	4.53 ± 0.60	9.64 ± 0.32	17.03 ± 0.10	16.89 ± 0.06	0.55 ± 0.14	2392.26 ± 354.01
	Pixel 3a	6.68 ± 0.61	10.71 ± 0.37	17.22 ± 0.15	17.14 ± 0.08	0.86 ± 0.12	1468.09 ± 82.55
	Fairphone 4	3.90 ± 0.00	4.76 ± 0.03	16.98 ± 0.06	17.45 ± 0.47	0.30 ± 0.01	2026.72 ± 50.08
Ubuntu Touch	OnePlus 6	2.85 ± 0.26	21.75 ± 2.91	12.21 ± 0.55	12.20 ± 0.52	1.61 ± 0.35	927.24 ± 482.98
	Pixel 3a	6.35 ± 0.10	26.79 ± 0.52	13.65 ± 0.78	11.52 ± 0.42	0.67 ± 0.03	877.47 ± 37.11
	Fairphone 4	1.93 ± 0.11	10.29 ± 0.14	14.43 ± 0.06	14.09 ± 0.13	0.23 ± 0.03	2761.97 ± 155.91
Mobian	OnePlus 6	2.87 ± 0.00	9.62 ± 0.18	17.31 ± 0.83	16.95 ± 0.19	0.45 ± 0.08	2950.69 ± 312.50
	Pixel 3a	6.95 ± 0.02	10.99 ± 0.10	16.71 ± 0.01	16.70 ± 0.01	0.41 ± 0.03	1794.95 ± 46.58
	Fairphone 4	1.96 ± 0.00	4.79 ± 0.03	16.69 ± 0.01	16.66 ± 0.01	0.21 ± 0.00	3040.94 ± 82.90

TABLE VI: Performance benchmarks for each operating system across devices

across every tested device and peaking at over 3 GB/s on the Fairphone 4. postmarketOS also significantly outperforms Android, particularly on legacy hardware like the Pixel 3a where it delivers nearly double the throughput. Ubuntu Touch presents highly variable results, struggling with lower throughput and high variance on the OnePlus 6 but scaling exceptionally well on the more modern Fairphone 4. Overall, the data demonstrates that stripping away heavy middleware allows mobile hardware to fully leverage the highly optimized Linux network protocols for superior data transmission speeds.

OS	**Device**	**Thermal (△°C)**	**Power (%)**
Android	OnePlus 6	3.62 ± 0.61	4.60 ± 0.37
	Pixel 3a	2.16 ± 0.76	2.50 ± 0.51
	Fairphone 4	3.68 ± 0.69	6.70 ± 1.12
postmarketOS	OnePlus 6	4.44 ± 0.61	4.00 ± 0.34
	Pixel 3a	3.32 ± 0.43	2.60 ± 0.60
	Fairphone 4	3.81 ± 0.49	9.50 ± 2.34
Ubuntu Touch	OnePlus 6	2.78 ± 0.65	3.00 ± 0.58
	Pixel 3a	3.04 ± 0.41	2.50 ± 0.38
	Fairphone 4	5.34 ± 0.18	5.70 ± 0.76
Mobian	OnePlus 6	4.77 ± 0.31	4.40 ± 0.37
	Pixel 3a	3.55 ± 0.41	3.40 ± 0.60
	Fairphone 4	-	-

TABLE VII: Efficiency benchmarks for each operating system across devices

5) Efficiency Benchmarks: The efficiency methodology includes a five-minute idle cooldown period. This is followed by ten minutes of intensive benchmarking to evaluate thermal and power performance. During these physical tests, all devices, without their protective cases, were placed on a table approximately 60 cm apart with no active airflow to ensure environmental consistency. All benchmarks were strictly executed with at least 30% battery capacity remaining. Thermal and power data for Mobian on the Fairphone 4 were unavailable at the time of testing due to lack of driver support and are consequently omitted from the table [46].

Table VII details the efficiency tests. Following the idle period, the thermal and power metrics reflect the hardware strain during the ten-minute intensive workload. Thermal output varied among the different operating systems and devices. On the OnePlus 6, Ubuntu Touch maintains a minimal temperature increase of 2.78 °C and consumes only 3.00% of the battery. In contrast, Android and Mobian push the exact same device to higher temperatures and consume over 4.00% of the battery. Power consumption shows even more significant variance on the Fairphone 4. Notably, postmarketOS on the Fairphone 4 drains 9.50% of the battery during the test, compared to Android consuming 6.70% and Ubuntu Touch consuming 5.70% on the same hardware. While baseline thermal dissipation is inherently constrained by the physical chassis design and internal materials of each specific phone, these paired fluctuations on identical hardware demonstrate that thermal

performance and power efficiency remain heavily dependent on software-level optimization and operating system power management.

C. MobiOSBench Score Results

To evaluate the impact of operating system architecture on identical hardware, the normalized pillar ratios and final MobiOSBench scores were calculated across the three distinct mobile devices. The framework applies a Neutral Baseline profile alongside three targeted user profiles to provide a mathematical control.

OnePlus 6	Android	postmarketOS	Ubuntu Touch	Mobian
Privacy	0.0558	0.1385	0.1595	0.1795
Resources	0.0264	0.0600	0.0445	0.0492
Performance	0.2890	0.4534	0.3242	0.5307
Efficiency	0.2451	0.2373	0.3463	0.2183
Daily Driver	145	221	207	227
Power User	149	252	204	271
Privacy Advocate	97	180	189	204
Neutral Baseline	101	173	168	179

TABLE VIII: OnePlus 6 pillar scores and MobiOSBench Scores

Table VIII presents the complete four-pillar results for the OnePlus 6 architecture. The operating systems rank in a clear hierarchical order under the Neutral Baseline and Daily Driver profiles. Mobian achieves the highest MobiOSBench scores, followed closely by postmarketOS, then Ubuntu Touch, with Android trailing significantly. Android suffers a severe and persistent penalty in the Privacy and Resources pillars. For instance, Android yields a Privacy Ratio of 0.0558 compared to Mobian's 0.1795. This mathematically reflects Android's heavy background data overhead, consistent with telemetry and ecosystem services.

