SafeHub: End-to-end encrypted Git hosting system

Easwar Vivek Mangipudi
Hoth Labs
easwarmangipudi@gmail.com

***Abstract*—Private repositories remain readable to Git hosts despite transport and at-rest encryption. We present SAFEHUB, an end-to-end encrypted Git hosting system. It encrypts repository contents and semantic metadata—file names, commit messages, authors, branches, issues, pull requests, and refs—so the host sees only ciphertext, opaque identifiers, lengths, and order.**

Each repository is a Messaging Layer Security (MLS) group, providing admin-mediated membership, post-compromise healing, and per-invite history windows. Ordinary Git behavior is preserved within each member's window: branches, merges, and blame still work. Forward-only members start from a join shallow snapshot rather than the full past.

Confidentiality alone is not enough: Git's hash-linked objects do not protect mutable refs. SAFEHUB records refs in an encrypted, device-signed, hash-chained manifest that detects rollback against a member's own anchor and host forks when members compare checkpoints; force-pushes require administrator co-signatures. We specify a single ideal functionality $\mathcal{F}_{\mathsf{safehub}}$ for the system and prove that SAFEHUB universally composably realizes it against a malicious server and adaptive member corruptions, in a hybrid model over group key agreement and certification, assuming secure erasure in the quantum random oracle model.

Our NIST PQ Category-5 Rust prototype measures full-stack `push`, `pull`, `fetch`, `clone`, `merge`, `rebase`, and `force-push` on a client-server pair of AWS Graviton4 hosts, together with the epoch rotation and consolidation that Git has no counterpart for. Against Git on its lowest-overhead native transport, wall-clock `push` runs $1.45\times$ plain Git at a 0.05 MB delta and $0.98\times$ at a 5 MB one, its marginal cost 46.7 ms/MB against Git's 49.3. We compare SAFEHUB with five other systems—plain Git, git-crypt, git-remote-gcrypt, and a reimplementation of the closest peer—over a single transport, with clients and remotes on separate hosts. The comparison separates designs whose cost follows the *edit* from designs whose cost follows the whole *file*. For a fixed 1 KiB edit, with the edited file growing from 10 KiB to 8 MiB, SAFEHUB's cost per update remains constant at 6.7 kB, because it seals the packfile Git has already built, whereas the systems that encrypt each file individually grow with the file and reach 8.39 and 13.4 MB. On that shared transport SAFEHUB is the fastest of the six at push, pull, fetch, merge, rebase, and force-push, each constant in history depth, and its stored size matches plain Git to within 0.2%, whereas the per-file encryption used by the other systems costs 13 to 21 times as much. The cost that does not amortize is `clone`, which grows with sealed history: a host that cannot read a repository cannot repack it.

***Index Terms*—End-to-end encryption, git, MLS, forge security, fork consistency, post-quantum cryptography**

I. INTRODUCTION

Source-code hosting concentrates sensitive organizational data on a small number of platforms. GitHub reports more than 180 million developer accounts and 630 million repositories; roughly four in five contributions occur in private repositories [1]. GitLab reports more than 50 million registered users and more than half of the Fortune 100 as customers across SaaS and self-managed deployments [2]. Both platform families protect private repositories with Transport Layer Security (TLS), provider-side encryption at rest, and access-control lists. None of these mechanisms, however, keeps the repository secret from the provider: the host holds the keys it needs to serve it, so operators, host compromises, and compelled disclosure can still read the entire repository plaintext.

Organizations seeking isolation from the provider therefore commonly self-host a forge. Doing so only moves the same trusted operator inside the organization, and it imposes operational costs unrelated to confidentiality. Removing that trusted point entirely calls for end-to-end encryption—but a Git host is not an ordinary encrypted store. Any such design must still accommodate repositories that outlive individual members, continuous-integration (CI) and search services, revocation, bounded history access, malicious-host integrity, and post-quantum protection against archived ciphertext.

Existing encrypted-Git tools do not provide this combination. git-crypt [3] leaves graph and path metadata host-readable and encrypts under a static, deterministic key, so identical files fingerprint across history and a key leak exposes the entire past and future; git-remote-gcrypt [4] encrypts whole repositories without authenticating refs (the name→tip map for branches and tags), so a malicious host can roll tips back or fork clients onto divergent views while packs still decrypt; and Keybase Git [5] used static per-team key generations without history windows or post-compromise security. Recent academic work encrypts Git on unmodified forges with proofs against a malicious server [6], but keeps revocation and bounded history access outside its model. None combines these properties with post-quantum parameters. The gap is not a missing cipher but a missing architecture: whoever controls the unlock keys must also cryptographically decide who receives them—the host has no say—and must govern how far back each member may read and which ref state is authentic.

A. Approach

SAFEHUB closes that gap with three structural observations. First, *Git is already a Merkle DAG*: a commit hash transitively authenticates the history beneath it, so end-to-end integrity of repository *contents* is inherited from object addressing; the only mutable state the host can attack is the map from ref names to tips. Second, *a repository is a natural administrated group*: bounded, admin-mediated membership with device

granularity—the setting modeled by administrated continuous group key agreement (A-CGKA) [7], [8] and implemented by Messaging Layer Security (MLS) [9]. Third, *history access is not all-or-nothing*: contractors, CI runners, and search indexers need the repository only from their join epoch forward (*forward-only* access), so an invitation should grant an explicit history interval rather than the entire past. Such a grant does not withhold the current code: a forward-only joiner receives a join shallow snapshot as of its join epoch. It withholds *superseded and deleted* content, and on real repositories that is almost everything: a forward-only grant withholds from the joiner 96–97% of every blob version ever committed and a third to a half of all paths ever present (section VI-C). Secrets are routinely committed and then removed rather than never committed [10], so a window bounds historical exposure, not present exposure.

These three observations map onto four composed layers in the SAFEHUB design (section IV). (1) A per-repository MLS group with client-verified administrator control (2) dual key regression (DKR) tokens that derive exactly each member's history interval (3) incremental Git bundles encrypted under fresh data-encryption keys (DEK s) and stored as ciphertext and (4) an encrypted, device-signed, hash-chained ref manifest that rejects rollback against a member's own anchor and detects host forks when members compare checkpoints. Force-pushes require administrator co-signatures. Pull requests, issues, and review travel as MLS application messages.

a) Untrusted host, org-trusted plaintext endpoints: SAFEHUB removes confidentiality trust from the hosting provider. The member devices—and CI, search, and consolidation fleets, when those services are required—remain organization-trusted plaintext endpoints. As in any end-to-end encrypted system, member devices already hold keys and plaintext; the server stores ciphertext and the sequencing metadata needed to order updates, so self-hosting is no longer required solely for confidentiality. Host compromise exposes only the explicit leakage of section IV-H—ciphertext, sizes, timing, operation type, repository identifier, control-channel account and device linkage, head sequence numbers, MLS framing, and compare-and-swap metadata. Compromise of a forward-only runner exposes plaintext and keys only within its post-join window; full-history members and administrators remain high-value endpoints.

B. Provable security in the universal composability (UC) framework

Four layers that are sound in isolation need not compose safely. End-to-end encrypted storage systems have repeatedly failed at component boundaries [11]–[13], and a Git host adds seams between bundles and epochs, refs and group state, and history windows and the DAG. We therefore specify whole-system security as an ideal functionality $\mathcal{F}_{\mathsf{safehub}}$—repository state, admin-gated membership, per-invite history windows, and an explicit leakage function limited to ciphertext sizes and timing—and prove that SAFEHUB UC-realizes $\mathcal{F}_{\mathsf{safehub}}$ against a malicious server and adaptively corrupted members under *secure erasure*, in a hybrid model that isolates inherited components (an administrated CGKA functionality $\mathcal{F}_{\mathsf{acgka}}$, a certificate authority, and a random oracle) from what we construct. UC composition lets larger systems invoke SAFEHUB as a subroutine. The full proof appears in appendices B and C.

C. Contributions

- **System design and protocol (section IV).** SAFEHUB uses A-CGKA-governed groups to progress epoch keys while granting each invite either a full-history clone or a join shallow snapshot through an explicit history window. Repository objects travel in encrypted bundles, whose stored ciphertext can be rewritten through admin-authorized consolidation (section IV-D), while collaboration and CI use MLS application messages. Fork-consistent RefHead manifests let a verifier reject rollback behind its own anchor, detect forks when members compare checkpoints, and enforce force-push policy.
- **Formal model (section V, appendix B).** We define the ideal functionality $\mathcal{F}_{\mathsf{safehub}}$, which gives a UC treatment of a version-control host with explicit leakage and history windows while exposing fork detection as an ideal-functionality event. Relative to formal E2EE storage and Group Key Progression [14], [15], $\mathcal{F}_{\mathsf{safehub}}$ extends the model with ref-manifest integrity and verifier-enforced authorization for force-pushes and consolidation. It further captures join shallow forward-only onboarding and requires consolidation not to expand decryptable history (appendix F-A).
- **UC proof in the quantum random oracle model (QROM) (theorem 1, appendix C).** Π_{safehub} UC-realizes $\mathcal{F}_{\mathsf{safehub}}$ in the $(\mathcal{F}_{\mathsf{acgka}}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{ro}})$-hybrid model against quantum polynomial-time (QPT) adversaries that control the server and adaptively corrupt members under secure erasure, even with quantum access to $\mathcal{F}_{\mathsf{ro}}$. The proof constructs a simulator and uses explicit abort conditions and a hybrid sequence to reduce history containment, post-compromise security and forward secrecy, RefHead integrity, admin authorization, committing AEAD, and DKR puncturing to standalone games. Although adaptive membership and export security under erasure remain a hybrid assumption on $\mathcal{F}_{\mathsf{acgka}}$ [7], [8], [16], [17], we prove the SAFEHUB-specific adaptive opening under erasure.
- **Post-quantum instantiation (§ VI-0a, appendix E).** We instantiate the protocol at NIST PQ Category 5, sizing public-key cryptography, hashes, MACs, and history-window checks for more than 128 bits of quantum security (definition 4 and table IX).
- **Prototype and evaluation (section VI).** We implement a NIST PQ Category 5 Rust prototype and evaluate it against five other systems—plain Git, git-crypt, git-remote-gcrypt, and a reimplementation of the closest peer—on one transport, with clients and remotes on separate AWS Graviton4 hosts and every clone verified against its source (section VI-A). On that common transport SAFEHUB is the fastest of the six at push, pull, fetch, merge, rebase, and force-push, and its stored size matches plain Git's,

because it seals the packfile Git has already compressed instead of encrypting each file. The same inability to read a repository makes `clone` the one operation whose cost is not amortized: it overtakes Git's beyond a hundred heads. We also measure the two operations Git has no counterpart for, epoch rotation at 25–29 ms and consolidation at 0.19 ms per head. Further experiments cover membership scaling, head-log contention, eleven attack scenarios, and history-window exposure on real repositories.

D. Organization

Sections II and III cover related work and the threat model. Sections IV and V present the design—including human key management (section IV-J)—and the UC theorem; section VI gives evaluation. Appendices hold preliminaries, full functionality and proof, design/eval supplements, PQ parameters, and residual limitations (appendix H).

II. Related Work

Table I compares SafeHub with Git forges and encrypted-Git systems. E2EE storage, key progression, and MLS appear in the survey below, not in the table. Prior work supplies the pieces separately—encrypted Git on unmodified forges (SGitChar/SGitLine), interval access to shared persistent data (GKP), and administrated membership with individual admin signatures (A-CGKA) [6], [8], [15]. SafeHub contributes the Git-specific composition: administrated MLS membership, per-invite history windows with join shallow snapshots, encrypted RefHead integrity with Compare-based fork detection, a NIST PQ Category 5 UC/QROM argument, and a full-stack prototype. We survey the six lines the design draws on, and in each case state what it does not claim.

a) Forge integrity: gittuf [18] decentralizes forge *integrity*: Update-Framework (TUF) [19]-style thresholds and delegations, an append-only Reference State Log, and client-verified policy over branches, tags, and paths, including on GitHub/GitLab. Repository objects remain host-readable. SafeHub provides the complementary property of host confidentiality while preserving Git semantics and cryptographically controlled membership, with integrity of the mutable ref map as a UC event (Forked). The lines compose: a deployment can witness RefHead digests in a gittuf-style log while keeping payloads encrypted under SafeHub epochs. The wider supply-chain attestation line—in-toto and Sigstore [20], [21]—is orthogonal in the same way: its attestations bind *who* produced an artifact and *how*, over plaintext the verifier can read, whereas SafeHub hides the artifact from the host. A SafeHub deployment can carry such attestations as MLS application records beside the review traffic, but their transparency-log publication is a plaintext channel and is out of this paper's scope. Self-hosting [22], [23] relocates rather than removes the trusted host, which is the assumption we set out to discharge.

b) E2EE Git and VCS: Li et al.'s SGitChar/SGitLine [6] (CCS'25) encrypt Git on *unmodified* forges: diff-then-encrypt keeps ciphertext proportional to the edit, and game-based definitions give confidentiality and repository unforgeability against a compromised Git server. Read or write access is granted by the owner wrapping one repository key, so a grant lasts until the repository is deleted, revocation is not in their model, and history is all-or-nothing. Gringotts [24] builds an E2EE VCS whose access control combines attribute-based encryption (ABE) with elliptic-curve digital signatures (ECDSA), under a covert-adversary rather than fully malicious server. The classical tools fix the key material instead: git-crypt [3] encrypts selected working-tree paths under a static symmetric key, git-remote-gcrypt [4] encrypts the whole remote under Pretty Good Privacy (PGP) keys, and Keybase git [5] uses per-team keys rotated by the Keybase service. None of these systems provides the four features SafeHub composes. The first two are administrated cryptographic membership whose removals carry analyzed post-compromise security—a group operation rather than a manual bulk re-key—and per-invite history windows for a contractor or CI runner. The last two are a fork-consistent encrypted manifest over the whole ref map and a UC version-control functionality. SGit authenticates refs per operation but carries no client-held anti-rollback anchor across operations, so a rolled-back or forked tip history is not detected the way an encrypted, hash-chained RefHead plus Compare detects it—which is what its ○ records. The classical tools leave the tip to the host. Deployment differs on an axis the table does not score: `git-crypt`, `git-remote-gcrypt`, and gittuf run against an unmodified forge, whereas SafeHub, Keybase, SGit, and Gringotts each require their own server. SGit argues standalone games. Gringotts argues a covert-server model. Encrypting the remote surface under per-repo MLS epochs also leaves the repository intact: old objects stay under prior epochs and new pushes seal under the new keys, so nothing has to be decrypted and re-encrypted in bulk (sections IV and V).

c) E2EE collaborative storage: Outside version control, a long line of work encrypts shared state against an untrusted server. SiRiUS [25] provides cryptographic sharing over untrusted storage; Metal and Titanium [26], [27] hide access metadata through multi-server oblivious RAM; Backendal et al. [14] formally analyze single-client E2EE cloud storage. These systems establish the confidentiality and metadata-hiding techniques we reuse, but they expose a file or object interface: they do not provide Git refs, force-push policy, history-limited clones, or administrated group collaboration, and the mutable, concurrently advanced ref map is precisely where our fork-consistency obligation arises.

d) Messaging, MLS/CGKA, and key progression: SafeHub takes its membership and window layers from this literature. MLS and CGKA analyses [7], [9], [16], [28], [29] underpin continuous group keys, and TreeSync [30] machine-checks authenticated MLS group state. We adopt administrated CGKA (A-CGKA) [8], which adds cryptographic administrator control, as our membership layer. Our $\mathcal{F}_{\mathsf{acgka}}$ hybrid assumption is exactly that interface under secure erasure and the QROM (path secrecy [17]; administration [8]). appendix C proves the *storage and VCS layer* above that hybrid in the

UC framework [31], including adaptive opening of $\mathsf{win}(u)$ under erasure. The window layer comes from key progression. Droplet [32] introduced dual-key regression for encrypted streams, and Group Key Progression [15] formalizes interval access to shared persistent data via dual key regression. SAFEHUB instantiates that progression for Git history windows (forward-only/ join shallow). As in GKP, $\mathsf{win}(u)$ is a range of epochs, not a position in a stream. Join shallow is a separate snapshot so a forward-only member can check out without older history. Concurrency at the membership layer has its own literature: CoCoA [33] admits concurrent group-key updates, and fork-resilient CGKA [34] tolerates divergent group states. SAFEHUB takes the opposite trade. Epoch changes are serialized through the same compare-and-swap head log that orders pushes. The group therefore has one authoritative epoch sequence, and divergence surfaces as the storage-layer Forked event rather than as a membership-layer fork.

e) Key transparency and fork consistency: Key transparency [35], [36] can optionally witness head hashes; it is not required for theorem 1. SUNDR [37] and SPORC [38] established fork consistency for untrusted stores, and the lineage continues through Venus, Depot, Frientegrity, Caelus, and Ghostor [39]–[43], whose witnesses, gossip, and periodic attestation address exactly the "detection needs client interaction" gap. SAFEHUB applies that approach to encrypted, epoch-bound ref manifests: the checkpoint-witness and periodic-Compare mechanisms transfer directly, since they act on head hashes rather than on plaintext, whereas server-side serialization does not transfer, because our server is untrusted and blind.

f) Attacks and PQ scope: Deployed E2EE cloud failures in MEGA, Nextcloud, and the CCS'24 ecosystem study [11]–[13] motivate whole-system proofs over per-layer heuristic arguments, since each of those breaks composes individually reasonable mechanisms into an insecure whole. We instantiate MLS at NIST PQ Category 5 [44], [45] and prove the storage layer in the QROM [46]–[50] (theorem 1 and appendix E-A); the full QPT/QROM argument is in appendix C.

TABLE I: Comparison of Git-based systems. ✓= property present; ○ = partial / weaker model / optional; × = absent. **E2EE**: contents confidential vs. a malicious host; **PCS**: cryptographic membership with post-compromise security; **Win.**: history windows / interval grants; **Ref**: encrypted tip integrity / fork consistency; **PQ**: PQ-ready suite; **Formal**: UC or formal malicious-server proof; **Path**: path-level policy. Forge-agnostic drop-in deployment is discussed in the text rather than scored here, being independent of the other columns.

System	E2EE	PCS	Win.	Ref	PQ	Formal	Path
GitLab self-hosted [22]	×	×	×	×	×	×	○
git-crypt [3]	○	×	×	×	×	×	×
git-remote-gcrypt [4]	✓	×	×	×	×	×	×
gittuf [18]	×	×	×	○	×	×	✓
Keybase git [5]	✓	×	×	○	×	×	×
SGitChar/Line [6]	✓	×	×	○	×	✓	×
Gringotts [24]	✓	×	×	○	×	○	×
SAFEHUB (this work)	✓	✓	✓	✓	✓	✓	×

III. SYSTEM AND SECURITY MODEL

A. System setup

A repository involves *members* $P_1, \ldots, P_n$ (devices), a distinguished subset of *administrators*, and a server Srv providing blob storage, an ordered head log, and MLS delivery. All communication passes through Srv. Members authenticate via credentials issued by a certificate authority $\mathcal{F}_{\mathsf{ca}}$.

Membership advances in *epochs*: each membership change or key rotation opens epoch e, whose MLS group secret is exported into a seed ss_e for the key-progression chains and an epoch MAC/collaboration key mk_e. Dual key regression (DKR) expands ss_e into per-epoch keys $\{K_e\}_{e \geq 1}$. Each member u holds compact interval tokens for a window $\mathsf{win}(u)$, a union of intervals $[from, to]$, and can derive K_e only for $e \in \mathsf{win}(u)$ (an invite grant is $[h, \infty)$; a removal caps the open interval). Every push seals under a fresh data-encryption key (DEK); K_e derives a per-head seal subkey K_e^{seq} that wraps the DEK and encrypts the refs. Each member-device occupies one MLS leaf with its own ML-DSA-87 signature key. Full notation: table VIII and appendix D-A.

B. Adversary model

The server is malicious: it may read, drop, reorder, replay, and fork messages, and may collude with corrupted and removed members. Security is argued against quantum polynomial-time (QPT) adversaries under computational UC [31] with quantum-accessible random oracles $\mathcal{F}_{\mathsf{ro}}$ (QROM) [46], [47], [49]. Git, MLS, and secure-erasure building blocks, together with the full UC/QROM setup and $(\mathcal{F}_{\mathsf{acgka}}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{ro}})$-hybrid interface, appear in appendices A and A-D.

Definition 1 (Secure erasure)**.** *An honest device* securely erases *ephemeral epoch material: after deriving and applying* (ss_e, mk_e)*, wrapping any required* DEK*, updating interval* DKR *tokens for* $\mathsf{win}(u)$*, and completing the push/fetch that consumes those exports, it zeroizes* ss_e*,* mk_e*, derived* K_e *values no longer needed for retained intervals, and past MLS epoch secrets the local stack no longer requires. It retains only persistent MLS member-device/group state and the* DKR *tokens authorizing* $\mathsf{win}(u)$*. A corruption therefore reveals exactly the party's* current, non-erased *state—idealized as the records in* $\mathsf{win}(u)$ *plus ongoing observation while corrupt.*

We prove adaptive member corruptions under definition 1 (the server is always adversarial). $\mathsf{ExposeState}$ reveals a device's state at a chosen time without ongoing observation—the interface used by PCS games after an honest post-exposure update. QPT adversaries with quantum-accessible $\mathcal{F}_{\mathsf{ro}}$, and the $(\mathcal{F}_{\mathsf{acgka}}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{ro}})$-hybrid interface, are summarized in appendices A and A-D; $\varepsilon_{\mathsf{acgka}}$ for the concrete Category-5 suite is currently *heuristic* (theorem 9). Assumptions and PQ margins: § VI-0a and appendices C and E.

