Scaling TrustChain to One Million Blocks on Mobile Devices

Storage Performance Evaluation and Benchmarking

Michiel Bakker

Supervisor(s): Johan Pouwelse, Bulat Nasrulin

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Michiel Bakker
Final project course: CSE3000 Research Project
Thesis committee: Johan Pouwelse, Bulat Nasrulin, Koen Langendoen

Abstract

Smartphones offer limited storage and memory, constraints that conventional blockchains struggle to meet, yet they are also the devices where user-owned transaction chains promise the most value. We present the first empirical study of **TrustChain**, a DAG-based per-peer ledger running entirely on mobile hardware. A Rust implementation and open-source benchmark evaluate how flush-interval batching (k) and lossless compression affect a single node scaling from 10^3 to 10^6 blocks.

On a Galaxy S8 and Pixel-6 emulator, RAM stays below 600 MB and compressed disk use below 0.5 GB at one million blocks (128 B payload). Insert latency remains interactive (< 8 ms) with `disk:100` flushing; moderate batching ($k \approx 500$) cuts CPU load by roughly 45 % without harming durability. Lightweight compression (LZ4-1, Zstd-1) trims space by 20 to 30 % at a sub-10 ms cost, with diminishing returns at higher levels. End-to-end tests show storage is never the bottleneck, raw UDP achieving a 7 ms median RTT.

Taken together, the implementation and measurement dataset provide a concrete reference for deploying DAG-based chains on smartphones and highlight opportunities for advancing mobile blockchain technology.

1 Introduction

Project context and team

This study forms one-fifth of a coordinated BSc research project carried out by **five TU Delft students**. Together we built a shared TrustChain code base. After this foundation was in place, each member investigated a separate performance aspect: **storage** (this paper), **latency**, **throughput**, **energy efficiency** and **robustness**. The five papers together give a complete view of running TrustChain on smartphones, and all metrics are directly comparable thanks to the common setup.

Why focus on *mobile* chains?

Smartphones are the most widely used, but also the most resource-constrained, networked devices in daily life. Even a mid-range phone offers only a few gigabytes of available storage, limited RAM, and storage systems whose random-write throughput rarely exceeds a few hundred IOPS [1]. Traditional blockchains, designed for workstation-class disks, perform poorly under these constraints: syncing Bitcoin's ≈500 GB history to a micro-SD card may take days, and the main bottleneck is storage I/O, not cryptography or bandwidth [2].

Why TrustChain on mobile?

DAG-based systems such as **TrustChain** [3] eliminate global consensus: each participant maintains a personal chain of signed blocks that can be verified pairwise. Server-grade studies show that DAG architectures can achieve much higher throughput than linear blockchains [4], and prior work such as Gromit [5] benchmarks blockchain scalability on general-purpose hardware. Yet despite these efforts, no public data exist on how DAG-based systems behave in mobile environments. Because limited space and random-write performance are key constraints on mobile, **storage behavior** is a critical aspect to understand before exploring networking or energy consumption.

We adopt a purely experimental methodology that is well-suited to a benchmarking study and implement TrustChain from scratch in Rust. Two independent test harnesses form the backbone of the evaluation:

1. **CLI Storage Harness:** inserts generated blocks without any network or UI layer and records timing, memory, and disk metrics.
2. **UI Round-Trip Test:** runs entirely on the phone, sending in-memory messages over raw UDP, QUIC (Iroh), or TFTP to place storage costs in a live network context.

All experiments run on a Samsung Galaxy S8 (ext4, physical device) and a Pixel-6 emulator (F2FS). The workload spans 10^3–10^6 blocks, four payload sizes (128–4096 B), five flush strategies (`memory`, `disk:`k with $k \in \{50, 100, 500, 1000\}$) and four compression levels (none, LZ4, Zstd 1/3/9).

Contributions

- **Mobile TrustChain implementation in Rust** An open-source implementation that runs unchanged on desktop, Android, and is ready for iOS embedding via the same JNI/FFI boundary.
- **Reproducible benchmarking toolkit** Two open-source harnesses (CLI runner and in-app round-trip interface) plus supporting automation tools for easy deployment on real devices or emulators via a single `cargo ndk/adb` command.
- **First empirical study of mobile TrustChain storage** A systematic analysis of how flush-interval and compression affect latency, CPU usage, and storage footprint from 10^3 to 10^6 blocks, offering practical insight for future mobile DAG-based platforms.