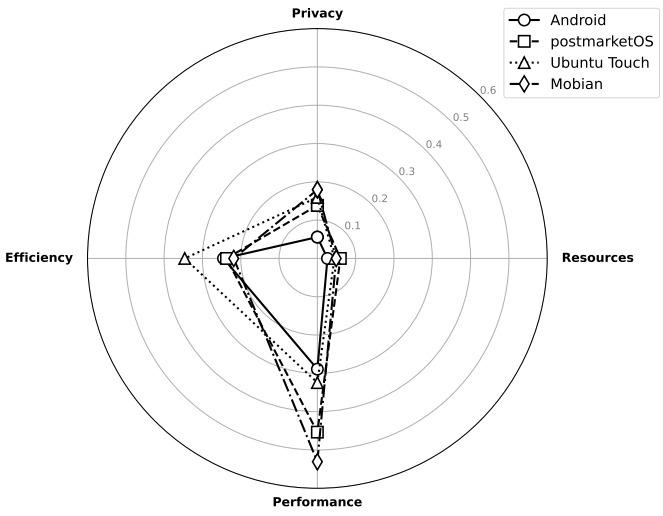


Fig. 3: Operating system pillar scores for the OnePlus 6

To visually synthesize these multidimensional results, Figure 3 illustrates the four-pillar performance of each operating system on the OnePlus 6 hardware. The radar chart clearly depicts Mobian dominating the Performance and Privacy axes, while Ubuntu Touch stretches furthest along the Efficiency axis. In contrast, the Android data polygon remains tightly constricted near the center of the chart across both the Privacy and Resources domains. This spatial representation allows for an immediate and intuitive understanding of the architectural trade-offs inherent to each operating system when deployed on identical hardware.

Pixel 3a	Android	postmarketOS	Ubuntu Touch	Mobian
Privacy	0.0783	0.1353	0.1565	0.1666
Resources	0.0315	0.0542	0.0434	0.0537
Performance	0.2440	0.3527	0.3502	0.4122
Efficiency	0.4303	0.3404	0.3627	0.2878
Daily Driver	171	218	215	225
Power User	153	220	214	242
Privacy Advocate	126	180	191	199
Neutral Baseline	127	172	171	181

TABLE IX: Pixel 3a pillar scores and MobiOSBench Scores

Table IX details the MobiOSBench evaluation on the Pixel 3a architecture and perfectly mirrors the hierarchical ranking of operating systems observed on the OnePlus 6. This consistent scaling validates that the baseline structural overhead of these operating systems remains largely static across different but mature hardware tiers. Android demonstrates a consistently lower Privacy Ratio of 0.0783 and a Resources Ratio of 0.0315. This further confirms the heavy architectural footprint of its managed runtime and persistent background services compared to native Linux-based operating systems such as Mobian.

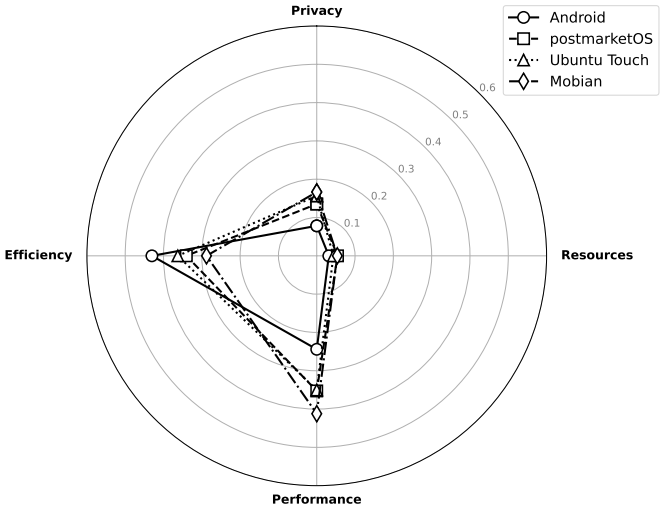


Fig. 4: Operating system pillar scores for the Pixel 3a

Transitioning to the Pixel 3a, Figure 4 provides the visual representation of the operating system performance across the four evaluation pillars. Similar to the previous device, the radar chart demonstrates Mobian maintaining a strong structural lead along the Performance and Privacy axes. However, a notable shift occurs within the Efficiency domain where Android stretches outward to claim the highest mathematical ratio. Despite this localized efficiency advantage, the Android data polygon remains severely restricted in the Privacy and Resources quadrants. This visualization reinforces the consistent architectural trade-offs of the platforms while highlighting how

specific hardware synergies can subtly alter individual pillar boundaries.

Fairphone 4	Android	postmarketOS	Ubuntu Touch
Privacy	0.0461	0.1427	0.1518
Resources	0.0275	0.0603	0.0439
Performance	0.4065	0.5402	0.7273
Efficiency	0.2014	0.1662	0.1813
Daily Driver	155	214	234
Power User	178	271	309
Privacy Advocate	91	176	190
Neutral Baseline	101	167	172

TABLE X: Fairphone 4 pillar scores and MobiOSBench Scores

Table X illustrates the standard four-pillar evaluation for the Fairphone 4 architecture. This device introduces a notable divergence from the established pattern due to asymmetric hardware optimization. Ubuntu Touch surges ahead in the Performance pillar by registering a high 0.7273 ratio. This fundamentally disrupts the hierarchy established on the older devices and suggests that Ubuntu Touch kernel drivers are more optimized for the Fairphone 4 chipset. The lack of functional battery driver support prevented the collection of Efficiency metrics for Mobian on this specific device configuration.