TABLE II: Core notation (full table: table VIII and appendix D-A).

Symbol	Meaning
RefHead	encrypted, hash-chained ref manifest
win(u)	history window for user u
forward-only	grant from join epoch onward
ss_e, mk_e	MLS-exported epoch seed / MAC key
K_e, K_e^{seq}	epoch key / per-head seal subkey
DEK	fresh per-push data-encryption key
$\mathcal{F}_{\mathsf{safehub}}$, Π_{safehub}	ideal functionality / protocol
$\mathcal{F}_{\mathsf{acgka}}$, $\mathcal{F}_{\mathsf{ca}}$, $\mathcal{F}_{\mathsf{ro}}$	CGKA / CA / RO hybrids

$\mathcal{F}_{\mathsf{safehub}}$ reveals operation type, ciphertext sizes, timing, repository identifiers, authenticated control-channel identity, compare-and-swap and head-sequence metadata, MLS group/device counts, and (when distinguishable) history-grant class—not plaintext content, paths, or review text (section IV-H).

C. Goals

SAFEHUB targets four joint goals against a malicious host. *Confidentiality:* repository objects, paths, commit metadata, refs, and collaboration text remain hidden from the server except for the explicit leakage of section IV-H. *Membership and windows:* only administrators gate invites and removals; each invite binds a history window win(u). *Ref integrity:* a client rejects any tip that rolls back or contradicts its own accepted anchor; two clients that have accepted divergent views detect the divergence when they compare checkpoints. Force-pushes require administrator co-signatures. *Post-compromise healing:* if a device's keys leak, honest devices then run an epoch key update (Rotate) and erase the superseded material of definition 1, retaining the tokens that authorize win(u). New content cannot be opened with the leaked keys. Dropping a member (Remove) caps that member's history window, stops the host from treating them as still in the group, and advances the epoch in the same transition. Tokens the member retained still open the epochs its capped intervals already authorized—the scope of forward secrecy, not a missing rekey. Availability is not a cryptographic goal: withholding is detectable but not preventable.

IV. SYSTEM DESIGN AND PROTOCOL

SAFEHUB is a Git hosting system whose server stores ciphertext and sequencing metadata only. Repository members hold the keys, and native Git workflows—`clone`, `push`, `fetch`, `branch`, `merge`, `rebase`, and `blame`—remain intact via the `sit` remote helper and `shub` administration CLI. Security goals appear in section III; the formal UC treatment in section V and appendices B and C, and its post-quantum instantiation in § VI-0a and appendix E; preliminaries are in appendix A. Table II collects the notation this section uses; full notation, secondary figures, numbered protocol listings, and the complete cookbook are in appendix D.

A. Design overview

The three observations of section I fix the architecture. Content- addressed Git objects already give Merkle integrity of history contents, so a malicious host that cannot forge hashes can attack only the mutable ref-name→tip map; an encrypted, signed, hash-chained RefHead protects that map without re-encrypting the object store on every push. Membership is an administrated group problem—MLS/A-CGKA already models client-verified invites, removals, and device revocations. History need not be all-or- nothing: DKR tokens unlock the epochs of a granted interval, whose boundaries are segment boundaries, for forward-only collaborators without a second ACL channel.

a) Design exclusions: These observations exclude four otherwise natural designs that would either return authority to the host or cost more than they save: path-level ACLs inside a single MLS group; full DAG re-encryption on every membership change; plaintext branch names on the server; and an out-of-band key-distribution channel for history grants (Welcome payloads carry interval tokens so membership and windows share one authentic path).

B. Architecture

A SAFEHUB deployment has three roles (fig. 1):

- **Members (devices).** Each device runs a local working copy and a SAFEHUB client (typically a sit remote helper). The client holds MLS/A-CGKA group state; DKR interval tokens for that member's history window; anti-rollback anchors (last-seen head sequence and epoch authenticator) in the OS keychain; and ordinary local objects compatible with Git.
- **Administrators.** A distinguished subset of members whose credentials appear on the group's admin roster. Only administrators may invite, remove, rotate keys for membership reasons, update the admin roster (SetAdmin), or co-sign non-fast-forward ref updates. Admin authority is cryptographic: every client verifies admin signatures against the roster in the MLS group context; the server has no authorization role.
- **Server.** A malicious-capable host providing (i) a content-addressed blob store for encrypted bundle chunks, (ii) an ordered *head log* of encrypted ref manifests appended by compare-and-swap, and (iii) MLS delivery. The server never receives plaintext objects, ref names, or group secrets.

Trust boundary. Confidentiality and integrity of repository state are end-to-end among members. The server is relied on for availability alone—and even withholding is detectable but not preventable. Credentials bind identities to ML-DSA-87 verification keys through $\mathcal{F}_{\mathsf{ca}}$. Figure 2 separates the untrusted host, the org-trusted plaintext fleet, and member devices (section I and appendix H).

a) Four protocol layers: Each repository is keyed by composing: (1) an administrated CGKA group (MLS) for membership and epoch secrets; (2) a DKR key-progression layer mapping epochs to transport and ref keys with per-invite history windows; (3) encrypted git-bundle transport under

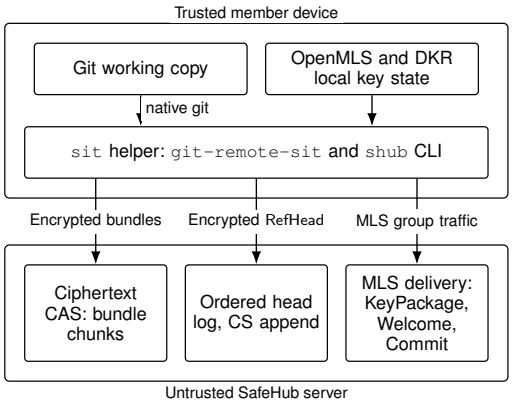

Fig. 1: SAFEHUB architecture. The untrusted server sequences and stores ciphertext but never holds repository keys. CAS = content-addressed storage; CS = compare-and-swap.

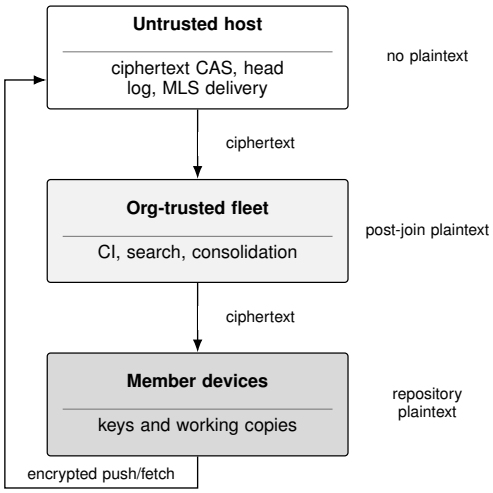

Fig. 2: Trust domains, ordered top to bottom by increasing plaintext exposure (gray shading): the host stores ciphertext only; the org-trusted fleet sees post-join plaintext as forward-only members; developer devices hold keys and working copies. CAS = content-addressed storage. Arrows are ciphertext delivery, not a trust chain: the org-trusted fleet is a set of forward-only members rather than a relay, and member devices reach the host directly.

fresh data-encryption keys (DEK s) wrapped under the current epoch's per-head seal subkey; and (4) an encrypted, device-signed, hash-chained, epoch-bound *ref manifest* (RefHead) that is the sole mutable map from ref names to tips. Collaboration traffic (pull requests, issues, review, and CI verdicts) travels as MLS application messages in the same group.

C. Git workflows

a) Create repository: An administrator initializes a local repository, creates an MLS group with themselves as the sole admin member-device, exports the first epoch secrets, seeds the DKR chains, and publishes an initial encrypted head to the server. The remote URL has the form `sit://host/org/repo`.

b) Clone: `sit clone` downloads the latest accepted RefHead; verifies the pusher signature, the epoch MAC under mk_e, and SHA-512 roots of referenced bundles; unwraps DEK s using keys derivable from the member's interval tokens; decrypts bundles; and imports them into the local object store. Refs are set from the decrypted manifest, and the accepted entry becomes the client's first anti-rollback anchor. Members with a *forward-only* history window receive a join shallow snapshot and cannot decrypt pre-join bundles.

c) Fetch and pull: `sit fetch` repeats those checks incrementally and additionally verifies chain descent from the client's last-seen anchor, decrypting only the bundles appended after it; refs are updated from the decrypted manifest and the new anchor is persisted. `sit pull` is that fetch followed by an ordinary Git merge in the working copy.

d) Merge: `sit merge` (and `git merge` on the same checkout) runs on the decrypted working copy once objects are already local—typically after `sit fetch`. It does not contact the host. The merge commit is sealed only when a later `sit push` publishes the new tip as an ordinary incremental bundle.

e) Rebase: `sit rebase` likewise rewrites commits locally on the decrypted DAG. New objects are sealed on the subsequent push; if the rewritten tip is not a fast-forward of the published ref, the update is a force-push and requires an admin co-signature, which verifiers check as in section IV-E. There is no host-side merge or rebase protocol: the server only stores ciphertext and sequences RefHead entries.

f) Push: `sit push` seals one incremental bundle and appends one authenticated RefHead entry (fig. 3):

1) $B \leftarrow$ `git bundle create`(lastAck..refs).
2) $\mathsf{DEK} \leftarrow \{0,1\}^{256}$; $C \leftarrow \mathsf{E}(\mathsf{DEK}, B)$ in chunked AEAD (default 4 MiB) with associated data binding $(\mathsf{repo}, \mathsf{push_id}, i, n)$; $\rho \leftarrow \text{SHA-512}(C)$.
3) Upload content-addressed chunks.
4) Form

$$\mathsf{RefHead}' = \big(\mathsf{seq}{+}1,\ \mathsf{E}(K_e^{\mathsf{seq}}, \mathsf{refs}'),\ \rho,\ \mathsf{wrap}(K_e^{\mathsf{seq}}, \mathsf{DEK}),\ H(\mathsf{RefHead}),\ e,\ \mathsf{MAC}(mk_e, \cdot),\ \mathsf{sig}\big)$$

and append it by compare-and-swap on $H(\mathsf{RefHead})$.

A concurrent push loses the compare-and-swap; the client fetches, merges with ordinary Git, and retries, so the head log rather than the host settles the order. Non-fast-forward updates require an admin co-signature; clients reject force-pushes lacking one. Verifiers recompute the fast-forward relation from decrypted refs, and a forward-only verifier limited to a join shallow snapshot that cannot decide ancestry treats the update as non-fast-forward and rejects it without a co-signature. Ref names appear only inside the encrypted manifest. Round-trip timing detail is in fig. 14 and appendix D; full Push/Fetch/Membership listings are in appendix D-C.

D. Administration and collaboration

a) Invite, remove, rotate: Administrators invite collaborators by KeyPackage, choosing a history grant h: full history

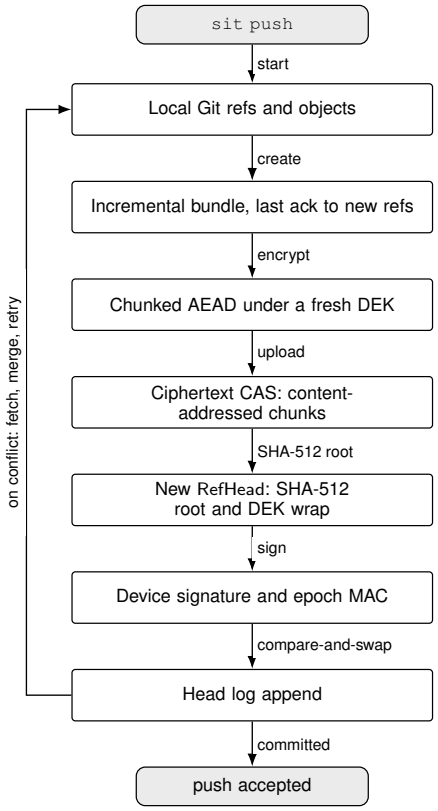

Fig. 3: Push path, read top to bottom from the `sit push` entry point. A client encrypts an incremental bundle, uploads content-addressed chunks, and atomically appends an authenticated RefHead; the run ends at *push accepted*. The single back edge on the left is the retry taken when the compare-and-swap loses, which returns to ordinary git merge. CAS = content-addressed storage.

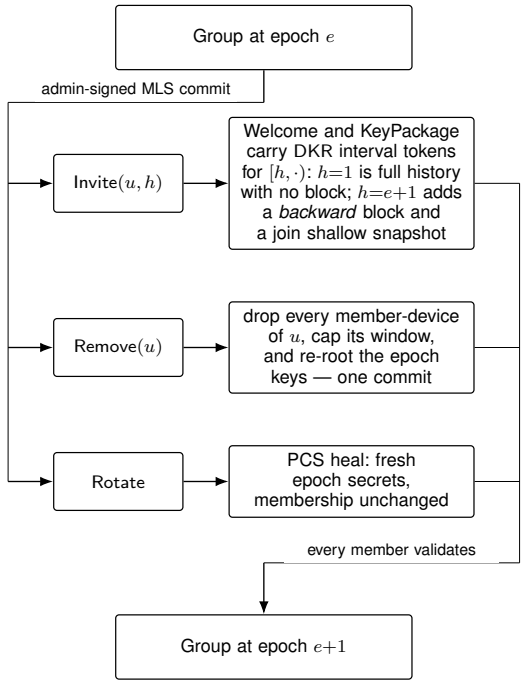

Fig. 4: MLS membership lifecycle. Each operation is an admin-signed, epoch-advancing MLS commit. Invite carries the history grant in the Welcome: a forward-only grant ($h{=}e{+}1$) adds a *backward* DKR block and a join shallow snapshot. Remove drops every member-device of u, caps its window, and re-roots the epoch keys in the same commit. Rotate is that epoch advance with membership unchanged. SetAdmin is a fourth admin-signed epoch-advancing commit, omitted here because it changes no window.

($h{=}1$) or forward-only ($h{=}e{+}1$). Remove is an admin-signed, epoch-advancing Commit: in one transition it drops every device leaf of the removed user, caps every open interval of win(u) at the pre-Remove epoch, and re-roots the epoch exports, so the later epochs are unreachable to the removed party even under server collusion. Capping the control plane without advancing the epoch would leave new payloads openable by retained state, which is why the rekey is part of the operation rather than a separate step. Rotate is the same epoch advance with membership unchanged, healing post-compromise security. SetAdmin advances the epoch without changing any window. All four are admin-signed MLS commits; every member validates before merging (fig. 4).

b) Pull requests and CI: Code review is not a second plaintext channel. Opening a pull request, commenting, and requesting changes are structured MLS application messages (type, ref, base, head, body) that reference commits by hash. Optional per-PR subgroups use resumption PSKs so review traffic can be scoped. The host delivers ciphertext envelopes only. Automation joins as ordinary members with forward-only windows: they can build and index content written after their join epoch, never before. Verdicts are application messages; revoking a compromised runner is one admin remove commit.

c) Consolidation: Consolidation is the admin-gated maintenance operation that compacts the encrypted history so stored ciphertext stops growing with push count. Each component is repacked under the epoch key that already sealed it, so compaction is per-epoch and no member gains access to an epoch outside win(u); re-sealing the history under the current epoch key would instead let any holder of that key read incorporated pre-join history, which is forbidden. It is therefore storage compaction for honest hosts rather than cryptographic erasure (appendix F-A). An administrator (or m-of-n quorum) issues a receipt, and the member performing the work—often a CI runner under policy—replays the incremental bundle log into that snapshot, marks superseded blobs collectible, and publishes it as an ordinary push. The snapshot carries a domain-separated digest binding the replaced prefix so window-limited verifiers can check it.

E. RefHead *integrity and force-push policy*

The mutable attack surface is the map from ref names to tips. Each accepted push appends an encrypted, device-

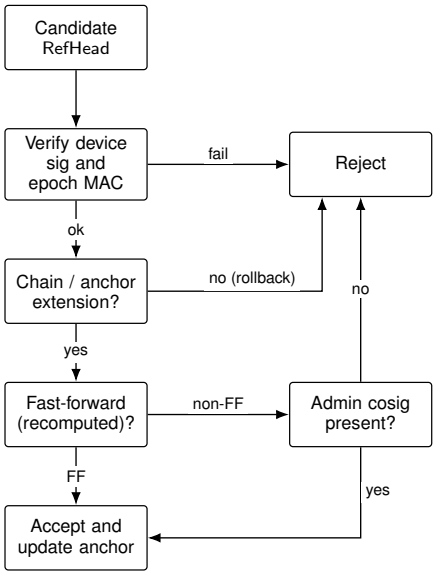

Fig. 5: RefHead verification state machine: signature and epoch MAC, chain/anchor extension, recomputed fast-forward, and admin co-signature on non-fast-forward updates. This path rejects rollback behind the verifier's own anchor; cross-party divergence surfaces on COMPARE, not here.

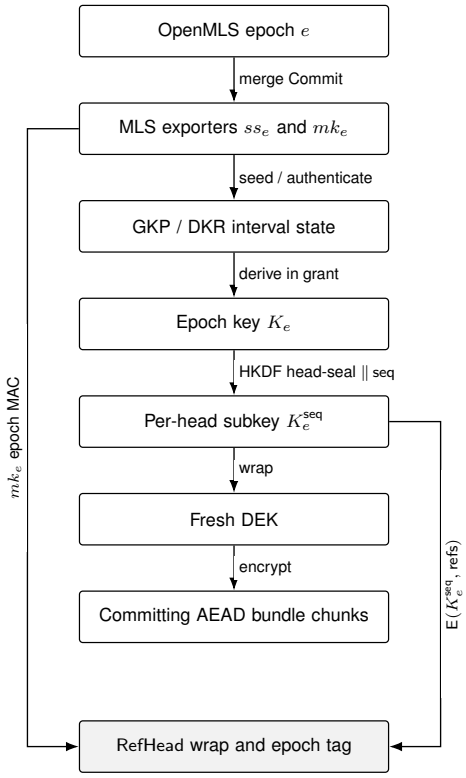

Fig. 6: Key schedule. OpenMLS exporters authenticate and seed per-epoch state; GKP/DKR interval tokens derive only authorized K_e values. K_e is never an AEAD key itself: each head derives a per-seq subkey K_e^{seq} that wraps the bundle DEK and encrypts refs, while mk_e authenticates the RefHead.

signed, hash-chained, epoch-MACed RefHead entry. Clients store an anti-rollback anchor $(\text{seq}, H(\text{RefHead}))$ in the OS keychain and reject any tip that is not a chain extension of that anchor. Chain extension is unconditional: an admin co-signature authorizes a non-fast-forward change to the decrypted refs, never a RefHead that fails to extend its predecessor. Each client additionally exports an authenticated RefHead checkpoint; when two honest clients run Compare over the authorized projection— the sequence range both windows are entitled to hold, aligned by seq and checked against per-epoch commitments—divergence surfaces as Forked (lemma 9 and theorem 4). Disjoint windows are incomparable, not consistent. Detection is therefore local and unconditional against a client's own anchor, and cross-party on Compare: a client that has accepted a divergent view learns of it when it next compares, not at the moment of acceptance. Checkpoints travel as MLS application messages, which the host may delay but cannot forge, so Forked follows once any overlapping honest checkpoint is delivered. Branch protection is a verifier property, not a server ACL: a malicious host can withhold updates, but a tip it rewrites is not one honest clients will continue to accept. Figure 5 summarizes the accept/reject checks.

F. Key schedule

On merging the commit that creates epoch e: $ss_e \leftarrow \text{Export}(\texttt{transport}) \in \{0,1\}^\lambda$ and $mk_e \leftarrow \text{Export}(\texttt{refs}) \in \{0,1\}^{256}$, both zeroized after use (definition 1). Interval DKR seeds a segment token from ss_e and derives $K_e \leftarrow \text{HKDF}_{\text{SHA-512}}(\text{token}, \texttt{safehub-v1:dkr-epoch:}e)$ only for epochs inside the member's granted interval. Rotate at e inserts a forward block; a forward-only join inserts a backward block (fig. 6). Key lifetimes: appendix D-B.

G. Client and server state

Each member holds MLS/A-CGKA state; DKR interval tokens; the local Git repository; and the last-seen $(\text{seq}, H(\text{RefHead}))$ with the last epoch authenticator. The server holds ciphertext blobs, the head log, KeyPackages, and MLS framing—never plaintext objects, branch names, or epoch keys.

H. Leakage surface

By design, the server learns operation type flags, ciphertext lengths, timing, repository identifiers, authenticated account/device linkage on the control channel, compare-and-swap outcomes, head sequence numbers, and MLS framing (including group/device counts). Full-history versus forward-only invitations are treated as distinguishable under size/timing leakage unless padded. Paths, commit messages, diffs, branch names, and review text remain hidden (section V, fig. 11, and appendix B).

I. Integration cookbook

Table III maps familiar forge capabilities to SAFEHUB answers. Capabilities that require host-side plaintext move to

TABLE III: GitHub-class capability → SAFEHUB answer (keys and residual). S = supported; C = client-side / org fleet; × = not supported. LFS = Git Large File Storage; PR = pull request.

Capability	SH	Keys	Residual
Private push/fetch	S	K_e, DEK	size/timing leak
Branch protection	S	admin cosig	malicious admin
Force-push gate	S	admin cosig	none
CI builds	C	forward-only member	runner plaintext
Code search	C	indexer member	index plaintext
PR / review text	S	MLS app msg	framing leak
Actions-as-a-service	×	×	out of scope
LFS / partial clone	×	×	table X
Path ACLs	×	×	use gittuf/plain
Org secrets store	C	separate	not UC

the org-trusted fleet; cryptographic membership and RefHead checks remain on the client (appendix H).

