KEIBIDROP

Post-Quantum Encrypted P2P File Transfer with a Cross-Platform Virtual Filesystem

Marius-Florin Cristian, CISSP
Founder, KeibiSoft SRL | Head of Security, Doczen

Pass the SALT | 2026 Lille

What is **KEIBI**DROP?

Demo

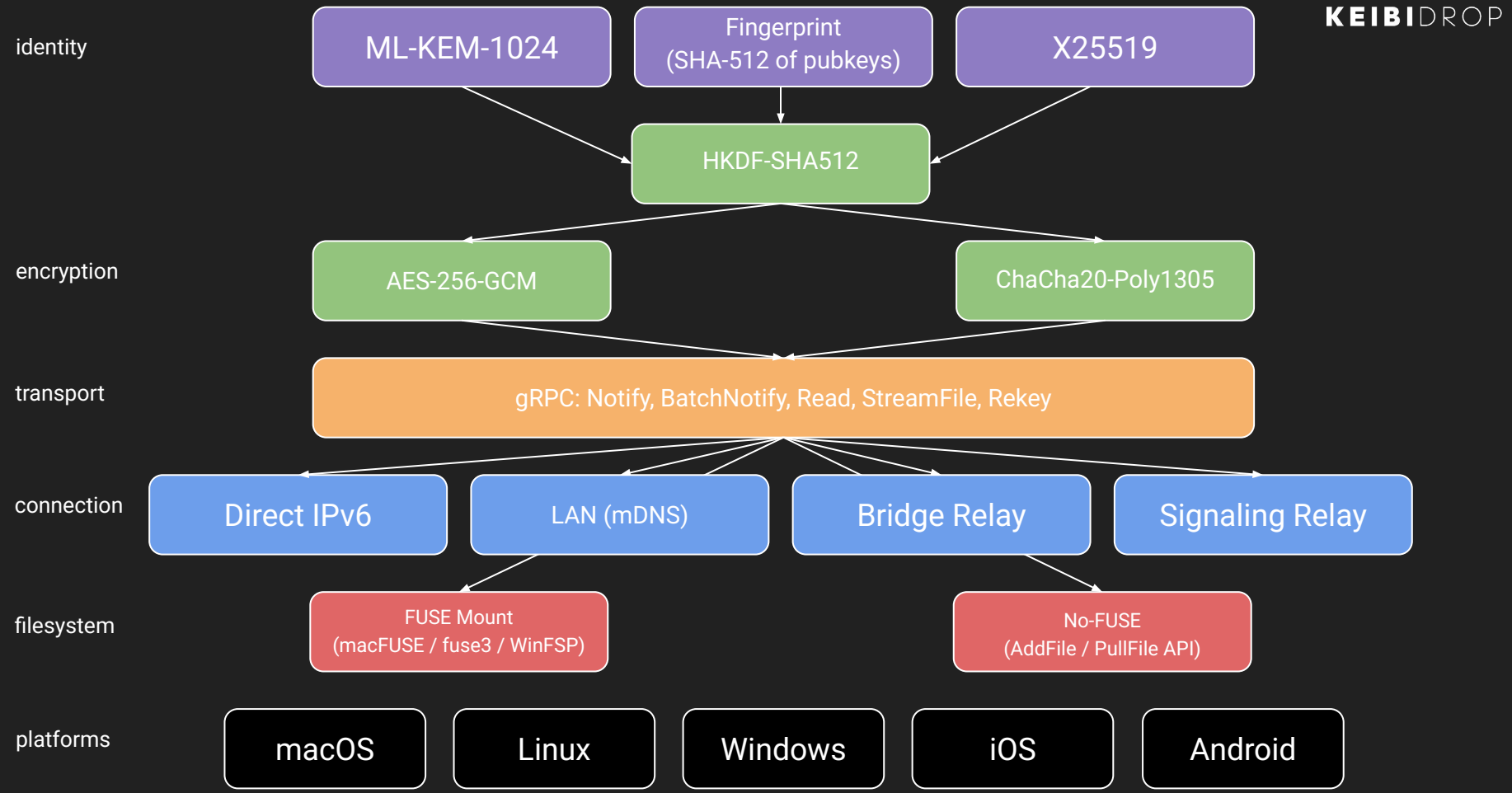
(Mac <-> Windows connect and work on demand)