Fairphone 4	Android	postmarketOS	Ubuntu Touch	Mobian
Privacy	0.0461	0.1427	0.1518	0.1750
Resources	0.0275	0.0603	0.0439	0.0400
Performance	0.4065	0.5402	0.7273	0.6537
Daily Driver	117	214	229	217
Power User	162	271	307	286
Privacy Advocate	76	176	187	194
Neutral Baseline	80	167	169	166

TABLE XI: Fairphone 4 scores without the Efficiency pillar

Table XI resolves the missing Efficiency pillar by providing a mathematically reweighted three-pillar profile comparison. This ensures a fair and scaled evaluation across all four operating systems on the Fairphone 4. By omitting the Efficiency pillar and equally redistributing the index weights across the remaining three pillars, the framework successfully accommodates the missing pillar without invalidating the comparative baseline. Under this reweighted Neutral Baseline, the three non-Android operating systems perform almost identically, as the Performance lead of Ubuntu Touch is offset by pillar deficits elsewhere under equal weighting; the lead instead emerges under the Performance-weighted Power User profile, where Ubuntu Touch attains 307 against 286 for Mobian and 271 for postmarketOS.

Finally, Figure 5 visualizes the performance distribution for the Fairphone 4. As noted in the numerical analysis, Ubuntu Touch exhibits a massive outward spike along the Performance axis, completely eclipsing the other operating systems in raw computational throughput. Android continues its trend of severe restriction in the Privacy and Resources quadrants while maintaining the lead in Efficiency. Crucially, this visualization also highlights the hardware-specific compatibility limits discussed earlier. Because of the lack of kernel support for battery drivers, Efficiency for Mobian is truncated and lacks a vertex on that specific axis. This distinct visual gap serves as a reminder that alternative mobile deployments

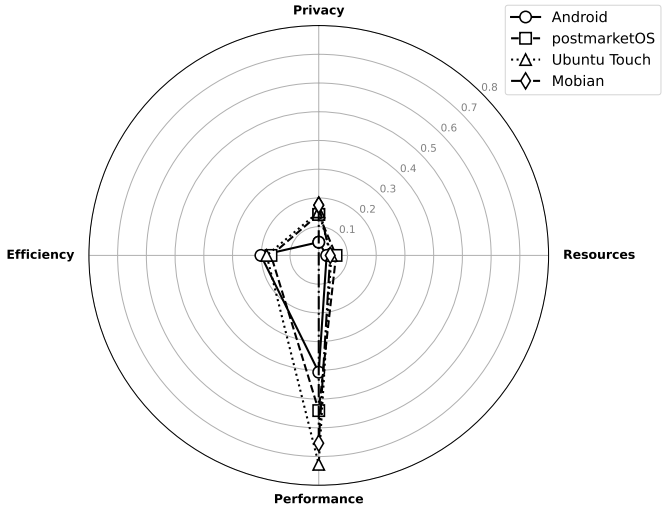


Fig. 5: Operating system pillar scores for the Fairphone 4

are still heavily constrained by device-specific driver support and kernel maturation.

The necessity of the MobiOSBench profiling system becomes evident when observing how final rankings shift based on mathematical weighting across all tested devices. For a standard user evaluating the Daily Driver profile, postmarketOS and Mobian remain highly competitive across the board. However, when the heavy weight is applied to the Privacy Advocate profile, the Android score collapses dramatically due to its background network footprint. On the standard Fairphone 4 evaluation, Android drops to a heavily penalized score of 91 while Ubuntu Touch maintains a robust 190.

This divergence effectively suggests that an operating system that is adequate for general compute can mathematically fail to meet the stringent requirements of a specialized user. A benchmark that yielded the same operating system ranking across all devices would be inherently blind to driver maturation and hardware-specific optimizations. The MobiOSBench framework successfully detects these nuances. Ultimately, the data indicates that while community-driven operating systems generally outperform Android in resource efficiency and privacy, raw computational throughput is strictly dictated by the specific software and hardware synergy of the deployment.

D. Framework Stability and Sensitivity Analysis

To empirically validate that the MobiOSBench scoring framework produces stable and reproducible conclusions under varying weight assignments, a One-At-A-Time (OAT) sensitivity analysis was conducted. OAT analysis systematically perturbs each input variable independently while holding others constant, isolating the individual contribution of each parameter to the output [42, 49, 50]. The OnePlus 6 was selected as a representative mid-range device spanning all four evaluated configurations. The Neutral Baseline profile, which assigns equal weight of 0.25 to all four pillars, was chosen as the analytical control due to its absence of domain-specific assumptions.

Each pillar weight was independently perturbed by ± 0.10, with the resulting delta equally redistributed among the remaining three weights to maintain a sum of exactly one across all scenarios. This produced eight perturbation scenarios alongside the base configuration, for nine total scenarios. The final MobiOSBench Score was recalculated for each operating system under each scenario, and systems were ranked ordinally from highest to lowest score.

Scenario	First	Second	Third	Fourth
Base	Mobian (179)	postmarketOS (173)	Ubuntu Touch (168)	Android (101)
Privacy +0.1	Mobian (179)	postmarketOS (168)	Ubuntu Touch (167)	Android (93)
Privacy -0.1	Mobian (179)	postmarketOS (178)	Ubuntu Touch (169)	Android (109)
Resources +0.1	Mobian (151)	postmarketOS (150)	Ubuntu Touch (141)	Android (85)
Resources -0.1	Mobian (212)	Ubuntu Touch (201)	postmarketOS (199)	Android (121)
Performance +0.1	Mobian (207)	postmarketOS (197)	Ubuntu Touch (183)	Android (116)
Performance -0.1	Mobian (155)	Ubuntu Touch (154)	postmarketOS (152)	Android (88)
Efficiency +0.1	Ubuntu Touch (185)	Mobian (184)	postmarketOS (180)	Android (114)
Efficiency -0.1	Mobian (174)	postmarketOS (166)	Ubuntu Touch (153)	Android (90)

TABLE XII: Sensitivity analysis (under weight perturbations)

The results shown in Table XII demonstrate strong framework stability across all nine scenarios. Android ranked last in every perturbation scenario without exception, confirming that its structural deficit in the Privacy and Resources pillars is robust to weight variation and is not an artifact of the specific weight configuration chosen. Minor positional variation was observed exclusively among the three non-Android operating systems, which exhibit far smaller score separations. Notably, Ubuntu Touch marginally surpassed Mobian under increased Efficiency weighting, directly reflecting its superior Efficiency pillar ratio of 0.3463 compared to 0.2183 for Mobian. This result validates that the framework correctly responds to genuine architectural differences rather than producing arbitrary rankings, as a system with a higher pillar score appropriately rises when that pillar receives greater weight. These findings confirm that MobiOSBench conclusions are robust to reasonable variations in user-defined weight assignments.