J. Human key management

Because no plaintext reaches the host, provider-mediated recovery becomes an organizational procedure. After device loss, revoke the device and invite its replacement with a forward-only or full-history grant; previously authorized ciphertext remains decryptable. Admin keys should use hardware or offline custody: a compromised administrator can change membership and authorize force-pushes until rotation. Legal-hold escrow is an ordinary full-history member and expands the plaintext coalition rather than creating a server backdoor.

V. UC SECURITY

We prove that Π_{safehub} UC-realizes $\mathcal{F}_{\mathsf{safehub}}$ against a malicious server and adaptive member corruptions under secure erasure, for QPT environments with quantum access to $\mathcal{F}_{\mathsf{ro}}$. Full functionality, simulator, hybrids, and games: appendices B and C; hybrid preliminaries: appendix A-D; PEKS scope: appendix H.

a) Threat and ideal functionality: Parties, QPT/QROM assumptions, adaptive corruptions under secure erasure (definition 1), and the $(\mathcal{F}_{\mathsf{acgka}}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{ro}})$-hybrid model are as in section III. Honest members run Π_{safehub}; in the ideal world they interact only with $\mathcal{F}_{\mathsf{safehub}}$ (fig. 11 and appendix B): member/admin sets, windows $\mathsf{win}(u)$, an append-only push sequence, and refs. Leakage is $\ell(op)$ (section IV-H); Forked fires on FETCH against a party's own anchor and on COMPARE between parties; corruption reveals current non-erased $\mathsf{win}(u)$.

b) Interface in brief: $\mathcal{F}_{\mathsf{safehub}}$ maintains per-repository state $(e, \mathsf{M}, \mathsf{Adm}, \mathsf{win}, L, \mathsf{refs}, d_P)$ and, on every accepted operation, sends $\ell(op)$ to $\mathcal{S}$ before committing at the linearization point the handler names. Its ten commands fall into four groups. CREATE, INVITE, REMOVE, ROTATE, and SETADMIN are admin-gated and move membership, epoch, and the history windows together: INVITE unions $[h, \infty)$ into $\mathsf{win}(u)$ for $h \in \{1, e{+}1\}$, and REMOVE is itself the rekey, capping every open interval of $\mathsf{win}(u)$ in the same transition that refreshes the epoch exports. PUSH, APPLY, and CONSOLIDATE extend or replace state: a push appends to L at the compare-and-swap linearization point, and an application record or a payload-preserving snapshot is recorded without widening any window. Admin authorization is required exactly when the *recomputed* $\mathsf{ff}{=}0$, or when the operation consolidates; the declared nonff flag is advisory and a push whose flag disagrees with the recomputation is rejected, for corrupt and honest callers alike. FETCH delivers the prefix authorized by $\mathsf{win}(P)$ and raises Forked on a rollback behind the caller's own anchor d_P, while COMPARE raises it on exchanged checkpoints that are not comparable within the shared window. CORRUPT and EXPOSESTATE open $\mathsf{win}(u)$ for a member-device of u, secure erasure leaving ephemeral exports outside that window unavailable. The authoritative handlers—honest and corrupt branches, leakage, shadow heads, forward-only grafts, and the adaptive-corruption bookkeeping the proof relies on—are fig. 11 and appendix B.

Theorem 1 (UC in the QROM, adaptive under secure erasure)**.** *Assume QPT EUF-CMA of ML-DSA-87; QPT IND-CPA, INT-CTXT, and key-commitment of the equivocable Encrypt-then-MAC transport; QPT SUF-CMA of HMAC-SHA-512-256 in the QROM; quantum collision resistance of SHA-512; and honest-client erasure. Then* Π_{safehub} *UC-realizes* $\mathcal{F}_{\mathsf{safehub}}$ *in the* $(\mathcal{F}_{\mathsf{acgka}}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{ro}})$*-hybrid QROM against a malicious server and adaptive member corruptions, with advantage bounded by* (1)*. The bound is stated over the ideal* $\mathcal{F}_{\mathsf{acgka}}$*; it carries no term for any concrete group-key protocol.*

$$\begin{aligned}&\left|\Pr[\mathsf{EXEC}_{\Pi_{\mathsf{safehub}},\mathcal{A},\mathcal{Z}}{=}1] - \Pr[\mathsf{EXEC}_{\mathcal{F}_{\mathsf{safehub}},\mathcal{S},\mathcal{Z}}{=}1]\right| \\ &\le \varepsilon_{\mathsf{acgka}} + q_p(\varepsilon_{\mathsf{ae}} + \varepsilon_{\mathsf{mac}}) + q_s\varepsilon_{\mathsf{sig}} \\ &\quad + q_{\mathsf{seg}}\varepsilon_{\mathsf{ow2h}} \\ &\quad + O\big(N_{\mathsf{prog}}\, q_{\mathsf{ro}}^2/2^{\lambda}\big) + q_v/2^{256} + \varepsilon_{\mathsf{coll}}.\end{aligned} \tag{1}$$

Terms: appendix C-A, definition 4, and table IX. The programming budget $N_{\mathsf{prog}} = N_{\mathsf{pad}} + N_{\mathsf{dkr}} + N_{\mathsf{open}}$ counts the oracle points the simulator reprograms—transport pads, DKR nodes, and adaptive openings—since a single collision term would not cover a multi-point reprogramming (appendix C-A); q_v counts epoch-tag verifications, whose multi-user term is separate from the outer AEAD's $\varepsilon_{\mathsf{mac}}$. Storage layer above MLS; $\varepsilon_{\mathsf{acgka}}$ replaces $\mathcal{F}_{\mathsf{acgka}}$ only under corollary 1.

Corollary 1 (Instantiating $\mathcal{F}_{\mathsf{acgka}}$, conditional)**.** *Instantiating* $\mathcal{F}_{\mathsf{acgka}}$ *with an A-CGKA whose adaptive labeled-export and Welcome-confidentiality advantage is* $\varepsilon_{\mathsf{acgka}}$ *adds that term to* (1)*. For the prototype's Category 5 MLS suite this quantity is an assumption, not a proved bound: the available analyses are selective or generic-group, and depth-dependent for adaptive CGKA (theorem 9 and appendix H).*

Corollary 2 (Guarantees delivered)**.** *Realizing* $\mathcal{F}_{\mathsf{safehub}}$ *yields history-window soundness; rollback rejection against a party's own anchor and fork detection on* COMPARE*; force-push/consolidation authorization; PCS/FS after any hon-*

est epoch-advancing Commit, Remove included; and a key-committing transport (appendix C-G).

VI. Implementation and Evaluation

The prototype[1] consists of a `sit` remote helper and the `shub` administration command. Because the helper carries Git's transport over `sit://`, unmodified `git` drives it and the whole set of Git operations remains available; the evaluation below exercises push, pull, fetch, clone, merge, rebase, and force-push. Both are built on a vendored OpenMLS [51] fork instantiated at a NIST PQ Category 5 private-use ciphersuite (`MLS_256_MLKEM1024_AES256GCM_SHA512_MLDSA87`), the equivocable committing transport AEAD of theorem 6, and dual key regression. The server provides three services: a blob store, a compare-and-swap head log, and MLS delivery. Unmodified `git` reaches the server through `git-remote-sit`. Encryption applies to a whole repository or not at all (table X and appendix H).

a) Post-quantum instantiation: SafeHub uses no classical public-key algorithm on the confidentiality or authorization path. Query budgets and security margins are given in appendix E; we require λ=384 for DKR and random-oracle nodes and do not accept 256-bit collision digests. Post-quantum KEM traffic arises from membership changes rather than from steady-state pushes, so its cost is reported separately in section VI-D.

b) Measurement method: All measurements were taken on AWS `c8g.8xlarge` instances (Graviton4, 32 vCPU, 61 GiB, Amazon Linux 2023, `rustc` 1.97.1). Clients and remotes occupy separate hosts within one subnet, measured at 0.28 ms round-trip time and 9.5 Gbit/s, so every timed operation crosses a real network path while remaining well clear of the network limit. Each reported value is the median of five repetitions, or three for gcrypt.

We compare six systems over a single HTTP transport: plain Git [52], `git-crypt` [3], `git-remote-gcrypt` [4], SafeHub, and SGitChar and SGitLine [6]. The last two were reimplemented from their specification because no artifact has been published (appendix H).

Three controls make the comparison sound. First, an operation is corrected by its own zero-payload floor measured on the same implementation. Second, every clone is unlocked and its contents compared against the source repository within the timed region: a `git clone` of a gcrypt remote returns a success status having checked out nothing, and `git-crypt` leaves ciphertext in the working tree until it is unlocked, so an unverified clone would credit both with work they had not performed. Third, an operation that a design cannot express is recorded as undefined rather than as zero. Where a comparison against plain Git alone is reported, Git is served by `git daemon` over native `git://`, without HTTP framing or CGI. That is the lowest-overhead transport available to Git and therefore the most demanding baseline.

[1]Implementation, evaluation harness, and published measurements: https://github.com/easwarvivek/safehub-artifact.

A single property explains most of the results. *A host that cannot read a repository cannot repack it. It therefore performs no unpacking when receiving a push and constructs no pack when serving a fetch, but it also cannot merge sealed bundles when serving a clone.* The same inability that reduces the cost of five operations is what raises the cost of the sixth.

c) Primitive costs: Every value in this paragraph is a median of 25 runs after three warm-up runs. Sealing and opening 1 KiB under the transport AEAD cost 11.9 and 11.8 μs; at 1 MiB they cost 8.09 and 8.07 ms. Computing a RefHead hash costs 1.04 μs, and verifying a 100-link RefHead chain costs 0.20 ms, which is negligible beside the transport it protects. The AES-256-GCM comparison in table XIII was taken with a separate in-process harness that reports the 1 MiB seal a few percent lower; the two sets of measurements are not pooled.

A. Operations

a) Push: To separate the cost of the payload from the cost of the repository it lands in, we hold the pushed payload at 50 KiB and the history at one head while the repository grows from 5 to 200 MB. Because the operation is identical at every point, a constant push time is the expected result, and a system whose push time grows is doing work proportional to the repository rather than to the payload. Git and SafeHub are constant to within 1.1× across this 40× range, and SafeHub is the faster of the two, at 42–46 ms against 97–103 (fig. 7a).

Three systems are not constant. gcrypt re-encrypts and re-uploads the entire remote on every push, so its cost is proportional to the repository by construction, rising from 426 to 909 ms. SGitChar and SGitLine reach 3.3 and 2.7 s respectively at 200 MB, a growth of 23 and 15×, because both read the whole working tree and compare it against a stored plaintext snapshot to find what changed, then rewrite that snapshot. It is this bookkeeping that scales, not the encryption: SGitChar encrypts only the delta it finds, and at a fixed 50 KiB payload neither system transmits more as the repository grows. `Git-crypt` ranges over 107–293 ms with no monotone trend.

Measured against Git on its native transport, the cost of encryption is visible but bounded. Over twelve repositories of 5–500 MB (fig. 8), push takes 1.45× as long as plain Git for a 0.05 MB delta and 0.98× for a 5 MB delta, with a marginal cost of 46.7 ms/MB against Git's 49.3. Encryption therefore does not appear per byte: the residual is a fixed per-push cost, comprising the DEK derivation, the AEAD seal, the epoch MAC, and one ML-DSA-87 signature, which a larger push amortizes. Varying the payload instead confirms this from the opposite direction: with the repository held at 4 MiB and the delta varied over 8 to 1024 KiB, the marginal cost is 0.0080 ms per additional KiB for SafeHub against 0.0155 for Git. Push is also insensitive to history depth: over 5 to 100 heads at a fixed 64 KiB delta it varies by 1 ms.

b) Pull and fetch: SafeHub is again the fastest of the six, at 50–63 ms against Git's 101–133, and its no-op fetch is constant at 31–33 ms against Git's 82–102 (fig. 7d). The reason

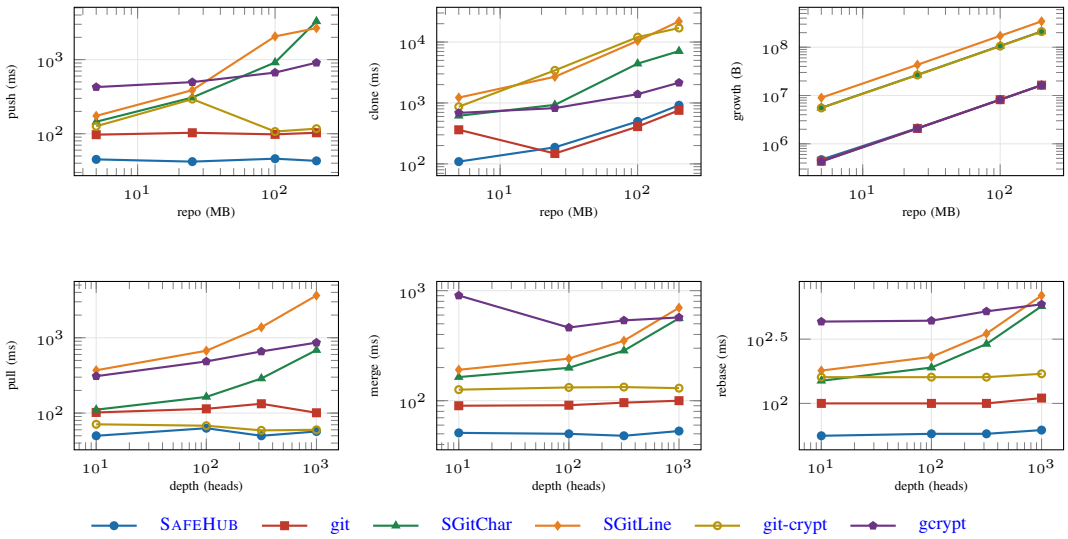

Fig. 7: Six systems, one transport. The top row holds the pushed payload at 50 KiB and the history at one head, varying only the size of the repository the push lands in: push *ought* to be flat there, so the row is a control that separates systems whose per-push cost is proportional to the repository from those where it is not. Git and SafeHub are flat to 1.1× across a 40× range; gcrypt re-encrypts the whole remote each time, and the SGit designs read and rewrite a whole-tree snapshot on every push, so all three scale with it. Clone and stored size depend on repository size by construction. The bottom row varies history depth instead. SafeHub is the fastest system on push, pull, merge and rebase and flat on both axes, and its storage growth matches plain Git's because it seals the packfile Git has already compressed. Clone is the exception: a host that cannot read a repository cannot repack it, so cost follows push count.

is that SafeHub serves ciphertext exactly as the writer sealed it, whereas Git reconstructs a pack for each request. The same difference explains the protocol floors, 18–30 ms against 64–74. Two earlier runs differed from these figures by as much as a factor of four. Three repetitions of a single configuration then returned 60 and 30 ms with no variation, in agreement with the reported values, so the discrepancy is attributable to those two configurations and not to the operation.

c) Merge, rebase, and force-push: The local Git work is identical in all six systems, so the difference between them lies in the push that follows. SafeHub is the fastest on all three operations and constant in history depth: merge takes 48–53 ms against Git's 90–100, rebase 56–62 against 100–110, and force-push 41–44 against 86–88 (fig. 7e–f, table IV). The administrator co-signature that a force-push requires is a single ML-DSA-87 signature, which is small compared with the transport it replaces. Force-push is undefined for the two SGit designs, because their ciphertext repository only ever appends: a rewrite of the plaintext history is recorded as a forward-only ciphertext commit, so the host can neither distinguish a rewrite from an ordinary push nor enforce branch protection.

d) Clone: Clone is the one operation whose cost the design does not amortize, and the one on which SafeHub is the slowest of the six. Its cost follows the number of pushes rather than the number of bytes, so it is measured against history depth (fig. 9). At 10 heads SafeHub is still the faster system, at 141 ms against plain Git's 267, because there is no pack to construct and too little history for replay to dominate. By 10^3 heads the ordering has reversed, at 13.1 s against 2.0.

Extending a single lineage to 10^4 heads at a fixed 64 KiB delta establishes the asymptotic behaviour: 173.9 s against Git's 3.21 s from a `gc`-packed remote, a factor of 54, with a least-squares fit of 17.3 ms per head against Git's 0.31. Leaving Git's own `receive.autogc` enabled halves the ratio to 19×, because part of the repacking is then charged to the replay instead of to the clone; it also makes Git's series non-monotonic in depth, so the packed configuration is the one to compare against. The measured cost per head *increases* with depth rather than remaining constant as modelled, because each replay leaves a packfile of its own and Git's object lookup scales with the number of packs. Refreshing a multi-pack-index periodically during replay removed the steepest part of that growth. Consolidation is the intended mitigation and does not yet provide it (appendix F-A), so these curves state the present, unmitigated cost.

e) Rotate and consolidation: Neither operation has a plain-Git counterpart. Remove is itself the rekey, and the epoch advance it performs costs 25–29 ms independently of repository size and of history depth. Consolidation constructs and signs its checkpoint receipt in time linear in the number of heads: 32 ms at 10 heads and 221 ms at 10^3, or 0.19 ms per head. Only the receipt path is measured (appendix F-A).

B. Storage

SafeHub seals the packfile after Git has compressed it, rather than encrypting each file before Git sees it. Its stored size is therefore at parity with plain Git: over a 200 MB

TABLE IV: Force-push, rotate and consolidation against history depth. A system absent from a block does not express that operation: force-push is undefined for the SGit designs, whose ciphertext repository only appends, so a host cannot tell a rewrite from an ordinary push; only SAFEHUB has rotate and consolidation.

operation	system	10	100	316	1000
`forcepush`	SAFEHUB	44	44	41	41
	git	86	88	87	87
	git-crypt	87	86	82	87
	gcrypt	429	460	483	606
`rotate`	SAFEHUB	29	27	25	26
`consolidate`	SAFEHUB	32	49	86	221

repository it adds 16.36 MB against Git's 16.38 (fig. 7c), and gcrypt, which likewise compresses before encrypting, adds 16.35. The three systems that encrypt each file add 210–343 MB for identical content, between 13 and 21 times as much, because ciphertext does not compress.

Measured against `gc`-packed Git over 5–500 MB (fig. 8), the ratio falls from 1.10× to 1.00×. The excess is almost entirely the head log: 557 KB of ML-DSA-87 signatures and chunk metadata for 100 pushes, against a measured excess of 541–566 KB, and it does not grow with repository size. The storage overhead is therefore attributable to post-quantum signatures rather than to encryption.

The same distinction governs the bytes each update places on the wire. With the edit fixed at 1 KiB and the file size increasing, Git and SGitChar remain constant because both transmit a delta, whereas `git-crypt` and SGitLine grow with the file because both re-encrypt it: at 8 MiB, Git transmits 843 bytes and `git-crypt` 8.39 MB for the same edit (table V). SAFEHUB admits no thin-pack measurement, its remote exposing no readable ref map, but on the growth measure it takes the constant side: 6.4 kB per update at 10 KiB and 6.7 kB at 8 MiB, against git-crypt's 11 kB rising to 8.39 MB. Figure 10 shows that this ordering is unchanged across four independent measures of update size—delta size, number of files touched, number of versions, and repeated revisions of one file—so it follows from what each design re-encrypts and not from a particular operating point. Spreading a fixed 1 KiB edit over 50 files reverses the comparison with Git, 21.9 kB against 54.6, because one sealed pack replaces the per-file objects.

C. What a forward-only grant withholds

A forward-only grant is justified only if the history it withholds is substantial. Across the `click`, `ripgrep`, and `git` histories it withholds 96–97% of every blob version the repository has ever contained, and between a third and a half of the paths that have ever existed.

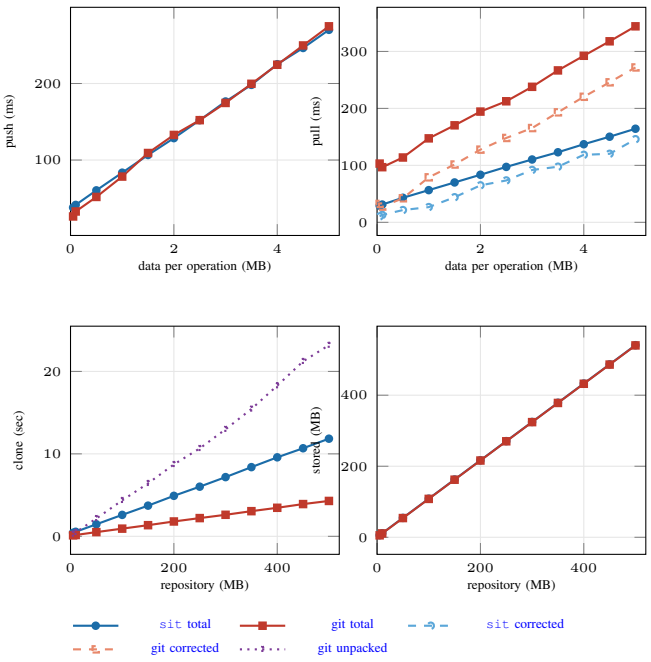

Fig. 8: SAFEHUB against plain Git over twelve repositories of 5–500 MB, 100 pushes each. Push and pull are plotted against the bytes each operation moves; clone and storage span the whole repository. Corrected pull subtracts each system's own no-op fetch. Push is shown as totals, having no measured no-op counterpart. Clone shows Git both `gc`-packed and as 100 pushes leave it. Git is served over native `git://`.