LAN and stream: https://youtu.be/9Wt0NMx2_I8

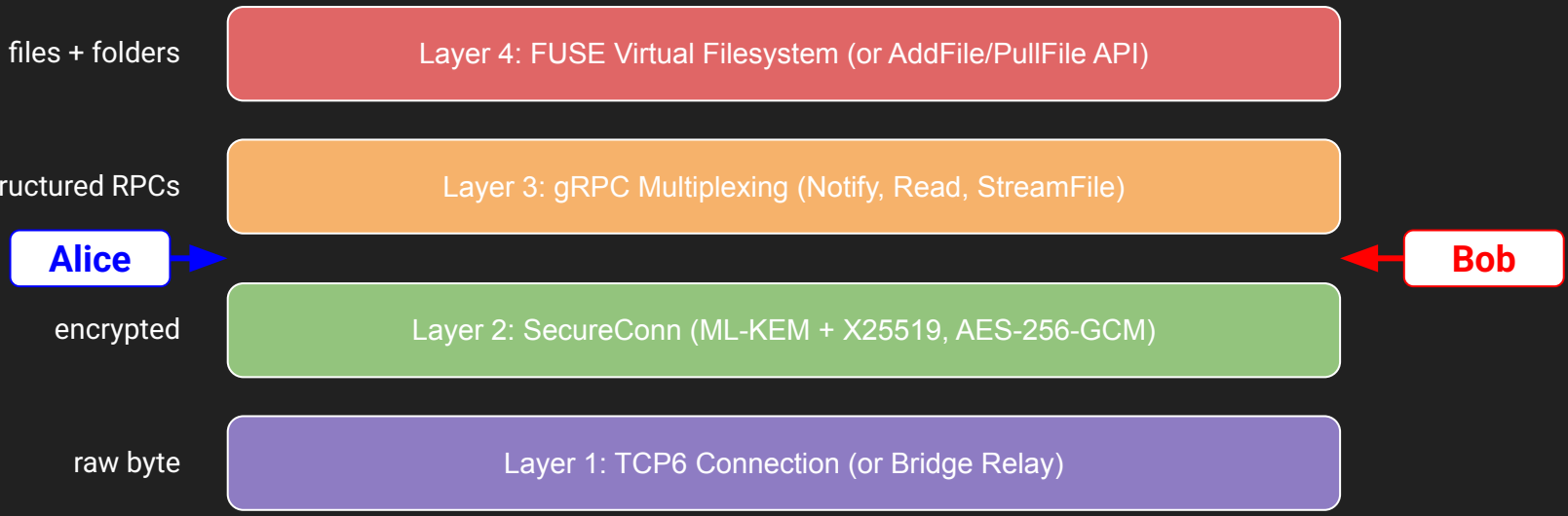
WAN (low bandwidth) git clone: https://youtu.be/h5fWS0kPEvs

WAN (low bandwidth) and steam: https://youtu.be/HwhrlRSl_LY

KEIBIDROP


identity
ML-KEM-1024
Fingerprint
(SHA-512 of pubkeys)
X25519
HKDF-SHA512
encryption
AES-256-GCM
ChaCha20-Poly1305
transport
gRPC: Notify, BatchNotify, Read, StreamFile, Rekey
connection
Direct IPv6
LAN (mDNS)
Bridge Relay
Signaling Relay
filesystem
FUSE Mount
(macFUSE / fuse3 / WinFSP)
No-FUSE
(AddFile / PullFile API)
platforms
macOS
Linux
Windows
iOS
Android