Paper outline

Section 1 gives project context and motivation. Section 2 formulates the mobile storage challenge and research question. Section 3 describes the TrustChain Rust core, JNI integration and storage model. Section 4 presents the CLI/UI test harnesses and reports results on scaling, batching, compression and end-to-end RTT. Section 5 interprets the findings and draws conclusions. Section 6 outlines future research directions.

2 Background and Problem Statement

Motivation

TrustChain's DAG design removes the need for global consensus, giving each peer an *append-only* chain that can be verified one-on-one [3]. Server-grade benchmarks show that such DAGs reach much higher throughput than traditional

blockchains because parallel chains avoid the miner bottleneck [4]. On smartphones, however, the main bottleneck shifts from consensus to *persistent storage*: most phones still only support a few hundred random writes per second on ext4 or F2FS storage [1]. To avoid excessive write amplification and flash wear, techniques like batching and compression become essential. A recent survey of blockchain storage methods highlights *flush interval* and *compression*, as the most effective ways to reduce write latency and storage use on constrained devices [6].

Key design parameters

- **Flush interval** k. Writing each block with an `fsync()` guarantees durability but takes $1\,\mathrm{ms}$ to $20\,\mathrm{ms}$ per call on mobile storage. Batching $k \gg 1$ blocks spreads out the cost but risks losing the last few blocks in a crash.
- **Compression level / codec**. Lightweight methods like LZ4 use very little CPU but only reduce size moderately, while Zstandard at level 9 can shrink usage significantly but requires much more processing. Finding a *balanced* trade-off between speed and space on mobile hardware remains open.

Scope and assumptions

1. **Node role**. We study a *full but non-archival* TrustChain node: it checks and stores every received block, builds local indexes, and writes to on-device storage. It doesn't serve historical data to others.
2. **Workload**. Blocks with payloads are added at a steady rate until reaching 10^6 blocks. This simulates a write-heavy pattern, worst case for mobile flash.
3. **Metrics**. We track (i) average *insert time* (wall-clock), (ii) *CPU time* per block, and (iii) final *storage usage*. Peak RAM is measured to ensure we stay within a 4 GB memory limit, though reducing RAM is not the main focus.

Problem statement

What performance and storage trade-offs arise from batching (flush interval k) and compression when a mobile TrustChain node grows from 10^3 to 10^6 blocks?

Hypotheses

H1: A modern smartphone can accommodate a *TrustChain* of one million blocks without exhausting its local storage.

H2: Applying lossless compression significantly reduces the on-disk footprint of the chain.

H3: Using lightweight compression together with moderate batching keeps average insert latency within interactive bounds for end-users.

The rest of the paper tests these hypotheses using the experimental setup described next in Section 3.

3 Design and Implementation

3.1 System Overview

TrustChain is implemented as a modular, cross-platform system composed of a Rust-based protocol core and an Android-facing Kotlin UI. The two layers communicate through a lightweight JNI interface, which lets the core handle cryptographic and validation tasks natively, while exposing only a minimal set of functions to the Java layer. This split keeps mobile-specific logic separate from core processing, and lets the same Rust library run unchanged on both desktop and Android. All performance-critical logic: block validation, hashing, signing and state updates runs in Rust for efficiency and portability. Rust's ownership model enforces memory-safety without a garbage collector while still matching C/C++ throughput in systems benchmarks [7; 8].

3.2 Rust Core Architecture

The Rust core is organized into a small number of modules, each with its own role:

- `model`: Defines the `Block` type and implements (de)serialization using Serde.
- `state`: Manages global chain state using `RwLock`-protected maps, including block histories, deduplication caches, and peer-specific indexes.
- `common`: Contains the public API used by JNI, including block proposal creation, validation logic, and error handling.
- `util`: Implements cryptographic operations such as SHA-256 hashing, Ed25519 signatures, and zero-copy helpers.

Thread safety is ensured through `RwLock`, and global state is initialized lazily with `OnceLock` and `Lazy`. For example, the deduplication cache uses a 30-second timeout keyed by message content, while peer indexing guarantees blocks are handled in proper sequence.

3.3 Data Model and Storage

Each TrustChain block includes:

- A payload (application-level message),
- Sender and receiver Ed25519 public keys,
- Corresponding digital signatures,
- A hash of the previous block,
- A monotonic index.