TABLE V: Bytes one **1 KiB edit** puts on the wire as the edited file grows, measured against the remote tip before the push. git and SGitChar track the *edit*; git-crypt and SGitLine track the *file*. SAFEHUB and gcrypt expose no readable ref map, so no thin pack can be measured for them.

file size (KiB)	10	100	1 M	4 M	8 M
git	590	601	640	727	843
SGitChar	1438	1450	1491	1594	1730
SGitLine	18 k	165 k	1.68 M	6.69 M	13.37 M
git-crypt	11 k	103 k	1.05 M	4.20 M	8.39 M

Repository	commits	blobs only in history	paths deleted before tip
`click`	3,328	5,304 (97.0%)	136 (45.6%)
`ripgrep`	2,327	4,898 (95.8%)	187 (46.6%)
`git`	85,328	160,474 (97.1%)	2,244 (32.2%)

Scans for credential markers found none in these scrubbed public mirrors, exhaustively for `click` and `ripgrep` and up to a 20,000-blob limit for `git`. The case for history windows over committed-and-later-deleted secrets therefore rests on measurements taken at the moment of leakage [10] rather than on these repositories.

a) Cost is not the only comparison: The six systems do not conceal the same information. Plain Git conceals nothing; `git-crypt` still exposes paths, the commit graph, and ref

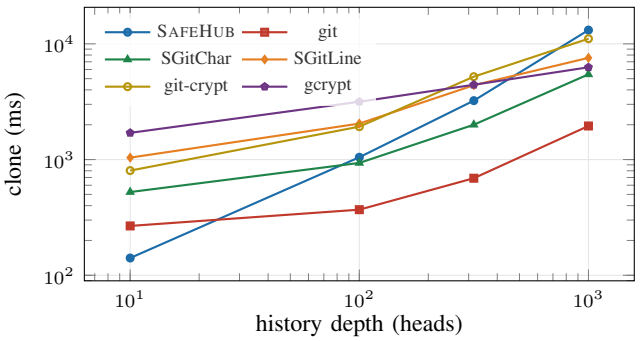

Fig. 9: Clone time against the number of commits in the history, six systems. Every clone is unlocked and its contents compared against the source inside the timer: a gcrypt clone exits successfully having checked out nothing, and git-crypt leaves ciphertext in the working tree until unlocked.

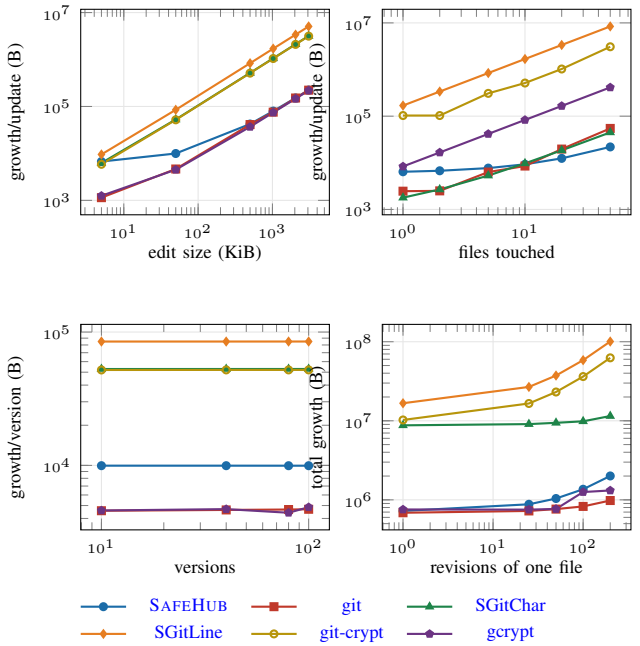

Fig. 10: How much storage the server gains from one update, as four different quantities grow: (a) the size of the edit, (b) the number of files one edit touches, (c) the number of versions pushed, and (d) repeated edits to a single file. SafeHub and plain Git follow the *edit*; git-crypt and SGitLine re-encrypt whole files, so they follow file size and file count instead. Storage growth is the measure used because it is the only one all six systems permit—wire bytes need a readable ref map, which SafeHub and gcrypt do not expose.

names; gcrypt and SafeHub expose only sizes and ordering. Concealment alone therefore does not distinguish SafeHub from gcrypt. The distinction lies in four properties that neither peer provides: post-compromise security for a dynamic group, per-invite history windows, verifier-enforced tip integrity, and a Category 5 instantiation. What the measurements above establish is the cost of providing them.

TABLE VI: Cost of growing a group to n members. Forward-only is the MLS work alone; full adds a 36 ms decryption representing a joiner's history.

n	Full (ms)	Forward-only (ms)
10	186	150
20	428	392
30	775	739
40	1191	1155
50	1689	1653
60	2259	2223
70	2926	2890
80	3699	3663
90	4540	4504
100	5448	5412

D. Membership scaling

Routine operations use symmetric keys; membership changes are where the post-quantum KEM traffic arises. Growing a Category 5 OpenMLS group from one to n members by invitation and join costs 150 ms at n=10 and 5.41 s at n=100 for a forward-only joiner (table VI). The growth is superlinear in n because each Welcome message carries a ratchet-tree path. A *full* joiner additionally opens a 5.0 MiB payload that represents the decryption of history, a constant 36 ms at every n; that component scales with the size of the history and not with the size of the group.

E. Summary

Table VII states the outcome for each operation. On a common transport, five of the six operations that plain Git also performs are faster under SafeHub, stored size is at parity, and clone is the one operation whose cost is not amortized. All four outcomes follow from the same property: a host that cannot read a repository cannot reorganize it, either on the writer's behalf or on the reader's.

VII. Conclusion

We presented SafeHub, an end-to-end encrypted Git hosting system that preserves Git semantics while moving confidentiality and tip integrity from the host to repository members. Administrated MLS groups, per-invite history windows, encrypted bundles, and authenticated ref manifests realize $\mathcal{F}_{\mathsf{safehub}}$ against a malicious host under the assumptions of theorem 1. The prototype quantifies transport, storage, and membership costs against five other systems on one transport (section VI). Gittuf [18] complements SafeHub on plaintext integrity policy; SafeHub can witness encrypted tip digests in an integrity log. Open problems include adaptive corruption without secure erasure, path-level grants, verifiable full-history consolidation against a malicious host that retains old ciphertext, and real multi-host WAN traces (appendix H).

TABLE VII: Each operation against plain Git. *Crossover* means the ratio passes 1 within the measured range. The *native* row compares against Git on its `git://` daemon, its lowest-overhead transport; the remaining rows place both systems on one HTTP transport.

Operation	vs Git	Outcome	Reason
`push`	0.41–0.47×	faster	the host neither unpacks nor repacks a sealed bundle
`push`, native	1.45–0.98×	crossover	fixed per-push cryptography, 46.7 ms/MB against 49.3; not per byte
`pull`	0.38–0.56×	faster	writer-side ciphertext is served unchanged; Git reconstructs a pack
`fetch` (no-op)	0.31–0.40×	faster	protocol floor only, 18–30 against 64–74 ms
`merge`	0.50–0.57×	faster	the local merge is identical work; the push that follows differs
`rebase`	0.56–0.58×	faster	as merge, with a non-fast-forward push of the rewritten branch
`force-push`	0.47–0.51×	faster	the administrator co-signature is one ML-DSA-87 signature
stored size	1.00–1.08×	parity	the packfile is sealed after Git compresses it, so ciphertext is ≈1:1
`clone`, 10 heads	0.53×	faster	no pack to construct, and too little history for replay to dominate
`clone`, 10^3 heads	6.7×	slower	a host that cannot read cannot merge sealed bundles: 17.3 ms/head against 0.31
`clone`, 10^4 heads	54×	slower	the same cause, not amortized and unbounded in the number of heads

Acknowledgements

AI tools were used to check the consistency of this paper, and to improve the quality of both the paper and the implementation.

References

[1] GitHub, "Octoverse 2025: A new developer joins GitHub every second," https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/, 2025.

[2] GitLab Inc., "Annual report on form 10-K for the fiscal year ended january 31, 2026," https://ir.gitlab.com/, 2026, >50 million registered users; >50% of Fortune 100 as customers; 10,682 Base Customers (≥$5,000 ARR) as of 31 January 2026. Covers GitLab SaaS and self-managed; not an aggregate repository count across all GitLab-like forges.

[3] A. Ayer, "git-crypt: Transparent file encryption in git," https://www.agwa.name/projects/git-crypt/, 2015.

[4] S. Whitton *et al.*, "git-remote-gcrypt: PGP-encrypted git remotes," https://spwhitton.name/tech/code/git-remote-gcrypt/, 2013.

[5] Keybase, "Encrypted git for everyone," https://book.keybase.io/git, 2017.

[6] Y.-N. Li, Y. Song, Q. Tang, and M. Yung, "End-to-end encrypted git services," in *ACM CCS*, 2025, also IACR ePrint 2025/1208.

[7] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis, "Security analysis and improvements for the IETF MLS standard for group messaging," in *CRYPTO*, 2020, also IACR ePrint 2019/1189.

[8] D. Balbás, D. Collins, and S. Vaudenay, "Cryptographic administration for secure group messaging," in *USENIX Security*, 2023, A-CGKA / administrated CGKA.

[9] R. Barnes, B. Beurdouche, R. Robert, J. Millican, E. Omara, and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol," RFC 9420, 2023.

[10] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it git? characterizing secret leakage in public GitHub repositories," in *NDSS*, 2019.

[11] J. Hofmann and K.-C. Truong, "End-to-end encrypted cloud storage in the wild: A broken ecosystem," in *ACM CCS*, 2024.

[12] M. Backendal, M. Haller, and K. G. Paterson, "MEGA: Malleable encryption goes awry," in *IEEE S&P*, 2023.

[13] M. R. Albrecht, M. Backendal, D. Coppola, and K. G. Paterson, "Share with care: Breaking E2EE in Nextcloud," in *EuroS&P Workshops*, 2024.

[14] M. Backendal, H. Davis, F. Günther, M. Haller, and K. G. Paterson, "A formal treatment of end-to-end encrypted cloud storage," in *CRYPTO*, 2024.

[15] M. Backendal, D. Balbás, and M. Haller, "Group key progression: Strong security for shared persistent data," IACR ePrint 2025/1028; EUROCRYPT 2026, 2025.

[16] J. Alwen, D. Jost, and M. Mularczyk, "Modular design of secure group messaging protocols and the security of MLS," in *IACR ePrint 2021/1083*, 2021.

[17] ——, "On the insider security of MLS," in *CRYPTO*, 2022, pp. 34–68.

[18] A. S. A. Yelgundhalli, P. Zielinski, R. Curtmola, and J. Cappos, "Rethinking trust in forge-based git security," in *NDSS*, 2025.

[19] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable key compromise in software update systems," in *ACM CCS*, 2010, pp. 61–72.

[20] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "in-toto: Providing farm-to-table guarantees for bits and bytes," in *USENIX Security*, 2019, pp. 1393–1410.

[21] Z. Newman, J. S. Meyers, and S. Torres-Arias, "Sigstore: Software signing for everybody," in *ACM CCS*, 2022, pp. 2353–2367.

[22] GitLab, "GitLab self-managed documentation," https://docs.gitlab.com/ee/, 2025.

[23] Gitea, "Gitea: Git with a cup of tea," https://about.gitea.com/, 2025.

[24] W. Xu, H. Ma, Z. Song, J. Li, and R. Zhang, "Gringotts: An encrypted version control system with less trust on servers," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 2, pp. 668–684, 2024.

[25] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing remote untrusted storage," in *NDSS*, 2003.

[26] W. Chen and R. A. Popa, "Metal: A metadata-hiding file-sharing system," in *NDSS*, 2020.

[27] W. Chen, T. Hoang, J. Guajardo, and A. A. Yavuz, "Titanium: A metadata-hiding file-sharing system with malicious security," in *NDSS*, 2022.

[28] B. Beurdouche, E. Rescorla, E. Omara, S. Inguva, and A. Duric, "The MLS Architecture," RFC 9750, 2025.

[29] J. Alwen, S. Coretti, D. Jost, and M. Mularczyk, "Continuous group key agreement with active security," in *TCC*, 2020, also IACR ePrint 2020/752.

[30] T. Wallez, J. Protzenko, B. Beurdouche, and K. Bhargavan, "TreeSync: Authenticated group management for messaging layer security," in *USENIX Security*, 2023.

[31] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *FOCS*, 2001.

[32] H. Shafagh, L. Burkhalter, S. Ratnasamy, and A. Hithnawi, "Droplet: Decentralized authorization and access control for encrypted data streams," in *USENIX Security*, 2020, pp. 2469–2486.

[33] J. Alwen, B. Auerbach, M. C. Noval, K. Klein, G. Pascual-Perez, K. Pietrzak, and M. Walter, "Cocoa: Concurrent continuous group key agreement," in *EUROCRYPT*, 2022.

[34] J. Alwen, M. Mularczyk, and Y. Tselekounis, "Fork-resilient continuous group key agreement," in *CRYPTO*, 2023.

[35] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing key transparency to end users," in *USENIX Security*, 2015.

[36] H. Malvai, L. Kokoris-Kogias, A. Sonnino, E. Ghosh, E. Oztürk, K. Lewi, and S. Lawlor, "Parakeet: Practical key transparency for end-to-end encrypted messaging," in *NDSS*, 2023.

[37] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *OSDI*, 2004.

[38] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: Group collaboration using untrusted cloud resources," in *OSDI*, 2010.
[39] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket, "Venus: Verification for untrusted cloud storage," in *ACM CCSW*, 2010.
[40] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," in *OSDI*, 2010.
[41] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten, "Social networking with Frientegrity: Privacy and integrity with an untrusted provider," in *USENIX Security*, 2012.
[42] B. H. Kim and D. Lie, "Caelus: Verifying the consistency of cloud services with battery-powered devices," in *IEEE S&P*, 2016.
[43] Y. Hu, S. Kumar, and R. A. Popa, "Ghostor: Toward a secure data-sharing system from decentralized trust," in *NSDI*, 2020.
[44] NIST, "Module-lattice-based key-encapsulation mechanism standard," FIPS 203, 2024.
[45] ——, "Module-lattice-based digital signature standard," FIPS 204, 2024.
[46] D. Boneh, Ö. Dagdelen, M. Fischlin, A. Lehmann, C. Schaffner, and M. Zhandry, "Random oracles in a quantum world," in *ASIACRYPT*, 2011, pp. 41–69.
[47] M. Zhandry, "Secure identity-based encryption in the quantum random oracle model," in *CRYPTO*, 2012, pp. 758–775.
[48] A. Ambainis, M. Hamburg, and D. Unruh, "Quantum security proofs using semi-classical oracles," in *CRYPTO*, 2019, pp. 269–295.
[49] M. Zhandry, "How to record quantum queries, and applications to quantum indifferentiability," in *CRYPTO*, 2019, pp. 239–268.
[50] J. Don, S. Fehr, C. Majenz, and C. Schaffner, "Security of the Fiat–Shamir transformation in the quantum random-oracle model," in *CRYPTO*, 2019, pp. 356–383.
[51] OpenMLS Contributors, "OpenMLS: A Rust implementation of the Messaging Layer Security protocol," https://github.com/openmls/openmls, 2025.
[52] Git Project, "Git: A distributed version control system," https://git-scm.com/, 2025.
[53] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full SHA-1," in *CRYPTO*, 2017, pp. 570–596.
[54] G. Leurent and T. Peyrin, "SHA-1 is a shambles: First chosen-prefix collision on SHA-1 and application to the PGP web of trust," in *USENIX Security*, 2020, pp. 1839–1856.
[55] Git Project, "Git hash function transition (SHA-256 object format)," https://git-scm.com/docs/hash-function-transition, 2025.
[56] D. Unruh, "Universally composable quantum multi-party computation," in *EUROCRYPT*, 2010, pp. 486–505.
[57] G. Brassard, P. Høyer, and A. Tapp, "Quantum cryptanalysis of hash and claw-free functions," in *LATIN*, 1998, pp. 163–169.
[58] A. Chailloux, M. Naya-Plasencia, and A. Schrottenloher, "An efficient quantum collision search algorithm and implications on symmetric cryptography," in *ASIACRYPT*, 2017, pp. 211–240.
[59] N. Bindel, M. Hamburg, K. Hövelmanns, A. Hülsing, and E. Persichetti, "Tighter proofs of CCA security in the quantum random oracle model," in *TCC*, 2019, pp. 520–551.
[60] L. K. Grover, "A fast quantum mechanical algorithm for database search," *Proc. STOC*, pp. 212–219, 1996.
[61] J. O'Connor, J.-P. Aumasson, S. Neves, and Z. Wilcox-O'Hearn, "BLAKE3," https://github.com/BLAKE3-team/BLAKE3, 2020.

APPENDIX A
PRELIMINARIES

Git, MLS, and secure-erasure facts used by SAFEHUB, plus the UC hybrid setup of theorem 1. Threat-model parties, corruptions, and leakage remain in section III; authoritative $\mathcal{F}_{\mathsf{safehub}}$ and the full proof are appendices B and C.

A. Git semantics used by SAFEHUB

Git stores immutable, content-addressed objects (blobs, trees, commits, tags) as a Merkle DAG under tip hashes; named *refs* are the sole mutable pointers, so the host can equivocate or roll back only the ref map—the surface RefHead protects (section IV). Wire ops: `clone`, `fetch`, `push`. Merge and rebase are local Git on the decrypted working copy after fetch; they reach the host only as a later push of the new history.

a) Object-ID hash agility: The DAG argument is only as strong as Git's object hash (historically SHA-1 [53], [54]). RefHead integrity does *not* rest on it: chain links, CAS addresses, and bundle roots are SHA-512, heads carry ML-DSA-87 device signatures, and $\mathcal{F}_{\mathsf{safehub}}$ treats payloads as opaque, so a chosen-prefix collision cannot forge a head, roll a verifier back, or move a tip (theorem 4 and lemma 8; those reductions never mention the object hash). Such a collision instead yields two contents under one ID *inside* an accepted bundle, which a member can exploit but a host that cannot read the bundle cannot; SAFEHUB inherits this from Git. We therefore support Git's SHA-256 object format [55] (`shub repo create --object-format=sha256`; `sit clone` matches the writer's format from the bundle header). The client shells out to `git bundle` and carries object IDs as opaque bytes; tip equality is verified end to end (`sha256-object-conformance-latest.json`). Under that configuration every hash the guarantees rest on is at Category 5. Cross-format interoperation with a SHA-1 remote remains a Git-level limitation. Packfiles and deltas are transport encodings only. In SAFEHUB the server never sees plaintext objects or ref names: objects travel as AEAD-sealed bundles and the ref map lives in the encrypted, hash-chained RefHead (section IV). Client-side `git` operates on a local plaintext store after decrypt.

B. MLS / administrated CGKA

MLS [9], [28] provides continuous group key agreement with ratchet-tree membership, epoch advancement, and exportable epoch secrets; administrated CGKA (A-CGKA) [8] adds cryptographic administrator control over Add/Remove/Update, with related analyses of active security, modular composition, and insider path secrecy [7], [16], [17], [29]. SAFEHUB maps each repository to one MLS group, taking from it device-granular authenticated membership, the labeled exports that seed DKR windows and transport keys, confidential Welcome payloads, application messages for collaboration traffic, and PCS/FS after honest updates—all through the $\mathcal{F}_{\mathsf{acgka}}$ interface of appendix A-D, so TreeKEM and IAS are never re-proved.

C. Secure erasure (threat-model use)

Definition 1 is the erasure hypothesis for adaptive corruptions in $\mathcal{F}_{\mathsf{safehub}}$ and for the simulator's opens of $\mathsf{win}(u)$. Operationally, once (ss_e, mk_e) are derived, any DEK wrapped, interval tokens updated, and the consuming push/fetch completed, an honest stack zeroizes those ephemerals and past MLS epoch secrets no longer needed for delivery, retaining only persistent MLS member-device/group material and the DKR tokens authorizing $\mathsf{win}(u)$. This is a hypothesis about honest client behaviour, not a proved property: every post-compromise claim is conditional on it. Adaptive corruption thus reveals only current non-erased state; ExposeState reveals

state at a chosen time without ongoing corruption. Without erasure, adaptive UC reduces to a selective residual (appendix H).