Architecture: Four Layers



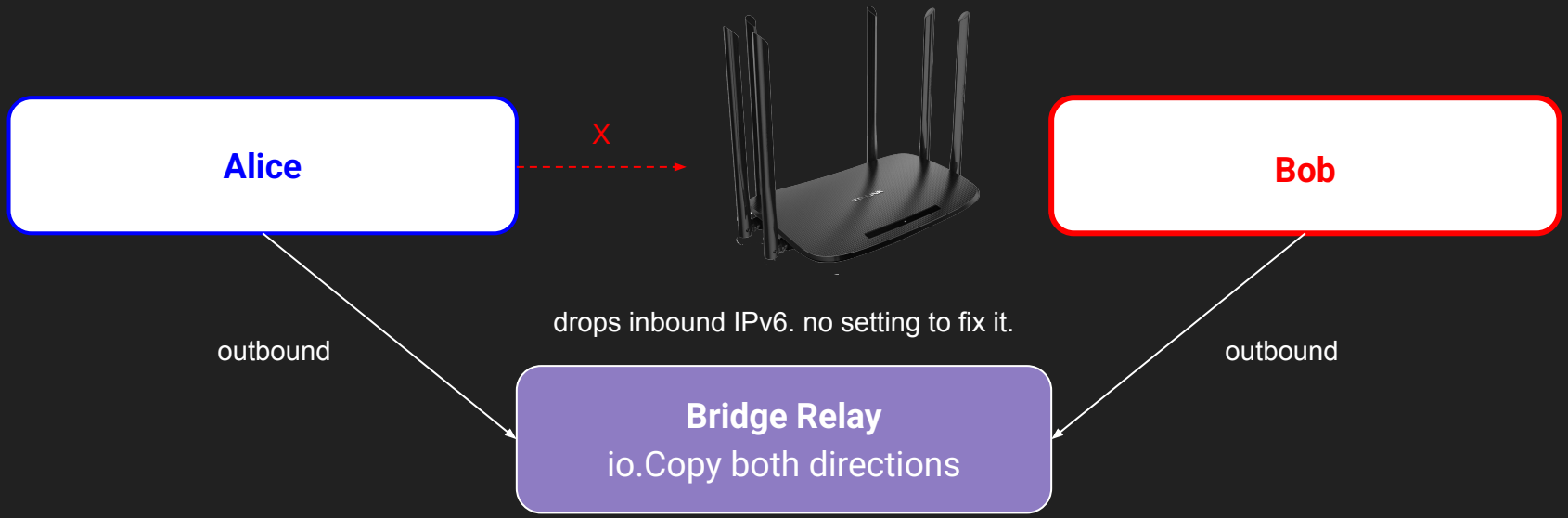
Each layer adds one thing. TCP moves bytes. SecureConn encrypts them. gRPC structures them into RPCs. FUSE makes them look like files.

We went IPv6-only

- 50% of the world has IPv6 now (Google stats, 2026)
- Global unicast addresses: both peers can reach each other directly
- No STUN. No TURN. No UPnP. No NAT traversal infrastructure.
- No IP metadata leaked to third-party STUN servers.

This worked. Direct peer-to-peer, no middleman, sub-millisecond connection setup on IPv6.
But then…

“But then, a shape-shifting master of darkness…”



Both peers connect outbound. The bridge forwards encrypted bytes it cannot read

Privacy model

The relay runs io.Copy. It forwards bytes it cannot decrypt.

Keys are ephemeral. Generated fresh each session. Never written to disk.

Fingerprint exchange is out-of-band. The relay never sees fingerprints.