E. Framework Portability (Case Studies)

To demonstrate the versatility of MobiOSBench beyond controlled smartphone environments, exploratory case studies were conducted targeting domains highlighted by recent literature: Internet of Things (IoT) edge devices and closed mobile ecosystems.

1) Case Study A: IoT and Smart Appliances (Raspberry Pi): A Raspberry Pi 4 Model B was used in this case study. The Raspberry Pi uses a Quad-core Cortex-A72 CPU (4x1.8 GHz), an integrated Broadcom VideoCore VI GPU, 8 GB RAM, and 32 GB via microSD card (SDHC). MobiOSBench required no fundamental methodological modifications to adapt to the Raspberry Pi OS environment, demonstrating the inherent portability of the native execution methodology. This Raspberry Pi model has no battery, and as such, the Efficiency pillar is not recorded or used. Even though this case study focused only on the Privacy, Resources, and Performance pillars, the framework still ensures that the evaluation remains statistically valid and scientifically rigorous.

Benchmark	Metric	Values
Information	OS Version	Debian 13
	Kernel Version	6.12.47+rpt
	CPU Governor	ondemand
	Application Manager	pi-packages
	Permissions Model	User-based
Security & Privacy	Mandatory Access Control	None
	Encryption Status	Disabled
	Connectivity Services	Bluetooth
	Location Services	Disabled
	Network Data Usage TX (KB)	0.62 ± 0.07
	Network Data Usage RX (KB)	68.48 ± 6.47
Resource	Boot Time (s)	29.86 ± 2.28
	Total Schedulable Tasks	361 ± 5
	Launchable Application Set	23
	Storage Usage (GB)	6.78 ± 0.01
	Memory Usage (GB)	0.47 ± 0.02
Performance	`gettid` System Call (s)	4.43 ± 0.02
	`fork` System Call (s)	0.46 ± 0.01
	CPU (s)	11.69 ± 0.01
	Memory Read (s)	7.22 ± 0.01
	Memory Write (s)	17.95 ± 0.01
	Disk Read (s)	46.83 ± 0.01
	Disk Write (s)	98.24 ± 1.54
	Graphics 2D (s)	16.67 ± 0.01
	Graphics 3D (s)	16.64 ± 0.01
	Network Latency (ms)	0.19 ± 0.01
	Network Performance (MB/s)	1117.52 ± 23.35

TABLE XIII: MobiOSBench benchmarks for Raspberry Pi

The baseline telemetry detailed in Table XIII reveals a system characterized by a contrast between exceptional network capabilities and severe local storage bottlenecks. While the framework recorded network performance exceeding 1 GB/s with extremely low latency, local data operations struggled significantly. Memory write operations required nearly 18 s due to the slower memory type. Disk write operations exceeded 98 s, highlighting the extreme input and output limitations inherent to utilizing a microSD card as the primary storage medium. Despite these severe penalties in local write operations, the system maintained minimal background memory overhead, consuming only 0.47 GB of available capacity and exhibiting a very low background network transmission rate.

Pillar Scores	
Privacy	0.1806
Resources	0.0662
Performance	0.2956
MobiOSBench Scores	
Daily Driver	171
Power User	195
Privacy Advocate	174
Neutral Baseline	152

TABLE XIV: Raspberry Pi scores without the Efficiency pillar

Table XIV highlights the architectural compromises exposed by translating the raw metrics through the user profile weighting system. Despite the exceptionally low network latency and good throughput, the massive memory write and disk read and write penalties cap the Power User profile score at 195. Concurrently, the limited ambient network activity yields

a strong Privacy Advocate score of 174, demonstrating that edge devices can maintain good data sovereignty even when localized compute performance is degraded.

A fundamental advantage of MobiOSBench is its capacity for dynamic profile reconfiguration. While the Raspberry Pi achieves a relatively high Power User score driven primarily by the networking component and the `fork` system call, a targeted focus on localized compute capabilities reveals severe limitations. By calibrating a custom evaluation profile focused exclusively on CPU, memory, and disk operations, researchers can accurately isolate these hardware constraints. Under this strict local processing profile, Mobian on the Fairphone 4 emerges as the most capable of the baseline smartphone deployments, with a localized pillar score of 0.1913. In contrast, the Raspberry Pi registers a degraded score of 0.0428. This extreme delta not only quantifies the severe storage bottlenecks of the edge device but also highlights the intrinsic flexibility of MobiOSBench. MobiOSBench accurately profiles this architecture as highly optimized for localized network relay tasks and ambient data, rather than intensive local data processing, in contrast to the smartphones evaluated earlier. By empowering researchers to dynamically adjust pillar weights, the framework ensures that evaluations can be precisely tailored to reflect specific operational requirements rather than relying on rigid generalized metrics or vague benchmark-specific scores.

2) Case Study B: Closed Ecosystems (iOS): An iPhone 16 Pro was utilized for this evaluation, featuring a Hexa-core Apple A18 Pro CPU (2x4.04 GHz, 4x2.42 GHz), integrated Apple GPU, 8 GB RAM, and 256 GB of storage. Despite the restrictive iOS environment, MobiOSBench ran without fundamental modification, further evidence of the portability of its native execution methodology.