D. UC framework and hybrid model

We use computational Universal Composability [31] with QPT machines, not Unruh quantum-message UC [56]: wire messages are classical bitstrings; parties may hold quantum state between classical interactions. Hashes, KDFs, HMAC, and SHA-512 are random oracles $\mathcal{F}_{\mathsf{ro}}$ accessible in superposition (QROM) [46], [47], [49]; signing, verification, KEM, and AEAD oracles stay classical. Environment $\mathcal{Z}$ drives execution; security requires a QPT $\mathcal{S}$ with $\mathsf{EXEC}_{\Pi_{\mathsf{safehub}},\mathcal{A},\mathcal{Z}} \approx \mathsf{EXEC}_{\mathcal{F}_{\mathsf{safehub}},\mathcal{S},\mathcal{Z}}$ (section V and appendix B). We work in the $(\mathcal{F}_{\mathsf{acgka}}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{ro}})$-hybrid model:

- $\mathcal{F}_{\mathsf{acgka}}$: ideal *administrated* CGKA (A-CGKA) [7], [8], [16], [29] with admin gates following Balbás–Collins–Vaudenay [8], bound to the group-context roster at the authorizing epoch. As a *standard hybrid assumption* (advantage $\varepsilon_{\mathsf{acgka}}$) we take adaptive labeled-export and Welcome security against QPT adversaries under definition 1 when instantiated by NIST PQ Category 5 MLS (ML-KEM-1024/ML-DSA-87), with insider path secrecy [17]; TreeKEM/IAS are not re-proved (SAFEHUB-specific adaptive composition: appendix C). Interfaces: Create, Add, Remove, Update, Rotate (admin-gated where appropriate), plus per-epoch Export(label) and joiner-confidential Welcome payloads. Concretely, for each uncompromised epoch e no QPT adversary corrupting adaptively under definition 1 distinguishes $\mathsf{Export}(e, \texttt{transport})$ from uniform $\{0,1\}^\lambda$ (DKR seed ss_e) or $\mathsf{Export}(e, \texttt{refs})$ from uniform $\{0,1\}^{256}$ (epoch MAC / collab key mk_e), given the MLS transcript and other exports (advantage $\varepsilon_{\mathsf{exp}}$); Welcome ciphertexts hide private payloads (DKR tokens included) from $\mathcal{A}$ and non-recipients.
- $\mathcal{F}_{\mathsf{ca}}$: binds identities to ML-DSA-87 verification keys.
- $\mathcal{F}_{\mathsf{ro}}$: quantum-accessible RO for DKR chains, KDFs, and HMAC modeling, output length λ=384 for DKR/KDF nodes. $\mathcal{F}_{\mathsf{ro}}$ is *programmable* by $\mathcal{S}$ alone, which is an interface and not a proof step: $\mathcal{S}$ may fix the image at a point neither $\mathcal{Z}$ nor $\mathcal{A}$ has queried, and the assignment is recorded so later queries return it. Consistency fails exactly when $\mathcal{A}$ already holds weight on a point $\mathcal{S}$ fixes—the event $\mathsf{Bad}_{\mathsf{ro}}$, bounded over N_{prog} points by lemma 3. Adaptive openings consume this interface (lemmas 1 and 4). Collision-critical content digests (CAS addresses, RefHead chain hashes, bundle roots) are modeled separately as ideal length-512 hashes with quantum collision resistance at the BHT/CNS margins [57], [58] of table IX (advantage $\varepsilon_{\mathsf{coll}}$ in eq. (1)), not as additional $\mathcal{F}_{\mathsf{ro}}$ outputs of length λ.

Each repository uses a distinct sid; UC composition yields independent $\mathcal{F}_{\mathsf{safehub}}$ instances.

Proposition 1 (Classical UC composition for QPT parties)**.** *Let every wire message and every scheduling decision be classical, let the simulator be straight-line, and let its oracle access be the programmable $\mathcal{F}_{\mathsf{ro}}$ above. Then classical UC composition [31] holds for QPT machines that carry quantum internal state across activations, and Unruh quantum-message UC [56] is not required. The three conditions are what the proof uses: classical scheduling makes the composed execution a classical interactive machine, straight-line simulation removes any need to rewind quantum state, and recorded programming keeps $\mathcal{F}_{\mathsf{ro}}$ consistent under superposition queries.*

Proof sketch. The composition theorem [31] quantifies over interactive machines exchanging classical messages under a classical schedule, and quantum internal registers do not leave that class: a party remains a map from classical input history to classical output, carrying the register between activations. The two side conditions are what the simulator needs. Straight-line simulation is required because the proof composes simulators by running them, and a party holding quantum state cannot be rewound; ours never rewinds, answering every handler from ℓ and the public statement. Recorded programming is required so the hybrid $\mathcal{F}_{\mathsf{ro}}$ is one fixed function throughout: the programmed points are part of the compressed oracle's state, so a later superposition query returns the programmed image. A quantum-message interface to $\mathcal{Z}$ would instead need Unruh's framework [56]. □

What theorem 1 assumes of $\mathcal{F}_{\mathsf{acgka}}$ and what it proves above MLS are itemized in appendix C-A; PQ margins in appendix E; QROM technique and tightness in appendix C.

E. Cryptographic building blocks

Committing AEAD (E, D) is the key-committing, message-equivocable HKDF-SHA-512 RO-pad Encrypt-then-MAC transform under an outer HMAC-SHA-512-256 tag (construction and games: theorem 6), so its key-commitment margin is the 48-byte commit block of table IX and not the AES-256 Grover floor—AES-256-GCM appears only inside the MLS ciphersuite (appendix G). Signatures are ML-DSA-87, SHA-512 is modeled in the QROM, and GKP/DKR uses λ=384 (appendix C-A and section IV).

APPENDIX B
THE FUNCTIONALITY $\mathcal{F}_{\mathsf{safehub}}$, IN FULL

Figure 11 is the authoritative functionality used by appendix C; the body (section V) summarizes the same interface in prose.

a) Session, parties, and interfaces: An instance is addressed by $\mathsf{sid} = (\mathsf{safehub}, \mathsf{repo}, \nu)$, where ν prevents cross-repository replay, and parties are devices, several of which may belong to one user. Every input carries sid and a monotonically increasing request identifier; malformed, duplicate, or cross-session inputs are ignored. $\mathcal{Z}$ supplies the ten commands to honest parties and receives their outputs, Forked and compare-and-swap retries included. $\mathcal{A}$ controls the server and network: it schedules every output, so it may delay or withhold one but not alter its contents, and it chooses fetch prefix indices k and may adaptively corrupt members under definition 1. Beyond $\ell(op)$ on each accepted operation it receives no plaintext payload, ref name, commit message, review text, or history bound for an honest uncorrupted window.

Functionality $\mathcal{F}_{\mathsf{safehub}}$ (authoritative). Interacts with parties and the ideal adversary $\mathcal{S}$. Maintains, per repository: epoch e; members M; admins $\mathsf{Adm} \subseteq \mathsf{M}$; a device-to-user map; history authorization $\mathsf{win}(u)$ as a union of disjoint closed epoch intervals; an append-only logical push sequence $L = (x_1, \ldots, x_t)$ with $x_j = (j, e_j, \mathsf{payload}_j, \mathsf{refs}_j, \mathsf{nonff}_j)$; canonical refs refs; application records; for every honest party P, the largest delivered prefix d_P and its branch b_P; a set Sh of *shadow heads* (passed the Push gate, lost the compare-and-swap; recorded with the position they descend from, never in L); and consolidation records that identify superseded physical payloads and a replacement payload with the same logical repository state (never changing L or any $\mathsf{win}(u)$). X is the corrupt *member-device* set; a user counts as corrupt exactly when one of its devices is in X.

Leakage. For an accepted operation op, send $(\mathsf{Leak}, \mathsf{sid}, \ell(op))$ to $\mathcal{S}$, where $\ell(op)$ contains its public type flag, ciphertext lengths, time, repository identifier, authenticated sender/device linkage when present on the control channel, compare-and-swap outcomes, head sequence, MLS-visible group and device counts, and (when distinguishable by size) the history-grant class $\{\mathsf{full}, \mathsf{from\text{-}now}\}$. Plaintext objects, ref names, commit messages, and review text are not leaked. Every accepted operation proceeds in one order: validate and compute the ideal transition, send $\ell(op)$, receive from $\mathcal{S}$ a transcript and schedule constrained by that leakage, then commit at the linearization point the handler names. A transcript is a function of the leakage, never an input to it. On a fetch, $\mathcal{S}$ chooses a prefix index $k \le t$. Delivery occurs only after $(\mathsf{Deliver}, P, k)$.

On (CREATE, repo) **from** P: require that no state exists for sid. If P is honest, set $e \leftarrow 1$, $\mathsf{M} \leftarrow \mathsf{Adm} \leftarrow \{P\}$, $\mathsf{win}(P) \leftarrow [1, \infty)$, $L \leftarrow \epsilon$, $\mathsf{refs} \leftarrow \emptyset$, receive an empty encrypted-RefHead transcript of leaked length from $\mathcal{S}$, and send $\ell(\text{CREATE})$ to $\mathcal{S}$. Corrupt issuers use the same transition on arguments they choose; $\mathcal{S}$ supplies transcripts and scheduling, never the resulting roster or windows. This holds for every handler below, so a corrupt administrator may issue any command it is authorized to issue and no state update a command cannot produce.

On (INVITE, u, h) **from** P_a: write e_0 for the epoch on entry and $e_1 = e_0 + 1$. Require $P_a \in \mathsf{Adm}$ and $h \in \{1, e_0 + 1\}$, evaluated against e_0. Add every authorized device of u to M, union $[h, \infty)$ into $\mathsf{win}(u)$, set $e \leftarrow e_1$, and receive a Welcome transcript of leaked length from $\mathcal{S}$. Payloads at epochs below h remain unauthorized for u. When $h = e_0 + 1$ (forward-only), record a *graft* g_u: a referenced encrypted object carrying the logical state as of e_0, bound to this Invite and admitted into u's view at h. It alters neither L nor any win, and FETCH delivers it ahead of the in-window entries. Send $\ell(\text{INVITE})$ to $\mathcal{S}$.

On (REMOVE, u) **from** P_a: require $P_a \in \mathsf{Adm}$. Write e_0 for the epoch on entry, $e_1 = e_0 + 1$. Remove all devices of u from M and Adm, cap every open interval in $\mathsf{win}(u)$ at e_0, set $e \leftarrow e_1$, refresh the epoch exports so the removed devices receive none of them, and receive MLS Commit framing of leaked length from $\mathcal{S}$. REMOVE is therefore itself the rekey: the cap and the key separation occur in one transition, and u cannot derive K_e for $e \ge e_1$ (appendix C-C). A later re-invite creates a new interval and does not reopen a capped one. Send $\ell(\text{REMOVE})$ to $\mathcal{S}$.

On (ROTATE) **from** P_a: require $P_a \in \mathsf{Adm}$. Increment e without changing membership or existing interval endpoints, and receive Commit framing of leaked length from $\mathcal{S}$. Every epoch-advancing Commit—INVITE, REMOVE, SETADMIN, and this one—refreshes the epoch exports. Send $\ell(\text{ROTATE})$ to $\mathcal{S}$.

On (SETADMIN, u, b) **from** P_a: require $P_a \in \mathsf{Adm}$, $u \in \mathsf{M}$, and valid roster authorization. Add or remove u from Adm according to b, increment e, and receive Commit framing of leaked length from $\mathcal{S}$. Send $\ell(\text{SETADMIN})$ to $\mathcal{S}$.

On (PUSH, $\mathsf{pred}, \mathsf{payload}, \mathsf{refs}', \mathsf{nonff}, \alpha$) **from** P: require $P \in \mathsf{M}$ and that pred name a node of the branch DAG $L \cup \mathsf{Sh}$. Over the object graph reachable from pred—the view being extended, not necessarily canonical L—set $\mathsf{ff} = 1$ iff every ref in refs' is unchanged or names a descendant of its value at pred. Creation, deletion, and a change of ref type are transitions like any other; deletion is never fast-forward, and a target absent from the object graph is unverifiable. Require $\mathsf{nonff} = 1 - \mathsf{ff}$, else reject; if $\mathsf{ff} = 0$ require a valid admin authorization α bound to $(\mathsf{sid}, e, \mathsf{refs}')$. The gate applies identically to honest and corrupt P. Compare-and-swap is the linearization point. $\mathcal{F}_{\mathsf{safehub}}$ takes (CASRESULT, $\mathsf{accept/retry}$) from $\mathcal{S}$: at most one push per pred is accepted; a loser is recorded as a *pending* node of Sh descending from pred, retaining its payload so a resubmission need not re-supply it. Pending is a property of the log, not of any party's view: a pending node becomes some party's accepted state only when FETCH delivers it. If P is honest and $\mathcal{S}$ returns accept, append $(t + 1, e, \mathsf{payload}, \mathsf{refs}', \mathsf{nonff})$, set $\mathsf{refs} \leftarrow \mathsf{refs}'$, and receive equal-length AEAD/RefHead transcripts from $\mathcal{S}$ matching $\ell(\text{PUSH})$. If P is corrupt and $\mathcal{S}$ returns accept, receive $(\mathsf{payload}, \mathsf{refs}', \mathsf{nonff})$ and the wire transcript from $\mathcal{S}$ and append/register them *only if* the recomputed gate above admits the update. Send $\ell(\text{PUSH})$ (including the compare-and-swap outcome) to $\mathcal{S}$.

On (APPLY, m) **from** P: require $P \in \mathsf{M}$. Append the PR, issue, review, or CI record m at epoch e to the append-only application log A, giving it index $|A| + 1$, and receive an equal-length collaboration ciphertext from $\mathcal{S}$. A is ordered and delivered independently of L: each party has a cursor a_P that $\mathcal{S}$ advances by $(\mathsf{DeliverApp}, P, j)$, releasing records of index $\le j$ whose epoch lies in $\mathsf{win}(P)$, so application delivery is not keyed to any push index. Send $\ell(\text{APPLY})$ to $\mathcal{S}$.

On (CONSOLIDATE, $J, \mathsf{snapshot}, \rho$) **from** P: require $P \in \mathsf{M}$ and that J names a prefix of payloads visible to P. Require that $\mathsf{snapshot}$ carries a valid consolidation binding digest for J (logical roots unchanged), and that ρ names a valid administrator (or m-of-n quorum) receipt authorizing the replacement of J. Record $\mathsf{snapshot}$ as a payload-preserving physical replacement for J, leaving L, refs, epochs, and windows unchanged, and receive a snapshot ciphertext of leaked length from $\mathcal{S}$. Each component of $\mathsf{snapshot}$ retains the epoch that sealed the entry it replaces, so a party can open exactly the components whose epochs lie in its window and consolidation widens no window. Send $\ell(\text{CONSOLIDATE})$ to $\mathcal{S}$.

On (FETCH) **from** P: require $P \in \mathsf{M}$. On adversarial choice $k \le t$, form the authorized view consisting of the graft g_P if recorded at Invite, then those x_j, $j \le k$, with $e_j \in \mathsf{win}(P)$, replacing an eligible physical prefix by a consolidation snapshot if $\mathcal{S}$ supplies the request bit. Refs delivered are those of the latest position in that view, not refs_k. An x_j whose merge-base lies below $\mathsf{win}(P)$ is *unverifiable* for P, treated as non-fast-forward and accepted only with a valid α. Withholding objects a head names is non-delivery of that entry. If $k \ge d_P$ and the selected branch diverges from P's view only above d_P, deliver it and set $d_P \leftarrow k$; if $k < d_P$, or the branch contradicts an entry at or below d_P, first deliver $\mathsf{Forked}(\mathsf{sid})$. Delivery follows $(\mathsf{Deliver}, P, k, b)$ from $\mathcal{S}$, where b selects the canonical branch or a continuation through Sh descending from k; serving one sets b_P and leaves L unchanged. A branch that verifies becomes P's accepted state whether or not it is canonical: a single FETCH raises only the local Forked above, and divergence between parties is a COMPARE event.

On (COMPARE, Q) **from** P: require $P, Q \in \mathsf{M}$. Each submits $\chi = (\mathsf{sid}, s, \{(j, h_j, p_j, e_j)\}_{j \le s})$: the sequence, head identifier, predecessor, and epoch of every entry in *that party's own delivered view* up to s, not of L. Comparison is the restriction of the two lists to $\mathsf{win}(P) \cap \mathsf{win}(Q)$ below $\min(s_P, s_Q)$, which a per-entry list admits and a single digest of the whole prefix would not. $\mathcal{S}$ may supply any χ for a *corrupt* party and may delay or withhold an honest party's, but may not forge one: checkpoints are authenticated under $\mathcal{F}_{\mathsf{acgka}}$ keys. Disagreement in that range delivers $\mathsf{Forked}(\mathsf{sid}, P, Q)$ to both; for honest P, Q this is exactly $b_P \ne b_Q$ diverging at or below $\min(s_P, s_Q)$.

On (EXPOSESTATE, d) **from** $\mathcal{S}$: reveal the current ideal state of member-device d (as under Corrupt) without adding d to X and without granting ongoing observation of future honest updates.

On (CORRUPT, d) **from** $\mathcal{S}$ *at any point*: add the *member-device* d to X and hand $\mathcal{S}$ the current ideal state of d under secure erasure: payloads and application records whose epochs lie in $\mathsf{win}(u)$ for u the owner of d, plus public metadata and any outputs already delivered to d. Erased ephemeral exports outside that window are unavailable. Corrupting an administrator device also grants its future authorized admin inputs; it does not retroactively authorize forged operations or reopen a capped window. While d stays corrupt, release the same abstract state at each later accepted update affecting $\mathsf{win}(u)$. Corruption is at device grain: $\mathsf{win}(u)$ is per user, so corrupting any device of u exposes that window, while leaf signing and decryption secrets are per device, so u's other leaves stay honest. Corruption is permanent: a Rotate replaces a leaf's keys but cannot end observation of a device still in X, and the post-compromise statement is therefore about transient EXPOSESTATE (definition 3). A handler's "corrupt" branch applies when the acting device lies in X.

Fig. 11: Ideal functionality $\mathcal{F}_{\mathsf{safehub}}$, written as one handler per command in the style of Canetti. An honest branch records ideal state and obtains its wire transcript from $\mathcal{S}$; a corrupt branch accepts the values $\mathcal{S}$ extracts. Leakage, compare-and-swap, history windows, fork detection, consolidation, and adaptive Corrupt and ExposeState are as used in theorem 1 and appendix C.

b) Batched pushes, races, and forks: The ideal append serializes concurrent pushes without granting the server authority over refs; in Π_{safehub} the loser fetches, merges, and resubmits. Availability is absent, since $\mathcal{A}$ may withhold outputs

forever and $k < t$ on FETCH withholds a suffix, and repository discovery is a separate MLS-protected directory application composing with $\mathcal{F}_{\mathsf{safehub}}$. Divergence is detected on COMPARE over exchanged checkpoints rather than delivered lengths—two lengths into one ideal L are always prefix-comparable, so a d_P/d_Q rule could never fire. It arises when $\mathcal{S}$ serves one party a shadow head from Sh while another follows the canonical branch ($b_P \neq b_Q$); both receive Forked when they compare, which may be after each has accepted its own branch. A local FETCH detects only rollback behind the caller's d_P.

c) Corruption, realizability, and realization notes: One modelling choice is load-bearing: $\mathcal{F}_{\mathsf{safehub}}$ *records* ideal payloads, windows, and authorization rather than deriving them from the wire ciphertexts $\mathcal{S}$ supplies, and samples ideal secrets independently. That is what lets $\mathcal{S}$ program AEAD/RO openings on legitimate exposure; a functionality deriving ideal keys from simulated ciphertext would fix those points and be unrealizable under adaptive corruption with erasure (appendix C). The correspondence to Π_{safehub} is a realization claim, not part of $\mathcal{F}_{\mathsf{safehub}}$: non-FF α is the ML-DSA-87 admin co-signature of theorem 5, bound to $(\mathsf{sid}, e, H(\mathsf{RefHead}), \mathsf{refs}')$; Invite Welcome carries DKR tokens for $[h, \infty)$ and a forward-only graft when $h = e + 1$; retry is compare-and-swap on $H(\mathsf{RefHead})$; transcripts are MLS framing, AEAD ciphertexts, and RefHead bytes.

d) What $\mathcal{F}_{\mathsf{safehub}}$ does not *promise:* $\mathcal{F}_{\mathsf{safehub}}$ grants an authorized administrator ideal admin power, leaves group size and device count in ℓ, and says nothing about availability, path- or file-level grants, PEKS over ciphertext, or plaintext after an authorized decrypt—the system-level readings of which are appendix H. Adaptive corruption under erasure is modeled by Corrupt above; TreeKEM/IAS enter only via $\mathcal{F}_{\mathsf{acgka}}$.

APPENDIX C
PROOF OF THEOREM 1

A. Proof overview, assumptions, and terms

Running $\mathcal{A}$ as a black box, QPT $\mathcal{S}$ (figs. 12 and 13) keeps a one-to-one correspondence with $\mathcal{F}_{\mathsf{safehub}}$ and rebuilds the server-facing transcript from ℓ and the public statement of each operation: equal-length zero-encryptions outside corrupt windows, simulated MLS/RefHead transcripts, corrupt-party extraction, prefix-or-fork delivery, and window-exact opens on Corrupt/ExposeState (lemma 1). Hybrids H_0–H_5 then replace exports, out-of-window ciphertexts, and acceptance checks until the ideal execution, each hop reducing to a primitive game or a QROM argument on hidden DKR segments. Theorem 1's hypotheses are discharged by theorem 6 (AEAD) and lemma 2 (HMAC, whose residual is HMAC-as-PRF under RO modelling, not an unconditional QROM result); $\mathcal{F}_{\mathsf{acgka}}$ stays hybrid (theorem 9 and corollary 1) and TreeKEM/IAS are not re-proved.

Budgets are polynomial with λ=384 and $q_{\mathsf{ro}} \leq 2^{64}$: q_p protected seals, q_s classical ML-DSA-87 signing queries, q_v epoch-tag verifications over n_{mac} keys, q_{ro} quantum $\mathcal{F}_{\mathsf{ro}}$ queries, q_{seg} segment roots, q_e protected epochs, n_{adm} admin keys, and the programming budget $N_{\mathsf{prog}} = N_{\mathsf{pad}} + N_{\mathsf{dkr}} + N_{\mathsf{open}}$ (lemma 3). Each primitive is charged once: $\varepsilon_{\mathsf{acgka}}$ absorbs $q_e\varepsilon_{\mathsf{exp}}$, Welcome confidentiality, and membership authenticity; $\varepsilon_{\mathsf{ae}}$ is IND-CPA, INT-CTXT, key commitment, and equivocation for the RO-pad transport, excluding the outer HMAC; $\varepsilon_{\mathsf{mac}}$ is multi-user HMAC-SHA-512-256 SUF-CMA; $\varepsilon_{\mathsf{sig}}$ is QPT EUF-CMA of ML-DSA-87; $\varepsilon_{\mathsf{ow2h}}$ is the per-segment O2H loss [48], [50], [59]; and $\varepsilon_{\mathsf{coll}}$ is quantum collision finding on 512-bit content digests [57], [58]. Concrete margins: definition 4, table IX, and appendix E-A.