The system supports two storage modes:

- **In-memory:** Blocks are stored in a `HashMap<String, Vec<Block>>`, keyed by public key.
- **Flush-to-disk:** Finalized blocks are saved as human-readable JSON files and written to disk in batches. Filenames are based on block hashes.

Compression algorithms such as Zstd and LZ4 are only applied during benchmarking and are not part of the main runtime logic.

3.4 Cross-Language Integration via JNI

To support Android, the Rust core makes selected functions available through a compact JNI interface:

- `tcCreateProposal`: Builds a signed block proposal.
- `tcProcessBlock`: Validates and stores a received block.
- `tcGetMyPubkey`: Returns the local Ed25519 public key.

JNI functions handle string conversion and error translation:

- Rust `Result<T, E>` types are converted to strings or codes that JNI can handle.
- Strings from Kotlin are validated as UTF-8 before use.
- Memory safety and thread safety are handled through Rust's type system.

JNI usage is kept minimal: once data is inside Rust, no further JNI calls are needed during processing. This avoids interference with garbage collection and helps the system stay efficient across platforms.

3.5 Transport Layer Abstraction

TrustChain separates networking from consensus logic by defining a shared transport interface. All supported protocols implement the same JNI-facing methods, so they can be swapped in or out without affecting the rest of the system.

Three transports are currently available:

- **Raw UDP:** A simple transport implemented in Kotlin using `DatagramSocket`. It uses a default port, retries messages up to three times with short delays, and supports a timeout for delivery. This mode is used for local and low-latency testing.
- **QUIC (via Iroh):** A Rust-based encrypted peer-to-peer transport built on the Iroh library. QUIC is a modern protocol, recently standardized, offering reliable streams over UDP with built-in encryption. Iroh handles discovery and connection management internally, making it well suited for high-latency or secure communication.
- **TFTP:** A file-based Rust transport that moves blocks via shared files. It works in setups where direct networking isn't available. Although not used in live tests, it functions correctly in limited or offline environments.

All transports implement the same API (e.g., `tcSendMessage`, `tcReceiveMessage`) and can be tested independently of the rest of the system. This makes it easy to compare different network setups in Section 4.

3.6 Portability and Extensibility

The Rust core is compiled to native Android libraries using `cargo-ndk`, and works across common mobile platforms (`arm64-v8a`, `x86_64`). The same codebase builds on desktop with no changes. Although the architecture supports future iOS integration via Swift FFI, that path hasn't been implemented yet.

Optional components are enabled using Cargo feature flags, which let developers include or exclude things like disk storage, benchmarking tools, or compression logic. These additions (e.g., performance logging or config switches) are only used in testing setups, not during normal app use, keeping the runtime lean and efficient.

4 Evaluation Setup and Results

4.1 Benchmarking Framework & Devices

All experiments were performed using a Samsung Galaxy S8 (Android 9, arm64) or a Pixel-6 "gphone64" emulator (Android 16). The Rust core is cross-compiled for `aarch64-linux-android` and included in a lightweight Kotlin UI; desktop builds run the same crate unchanged. Two separate test harnesses are used:

(A) Isolated Storage CLI A Rust binary (*∼600 lines*) runs the storage layer without network or UI. Each run:

(1) generates dummy double-signed blocks,

(2) saves them using the chosen storage mode, and

(3) records wall/CPU time, peak memory usage, and disk statistics to CSV/JSON.

Configurable parameters.

- **Blocks** N: 10^3–10^6
- **Payload**: 128–4096 B
- **Mode**: `memory` or `disk:`k (flush every k blocks)
- **Compression**: `none`, `lz4:1`, `zstd:1,3,9`

A fixed PRNG seed guarantees reproducible payloads. The same executable can run on both physical Android devices and emulators. Passing the optional `{suite` flag benchmarks all combinations of $(N, \text{, payload, , mode, , compression})$.

(B) UI-RTT Benchmark. A lightweight UI allows the user to pick a transport and launch a round-trip test directly from the handset. The entry screen offers three buttons: *Iroh*, *UDP*, and *TFTP*. These switch the underlying transport at runtime. The following "Simulation Config" form lets the user enter payload size, message rate, and test duration, and provides a single `START SIMULATION` button that begins a loop of in-memory send/echo messages. Each leg is timestamped inside the Rust layer; the UI periodically polls the running process and saves the results as CSV files on the device.