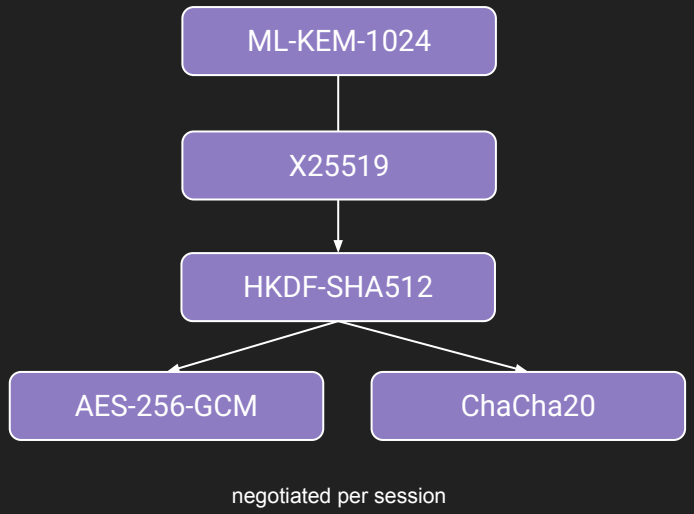
Registration blobs are encrypted client-side. Lookup key is HKDF-derived.

Relay TTL is 10 minutes. After that, blobs are garbage collected.

No analytics. No crash reports. No telemetry. No logs shipped anywhere.

relay_keepalive.go

Post-Quantum Key Exchange



Ephemeral keys, no certificates.
Fingerprint = SHA-512 of public keys.
Verified out-of-band (Signal, paper, QR).
Re-key every 1 GB or 1M messages.

secure_conn.go
handshake.go

The wire format:

Every message on the wire is one AEAD record

prefix OUTB/INBD (direction)· counter monotonic per direction

- length covers nonce + ciphertext + tag, not itself.
- Rejected before allocation if it exceeds 20 MiB. No OOM from a forged header.

secureconn.go
symmetric.go

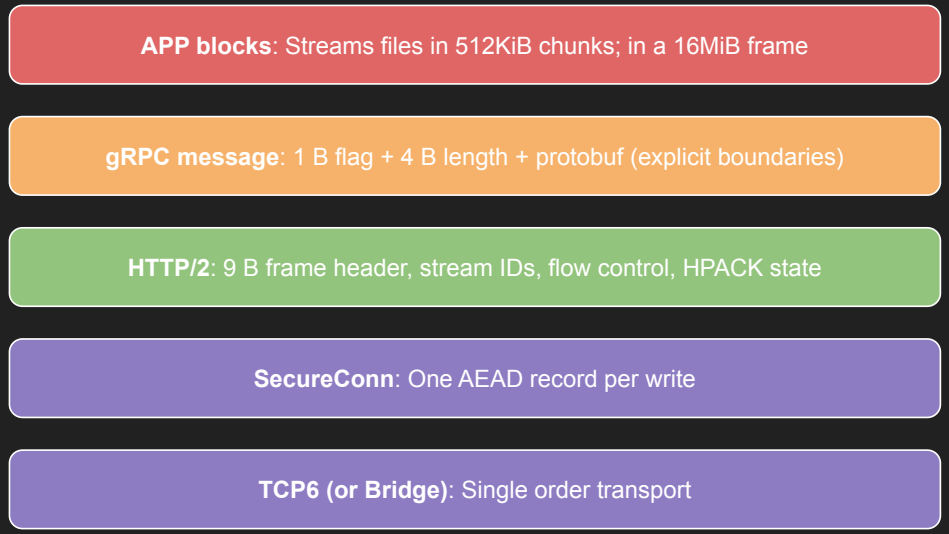
- AEAD: ChaCha20-Poly1305 or AES-256-GCM, per session. 12 B nonce, 16 B tag.
- Re-key at 1 GB / 1M msgs: fresh key, counter resets, never a (key, nonce) repeat.

“Can’t you just rearrange the blocks?”

Each record is authenticated on its own, but the reader takes the nonce from the wire and keeps no receive counter.

So the AEAD layer alone will not reject a reordered or replayed record.