B. QROM preliminaries, freshness, and bad events

Only $\mathcal{F}_{\mathsf{ro}}$—and SHA-512 when modelled as RO for DKR, KDF, and HMAC—is quantum-accessible; wires, signing, verification, KEM, and AEAD oracles are classical (appendix A-D and proposition 1).

a) QROM tools and scope: Two tools do the work. *Compressed oracles* [49] answer superposition queries and record classical extracts, letting $\mathcal{S}$ program an image only at unqueried inputs and abort into $\mathsf{Bad}_{\mathsf{ro}}$ otherwise. *Semi-classical O2H / measure-and-reprogram* [48], [50], [59] bounds amplitude on a hidden DKR segment preimage, union-bounded over q_{seg} segments at $\varepsilon_{\mathsf{ow2h}}$ each; we claim no asymptotic tightness beyond those references.

Definition 2 (Exposed and protected epochs)**.** *Fix a (possibly empty) initially corrupt device set X_0; $\mathcal{A}$ may grow X adaptively under definition 1. Write E_{xs} for the epochs revealed by* ExposeState *calls. An epoch e is* exposed *if $e \in \bigcup_{d \in X} \mathsf{win}(u(d)) \cup E_{\mathsf{xs}}$, where $u(d)$ is the owner of device d; otherwise it is* protected*; protected ciphertexts, exports, and DKR nodes are the challenge objects below.* ExposeState *counts even though it does not add its device to X: it hands the environment real plaintext, so keying on X alone would leave those epochs protected and the zeroing hybrids would owe an opening the simulator cannot give. What X adds beyond it is ongoing observation.*

Definition 3 (PCS freshness)**.** *In game $\mathsf{G}_{\mathsf{pcs}}$, a challenge epoch $e' > e$ is* fresh *if: (i) an honest epoch-advancing Commit occurred at epoch e (standalone Rotate, Remove, Invite, or SetAdmin); (ii)* ExposeState *reveals state only for times $t \leq e$; and (iii) the adversary does not observe the honest post-exposure update path that refreshes (ss_e, mk_e) and inserts the fresh forward segment root. Exposure is stated over* ExposeState *because $\mathcal{F}_{\mathsf{safehub}}$ has no* Uncorrupt*: a device in X stays there, so "a corruption that ends" has no ideal counterpart, whereas* ExposeState *is exactly the transient compromise the game needs. Freshness is vacuous if a corrupted admin authorizes the update.*

b) Bad events: We condition throughout on the complement of $\mathsf{Bad}_{\mathsf{sig}}$, $\mathsf{Bad}_{\mathsf{mac}}$, $\mathsf{Bad}_{\mathsf{coll}}$, $\mathsf{Bad}_{\mathsf{aead}}$, and $\mathsf{Bad}_{\mathsf{ro}}$—the five abort conditions fig. 13 enumerates, each with the game it wins. Honest clients require valid author signature, epoch tag, predecessor/anchor consistency, and admin co-signature on non-fast-forward heads (section IV-E).

C. DKR semantics (matching the protocol)

Each contiguous DKR segment has an independent root $token \in \{0,1\}^\lambda$; for $(from, to, token)$ and $e \in [from, to]$, $K_e \leftarrow \mathsf{HKDF}_{\mathsf{SHA\text{-}512}}(token, \texttt{safehub-v1:dkr-epoch:}e) \in \{0,1\}^{256}$ (equivalently a domain-separated $\mathcal{F}_{\mathsf{ro}}$ evaluation on $(token, e)$).

- An epoch-advancing Commit starts a new segment at e under a fresh root, taken from the MLS transport export of that epoch. Earlier tokens keep $[from, to]$ under the *old* root and cannot derive post-block keys, because deriving the new root would mean deriving an export the Commit did not deliver to them. REMOVE is such a Commit (fig. 11), so the cap on $\mathsf{win}(u)$ and the key separation coincide.
- $\mathsf{backward_block}(h)$ (forward-only Invite, $h{=}e{+}1$) likewise samples a fresh root and returns $[h, \cdot]$ under it. Widening $from$ cannot recover pre-join K_e (prior root).

A segment is re-rooted once its span would exceed the capacity N of table X; because every membership change advances the epoch and so starts a segment, that bound is reached only by N consecutive epochs with no membership change, and it never narrows an existing grant.

Segment separation is therefore cryptographic and $[from, to]$ within one segment is not: a token evaluates the derivation at any e, so the endpoints bound an honest holder and not an adversarial one. This suffices because every event that creates or narrows a window—a forward-only Invite, and a REMOVE—starts a segment, so the boundaries of $\mathsf{win}(u)$ coincide with boundaries between roots. Exactness is claimed at that granularity: what a holder can derive is the union of the segments whose roots it holds. Model assumed by theorems 2, 7 and 8 and lemma 7.

D. Simulator algorithm

The QPT simulator $\mathcal{S}$ (figs. 12 and 13) provides one handler for each command of $\mathcal{F}_{\mathsf{safehub}}$'s adversary interface (fig. 11) and UC-emulates the server-facing view from $\mathcal{F}_{\mathsf{safehub}}$'s leakage and delivery interfaces alone, under adaptive corruptions with secure erasure.

Lemma 1 (Corruption consistency under secure erasure)**.** *Under definition 1, when $\mathcal{S}$ opens on $\mathsf{Corrupt}(d)$ or $\mathsf{ExposeState}(d)$ for a device d of u: for $e \in \mathsf{win}(u)$ it returns tokens whose $\mathsf{HKDF}(token, \texttt{safehub-v1:dkr-epoch:}e)$ equals the simulated sealing key, and opens each published in-window ciphertext to the plaintext $\mathcal{F}_{\mathsf{safehub}}$ discloses by programming the pad at that ciphertext's points (theorem 6(iii)); for $e \notin \mathsf{win}(u)$ the state holds no root for that segment and those ciphertexts remain zeros under independent keys. Later growth of X only enlarges $\bigcup_{d \in X} \mathsf{win}(u(d))$, programmed lazily on first open. The distribution is that of the real execution given $\neg\mathsf{Bad}_{\mathsf{ro}}$.*

Proof. By definition 1 an honest device retains persistent MLS state and the tokens for $\mathsf{win}(u)$, which is what $\mathcal{F}_{\mathsf{safehub}}$'s Corrupt interface returns. An in-window root is thus either already used to seal a real payload, where pad programming realizes the open and contradicts no commitment to zeros because $H_3 \to H_4$ zeros only out-of-window payloads (lemma 10), or sampled lazily on first open. Equivocation here is unavoidable rather than convenient: the retained token lets $\mathcal{A}$ recompute K_e and unwrap the DEK, so a *published* ciphertext must open to the disclosed plaintext, which the equivocable pad permits and AES-256-GCM does not. Out-of-window zeros use keys never claimed equal to an ideal key, so opening $\mathsf{win}(u)$ never reprograms them. Consistency across all such openings is lemma 3. □

E. Supporting lemmas (UC hybrids)

Each hybrid hop below rests on one of the following.

Lemma 2 (HMAC-SHA-512-256 SUF-CMA in the QROM)**.** *Model SHA-512 as a quantum-accessible random oracle and let* HMAC-SHA-512-256 *be HMAC truncated to 256 bits. Against a QPT adversary making q_{mac} classical MAC and q_{ro} quantum RO queries against n_{mac} keys, and attempting q_v verifications, the multi-user SUF-CMA advantage is at most*

$$\varepsilon_{\mathsf{mac}} \le n_{\mathsf{mac}}\varepsilon_{\mathsf{PRF}} + O\big(q_{\mathsf{ro}}^2/2^{512}\big) + q_v/2^{256},$$

where $\varepsilon_{\mathsf{PRF}}$ is the PRF advantage of HMAC under the same RO modeling (standard HMAC-as-PRF lifting with classical tagging oracle). Online tag guessing contributes at most 2^{-256} per verification.

Proof. In $\mathsf{G}_{\mathsf{mac}}$ the adversary holds classical MAC/Vrfy oracles under n_{mac} independent keys and quantum $\mathcal{F}_{\mathsf{ro}}$, and wins on a fresh verifying (m^*, τ^*). Replacing each HMAC_K by a random function on classical queries costs $\varepsilon_{\mathsf{PRF}}$ per key by the Bellare–Canetti–Krawczyk lifting, hence the n_{mac} factor; tagging stays classical, so no SUF-CMA against a quantum tagging oracle is claimed. A forgery then needs a colliding RO transcript on the truncated output, $O(q_{\mathsf{ro}}^2/2^{512})$ [49], or a tag guess, 2^{-256} per attempt and $q_v/2^{256}$ over all of them. The residual is HMAC-as-PRF in the RO model: full QROM indifferentiability of the SHA-512 Merkle–Damgård construction is not known unconditionally. Epoch tags bind $(\mathsf{sid}, \mathsf{seq}, H(\mathsf{enc_refs}), \mathsf{bundle_root}, \mathsf{prev}, \mathsf{mls_epoch})$ under mk_e and outer AEAD tags bind $(\mathsf{nonce}, \mathsf{body}, \mathsf{AD})$: the same construction under independent keys, which is why one term covers both. □

Lemma 3 (Multi-point reprogramming)**.** *Let $\mathcal{S}$ program $\mathcal{F}_{\mathsf{ro}}$ at N points of λ bits, each chosen after the adversary has made at most q_{ro} quantum queries. Then no QPT distinguisher separates the programmed oracle from one where those images were sampled at the outset, except with probability $N \cdot O(q_{\mathsf{ro}}^2/2^\lambda)$, which is $\Pr[\mathsf{Bad}_{\mathsf{ro}}]$.*

Proof sketch. Hybridize over the points: in D_t the first t are programmed on demand and the remainder are sampled in advance. D_t and D_{t+1} differ at one point x_{t+1}, so a distinguisher between them is a measure-and-reprogram adversary at x_{t+1} [50]. In the compressed oracle [49] the database carries

Simulator $\mathcal{S}$ (state and handlers isomorphic to $\mathcal{F}_{\mathsf{safehub}}$; continuation in fig. 13). $\mathcal{S}$ runs a copy of $\mathcal{A}$, relays between $\mathcal{Z}$ and $\mathcal{A}$, plays honest parties toward $\mathcal{A}$, controls $(\mathcal{F}_{\mathsf{acgka}}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{ro}})$, and interacts with $\mathcal{F}_{\mathsf{safehub}}$ on its adversary interface (fig. 11 and appendix B).

State. sid; maps dev, epoch; simulated MLS transcript T; head-chain table H[seq]=(prev, e, author, nonff, admin, tag, sig); per-party anchors d_P; simulated keys ss_e, mk_e, K_e, DEK (lazy); DKR segment table Seg; programmed RO record R; compare-and-swap tip H_{tip}; leak buffer; corrupt set X (on $\mathsf{Init}(X_0)$ store $X \leftarrow X_0$; grown on Corrupt).

Corrupt issuer (all operations). When $\mathcal{A}$ issues an operation as a corrupt party, $\mathcal{S}$ applies exactly the gate an honest client would—the IAS/admin check for membership operations, the RefHead verification of section IV-E for the rest; on failure it ignores the operation, or **aborts** where fig. 13 maps that failure to a game win. On success it extracts the operation's contents—the initial roster/window assignment for CREATE, the Welcome payload for INVITE, the Commit contents for REMOVE/ROTATE/SETADMIN, (payload, refs′, nonff), m, or $(J, \mathsf{snapshot})$ for PUSH/APPLY/CONSOLIDATE —and supplies them, with the wire transcript, as the corrupt party's *arguments* to $\mathcal{F}_{\mathsf{safehub}}$, which then runs the same transition it runs for an honest issuer. $\mathcal{S}$ never supplies a resulting roster, window, or log entry. Only the honest branches differ per operation, below.

On (CREATE leakage L) **from** honest creator: spawn a simulated $\mathcal{F}_{\mathsf{acgka}}$ session, set e=1, seed Seg with one forward and one backward segment, sample ss_e, mk_e for e=1, program $\mathcal{F}_{\mathsf{ro}}$ only on unqueried points, publish an empty encrypted RefHead of leaked length to $\mathcal{A}$, and set H_{tip}.

On (INVITE, u, h leakage L) **from** honest admin: if $P_a \notin$ Adm in the ideal roster and the submitted IAS signature is not from a corrupted admin key: **abort** (sig forgery). Else emit a Welcome of leaked length whose payload is simulated interval tokens for $[h, \infty)$ (fresh backward block iff h=e+1). Update Seg and dev.

On (REMOVE, u / ROTATE / SETADMIN leakage L) **from** honest admin: same IAS gate. Each of these advances the epoch, so each refreshes ss_e, mk_e and starts a fresh DKR segment; only the roster effect differs. On Remove: drop the leaves, cap the window, and withhold the new exports from the dropped leaves. On Rotate: leave membership and windows unchanged. On SetAdmin: update the simulated admin roster. Emit MLS Commit framing of leaked length.

On (PUSH leakage L) **from** honest member: obtain from $\mathcal{F}_{\mathsf{safehub}}$ the leakage together with the *public statement* of the operation: the policy result, the recomputed ff, the predecessor and sequence, the refs commitment, the roster digest, and the compare-and-swap outcome. These are what make a verifying RefHead constructible without plaintext refs; ref names and objects are never disclosed. Emit AEAD of equal-length zeros under fresh simulated keys (bundle, refs) and append a simulated RefHead with prev=H_{tip}, epoch tag under mk_e, device ML-DSA-87 signature, and admin co-sig iff the recomputed ff $= 0$—the recomputation, not $\mathcal{A}$'s declared bit, is the gate, for corrupt issuers too. Attempt the compare-and-swap: if $\mathcal{A}$ rejects tip, return (CASRESULT, retry) to $\mathcal{F}_{\mathsf{safehub}}$, which returns an ideal retry (no plaintext). Else return (CASRESULT, accept), set H_{tip}, and notify $\mathcal{F}_{\mathsf{safehub}}$ of acceptance.

On (APPLY leakage L) **from** honest member: emit a collaboration ciphertext of the leaked length under mk_e and append it to the simulated application log at the index $\mathcal{F}_{\mathsf{safehub}}$ assigns. No RefHead is appended and no compare-and-swap runs: an application record is not a push.

On (CONSOLIDATE leakage L) **from** honest member: emit a snapshot ciphertext of the leaked length whose components carry the epochs $\mathcal{F}_{\mathsf{safehub}}$ reports, together with the receipt and binding digest. The logical log is unchanged, so this handler adds no entry to the simulated L; a consolidation that altered it would be a push, which the protocol forbids.

On (FETCH / DELIVER, P, k, b) **from** $\mathcal{A}$: if $\mathcal{A}$ withholds a suffix, deliver the matching ideal prefix k. If $\mathcal{A}$ serves a head that verifies but does not extend the canonical branch, issue (Deliver, P, k, b) naming the corresponding node of Sh, which P then accepts; if $\mathcal{A}$ rolls a party behind d_P, request Forked from $\mathcal{F}_{\mathsf{safehub}}$ before delivery. Accept a head only if signature, tag, chain, and—when the recomputed ff $= 0$, including the undecidable case where the recipient's window begins above the merge-base—admin co-sig verify under simulated state; else **abort** (fig. 13).

On (COMPARE, P, Q) **from** honest parties: compare the checkpoints the parties exchange, not their delivered lengths. If the exchanged checkpoints are not prefix-comparable on their common projected range, request Forked(sid, P, Q) from $\mathcal{F}_{\mathsf{safehub}}$. Both parties may already have accepted their own branch: this handler detects the divergence, it does not precede it.

Fig. 12: QPT UC simulator $\mathcal{S}$, with one handler for each command of $\mathcal{F}_{\mathsf{safehub}}$'s adversary interface. Corruptions are adaptive under secure erasure (theorem 1 and fig. 11).

Simulator $\mathcal{S}$ (QROM, leakage, extraction, aborts; continuation of fig. 12).

Leakage use. On every accepted ideal op, receive ℓ=(type, |ct|, time) plus MLS-visible group/device counts; choose ciphertext lengths and timing to match ℓ exactly. Never read ideal plaintext outside Corrupt / ExposeState.

QROM ($\mathcal{F}_{\mathsf{ro}}$). Answer superposition queries with a compressed/recording oracle [49]. Lazily sample DKR nodes and K_e on first classical extract; **reprogram** only points not yet queried. Crossing a Remove (forward) or forward-only Invite (backward) boundary uses a fresh independent segment root (appendix C-C). Since $\mathcal{F}_{\mathsf{safehub}}$ samples ideal secrets independently of wire transcripts (fig. 11), $\mathcal{S}$ may program AEAD/RO openings on legitimate exposure without fixing those points inside $\mathcal{F}_{\mathsf{safehub}}$.

On (EXPOSESTATE, d) **from** $\mathcal{A}$; $\mathcal{F}_{\mathsf{safehub}}$ returns transient ideal state: open simulated state for epochs in win(u), u the owner of member-device d, exactly as under Corrupt, without adding d to X and without ongoing observation.

On (CORRUPT, d) **from** $\mathcal{A}$ at any time; $\mathcal{F}_{\mathsf{safehub}}$ returns d's abstract state: add the member-device d to X. Call $\mathcal{F}_{\mathsf{safehub}}$.Corrupt($d$); open simulated state for epochs in win(u) so tokens derive exactly those K_e; leave out-of-window ciphertexts as zero-encryptions under independent keys. Erased ephemeral exports need not be opened (definition 1). Forward $\mathcal{F}_{\mathsf{acgka}}$ corruption for the single leaf d, not every device of u: windows are per user, so the opening already covers what d reads while u's other leaves stay honest. Repeat openings at each later epoch while d stays corrupt.

Extraction / abort conditions (exhaustive; each yields a concrete QPT game win, else abort):
(i) *Sig*: honest accept of device/admin/ML-DSA-87 statement never signed by an authorized key (incl. non-FF admin co-sig) $\mapsto$ theorems 4, 5 and 9.
(ii) *MAC*: epoch tag verifies without prior MAC($mk_e, \cdot$) $\mapsto$ lemma 2 and theorem 4.
(iii) *Coll*: distinct heads or CAS digests collide under SHA-512 $\mapsto$ theorem 4 ($\varepsilon_{\mathsf{coll}}$).
(iv) *RO*: $\mathcal{A}$ outputs a hidden DKR preimage before programming $\mapsto$ lemma 7 and theorem 8.
(v) *AEAD*: committed ciphertext opens under a second key to a different in-window plaintext (key-commitment / INT-CTXT) $\mapsto$ theorem 6 and lemma 10.

Invariant. Conditioned on no abort, the simulated server transcript is identically distributed to the ideal execution of $\mathcal{F}_{\mathsf{safehub}}$ with $\mathcal{S}$.

Fig. 13: QPT UC simulator $\mathcal{S}$, continued from fig. 12: oracle programming in the QROM, leakage, adaptive corruption under secure erasure, and the mapping from each abort to the game that bounds it.

weight at most $O(q_{\mathsf{ro}}^2/2^\lambda)$ on any particular λ-bit preimage that $\mathcal{S}$ has not itself revealed, and $\mathcal{S}$ programs only such points, so each hop costs that much. Summing over N gives the claim. The bound is linear in N because the points are distinct by

construction—distinct pad $(\mathsf{nonce}, \mathsf{AD}, \ell, i)$ tuples and distinct DKR $(token, e)$ tuples—so the hops are independent and no single point is reprogrammed twice. □

Lemma 4 (Adaptive opening)**.** *Consider the experiment in which the adversary receives challenge seals for epochs of its choosing, adaptively issues* ExposeState *and* Corrupt*, receives the tokens those calls disclose, and demands that every published in-window ciphertext open to the disclosed plaintext. Its advantage over the ideal execution is at most* $q_p\varepsilon_{\mathsf{ae}} + N_{\mathsf{open}} \cdot O(q_{\mathsf{ro}}^2/2^\lambda)$.

Proof sketch. The experiment is single: challenge seals, exposure, token reveals, and openings share one set of oracles, and the partition of epochs into exposed and protected is read off the transcript rather than fixed in advance. This is what plain IND-CPA cannot provide, since it never permits opening a challenge ciphertext after the distinguisher has seen it. Each demanded opening is one application of theorem 6(iii): programming the pad at that ciphertext's points makes it open to the target plaintext, with the same distribution as a fresh seal of that plaintext. Consistency across all N_{open} of them is lemma 3, which is where the second term comes from. Ciphertexts that remain protected for the whole experiment are never opened, so they are covered by the IND-CPA and INT-CTXT halves of $\varepsilon_{\mathsf{ae}}$, hybridized over $\le q_p$ seals. An epoch exposed *after* its ciphertexts were replaced by zeros is the case that forces the joint treatment: the reduction reclassifies it at the moment of exposure and serves the opening from lemma 1, which is sound because $H_3 \to H_4$ zeros only out-of-window payloads. □

Lemma 5 (Multi-user adaptive EUF)**.** *Let* n_{adm} *admin keys have honest signing oracles, let the adversary expose any of them at any time, and let a win be a fresh verifying statement under a key that was honest at the epoch it authorizes. The advantage is at most* $q_s\varepsilon_{\mathsf{sig}}$*, where* q_s *counts authorization points* $(key, epoch)$.