In summary, the CLI isolates *storage* while the UI benchmark gives a single RTT datapoint to place storage costs in a real network context; all later sections draw exclusively on one of these two harnesses.

4.2 Round-Trip Baseline (Network + Storage)

The storage experiments in Section 4 leave out all network effects. To better understand how storage performance compares to the rest of the system, we measured an end-to-end round-trip time (RTT) between a Pixel-6 emulator (sender) and a Galaxy S8 physical device (receiver). Both devices run the same Rust core and Kotlin UI. The emulator runs on the desktop screen, while the phone is connected to the computer via USB. We ran the same test for both Iroh and plain UDP.

Every $10\,\text{s}$ the emulator sends a single $128\,\text{B}$ payload. The phone receives, validates, stores it in memory, and immediately echoes it back. Timestamps recorded at `send_benchmark_sc` and `processing_message` provide an application-level RTT, which we can directly compare to the isolated-storage insert times.

Result. Figure 1 shows the results for the two transport options in our setup. Raw UDP completes the round trip in a median of around 7 ms with almost no outliers, while encrypted QUIC (Iroh) comes in at about 56 ms and shows a noticeable long tail.

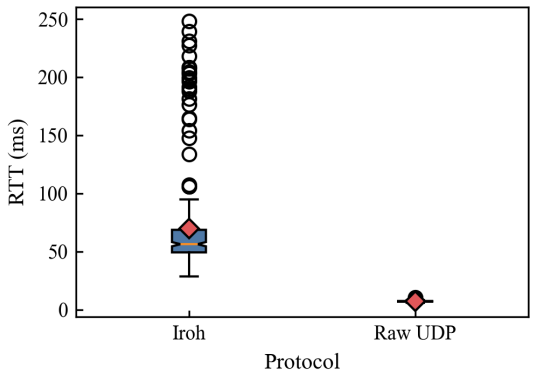

Figure 1: Application-level RTT for bursts of ten 128 B blocks emitted each second. Each box summarises 300 messages (n=300).

4.3 Isolated Storage Results

This section isolates the storage layer, which allows to dive deeper into its behavior without interference from networking or consensus logic. All experiments were executed using the stand-alone Rust CLI on the Samsung Galaxy S8 (physical) and the "gphone64" Pixel 6 emulator. Unless stated otherwise, we disable compression and fix the flush interval to `disk:100`.

4.3.0 Statistical baseline. Before analyzing larger configurations, we ran the two main storage modes (`memory` and `disk:100`) **five times** at N=100,000 blocks and collected 95 % confidence intervals. Figures 2 and 3 show that the resulting interval bars are barely visible, below 2 % of the mean in all cases. For the rest of this section, we therefore report the *mean* of a single representative run, as the variance is statistically negligible. This choice also helped reduce the total time needed to run the larger benchmarks.

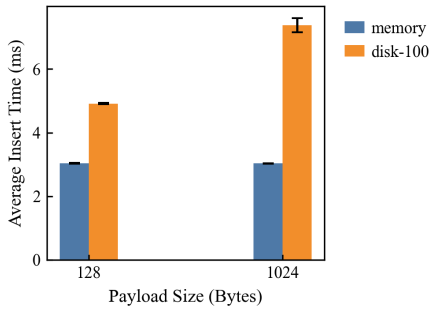

Figure 2: Average insert time per block at N=100,000 blocks with 95 % CI (n=5) for the two principal storage modes.

4.3.1 Scaling to 1M Blocks. Figure 4 shows the average insert time as the chain scales from 1 k to 1 M blocks, using the emulator with 128-byte payloads. Insert time increases linearly with chain length ($R^2 > 0.99$), confirming an $\mathcal{O}(N)$ cost model. In-memory mode stays efficient, remaining below 4 ms/block even at one million blocks, while `disk:100` passes 8 ms/block over the same range.

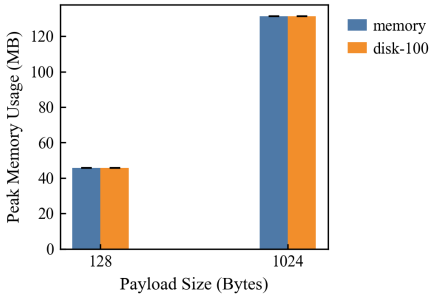

Figure 3: Peak RAM and final disk usage at N=100,000 blocks with 95 % CI (n=5).