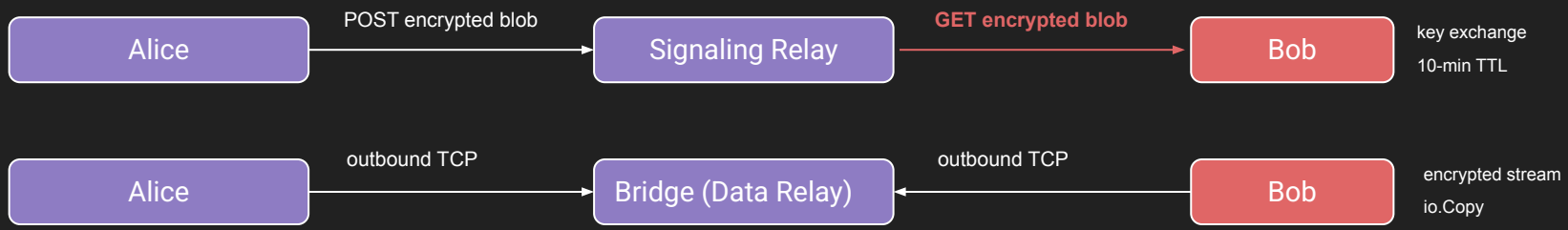
But why this is only a denial of service, and never forged data?



a) TCP is the only path records take: in sequence, no duplication, no side channel.
A benign network never reorders; a hostile one must actively rewrite the stream.
b) gRPC over HTTP/2 is strictly ordered and framed. A record moved, dropped, or replayed corrupts the HTTP/2 framing → protocol error → teardown.

The tag means only whole, valid records can be shuffled at all, no forging or splicing inside one.
Worst case: the connection dies. The adversary stops progress; it never yields a forged message.

Two relays, both blind



Signaling relay stores lookup_key → encrypted_blob. Cannot decrypt. Cannot correlate sessions. Blobs expire in 10 minutes.

Data relay copies bytes between two outbound connections. Cannot decrypt. Sees source IPs in logs, nothing else.

What if we hijack the kernel?

Intercept every open(), read(), write()
and serve files from another computer.

10 syscalls to build a filesystem

Getattr	Release
Open	Mkdir
Create	Readdir
Read	Rename
Write	Flush

That's the core.

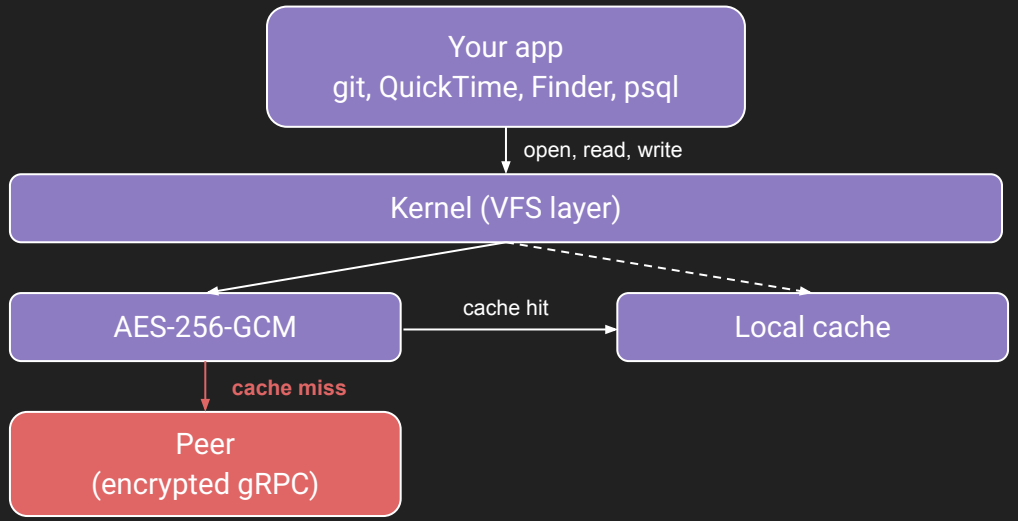
macFUSE, fuse3, WinFSP

all share these 10.