Proof sketch. The reduction generates all n_{adm} keypairs, embeds the EUF-CMA challenge key at one authorization point drawn uniformly from the $\le q_s$ points, and answers every other signing or exposure query with the keys it holds. A forgery satisfying the win condition names a key honest at that epoch; with probability $1/q_s$ it names the embedded point, and the reduction outputs it. Hence $\mathrm{Adv} \le q_s\varepsilon_{\mathsf{sig}}$, and the guessing factor is the multi-user loss rather than an uncharged abort. Nothing is lost by exposure of other keys, since the reduction holds them. A statement signed by a key already corrupt at the authorizing epoch fails the win condition: it is an authorized action, not a forgery this game must exclude. □

Lemma 6 (Export replacement, QPT)**.** *In uncorrupted epochs,* (ss_e, mk_e) *are indistinguishable from uniform of their respective lengths (*λ *and* 256 *bits) to any QPT adversary; loss* $q_e\varepsilon_{\mathsf{exp}}$*, absorbed into the hybrid assumption* $\varepsilon_{\mathsf{acgka}}$ *(not an extra summand in eq.* (1)*).*

Proof. Let E_0 be the real labeled exports for every epoch; E_i replaces exports of the first i *protected* epochs (definition 2) by independent uniform strings, already-exposed epochs staying real. On $E_{i-1} \to E_i$, $\mathcal{A}_{\mathsf{exp}}$ embeds the $\mathcal{F}_{\mathsf{acgka}}$ labeled-export challenge at protected e_i, forwards MLS transcript and adaptive Corrupt/ExposeState under definition 1, uses the challenge as (ss_e, mk_e) for e_i, and outputs $\mathcal{Z}$'s bit; each hop costs $\varepsilon_{\mathsf{exp}}$ (theorem 9). Secure erasure opens only current non-erased state (lemma 1), so already-exposed epochs lie outside the challenge set; union bound $q_e\varepsilon_{\mathsf{exp}} \le \varepsilon_{\mathsf{acgka}}$. □

Lemma 7 (Window soundness and completeness in the QROM)**.** *For every adaptively grown corrupt set* X *under definition 1, keys derivable from* $\bigcup_{d\in X} \mathsf{tokens}(u(d))$ *equal* $\{K_e : e \in \bigcup_{d\in X} \mathsf{win}(u(d))\}$ *(same* K_e *family), except with advantage at most* $q_{\mathsf{seg}}\varepsilon_{\mathsf{ow2h}} + O(q_{\mathsf{ro}}^2/2^\lambda)$.

Proof. Completeness. By appendix C-C and theorem 7 an honestly issued $(from, to, token)$ derives every $\mathsf{HKDF}(token, \texttt{safehub-v1:dkr-epoch:}e)$ with $from \le e \le to$; Invite/Remove maintain $\mathsf{win}(u)$ as a union of such intervals and Welcome/key-log delivery places exactly those tokens at honest joiners.

Soundness. If a QPT adversary holding all corrupt tokens outputs or distinguishes K_{e^*} for $e^* \notin \bigcup_{d\in X} \mathsf{win}(u(d))$, let s^* be the segment root for e^* (appendix C-C). Forward/backward blocks sample independent roots, so s^* is absent from every corrupt token; extracting K_{e^*} wins single-segment $\mathsf{G}_{\mathsf{dkr\text{-}ow}}$ (theorem 8). Hybridizing over $\le q_{\mathsf{seg}}$ hidden segments plus the compressed-oracle collision term gives the bound. □

Lemma 8 (RefHead authenticity, QPT)**.** *No QPT adversary causes acceptance outside the ideal prefix-or-fork rule, and none causes an honest party to accept a tip that rolls back or contradicts its own anchor, except with probability* $q_s\varepsilon_{\mathsf{sig}} + q_p\varepsilon_{\mathsf{mac}} + \varepsilon_{\mathsf{coll}} + \varepsilon_{\mathsf{ae}}$.

Proof. Run $\mathsf{G}_{\mathsf{ref}}$ (theorem 4) on the adaptively corrupted server transcript; a head accepted outside $\mathcal{F}_{\mathsf{safehub}}$'s prefix-or-fork rule is a win there. Case reduction: unsigned member-device or admin statement↦EUF-CMA; fresh valid epoch tag↦lemma 2; inconsistent or second-key opening↦theorem 6; colliding encodings↦SHA-512 collision. Absent these, chain, anchor, non-FF, and CAS implement the ideal rule, and a rollback or contradiction below d_P is refused by the anchor check alone, without reference to any other party. □

Lemma 9 (Fork detection on COMPARE, QPT)**.** *Let* P, Q *hold accepted views that are not prefix-comparable on* $\mathsf{win}(P) \cap \mathsf{win}(Q)$*. Then* $\textsc{Compare}(P, Q)$ *delivers* Forked *except with the probability of lemma 8. Detection is not claimed before the second acceptance: each party may already have accepted its own branch, and the adversary may delay* COMPARE *indefinitely.*

Proof. Checkpoints are authenticated under $\mathcal{F}_{\mathsf{acgka}}$ keys, so $\mathcal{A}$ may withhold or delay one but cannot forge it (fig. 11 COMPARE). Given two delivered checkpoints, disagreement

on the common projected range is decided entry by entry, so either some accepted head lies outside the prefix-or-fork rule—covered by lemma 8—or the restriction of the two lists differs and $\mathcal{S}$ requests Forked (fig. 12). Silent equivocation therefore survives only for as long as no overlapping honest checkpoint is exchanged. □

Lemma 10 (Payload substitution, QPT)**.** *Conditioned on lemmas 6 and 7, zeroing protected ciphertexts changes QPT advantage by at most $q_p\varepsilon_{\mathsf{ae}}$ (IND-CPA component; INT-CTXT / key-commitment charged in $H_4 \to H_5$ / theorem 4, not again here).*

Proof. In $\mathsf{G}_{\mathsf{pay}}$ the challenger prepares the H_3 hybrid and, for each protected AEAD seal, bit b selects real plaintext vs equal-length zeros under the same key and AD. $\mathcal{A}_{\mathsf{ae}}$ is a multi-user IND-CPA adversary against theorem 6: embeds challenge seals at protected epochs, forwards classical Enc, and uses lemmas 6 and 7 to keep challenge keys independent of the corrupt-window view. With $\leq q_p$ seals the hybrid costs $q_p\varepsilon_{\mathsf{ae}}$; integrity side-conditions deferred to $H_4 \to H_5$; in-window seals stay real so Corrupt opens remain consistent (lemma 1). □

F. Hybrid sequence

Each hop transforms real H_0 towards ideal H_5, names its reduction, and preserves adaptive corruptions under definition 1.

$H_0 \to H_1$ **(uncompromised exports).** Replace every uncompromised (ss_e, mk_e) by uniform strings. $\mathcal{A}_1$ embeds the $\mathcal{F}_{\mathsf{acgka}}$ labeled-export challenge at a random protected epoch (lemma 6 and theorem 9); erasure keeps already-exposed epochs out of the challenge set. Bound $q_e\varepsilon_{\mathsf{exp}}$, charged to $\varepsilon_{\mathsf{acgka}}$.

$H_1 \to H_2$ **(out-of-window grants).** Replace protected key-log deltas and Welcome grants outside corrupt windows by equal-length zeros. $\mathcal{A}_2$ embeds the $\mathcal{F}_{\mathsf{acgka}}$ Welcome challenge and reuses H_1's idealized exports; absorbed into $\varepsilon_{\mathsf{acgka}}$, with forged admin-gated membership mapping to theorems 5 and 9.

$H_2 \to H_3$ **(out-of-window** DKR**).** Replace out-of-window DKR nodes and K_e by lazily sampled $\mathcal{F}_{\mathsf{ro}}$ values. $\mathcal{A}_3$ is lemma 7's adversary: embed a random protected segment root as the $\mathsf{G}_{\mathsf{dkr\text{-}ow}}$ challenge (theorem 8), answer corrupt-window token oracles from the remaining roots, and abort if a classical extract precedes programming. Fresh segment roots (appendix C-C) keep the hidden root absent from corrupt tokens. Bound $q_{\mathsf{seg}}\varepsilon_{\mathsf{ow2h}} + N_{\mathsf{dkr}} \cdot O(q_{\mathsf{ro}}^2/2^\lambda)$ (lemma 3).

$H_3 \to H_4$ **(protected payloads).** Replace plaintexts outside corrupt windows by equal-length zeros. Because $\mathcal{Z}$ may expose an epoch after its ciphertexts are published, the hop is the single experiment of lemma 4: it reclassifies as those queries arrive and pays N_{open} programmed points on top of $q_p\varepsilon_{\mathsf{ae}}$.

$H_4 \to H_5$ **(ideal acceptance).** Route acceptance, membership, authorization, CAS retries, Fetch/Deliver, Compare/Forked, and corruption through $\mathcal{F}_{\mathsf{safehub}}$ via figs. 12 and 13. $\mathcal{A}_5$ outputs the game witness on abort and, absent one, an honest accept outside the ideal rule (lemma 8 and theorem 4), an undetected cross-party fork (lemma 9), or an unsigned authorization (theorem 5 and lemma 5); open consistency is lemma 1. Then $H_5 \equiv \mathsf{EXEC}_{\mathcal{F}_{\mathsf{safehub}},\mathcal{S},\mathcal{Z}}$.

a) Advantage accounting: Summing the hops gives eq. (1) with no primitive counted twice. Confidentiality and integrity draw on one $\varepsilon_{\mathsf{ae}}$ over one q_p budget; outer-AEAD and epoch-tag HMAC share $\varepsilon_{\mathsf{mac}}$; the programmed points of the two QROM hops sum into N_{prog}; $q_e\varepsilon_{\mathsf{exp}}$ is absorbed into $\varepsilon_{\mathsf{acgka}}$; and certification carries no summand, being idealized in $\mathcal{F}_{\mathsf{ca}}$.

G. Game-based security definitions and proofs

Standalone games modularizing the UC theorem. All adversaries are QPT; hash/KDF/HMAC modeling uses the QROM unless stated otherwise.

a) Game $\mathsf{G}_{\mathsf{hist}}$ (history containment).: Challenger runs honest Invite with grant $h{=}e{+}1$ (fresh backward_block root; appendix C-C), encrypts challenge payloads under K_{e^*} for a random $e^*{<}h$, and gives the adversary the forward-only tokens, all post-join ciphertexts, and quantum RO access. Bit b: real payload vs zeros. Adversary outputs b'.

Theorem 2 (History IND under QROM)**.** *Any QPT adversary against $\mathsf{G}_{\mathsf{hist}}$ has advantage at most $q_{\mathsf{seg}}\varepsilon_{\mathsf{ow2h}} + O(q_{\mathsf{ro}}^2/2^\lambda) + q_p\varepsilon_{\mathsf{ae}}$.*

Proof. Step 1 (window). If the adversary derives K_{e^*} from the forward-only token, it wins $\mathsf{G}_{\mathsf{dkr\text{-}ow}}$ on the pre-join segment (theorem 8): the joiner token uses an independent root, so widening $from$ does not reveal the pre-join root. Loss $q_{\mathsf{seg}}\varepsilon_{\mathsf{ow2h}} + O(q_{\mathsf{ro}}^2/2^\lambda)$ by lemma 7.

Step 2 (AEAD). Else K_{e^*} is independent of the adversary's view. Replace the challenge plaintext by zeros via multi-user IND-CPA of the committing AEAD (theorem 6 and lemma 10). Loss $q_p\varepsilon_{\mathsf{ae}}$. □

b) Game $\mathsf{G}_{\mathsf{pcs}}$ (PCS / FS).: After an honest epoch-advancing Commit at epoch e — a standalone Rotate, or a Remove, Invite, or SetAdmin, each of which samples the forward segment root itself — the adversary is given an ExposeState transcript of a device at a time $t \leq e$ (transient exposure, not ongoing corruption) and all prior server transcripts. The freshness condition is definition 3. Challenge bit encrypts a fresh payload under $K_{e'}$ for $e'{>}e$ (post-event) vs zeros.

Theorem 3 (PCS/FS after Rotate)**.** *Advantage $\leq \varepsilon_{\mathsf{exp}} + q_{\mathsf{seg}}\varepsilon_{\mathsf{ow2h}} + O(q_{\mathsf{ro}}^2/2^\lambda) + q_p\varepsilon_{\mathsf{ae}}$.*

Proof. Exports. Honest Rotate refreshes $\mathcal{F}_{\mathsf{acgka}}$ exports for epochs $> e$. Under freshness, ExposeState does not include those exports; distinguishing costs $\varepsilon_{\mathsf{exp}}$ (lemma 6 and theorem 9).

Forward block. Every epoch-advancing Commit samples a fresh independent forward segment root (appendix C-C) *and* replaces (ss_e, mk_e), so pre-event tokens cannot derive $K_{e'}$ except by winning $\mathsf{G}_{\mathsf{dkr\text{-}ow}}$ (theorem 8). Remove is such a Commit and therefore samples the fresh root itself, rather than depending on a separately issued Rotate.

AEAD. Conditioned on independent $K_{e'}$, IND-CPA yields $q_p\varepsilon_{\mathsf{ae}}$. □

c) Game $\mathsf{G}_{\mathsf{ref}}$ *(*RefHead */ fork).:* Adversary sees the head log and may inject candidates. Wins by causing an honest client to accept a head outside the prefix-or-fork rule, or by producing two accepted non-prefix-comparable heads without Forked.

Theorem 4 (RefHead unforgeability)**.** *Advantage* $\leq q_s\varepsilon_{\mathsf{sig}} + q_p\varepsilon_{\mathsf{mac}} + \varepsilon_{\mathsf{coll}} + \varepsilon_{\mathsf{ae}}$.

Proof. Let H be a head accepted by an honest verifier (section IV-E), and let H_0 be its canonical core: every field except pusher_sig and admin_cosig, which the chain link SHA-512(encode(H)) does cover. Acceptance requires each of the following, so a head no honest party produced yields one of the listed wins.

Sig (device). $\mathsf{Verify}(\mathsf{vk}_P, H_0, \mathsf{pusher_sig}) = 1$ for the roster-bound leaf at epoch e. If honest P never signed H_0, embedding the challenge key as that leaf among $\leq q_s$ signing queries gives an ML-DSA-87 forgery: $q_s\varepsilon_{\mathsf{sig}}$.

Sig (admin non-FF). The gate is the *recomputed* relation: $\mathsf{ff} = 1$ iff refs' extends the verifier's refs by fast-forward, and $\mathsf{ff} = 0$ both when it does not and when ancestry is undecidable because the merge-base lies below the graft. A head with $\mathsf{nonff} \neq 1 - \mathsf{ff}$ is rejected, and $\mathsf{ff} = 0$ requires a co-signature from the authorizing roster, forging which wins theorem 5. Keying on the declared bit instead would leave force-push undischarged.

MAC. A verifying tag with no prior honest $\mathsf{MAC}(mk_e, \cdot)$ wins lemma 2: $q_p\varepsilon_{\mathsf{mac}}$.

AEAD. enc_refs and dek_wrap are committing under K_e^{seq}; an opening inconsistent with an honest seal, or a second opening key, wins theorem 6: $\varepsilon_{\mathsf{ae}}$.

Coll. Colliding head encodings or CAS digests under SHA-512 win the quantum collision game: $\varepsilon_{\mathsf{coll}}$.

Anchor. A tip that does not extend the local anchor $(\mathsf{seq}, H(\mathsf{RefHead}))$ is refused outright, so a CAS advertisement that truncates the log is a withheld suffix. Two non-prefix-comparable valid chains may both be accepted, by different parties; that is not a win here and is the subject of lemma 9. Leaf keys come from the roster at the head's mls_epoch, and IAS-gated membership authenticity is charged to $\mathcal{F}_{\mathsf{acgka}}$ (theorem 9) rather than counted again. □

d) Game $\mathsf{G}_{\mathsf{cosig}}$ *(admin authorization).:* An adversary without a corrupted administrator wins by producing either a non-fast-forward RefHead or a consolidation receipt accepted by an honest verifier. Both are ML-DSA-87 signatures by an administrator over a domain-separated statement, so one game covers them.

Theorem 5 (Admin authorization EUF)**.** *Advantage* $\leq q_s\varepsilon_{\mathsf{sig}}$ *under QPT EUF-CMA of ML-DSA-87 (standard-model signature assumption; messages classical). The bound covers non-fast-forward co-signatures and consolidation receipts alike; it therefore introduces no summand beyond the* $q_s\varepsilon_{\mathsf{sig}}$ *already present in eq.* (1).

Proof. Reduction $\mathcal{A}_{\mathsf{cosig}}$. $\mathcal{A}_{\mathsf{cosig}}$ receives an EUF-CMA challenge key vk^*, plants it as an honest administrator verification key (via $\mathcal{F}_{\mathsf{ca}}$), and answers classical signing queries for honest admin operations by the EUF-CMA signing oracle. When the adversary outputs a non-FF head, or a consolidation receipt, accepted by an honest verifier, the embedded admin_cosig is a signature on a new statement (domain-separated `admin-cosig` binding $\mathsf{sid} \,\|\, e \,\|\, \mathsf{op} \,\|\, \ldots \,\|\, H(\mathsf{roster})$) that was never queried. Output that forgery. The game is multi-user and adaptive, with the loss carried in $q_s\varepsilon_{\mathsf{sig}}$ rather than in an uncharged abort (lemma 5). □

e) Game $\mathsf{G}_{\mathsf{aead}}$ *(equivocable committing AEAD).:* Multi-user IND-CPA, INT-CTXT, key-commitment, and *message equivocation* games for the random-oracle-pad Encrypt-then-MAC transform, with HMAC modeled via QROM.

Theorem 6 (Equivocable committing AEAD security)**.** *Model* $\mathsf{HMAC}_{K_{\mathsf{enc}}}$ *(the pad) as a programmable random oracle and assume QPT SUF-CMA of HMAC-SHA-512-256 in the QROM. With per-head seal subkeys and deterministic counters, the transport transform is (i) IND-CPA and INT-CTXT with advantage* $\varepsilon_{\mathsf{ae}} + \varepsilon_{\mathsf{mac}}$*; (ii) key-committing with the stated BHT/CNS margins; and (iii)* message-equivocable*: given a target plaintext* m' *of the sealed length, a published ciphertext, and the sealing key, a simulator programs one pad point so the ciphertext opens to* m'*, identically distributed to a fresh seal of* m' *unless a pad preimage was already queried (*$\mathsf{Bad}_{\mathsf{ro}}$*). Here* $\varepsilon_{\mathsf{ae}}$ *collects IND-CPA and INT-CTXT and* $\varepsilon_{\mathsf{mac}}$ *is HMAC SUF-CMA (lemma 2); in eq.* (1) *they are the separate summands of* $q_p(\varepsilon_{\mathsf{ae}} + \varepsilon_{\mathsf{mac}})$.

Proof. Construction. $\mathsf{body} = \mathsf{pad} \oplus (\mathsf{cblock} \,\|\, m)$ with $\mathsf{pad_key} = \mathsf{HMAC}_{K_{\mathsf{enc}}}(\texttt{aead-pad} \,\|\, \mathsf{nonce} \,\|\, \mathsf{AD} \,\|\, \ell)$, $\mathsf{pad}_i = \mathsf{SHA\text{-}512}(\mathsf{pad_key} \,\|\, i)$, outer HMAC-SHA-512-256 over $(\mathsf{AD}, \mathsf{body})$ under K_{mac}, and $K_{\mathsf{enc}} \| K_{\mathsf{mac}} \leftarrow \mathsf{HKDF}(K, \texttt{safehub-v1:aead-keys})$.

(i). Encrypt-then-MAC: a verifying body never honestly produced is an HMAC forgery (lemma 2), and tagging is classical throughout. For confidentiality the pad is a random function of $(\mathsf{nonce}, \mathsf{AD}, \ell, i)$, so on non-repeating inputs body is uniform and independent of m; per-head K_e^{seq} and deterministic counters keep inputs unique under the 2^{32} cap, on which honest clients rotate.

(ii). Two openings $(K \neq K', C, \mathsf{AD})$ must agree on the 48-byte commit block under independent key derivations, which is the stated BHT/CNS margin. Equivocation programs the message under a *fixed* key and so leaves key binding untouched.

(iii). To open a published $(\mathsf{nonce}, \mathsf{body}, \tau)$ to m', program the pad so that $\mathsf{body} = \mathsf{pad} \oplus (\mathsf{cblock} \,\|\, m')$; τ is unchanged because it authenticates body, which is unchanged. The result is distributed as an honest seal of m' unless a pad preimage was already queried, and consistency across many such openings is lemma 3. Seal keys are independent across heads and epochs,

so hybridizing over $\leq q_p$ seals gives the q_p factor; epoch tags reuse this HMAC under independent mk_e and share $\varepsilon_{\mathsf{mac}}$. □

f) Game $\mathsf{G}_{\mathsf{dkr\text{-}ow}}$*.:* Given tokens that exclude a hidden segment, the adversary cannot derive a node on that segment.

Theorem 7 (DKR interval unlock correctness)**.** *Honestly issued tokens for* $[a, b]$ *derive every* K_e *with* $a \leq e \leq b$*. What* any *holder can derive is the union of the segments whose roots it holds; crossing a segment boundary reduces to theorem 8. Exactness is claimed at segment granularity, since every window-creating or window-narrowing event starts a segment (appendix C-C, appendix H).*

Proof. Immediate from appendix C-C: $\mathsf{derive}(I, e)$ is a deterministic function of $(token, e)$ on $[from, to]$ and rejects elsewhere, and each block samples a fresh independent root, so tokens issued earlier derive nothing under it. This is what prevents a pre-Rotate token from reaching a post-Rotate epoch, and a forward-only token from reaching a pre-join epoch even when the adversary widens $from$. Cross-segment derivation needs the other root (theorem 8). □

Theorem 8 (DKR puncturing / one-wayness in QROM)**.** *Advantage against* $\mathsf{G}_{\mathsf{dkr\text{-}ow}}$ *is at most* $q_{\mathsf{seg}}\varepsilon_{\mathsf{ow2h}} + O(q_{\mathsf{ro}}^2/2^\lambda)$.