While the extreme sandboxing paradigm of iOS restricts access to specific system-level telemetry, this case study proves the framework can gracefully adapt to missing variables. MobiOSBench strictly evaluates production environments; therefore, rooting or bypassing native security models is deliberately prohibited. By evaluating the system exactly as deployed to the end user, the framework outputs highly accurate comparative profiles regardless of the ecosystem constraints.

Table XV details the baseline metrics for the iOS architecture. Because iOS is built upon the XNU kernel, the reported version reflects the Darwin subsystem rather than a mainline Linux deployment. Furthermore, the mandatory access control, App Sandbox, is fundamentally built upon the TrustedBSD architecture. MobiOSBench effectively captures the strict security posture of the operating system, verifying system parameters such as the proprietary governor and mandatory access control via compiler directives and high-level framework queries.

Due to the closed ecosystem, iOS restricts programmatic access to several metrics. Boot time and storage usage were therefore captured manually, and because XNU does not expose the `SYS_gettid` interface, the system call benchmark instead invokes the functionally equivalent `SYS_thread_selfid`, preserving the measurement of raw user-to-kernel context switching latency. Total Schedulable Tasks, thermal data, and the `fork` system call could not be measured in any form; rather than failing, MobiOSBench dynamically masks these obfuscated variables.

Benchmark	Metric	Values
Information	OS Version	iOS 26.3.1
	Kernel Version	25.3.0 (Darwin)
	CPU Governor	Proprietary
	Application Manager	Apple App Store
	Permissions Model	Per-app, per-service
Security & Privacy	Mandatory Access Control	App Sandbox
	Encryption Status	Enabled
	Connectivity Services	NFC, Bluetooth
	Location Services	Enabled
	Network Data Usage TX (KB)	14.38 ± 6.24
	Network Data Usage RX (KB)	68.76 ± 17.90
Resource	*Boot Time (s)*	*18.60 ± 0.18*
	Total Schedulable Tasks	-
	Launchable Application Set	*44*
	Storage Usage (GB)	*27.80*
	Memory Usage (GB)	5.93 ± 0.02
Performance	`gettid` System Call (s)	1.05 ± 0.00
	`fork` System Call (s)	-
	CPU (s)	4.17 ± 0.00
	Memory Read (s)	1.32 ± 0.02
	Memory Write (s)	1.19 ± 0.02
	Disk Read (s)	1.33 ± 0.00
	Disk Write (s)	1.36 ± 0.01
	Graphics 2D (s)	14.95 ± 0.21
	Graphics 3D (s)	16.67 ± 0.00
	Network Latency (ms)	0.40 ± 0.00
	Network Performance (MB/s)	5221.56 ± 91.15
Efficiency	Thermal Performance (Δ°C)	-
	Power Consumption (%)	6.70 ± 0.35

TABLE XV: MobiOSBench benchmarks for the iPhone 16 Pro

The metrics highlight a massive storage footprint of 27.80 GB for the base system and pre-installed applications, and background memory usage of 5.93 GB, reflecting the heavy resource allocation required to maintain local artificial intelligence services, system responsiveness, application and system caches, and other system features. Furthermore, the framework recorded exceptional performance metrics, including a 5221 MB/s network throughput and outstanding read and write operations.

Pillar Scores	
Privacy	0.0632
Resources	0.0522
Performance	1.1552
MobiOSBench Scores	
Daily Driver	255
Power User	381
Privacy Advocate	133
Neutral Baseline	156

TABLE XVI: iPhone 16 Pro scores without the Efficiency pillar

Table XVI highlights the architectural priorities of the Apple ecosystem, revealed by translating the raw metrics through the user profile weighting system. To maintain mathematical parity due to the restricted thermal telemetry, the Efficiency pillar was cleanly omitted from the final scoring. Despite the system's unparalleled performance, its heavy baseline storage utilization and ambient network data keep the Privacy Advocate and

Neutral Baseline profiles at levels comparable to previously evaluated systems.

The Power User profile, however, achieves an exceptional score of 381. Relying on the dynamic reconfiguration capabilities of MobiOSBench, researchers can directly compare this closed ecosystem to the highest-performing open architecture from the baseline study, Ubuntu Touch on the Fairphone 4. The comparison is placed on equal footing by masking the `fork` system call and Total Schedulable Tasks and by excluding the Efficiency pillar on both systems; each of these restrictions originates from the telemetry limits of iOS and is mirrored on Ubuntu Touch solely to preserve parity. Under these matched conditions, Ubuntu Touch achieves a Power User score of 346. The delta between these scores primarily reflects the significant generational and tier disparity between the modern flagship Apple silicon and the older mid-range hardware of the baseline devices. This contrast simultaneously demonstrates that MobiOSBench scales effectively to capture maximum computational throughput and accurately profile a vertically integrated ecosystem while preserving the underlying mathematical framework.

The iOS operating system heavily prioritizes security and maximum performance, trading lightweight resource efficiency for a premium user experience. This case study successfully proves that MobiOSBench can evaluate deeply restricted architectures without compromising its strict comparative methodology.

F. Comparison with Existing Benchmarking Frameworks

Table XVII compares MobiOSBench against existing benchmarking frameworks and tools across eight dimensions. Mobile OS Comparison indicates whether the tool is explicitly designed to evaluate fundamentally different mobile operating system architectures on identical physical hardware, rather than simply executing across multiple platforms. Identical Hardware indicates whether the evaluation methodology itself holds physical silicon constant across comparisons. Mobile-Specific indicates whether the tool targets mobile device characteristics rather than general-purpose computing. The remaining columns reflect open-source availability and whether the tool incorporates a dedicated Privacy pillar, Resources pillar, Efficiency pillar, and configurable user-weighted scoring profiles.