Figure 5 shows the peak RAM usage in `memory` mode for the same experiment. Usage grows linearly with chain length, reaching around 600 MB at one million blocks. This suggests a rough estimate: if this trend continues, devices with 1 GB of free RAM could handle roughly 1.5–1.8 million blocks in pure memory mode. That number would be significantly lower for larger payload sizes.

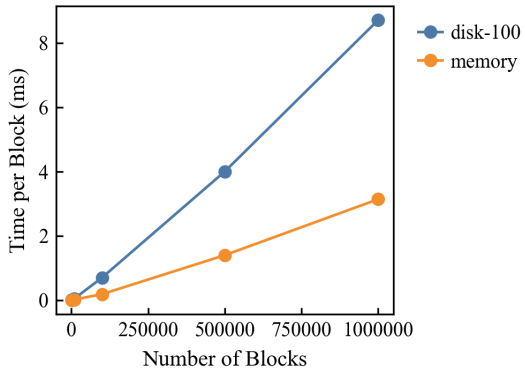

Figure 4: Average insert time per block versus chain length (128 B payload, emulator).

4.3.2 Impact of Flush Intervals. Figure 6 shows the CPU time per block for five storage modes at N=100,000 blocks and two payload sizes, using the emulator. Each `disk-N` configuration flushes to disk every N blocks. The `disk-50` setting shows the highest overhead, reaching up to 2.4 ms/block at 1 kB payloads. In comparison, `disk-500` and `disk-1000` both significantly reduce overhead, approaching the performance of `memory` while maintaining reasonable durability. The data shows diminishing returns beyond `disk-500`, which represents a practical sweet spot in terms of the latency–durability trade-off.

4.3.3 Payload Scaling Effects. To understand how payload size impacts performance, we benchmarked insert cost and

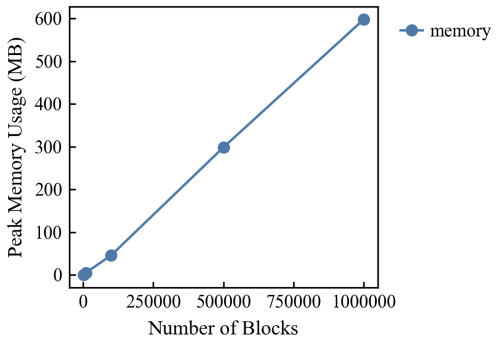

Figure 5: Peak memory usage for `memory` mode as a function of chain length (128 B payloads, emulator).

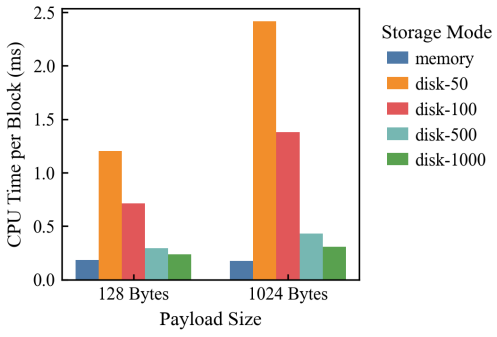

Figure 6: CPU time per block at N=100,000 blocks for five storage modes (emulator). Each `disk-N` mode flushes to disk every N blocks.

disk usage as a function of block count for five payload configurations (128 to 4096 B) using the emulator. The results in Figures 7 and 8 show near-perfect linear scaling in both dimensions, consistent with constant-time encoding and deterministic flush overheads.

Figure 7 shows that disk usage increases linearly with the number of blocks and is largely driven by the payload size. At 100,000 blocks, moving from 128 B to 4096 B payloads leads to a several-hundred-megabyte increase in disk space, showing that the majority of storage cost comes from the payload itself once sizes exceed around 1 kB.

Meanwhile, Figure 8 reveals that CPU time per block also increases linearly with block count and clearly separates by payload size. At 100k blocks, the per-block cost nearly triples between the smallest and largest payloads. This is expected, as larger payloads require more I/O and serialization work per insert.

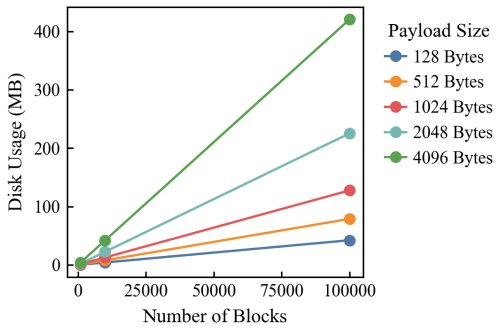

Figure 7: Disk usage growth as a function of number of blocks across five payload sizes (emulator). Disk cost scales linearly, with payload size as the dominant factor.