*Each platform needs its own mount flags to behave correctly.

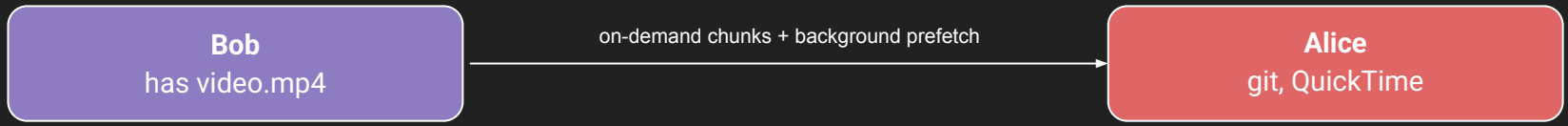
fuse_directory.go
api.go

FUSE Virtual Filesystem



cat /mount/peer/README.md triggers a gRPC stream. The application has no idea.
fuse_directory.go
api.go

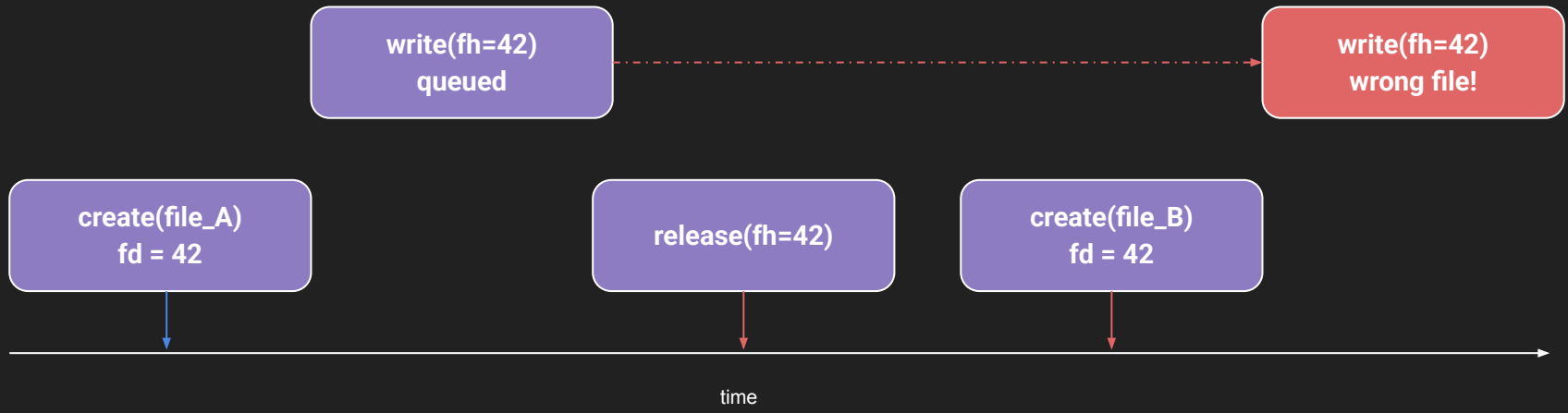
Stream a 4 GB video from a peer



Chunk bitmap (512 KiB blocks):

Playback starts in seconds. Full file downloads in background.
Tested with .mp4, .mov, .webm over WAN.