The comparison reveals a clear methodological gap. General-purpose tools such as lmbench, SPEC, Sysbench, Phoronix, PassMark, AnTuTu, and Geekbench execute workloads natively but are designed to characterize hardware performance rather than isolate operating system behavior, and none incorporate dedicated evaluation pillars or configurable profiles. Phoronix partially supports identical-hardware comparison: it is routinely used to compare desktop and server operating systems on the same machine, though hardware constancy is supplied by the operator rather than enforced by the methodology. Cider [2] holds hardware constant and explores operating system-level performance differences, mainly leveraging PassMark as its benchmarking instrument, but its scope is confined to the Android-iOS boundary and omits privacy, resource, and efficiency evaluation. AnROM [13] advances the methodology within the Android ecosystem, incorporating some battery, memory, and privacy-related checks, but remains confined to a single operating system family without cross-platform comparison or configurable scoring. MobiOSBench is the only tool to simultaneously satisfy all eight dimensions, extending the fixed-hardware evaluation paradigm across architecturally divergent mobile operating systems while providing a mathematically configurable scoring framework tailored to diverse developer and user priorities.

VI. Discussion

The deployment of MobiOSBench across identical hardware successfully isolated software execution from physical device variability, revealing critical insights into how disparate operating systems manage resources, enforce privacy, and handle compute tasks.

A. Privacy and the Full-Stack Resource Tax

MobiOSBench evaluates the full native operating system stack as provided to end users, including runtimes like the Android Runtime and glibc. This approach exposes a severe, consistent penalty for Android in the Privacy and Resources pillars. Driven by persistent background telemetry, heavy ecosystem integration, and massive memory footprints, Android registered the lowest ratios across all devices. Conversely, native Linux-based systems like Mobian and postmarketOS lack mandatory telemetry, maintaining extremely lean idle states that significantly reduce unnecessary CPU cycles and memory consumption.

B. Performance Dynamics, Driver Maturity, and Hardware Synergy

A broader methodological consideration involves the relationship between operating system efficiency and driver maturity. The framework evaluates each operating system as delivered to end users on target hardware, treating driver support as an integral component of the operating system deployment rather than an external confound. Driver support is not an external dependency imposed on an operating system project, but rather a deliberate engineering and resource allocation decision made by operating system maintainers for each supported hardware target. An operating system that ships with immature or incomplete drivers for a given device is making a deployment decision that directly shapes the user experience on that device. MobiOSBench evaluates this deployment reality rather than a hypothetical operating system kernel evaluated independently of its hardware integration layer, an approach that does not reflect what users, developers, or researchers encounter in practice. Rather than treating driver immaturity as noise to be filtered out, MobiOSBench surfaces it as actionable diagnostic information.

While the lean architecture of Mobian dominates execution metrics on the OnePlus 6 and Pixel 3a, the performance characteristics observed with Ubuntu Touch on the Fairphone 4 show how driver optimization appears to contribute to significant computational gains on that specific hardware target.

Framework	Mobile OS Comparison	Identical Hardware	Mobile-Specific	Open-Source	Privacy Pillar	Resources Pillar	Efficiency Pillar	Config. Profiles
lmbench [5]	×	×	×	✓	×	×	×	×
SPEC [6]	×	×	×	×	×	×	×	×
Sysbench [7]	×	×	×	✓	×	×	×	×
Phoronix [8]	×	∼	×	✓	×	×	×	×
Geekbench [9]	×	×	×	×	×	×	×	×
AnTuTu [10]	×	×	×	×	×	×	×	×
PassMark [11]	×	×	×	×	×	×	×	×
Cider [2]	∼	✓	✓	×	×	×	×	×
AnROM [13]	×	✓	✓	✓	∼	∼	∼	×
MobiOSBench	✓	✓	✓	✓	✓	✓	✓	✓

TABLE XVII: MobiOSBench comparison with benchmarking tools. ✓ = supported, ∼ = partially supported, × = not supported

This illustrates precisely the kind of insight that enables operating system developers to focus engineering effort where it produces the greatest real-world impact, rather than evaluating their systems in isolation from the hardware they ship on.

C. Electronic Waste and Hardware Longevity

MobiOSBench yields a compelling outcome for hardware sustainability: aging smartphones are rarely bottlenecked by physical silicon, but rather by the accumulating resource overhead of commercial operating systems. Deploying Mobian or postmarketOS on older devices like the OnePlus 6 yields higher compute performance and lower battery consumption than native Android. This mathematically validates lightweight operating systems as a viable, eco-friendly pathway to extend device lifespans and combat electronic waste.

D. Implications for Users and Developers

The weighting profiles prove that no universally superior mobile operating system exists. While Android remains highly capable for mainstream applications, its score drops sharply under the Privacy Advocate profile. Transitioning to alternative operating systems significantly improves resource economy and ambient privacy, provided users align their operating system choice with hardware compatibility. For example, Ubuntu Touch is better suited for newer hardware, while Mobian is preferable for aging devices.

E. Framework Adaptability and Extensibility

The modular architecture of MobiOSBench is designed to accommodate additional workload categories without restructuring the existing benchmark suite or scoring methodology. This extensibility is already demonstrated in the evaluation, where missing efficiency metrics for Mobian on the Fairphone 4 were handled by reweighting the remaining pillars without invalidating cross-operating system comparisons. This same architectural property enables future work targeting operating system-level hardware abstraction layer tests such as media pipeline latency, camera subsystem throughput, and audio buffer management. Unlike application-level workloads, these tests evaluate specific, well-defined operating system subsystems through standardized hardware interfaces, preserving the isolation methodology that underpins MobiOSBench while providing additional behavioral context. Such extensions represent a natural next step that the framework's design actively enables rather than a modification of its core methodology.

This modular design is not merely a future consideration but a present necessity, as the evaluated platforms already exhibit significant hardware and feature heterogeneity. Not all operating systems support identical hardware interfaces across all devices, as evidenced by the absence of NFC support in postmarketOS on the Fairphone 4 and the lack of battery instrumentation on the Raspberry Pi deployment. MobiOSBench handles this heterogeneity through the same reweighting mechanism that would accommodate future workload extensions, ensuring that cross-platform comparisons remain valid regardless of which specific metrics are available on a given deployment target.