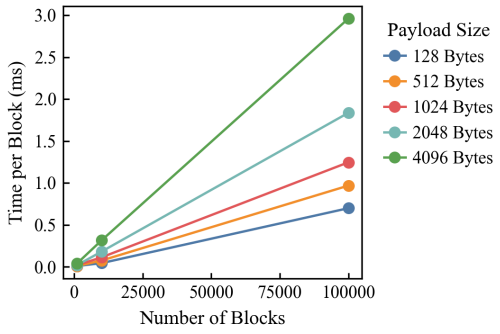

Figure 8: Average insert time per block versus number of blocks for different payload sizes (emulator). CPU cost increases linearly with chain length, and is higher for larger payloads.

4.4 Compression Study

Benchmark Setup. To evaluate the trade-offs of on-device compression for TrustChain storage, we benchmarked four compression modes on a physical Samsung Galaxy S8: `zstd:1`, `zstd:3`, `zstd:9`, and `lz4:1`, against a baseline without compression: `none`. At five different payload sizes (128,

512, 1024, 2048, 4096 bytes), each configuration was run once using 100,000 blocks. All benchmarks used the same `disk:100` flush interval and were executed under stable runtime conditions to minimize noise.

CPU Overhead. Figure 9 reports the average CPU time per block for each compression method and payload size. As expected, CPU cost increases with compression level. The `zstd:9` configuration shows the highest overhead, on smaller payloads, it processes each block more than five times slower than when no compression is used. On the other hand, `zstd:1` and `lz4:1` introduce moderate costs, typically staying under 10 ms per block.

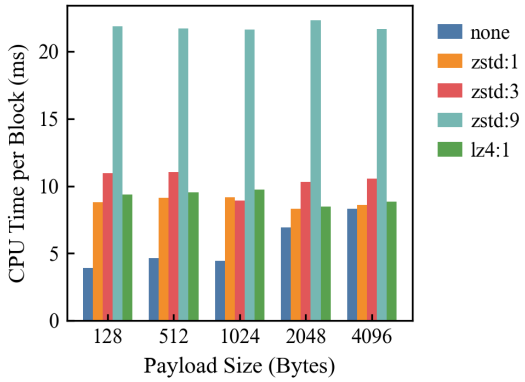

Figure 9: Impact of compression mode on average CPU time per block after 100,000 blocks (Samsung Galaxy S8), shown across varying payload sizes.

Profiling. To confirm that the slowdown is caused by the codec, we profiled the last 60 s of the `disk:100`, 128 B run on the Galaxy S8 using `simpleperf`: once with no compression and once using `zstd:3`.

No compression. The most active application-level functions together account for about 61 % of execution time:

- System RNG reseed: 5.3 %
- Two `serde_json` helpers (map serialization): 5.3 % in total
- All remaining items (`memcpy`, allocator calls, bookkeeping) stay below 2 %

With `zstd:3`. Enabling `zstd:3` compression shifts the profile noticeably:

- Bulk memory operations (`memset`/`memcpy`) rise from 9 % to 21 %
- `ZSTD_compressBlock_doubleFast` and related calls: 7 %
- Storage-related functions (`flush_to_disk`, `serialize_entry`, etc.) stay below 4 %, similar to the baseline

Take-away. Roughly 28 % of total CPU time now goes into compression itself or the extra memory operations it requires—closely aligning with the 25 % to 30 % runtime impact observed in Fig. 9. No new processing or hashing bottlenecks are introduced, confirming that compression, not file I/O, is responsible for the extra overhead. The lighter codecs (`zstd:1`, `lz4:1`) follow the same pattern but with lower overhead, reinforcing their suitability in situations where disk space matters but performance must remain responsive.

Disk Savings. Figure 10 shows the total disk usage after flushing all blocks to storage. Compression reduces disk usage consistently across all payload sizes. For example, at 4096-byte payloads, `zstd:1` lowers usage from around 0.42GB to under 0.30GB. The savings become more noticeable as payload size increases. This trend is expected since larger payloads tend to contain more repeated patterns, giving compression algorithms more opportunities to reduce file size.