The file descriptor reuse race



PostgreSQL initdb: 224 files/second. 50% data corruption.
Fix: Use monotonic opaque handles ID's
fuse_directory.go

Performance

Layer	Optimization	Throughput
Raw TCP (baseline)	cache AEAD cipher, single write syscall	19,400 MB/s
SecureConn	push-based stream, batch 64 notifications	1,740 MB/s
gRPC streaming	kernel read size 4K → 2M, cap 8 parallel streams	662 MB/s
FUSE round-trip	FUSE round-trip	276 MB/s
Laptop → VPS (WAN)	gRPC window tuning, 500 Mbps line	55 MB/s
iPhone → Mac (WiFi)	gomobile xcframework	47 MB/s
Through bridge relay	Timisoara relay, encrypted	43 MB/s

Catherine McGeoch: “change one thing, measure, proceed.”
We profiled layer by layer and optimized each bottleneck.

secure_conn.go
Benchmark Matrix (May 2026)

The magic lies in hiding the latency, not maximizing the bandwidth

The table proves raw speed. But whether a remote file **feels** local is decided by two latency walls, and more bandwidth moves neither.

Same machine

the filesystem boundary
set by **block size**, not the link

4 KB reads: 300 MB/s
⇒ 2 MiB reads: 3,400 MB/s

Across the world

the network boundary
set by **round-trip time**, not the wire

512 KiB per round trip:
~2 MB/s no matter the link

You don’t buy speed. You **hide latency**.

Boundary 1: the filesystem (block size, no link yet)

FUSE advertises a block size (**st_blksize**); cp and dd size their reads from it. Advertise 4 KB and every read is a syscall; advertise the right size and reads go big.

Advertised block	Throughput	Write calls / GB
4 KiB (kernel default)	300 MB/s	~25,600
2 MiB	3,400 MB/s	~1,000

One number, 10×. Same machine, same disk, no network involved.

Optimal per platform:

- macOS Intel 2 MiB, M-series up to 10 MiB
- Linux 256 KiB, Windows (WinFsp) up to 10 MiB

The ceiling: past~16 MiB, memory pressure and Translation Lookaside Buffer thrashing bites (16 parallel × 64 MiB = 1 GB of buffers on 8 GB RAM).

Block Size Performance

Boundary 2: the network (the round-trip floor)

On demand, one 512 KiB chunk per read is one round trip per chunk. The reader waits; the wire sits idle. Throughput settles near 2 MB/s, independent of the link (here up to 1 Gbit/s).

Link	RTT	Naive 512 KiB/RTT	16 MiB read-ahead
Timisoara VPS to Singapore	200 ms	2.05 MB/s	**30.4 MB/s** (15x)
Iasi home to Singapore	330 ms	1.65 MB/s	**28.8 MB/s** (17x)

Fetch one gRPC-frame-sized 16 MiB block per miss and prefetch the next: one round trip amortizes over 32 chunks, and on-demand reads reach the bulk band, 22 to 30 MB/s. Time to first byte~0.5 s.

Real RTTs: 24 ms Iasi to Timisoara, 200 ms Timisoara to Singapore, 330 ms residential to Singapore.

The Singapore WAN Report

How much do you need? The Blu-ray test

To never stall, the fetch must beat the consumer's rate. Blu-ray video peaks near 40 Mbit/s; we pace playback at 3 MB/s and count stalls on a cold mount.

Platform	Read ahead	Freezes	Total frozen
Windows / WinFsp (100MB)	off	5	**2.6 s**
	on	**0**	0 ms
Linux / libfuse (200 MB)	off	13	23.6 s
	on	**1**	**1.5 s**

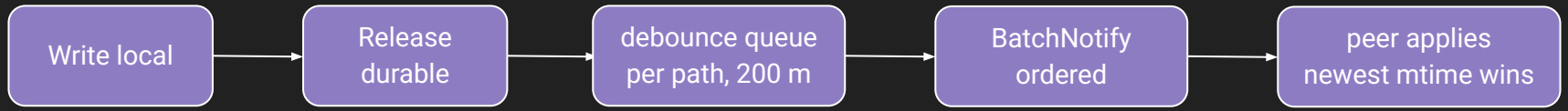
Window: 64 MiB (four 16 MiB blocks), fetched one ahead of the read head.

The catch (self-congestion): prefetch too hard and you saturate the link, the urgent read queues behind the bulk, and the latency you hid comes back. So blocks are fetched one at a time: the window has a ceiling, not just a floor.