F. Threats to Validity

The primary internal threat involves driver-level maturity, which manifests in two distinct forms. The first is a potential measurement accuracy concern. Because alternative operating systems rely on community-developed drivers, it cannot be fully ruled out that hardware sensor reporting accuracy varies across operating systems, particularly for efficiency metrics such as thermal states. This represents a theoretical source of variance rather than a confirmed measurement error. Known driver-related anomalies, such as the boot delay on Mobian's Fairphone 4 configuration and the missing efficiency metrics on the same device, are explicitly identified and disclosed throughout the evaluation. The second is a scope consideration. Results reflect the operating system as deployed on target hardware, including its driver support layer, rather than an isolated operating system kernel. This is an intentional design decision, as MobiOSBench evaluates deployment reality rather than a theoretical hardware-independent operating system abstraction. Externally, the study is bounded by the three selected devices, and ongoing mainline kernel development could alter future performance ratios. Finally, the Security Risk Index relies on subjective demerit assignment, which may not perfectly map to production threat susceptibility.

VII. Conclusion

Evaluating mobile operating systems requires isolating software architecture from hardware capabilities to prevent the

conflation of physical silicon performance with operating system efficiency. This research addresses this gap through MobiOSBench, a standardized and flexible evaluation framework that measures the full native software stack by executing workloads directly through native interfaces while holding physical hardware variables constant. Moving beyond one-dimensional scores, MobiOSBench evaluates Privacy, Resources, Performance, and Efficiency, utilizing mathematically weighted user profiles to translate raw telemetry into actionable insights. Extensive evaluations of four distinct mobile operating systems systematically deployed across three different physical devices, alongside targeted case studies on iOS and a Raspberry Pi, validate this approach. The results confirm that MobiOSBench reliably exposes the systemic trade-offs unique to each architecture, proving that software design profoundly impacts device capabilities independently of the silicon. A sensitivity analysis further confirms that these conclusions are robust to reasonable variation in user-defined weight assignments, and a systematic comparison against existing benchmarking tools establishes the framework's novelty relative to prior work. Ultimately, MobiOSBench provides a mathematically rigorous methodology for cross-architecture comparisons, offering a quantifiable pathway to optimize resource allocation, extend the operational lifespan of aging mobile hardware, and foster a more sustainable software ecosystem.

CRediT authorship contribution statement

Naser AlDuaij: Writing - review & editing, Writing - original draft, Visualization, Validation, Software, Methodology, Investigation, Data curation, Conceptualization, Project administration.

Declaration of competing interest

The author declares that he has no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The MobiOSBench benchmarking suite and raw research data are publicly available at [51]. The repository includes platform-specific benchmark implementations for all evaluated operating systems, raw telemetry from all benchmark runs, and confidence interval calculations for every device and operating system configuration.

ACKNOWLEDGMENTS

The author would like to thank Kuwait University and its College of Science for their support and facilities. The author used Claude to proofread this paper.

REFERENCES

[1] Motorola. Motorola announces a partnership with GrapheneOS Foundation. https://motorolanews.com/motorola-three-new-b2b-solutions-at-mwc-2026.

[2] J. Andrus, A. Van't Hof, N. AlDuaij, C. Dall, N. Viennot, and J. Nieh, "Cider: Native Execution of iOS Apps on Android," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, Salt Lake City, UT, Mar. 2014, pp. 367–382.

[3] N. AlDuaij, A. Van't Hof, and J. Nieh, "Heterogeneous Multi-Mobile Computing," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '19, Seoul, Republic of Korea, Jun. 2019, pp. 494–507.

[4] Linux Code. Linux Based Operating Systems for Mobile and Tablet Devices. https://thelinuxcode.com/linux_based_operating_systems_mobile_tablet_devices.

[5] L. McVoy and C. Staelin, "lmbench: portable tools for performance analysis," in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ser. ATC '96, San Diego, CA, USA, Jan. 1996, pp. 23–38.

[6] Standard Performance Evaluation Corporation (SPEC). Welcome to SPEC. https://www.spec.org.

[7] akopytov. sysbench. https://github.com/akopytov/sysbench.

[8] Phoronix Media. Phoronix Test Suite. https://phoronix-test-suite.com.

[9] Primate Labs Inc. Geekbench 6 - Cross-Platform Benchmark. https://www.geekbench.com.

[10] Beijing Antutu Technology Company Limited. AnTuTu Benchmark: Performance Testing. https://www.antutu.com/web.

[11] PassMark Software. PassMark Software. https://passmark.com.

[12] F. Khomh, H. Yuan, and Y. Zou, "Adapting Linux for mobile platforms: An empirical study of Android," in *2012 28th IEEE International Conference on Software Maintenance*, ser. ICSM '12, Trento, Italy, Sep. 2012, pp. 629–632.

[13] N. AlDuaij, "AnROM: A Methodology and Comparative Study of Custom Android Systems," *Baltic Journal of Modern Computing*, vol. 13, no. 4, pp. 894–918, 2025.

[14] K. Sareen, S. M. Blackburn, S. S. Hamouda, and L. Gidra, "Memory Management on Mobile Devices," in *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM '24, Copenhagen, Denmark, Jun. 2024, pp. 15–29.

[15] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "MLPerf inference benchmark," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20, Virtual Event, May 2020, pp. 446–459.

[16] V. Janapa Reddi, D. Kanter, P. Mattson, J. Duke, T. Nguyen, R. Chukka, K. Shiring, K.-S. Tan,

M. Charlebois, W. Chou, M. El-Khamy, J. Hong, T. St John, C. Trinh, M. Buch, M. Mazumder, R. Markovic, T. Atta, F. Cakir, M. Charkhabi, X. Chen, C.-M. Chiang, D. Dexter, T. Heo, G. Schmuelling, M. Shabani, and D. Zika, "MLPerf Mobile Inference Benchmark: An Industry-Standard Open-Source Machine Learning Benchmark for On-Device AI," in *Proceedings of 5th Machine Learning and Systems Conference*, ser. MLSys '22, Santa Clara, CA, USA, Aug. 2022, pp. 352–369.