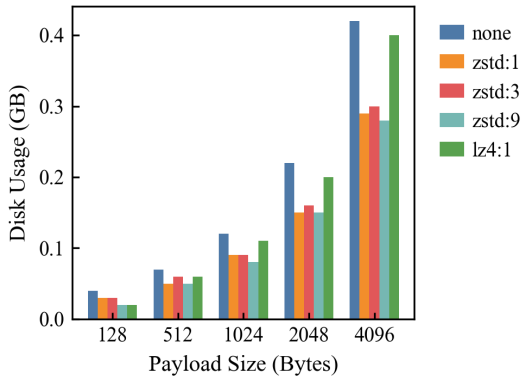

Figure 10: Impact of compression mode on disk usage after 100,000 blocks (Samsung Galaxy S8), shown across varying payload sizes.

Compression Trade-off Summary. Compression reduces the amount of disk used but introduces CPU overhead, with a clear trade-off between disk usage and processing time. For use cases that prioritize storage savings, `zstd:1` offers a good balance. Latency-sensitive deployments may prefer `none` or `lz4:1` for their faster runtime performance. Industry measurements on embedded cores show exactly the same trend: LZ4 and low-level Zstd give a 20–30% space reduction at sub-10 ms latency, while high-level Zstd wins on ratio only when extra CPU cycles are acceptable [9].

Responsible Research Practices

Reproducibility and open access. All source code, benchmark harnesses, and plotting scripts are available in a public Git repository[1]. The full experimental workflow, including device setup, flush intervals and compression settings is detailed in Section 3. Results may vary slightly on other hardware or Android versions, but all parameters are documented so that equivalent devices should reproduce the reported trends.

Synthetic data only. No real user data are ever processed. All blocks are produced from a fixed PRNG seed, giving reproducible and entirely synthetic payloads and keys.

[1] https://github.com/mbakker520/smartphone-trustchain/tree/storage_benchmarking

Licensing. The Rust core and Kotlin UI are released under a liberal Apache-2.0/MIT dual license and rely exclusively on open, royalty-free algorithms such as Ed25519, SHA-256, Zstandard, and LZ4.

Potential for misuse. While this project is intended for research and benchmarking purposes only, it is technically possible to embed the code in closed-source or privacy-obscuring peer-to-peer applications. We acknowledge this risk and note that no safeguards or usage restrictions are built into the system. The software is made available as-is for transparency and academic reproducibility, with no warranty or control over downstream use.

5 Discussion and Conclusion

This project set out to evaluate whether a mobile TrustChain node can scale from a few thousand to one million blocks while remaining responsive and storage-efficient under typical smartphone constraints. We focused on two core storage parameters: flush-interval batching and lossless compression. And analyzed their impact across CPU usage, latency, disk footprint, and memory scalability.

Feasibility and scalability. Our experiments confirm that a one-million-block chain fits comfortably on modern smartphones. With 128 B payloads, total disk usage is projected at ≈ 0.48 GB, and memory mode remains below 600 MB (Fig. 5), allowing full in-RAM operation on devices with ≥ 1 GB free. Insert latency stays under 8 ms even in disk-backed mode, and network round-trips over raw UDP remain fast (median RTT ≈ 7 ms), showing that storage never becomes the primary bottleneck.

Flush interval trade-offs. We observed a clear performance "knee" at `disk:500` (Fig. 6), where per-block CPU cost drops by $\approx 45\%$ compared to `disk:50`. Intervals beyond 500 result in reduced returns, suggesting that `disk:500` offers the best trade-off between durability and speed for most mobile deployments. Aggressive flushing increases CPU overhead substantially without significant benefit, and applications should avoid it unless strict crash-tolerance is required.

Compression benefits and costs. Both Zstd 1 and LZ4 provide consistent 20–30% storage reduction (Figs. 9, 10), with sub-10 ms latency on mid-range devices. Profiling confirms that the added cost is due to compression itself, not I/O or cryptographic overhead. LZ4 adds negligible delay and is suitable even for interactive use. For background syncing or archival storage, Zstd 1 offers improved compression at acceptable runtime cost. Zstd 9 reduces space further but exceeds $5\times$ the CPU time of uncompressed inserts, making it impractical for most mobile deployments.