Smooth On-Demand Playback

How an edit propagates

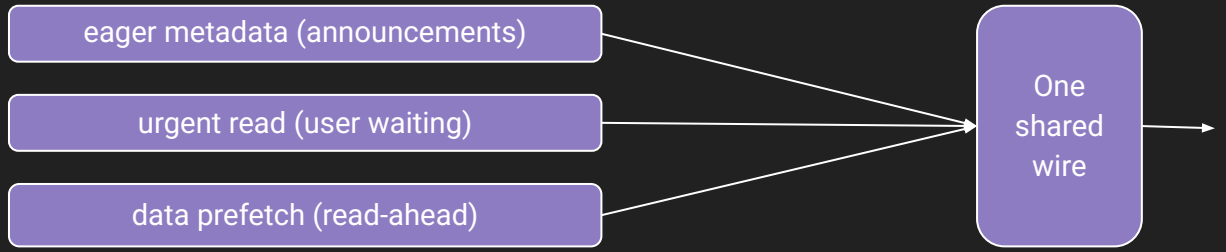
You edit a file on the mount. The write lands locally; on Release, once the write is durable, the change is published to the peer.



- A burst of writes to one file coalesces into one announcement (git-lfs writing an object in chunks does not restart the peer ten times). Removes and renames go immediately; creates and edits debounce.
- Order is last-write-wins by modification time (UnixNano); the receiver rejects any notification older than its current state for that path.
- Design space, per workload: per-chunk non-cryptographic hashes (xxh3) invalidate only the changed region, so a one-byte edit does not re-fetch 40 GB. If clocks cannot be trusted, a logical (Lamport) clock can order edits instead of the wall clock.

Prefetch has to leave room

KEIBIDROP streams file metadata **eagerly**, ahead of any read, so the whole tree is browsable at once. That announcement stream shares one wire with data prefetch and with the read the user is waiting on.



- Prefetch too hard, saturate the wire, and the metadata stream starves. A git clone or a git-lfs checkout, hundreds of creates and renames that need their announcements to keep flowing, stalls or fails.
- So prefetch is **paced**: one block at a time, backing off when the link is busy, so metadata and the demand read always keep headroom.
- Aggressive enough to hide the round trip (Boundary 2), gentle enough not to starve the pipeline. That is the ceiling on the read-ahead window.

Runs on everything

macOS | Linux | Windows | iOS | Android

Desktop GUI (Rust + Slint) | CLI (keibidrop-cli) | Agent (kd)
One Go engine. Same crypto. Same FUSE handler (cgofuse)

Open problems: no one design fits all

a) **Atomic saves, edit regions**. Apps save by copying the whole file to a temp and renaming it over the original. Track which regions changed,
or a one-byte edit re-fetches the whole 40 GB. (wire-protocol change; handling depends on each app's write pattern.)

b) **Mobile connections move**. Phones drop and switch networks; TCP dies with the path. Run the same post-quantum handshake over **QUIC / HTTP-3** (survives an address change), gRPC on top. (doable, not yet built.)

c) **More than two parties.** A real chain of custody needs multi-peer sharing, fallback or on-demand relays (a unikernel spun up near both ends to forward bytes), and from that, versioning. (every party adds trust and consistency questions.)

d) **Datasets bigger than the disk**. The cache can outgrow local storage. Evict cold chunks and keep only the working window. (what to keep, re-fetch cost, thrashing: a whole new set of problems.)

No single design hits every use case. These are tradeoffs, not missing features.

KEIBIDROP

Post-quantum encrypted. Peer-to-peer. Open source.

Web https://keibidrop.com
GitHub https://github.com/KeibiSoft/KeibiDrop
Homebrew brew tap keibisoft/keibidrop && brew install keibidrop
Chocolatey choco install keibidrop
License MPL-2.0

Contact marius@keibisoft.com

Questions?