[17] Apple Inc. iOS 26 Preview. https://www.apple.com/os/ios.

[18] Huawei. HarmonyOS NEXT. https://consumer.huawei.com/cn/harmonyos-next.

[19] Google LLC. Android — Do More With Google on Android Phones & Devices. https://www.android.com.

[20] Red Hat, Inc. Mobility - Fedora Project Wiki. https://fedoraproject.org/wiki/Mobility.

[21] Kai OS Technologies. Internet for everyone. KaiOS brings the best of smartphones to affordable devices. https://www.kaiostech.com.

[22] WebOS Ports. WebOS-Ports (LuneOS). https://webos-ports.org/wiki/Main_Page.

[23] Maemo Leste. A FOSS Linux Distribution for Mobile Devices. https://maemo-leste.github.io.

[24] Manjaro GmbH & Co. KG. Manjaro-ARM. https://wiki.manjaro.org/index.php/Manjaro-ARM.

[25] Mobian. mobian. https://mobian-project.org.

[26] Nemo Mobile. NemoMobile Bringing freedom to mobile. https://nemomobile.net.

[27] postmarketOS. The Linux distribution for mobile devices and beyond... https://postmarketos.org.

[28] Purism. PureOS. https://pureos.net.

[29] Jolla. Sailfish OS is a Linux-based European alternative to dominating mobile operating systems. https://sailfishos.org.

[30] UBports. A whole mobile operating system experience, that is yours. https://www.ubuntu-touch.io.

[31] Apple Inc. xnu. https://github.com/apple-oss-distributions/xnu.

[32] The Educative Team, "OS Design: Monolithic vs. microkernel architecture," https://learningdaily.dev/os-design-monolithic-vs-microkernel-architecture-78981dd41c49, Oct. 2024.

[33] Caffeine, "The ultimate Linux phone guide for consumers," https://medium.com/@camden.o.b/the-ultimate-linux-phone-guide-28d0cce9929, Jul. 2023.

[34] Halium Project. Halium. https://halium.org.

[35] Mer, et al. libhybris. https://github.com/libhybris/libhybris.

[36] Linux Kernel Organization. Linux CPUFreq Governors. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[37] Palo Alto Networks, Inc. ICMPv6 Rate Limiting. https://docs.paloaltonetworks.com/pan-os/11-0/pan-os-networking-admin/session-settings-and-timeouts/icmp/icmpv6-rate-limiting.

[38] N. AlDuaij, "DepAPI: The performance impact of the evolution of APIs," *Journal of Computer Languages*, vol. 85, p. 101353, Nov. 2025.

[39] K. M. Kim, Y. S. Jeong, and I. C. Bang, "Thermal analysis of lithium ion battery-equipped smartphone explosions," *Engineering Science and Technology, an International Journal*, vol. 22, no. 2, pp. 610–617, Apr. 2019.

[40] Google LLC. Thermal mitigation. https://source.android.com/docs/core/power/thermal-mitigation.

[41] J. L. Hennessy, D. A. Patterson, and C. Kozyrakis, *Computer Architecture: A Quantitative Approach*, 7th ed. Oxford, England: Morgan Kaufmann, Oct. 2025.

[42] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: the correct way to summarize benchmark results," *Communications of the ACM*, vol. 29, no. 3, pp. 218–221, Mar. 1986.

[43] H. Kang and W. Shin, "Do Smartphone Power Users Protect Mobile Privacy Better than Nonpower Users? Exploring Power Usage as a Factor in Mobile Privacy Protection and Disclosure," *Cyberpsychology, Behavior, and Social Networking*, vol. 19, no. 3, pp. 179–185, Jan. 2016.

[44] S. Welber, V. Zhao, C. Dolin, O. Morkved, H. Hoffmann, and B. Ur, "Do Users Have Contextual Preferences for Smartphone Power Management?" in *Proceedings of the 29th ACM Conference on User Modeling, Adaptation and Personalization*, ser. UMAP '21, Jun. 2021, pp. 44–54.

[45] postmarketOS Wiki. Fairphone 4 (fairphone-fp4). https://wiki.postmarketos.org/wiki/Fairphone_4_(fairphone-fp4).

[46] InstallingDebianOn. Fairphone 4. https://wiki.debian.org/InstallingDebianOn/Fairphone/Fairphone4.

[47] Tulie Finley-Moise, "VSync: What It Is and When to Use It for Better Gaming," https://www.hp.com/us-en/shop/tech-takes/vsync-should-i-turn-it-on-or-off, Aug. 2024.

[48] Y. Wu, D. Du, C. Xu, Y. Xia, M. Fu, B. Zang, and H. Chen, "D-VSync: Decoupled Rendering and Displaying for Smartphone Graphics," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25, Rotterdam, Netherlands, Mar. 2025, pp. 326–341.

[49] M. D. Morris, "Factorial Sampling Plans for Preliminary Computational Experiments," *Technometrics*, vol. 33, no. 2, pp. 161–174, May 1991.

[50] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 20–34, Jan. 2006.

[51] N. AlDuaij, "The MobiOSBench Benchmarking Suite and Raw Research Data," Jul. 2026. [Online]. Available: https://doi.org/10.5281/zenodo.21347610

BIOGRAPHY SECTION

Naser AlDuaij is an Assistant Professor of Computer Science at Kuwait University and a consultant. He was previously a Visiting Scholar in CSAIL at the Massachusetts Institute of Technology. He received two B.S.E. degrees in

Computer Science and Computer Engineering, with Mathematics and Physics minors, from the University of Michigan, Ann Arbor. He holds an M.S. degree from Yale University, and M.S., M.Phil., and Ph.D. degrees in Computer Science from Columbia University. Prior to his graduate studies, he worked as a software engineer for VMware Inc. His research interests include the Internet of Things, operating systems, security, mobile and ubiquitous computing, virtualization, and embedded systems.