Memory limits and hybrid strategies. RAM usage grows linearly with chain size, reaching ≈ 600 MB at one million blocks in memory mode (Fig. 5). This imposes a practical cap of 1.5–1.8 million blocks on devices with 1 GB of available memory. To support longer histories, applications should adopt hybrid memory/disk strategies that retain recent blocks in RAM and flush older ones to disk.

Real-world variance and generalizability. All results were obtained under controlled conditions using synthetic data, fixed insert rates, and isolated benchmarks. In real deployments, performance may degrade due to background processes, thermal throttling, or variable network quality. The TFTP transport was excluded from benchmarking due to unreliability on mobile hardware. Raw UDP performed consistently well, while Iroh (QUIC) introduced additional latency (~ 55 ms median RTT). Tests on ext4 (Galaxy S8) and F2FS (Pixel emulator) represent two common Android setups, but further validation is needed on iOS (APFS), lower-end devices, and newer UFS-based flash controllers.

Hypothesis verdicts

H1: **(capacity)** upheld: a one-million-block TrustChain fits comfortably within the RAM and flash constraints of modern smartphones.

H2: **(space savings)** upheld: lightweight compression reduces disk usage by 20–30% with acceptable CPU overhead.

H3: **(interactive latency)** upheld: under moderate batching and compression, per-block insert latency remains below 10 ms, even at high block counts.

Final remarks. This work demonstrates that DAG-based, user-owned blockchains like TrustChain can operate efficiently on real mobile hardware, provided they apply batching and lightweight compression. The benchmarks and profiling tools developed here provide a basis for future mobile ledger systems. Ongoing work should focus on adaptive flushing, cross-platform validation, and energy-aware optimizations to prepare TrustChain for production-ready, real-world deployments.

6 Future Work

Two storage related possible future additions:

- **Binary block encoding.** Replace JSON payloads with a compact binary format (e.g., Protocol Buffers or FlatBuffers) to reduce on-disk size and speed up parsing.
- **Adaptive flush controller.** Replace the fixed `disk:k` setting with a feedback loop that stretches flushes on *Wi-Fi + charge* and shortens them under interactive load, balancing tail latency and flash wear.

And in addition to the latency-, throughput-, energy-, and robustness- studies already completed more general possible additions:

- **Cross-platform port & API parity.** Compile the Rust core for `aarch64-apple-ios`, publish Swift bindings, and confirm that mobile nodes can sync, validate, and expose the same GraphQL/event interfaces as desktop peers.
- **Large-scale interoperability tests.** Deploy mixed fleets of phones, tablets, and desktops (Wi-Fi, 5G, Ethernet) to exchange chains, measure fork-resolution latency, and uncover protocol edge-cases under thousands of concurrent connections.

References

[1] Joungyoul Lee. Initial F2FS performance results. Linux Kernel Mailing List post, October 2012. Message to linux-kernel@vger.kernel.org.

[2] Initial sync extremely slow – use ssd. Bitcoin Stack Exchange, Q&A post, 2020.

[3] Pim Otte, Martijn de Vos, and Johan Pouwelse. TrustChain: A sybil-resistant scalable blockchain. *Future Generation Computer Systems*, 107:770–780, 2020.

[4] Fabian Kahmann, Maximilian Bader, Manuel Brack, Wiebke Sahlmann, and Peter Herrmann. Performance comparison of dag-based ledgers and blockchain platforms. *Computers*, 12(12):257, 2023.

[5] Bulat Nasrulin, Martijn de Vos, Georgy Ishmaev, and Johan Pouwelse. Gromit: Benchmarking the performance and scalability of blockchain systems. In L. O'Conner, editor, *Proceedings of the 4th IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS 2022)*, pages 56–63. IEEE, 2022.

[6] Bingzhe Li, Qian Wei, Wanli Chang, Zhiping Jia, Zhaoyan Shen, and Zili Shao. Blockchain data storage optimisations: A comprehensive survey. *ACM Computing Surveys*, 57(1):1–37, 2024.

[7] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Safe systems programming in rust. *Communications of the ACM*, 64(4):21–28, 2021.

[8] William Bugden and Ayman Alahmar. The safety and performance of prominent programming languages. *International Journal of Software Engineering and Knowledge Engineering*, 32(5):713–744, 2022.

[9] Calliope-Louisa Sotiropoulou. Evaluating lossless data compression algorithms and cores. CAST Inc. Technical White Paper, 2025. Available: https://www.cast-inc.com/blog/white-paper-evaluating-lossless-data-compression-algorithms-and-cores.