Proof. Game. Challenger samples independent roots $s_1, \ldots, s_{q_{\mathsf{seg}}} \leftarrow \{0,1\}^\lambda$; adversary receives tokens for every segment but a hidden index j^* plus quantum access to $\mathcal{F}_{\mathsf{ro}}$, and wins by outputting a hidden-segment node $y = \mathcal{F}_{\mathsf{ro}}(s_{j^*} \,\|\, \mathsf{label})$ (or $\mathsf{HKDF}(s_{j^*}, \texttt{safehub-v1:dkr-epoch:}e)$) or distinguishing one from uniform.

Reduction. Model each segment derivation as an independent RO-derived family under the compressed oracle [49]. Any QPT adversary that outputs or distinguishes a hidden-segment node must place non-negligible amplitude on s_{j^*}, reducing to semi-classical O2H / measure-and-reprogram [48], [50], [59] at per-segment loss $\varepsilon_{\mathsf{ow2h}}$ (no tighter asymptotic claim). Rotate (forward) and forward-only Invite (backward) sample independent roots (appendix C-C), so a hybrid over $\leq q_{\mathsf{seg}}$ hidden segments plus $O(q_{\mathsf{ro}}^2/2^\lambda)$ gives the union bound. Does not rely on lemma 7 and theorem 2, which reduce *to* this theorem. □

g) Game $\mathsf{G}_{\mathsf{acgka}}$ *(membership authenticity).:* Black-box game for labeled export indistinguishability and Welcome confidentiality of $\mathcal{F}_{\mathsf{acgka}}$ against QPT adversaries under adaptive corruptions and secure erasure. This is a *standard hybrid assumption* on an ideal administrated CGKA (A-CGKA), instantiated by NIST PQ Category 5 MLS (administration [8]; MLS/CGKA and path secrecy [7], [16], [17]).

Theorem 9 (Membership authenticity via $\mathcal{F}_{\mathsf{acgka}}$)**.** *If* $\mathcal{F}_{\mathsf{acgka}}$ *is QPT-secure in the sense of section III (adaptive under secure erasure; administration [8]; MLS/CGKA [7], [16], [17]; advantage* $\varepsilon_{\mathsf{acgka}}$*), then forged admin-gated Add/Remove/Rotate that pass IAS checks imply either an ML-DSA-87 forgery or a break of* $\mathcal{F}_{\mathsf{acgka}}$*. Honest membership transcripts are indistinguishable from the hybrid ideal interface used by* $\mathcal{S}$*.*

Proof. The $\mathcal{F}_{\mathsf{acgka}}$ interface is as specified in appendix A-D, and TreeKEM, path secrets, and IAS are not re-proved. What is open is the concrete bound: for the Category-5 suite no adaptive-plus-erasure QROM export or Welcome bound is known ([7], [16] selective or generic-group, [17] adaptive with depth-dependent loss, [8] administration rather than PQ adaptive exports), so $\varepsilon_{\mathsf{acgka}}$ is heuristic and appears only under corollary 1. This localizes one summand rather than weakening theorem 1, which is a theorem *in* the hybrid. Only $H_0 \to H_1$ and $H_1 \to H_2$ consume it: an Add/Remove/Rotate passing IAS checks without honest-admin authorization yields either an ML-DSA-87 forgery (theorem 5 and lemma 5) or a break of $\mathcal{F}_{\mathsf{acgka}}$ membership authenticity, while adaptive corruption for storage and windows is handled by $\mathcal{S}$ (lemma 1) with no appeal to an external adaptive UC theorem for TreeKEM. □

H. Tightness

Loss is dominated by terms linear in q_p, q_s. The quadratic term is harmless here: at λ=384 and $q_{\mathsf{ro}} \leq 2^{64}$, $N_{\mathsf{prog}} q_{\mathsf{ro}}^2/2^\lambda \approx N_{\mathsf{prog}} 2^{-256}$, so integrity stays above 128 bits (definition 4 and table IX) for any polynomial programming budget while AES-256 remains at its Grover floor of 2^{128}. O2H losses stay parameterized per segment; we claim no asymptotic tightness beyond the cited bounds.

APPENDIX D
DESIGN SUPPLEMENTS

This appendix collects the full notation table, secondary design figures, and numbered protocol listings. Body summaries appear in section IV; the capability map is table III.

A. Full notation

B. Secondary design figures

Key schedule and RefHead verification are in the body (figs. 5 and 6).

a) Key lifetimes: Along the derivation chain of fig. 6, each key lives for a different span. The MLS epoch holds until the next Commit or Rotate, and the exports (ss_e, mk_e) and the epoch key K_e live exactly for epoch e. Below K_e, the per-head subkey K_e^{seq} is capped at 2^{32} seals and K_e^{cas} is a separate `cas-obj` derivation rather than mk_e, while a DEK lives for one push. Off the chain, DKR tokens live for the grant $[h, \infty)$ the invite issued, and admin keys are long-lived under HSM custody until rotated.

C. Push, Fetch, Merge, Rebase, and Membership

a) Push: The steps are listed in section IV-C; three details are expanded here. Chunk encryption binds $\mathsf{aad} = (\mathsf{repo}, \mathsf{push_id}, i, n)$, so a chunk cannot be replayed into another push or another position; uploads are content-addressed and therefore idempotent under retry; and sig is the pusher's ML-DSA-87 device signature under its own domain separator. Verification recomputes the fast-forward relation from decrypted old and new refs against the local Git DAG whenever

TABLE VIII: Terms, notation, and abbreviations used throughout the paper.

Git objects and the ref store	
ref	a named branch or tag pointer
tip	the commit a ref currently names
refs	the ref-name → tip map for a repository
RefHead	encrypted, hash-chained ref manifest carrying refs; the sole mutable state the host can attack
seq	sequence number of a RefHead entry in the head log
nonff	flag marking a non-fast-forward (force-push) update
Forked	event raised when two accepted views are not prefix-comparable
Membership, epochs, and history	
e	epoch index; a new epoch opens on each membership change or key rotation
win(u)	history window: the union of disjoint epoch intervals user u may decrypt
forward-only	grant decrypting only from the join epoch onward
leaf	MLS ratchet-tree position occupied by one member-device
member-device	a user's device holding one MLS leaf and its credential key
device-signed	signed under the producing member-device's leaf credential key, hence attributable to that one device
sid	session identifier; one instance per repository
Keys and cryptographic values	
λ	security parameter (384 for DKR/KDF outputs)
ss_e	per-epoch seed exported from MLS, feeding the DKR chains
mk_e	per-epoch MAC and collaboration key, held by every member
K_e	epoch key derived from the DKR sequence; never an AEAD key itself
K_e^{seq}	per-head seal subkey HKDF(K_e, `head-seal`‖seq); wraps DEK s and encrypts refs
DEK	data-encryption key, fresh per push, sealed under K_e^{seq}
ML-KEM-1024	NIST PQ Category-5 key-encapsulation mechanism (FIPS 203)
ML-DSA-87	NIST PQ Category-5 signature scheme (FIPS 204)

Formal model	
$\mathcal{F}_{\mathsf{safehub}}$	ideal functionality for encrypted Git hosting
Π_{safehub}	the SAFEHUB protocol realizing $\mathcal{F}_{\mathsf{safehub}}$
$\mathcal{F}_{\mathsf{acgka}}$	administrated CGKA hybrid functionality
$\mathcal{F}_{\mathsf{ca}}$	certificate authority binding identities to ML-DSA-87 keys
$\mathcal{F}_{\mathsf{ro}}$	quantum-accessible random oracle
ℓ	leakage function of the ideal functionality
Srv	the (malicious) hosting server; a protocol role, distinct from $\mathcal{A}$
M	member set of a repository
Adm	administrator set, Adm ⊆ M
L	append-only logical push sequence held by $\mathcal{F}_{\mathsf{safehub}}$
d_P	largest prefix length of L delivered to party P
X	set of corrupt member-devices
$\mathcal{A}$	adversary
$\mathcal{S}$	simulator
$\mathcal{Z}$	environment
Abbreviations: storage	
CAS	content-addressed storage
CS	compare-and-swap (atomic conditional append)
LFS	Git Large File Storage
Abbreviations	
MLS	Messaging Layer Security
A-CGKA	administrated CGKA
DKR	dual key regression
GKP	group key progression
PCS	post-compromise security
FS	forward secrecy
BHT	Brassard–Høyer–Tapp quantum collision search
CNS	Chailloux–Naya-Plasencia–Schrottenloher collision search (low memory)

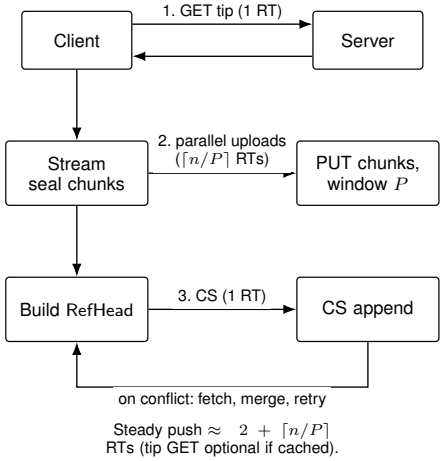

Fig. 14: Round trips required by a push: a tip fetch, omitted when the tip is cached; a parallel chunk upload of window P; then a compare-and-swap append. Each conflict retry adds one fetch and one merge. CS denotes compare-and-swap and RT a round trip.

the merge-base is present. A grafted forward-only verifier whose DAG starts at the graft cannot decide ancestry when the merge-base lies below it; that update is *unverifiable*, is treated as non-fast-forward, and is rejected unless an admin co-signature authorizes it. Signatures cover one canonical *core*—sid, epoch, operation, sequence, predecessor head, refs commitment, bundle root, roster digest, and policy version—which excludes the signature fields themselves, so neither the device signature nor the co-signature covers a value containing it. The co-signature is bound to that core; binding it to H(RefHead) would be circular, since RefHead carries the co-signature. Including the roster digest is what stops a co-signature issued under one roster from authorizing the same refs under another. A sender-declared non_ff bit disagreeing with the recomputed relation is rejected, and the co-signature authorizes ref policy only: the RefHead chain must extend its predecessor unconditionally.

b) Fetch: Fetch the latest RefHead; verify signature, chain descent from the last-seen head, epoch tag under mk_e, and (for each new bundle) the SHA-512 root; unwrap DEK s with $K_{e'}$ from interval state; decrypt; import objects; update refs per the manifest; persist the new anchor.

c) Merge and rebase: Local Git on the decrypted working copy (section IV-C). They are not wire operations and have no host protocol: a later Push seals the resulting objects, and a rebase that rewrites a published tip is a non-fast-forward Push (admin co-signature).

d) Membership: Invite(u, h): admin fetches u's KeyPackages via $\mathcal{F}_{\mathsf{ca}}$-bound credentials and runs $\mathcal{F}_{\mathsf{acgka}}$.Add. A KeyPackage is a pre-join credential and carries no repository secret; the interval grant travels in the confidential Welcome, whose secrecy is the joiner's, and a backward block is inserted iff $h{=}e{+}1$. For forward-only grants the inviter also publishes the graft as a content-addressed encrypted object, named by identifier, digest, epoch, and authorization, sealed inside u's window. Remove(u): $\mathcal{F}_{\mathsf{acgka}}$.Remove over all of u's leaves, a window cap, and the epoch advance that separates the keys, in one commit. Rotate: epoch advance without membership

change. SetAdmin(u, b): admin-gated roster update. All four are admin-signed; clients validate admin signatures against the group-context roster before merging.

APPENDIX E
PARAMETER SUMMARY

Recommended suite (NIST PQ Category 5; split claim of definition 4):

`MLS_256_MLKEM1024_AES256GCM_SHA512_MLDSA87`

with application digests and tags sized for >128-bit PQ security under BHT/QRACM, CNS/low-QRACM [57], [58], and QROM window bounds, while AES-256 confidentiality inside the MLS ciphersuite remains at the Grover floor 2^{128} [60]. Object sizes for ML-KEM-1024 (FIPS 203) [44] and ML-DSA-87 (FIPS 204) [45] are table XII. Transport confidentiality comes from the HKDF-SHA-512 RO-pad Encrypt-then-MAC with a 48-byte (λ=384) commit block and outer HMAC-SHA-512-256, AES-256-GCM appearing only inside MLS (appendix G); SHA-512 covers MLS KDF/hash, the RefHead chain, and CAS/bundle roots; DKR/RO outputs are λ=384 bits under the domain prefix `safehub-v1:`.

Definition 4 (Post-quantum claim (split))**.** *Under query budgets $q_{\mathsf{ro}} \le 2^{64}$, $q_{\mathsf{seg}} \le 2^{20}$, $q_e, q_p, q_s \le 2^{40}$ (enterprise lifetime upper bounds; appendix C-A):*

- $>$ 128***-bit target:*** *asymmetric (ML-KEM-1024/ML-DSA-87), 512-bit digests (BHT/CNS/Grover preimage [57], [58], [60]), tags/MACs, QROM window soundness, and committing-AEAD key commitment under CNS are parameterized so concrete quantum attack cost (or QROM failure probability) is* strictly above 128 *bits.*
- $=$ 128 ***Grover floor:*** *AES-256 transport confidentiality is exactly the NIST PQ Category 5 Grover key-search cost 2^{128} and is* not *claimed higher. Committing-AEAD key commitment under BHT meets $\approx 2^{128}$ (same floor); CNS gives $\approx 2^{153}$.*

table IX records which row is which.

TABLE IX: Post-quantum margins, with the prototype site for each entry given as `crate/file:fn`. Digests are n=512 bits, λ=384, t=256, and the AEAD commit block is 48 bytes. The quantum-attack column gives the cost of a concrete quantum algorithm and the QROM column the soundness of the proof model; the claim is more than 128 bits, measured against the AES-256 Grover floor.

Attack / bound	Model	Bits	Claim	Trace
SHA-512 preimage	Grover $2^{n/2}$	256	>128	`ids.rs:of_ciphertext`
SHA-512 collision	BHT $2^{n/3}$ (+QRACM)	170	>128	`ids.rs:of_ciphertext`
SHA-512 collision	CNS $2^{2n/5}$ (low QRACM)	204	>128	`refs.rs:hash`
Multi-target ($T \le 2^{64}$)	QRACM	224	>128	`ids.rs:of_ciphertext`
Epoch / outer HMAC tag	online guess	256	>128	`aead.rs:seal`
Committing AEAD (48 B)	BHT / CNS	128/153	floor/>128	`aead.rs:seal`
DKR window soundness[†]	QROM	>128	>128	`dkr.rs`
AES-256 key search, MLS suite	Grover	128	$=128$ floor	OpenMLS
RefHead compare-and-swap append	BHT / CNS on chain hash	170/204	>128	`local.rs:cas_append`
ML-KEM-1024/ ML-DSA-87	NIST PQ Cat. 5	>128	>128	OpenMLS / `mldsa.rs`

[†]With λ=384, $q_{\mathsf{ro}} \le 2^{64}$: $q_{\mathsf{ro}}^2/2^{\lambda} = 2^{-256}$ (appendix C-H).

A. Scope of the post-quantum claim

Three claims, in decreasing strength. *UC security against QPT adversaries in the QROM* is theorem 1, under the assumptions of appendix C-A. *Concrete margins* are the split of definition 4, row by row in table IX. *Residual limits* are appendix H, of which three are inherent rather than engineering gaps: adaptive UC without secure erasure is open, $\mathcal{F}_{\mathsf{acgka}}$ adaptive export under erasure is a hybrid assumption [7], [8], [17], and O2H losses are not asymptotically linear.

a) Operational mitigations (not theorems): Enforce definition 1; prompt admin Remove/Rotate; schedule consolidation after Remove; prefer forward-only grants for CI/contractors; keep admin keys offline/HSM where possible. These shrink residual windows when erasure is imperfect; they do not replace the erasure hypothesis.

B. Quantum attack-cost models

Table IX tabulates the costs; the models behind them are Grover preimage $\Theta(2^{n/2})$, BHT collision $\Theta(2^{n/3})$ with QRACM [57], and CNS $\Theta(2^{2n/5})$ with polynomial quantum memory [58], each at n=512. The digest length is forced by the weakest of these: BHT against a 256-bit digest would leave only 85 bits, which is why the design uses 512 throughout. Protocol soundness then aggregates QPT signature, MAC, and AEAD advantages, quantum collision finding, and the window term of lemma 7, with 256-bit tags giving ideal forgery 2^{-256} per online verification and the QROM budget working out as in appendix C-H.

TABLE X: Operational ceilings and defaults (policy, not security constants). Changing chunk size or consolidation cadence affects traffic leakage and residual ciphertext lifetime, not ideal epoch windows.

Limit / parameter	Value / note
Max evaluated repo	5–500 MB sweep; 100 files, 100 pushes
Concurrent pushers	Compare-and-swap retry ≤ 8
Members (eval)	Join sweep n=10..100
DKR segment capacity N	2^{20}; exhaustion re-init (no window widen)
Bundle chunk / K_e^{seg} seal cap	4 MiB; 2^{32} seals / head subkey
Consolidation	64 pushes or 30 days; immediate after Remove
Anti-rollback / exports / domain	OS keychain; `transport,refs`; `safehub-v1:`
Collision digests / tags / λ / AEAD block	SHA-512; 256-bit; 384; 48 B
Git object format	`sha1` or `sha256` (`repo create --object-format`); clone follows the writer
Unsupported Git / platforms	LFS, partial clone; Windows not claimed
Adversary model	QPT + QROM; adaptive+erasure

The post-quantum MLS ciphersuite is draft/experimental in OpenMLS; a private-use code point is acceptable for a closed prototype, while interoperable deployment must track the final registry. Under defaults ($N = 2^{20}$, Rotate-driven re-seeds), q_{seg} typically tracks membership churn and stays $\ll 2^{20}$ in enterprise lifetimes (definition 4). Residuals: adaptive corruption without erasure, and $\mathcal{F}_{\mathsf{acgka}}$ as a QPT adaptive+erasure assumption (appendices E-A and H). The reference implementation aligns CAS digests, λ=384 DKR/RO outputs, and domain separation with this appendix (SHA-512 content addresses; BLAKE3 [61] only for non-security local indexing).

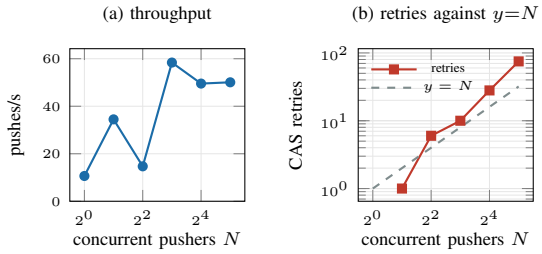

Fig. 15: Head-log contention: N real clients pushing to N distinct branches against one server. A single compare-and-swap log per repository serializes every push in that repository, which a plaintext forge does not do. No push was lost at any N (0 failures in total, within the retry budget of 8); throughput saturates rather than scaling, and retries grow faster than N. A starvation configuration (one 8 MiB push against four concurrent small pushes) completed in 1.02 s. These are single-host measurements: at $N{=}32$ the clients contend with the server for the same cores, so the flattening combines client scheduling with head-log contention.

APPENDIX F
EXTENDED EVALUATION

Material supporting section VI: head-log contention, attack reconstructions, forward-only exposure, consolidation, deployment, and secondary plots. Per-invite cost against current n runs 17 ms at $n{=}10$ to 91 ms at $n{=}100$, and 24–35 ms across the 20–30-member private-team range, which table VI states cumulatively.

A. Consolidation after Remove

Remove is itself the rekey: it revokes membership, caps the window, and refreshes the epoch exports in one operation, so a removed member cannot derive K_e for any later epoch. Consolidation is a separate, admin-authorized honest-storage compaction that acts *within* an epoch: every component keeps the key that already sealed it, so a checkpoint grants its reader nothing outside that reader's window. What bounds clone growth is that checkpoint—one self-contained tree as of sequence s, after which readers replay only the tail—and accepting it is safe only if a reader can distinguish a real checkpoint from a fabricated one without performing the replay it is avoiding. The receipt therefore binds the head hash at s, a digest of the ref map sealed inside that head, and the checkpoint's own root; a reader opens the anchor head it already holds and recomputes the digest in constant work. The objects need no separate commitment, since the ref map names content-addressed Git object ids.

The library implements receipt construction, admin and quorum signing, window-scoped Merkle verification, and the anchor check under eighteen tests with each compaction guard mutation-checked, and the CLI reaches it: `shub repo consolidate` plans the compaction over the head chain and signs the receipt in 0.19 ms per head. No server path applies that receipt, so consolidation reclaims no storage and the clone curve of § VI-A0d stands unmitigated—a consequence of what is implemented rather than of a measurement. The consolidation figures we report are therefore the receipt path alone, and bounding clone growth remains outstanding implementation work. Compaction would not in any case cryptographically erase ciphertext a malicious host chose to retain (appendix H).

B. Concurrency and attack scenarios

A single compare-and-swap head log serializes every push to a repository, whereas in a plaintext forge pushes to disjoint branches do not contend. With N clients pushing to N distinct branches concurrently, no push was lost at any N within the retry budget of 8 (table X):

N	wall (s)	accepted	failed	CAS retries
1	0.094	1	0	0
2	0.058	2	0	1
4	0.272	4	0	6
8	0.137	8	0	10
16	0.323	16	0	28
32	0.639	32	0	75

Retries grow faster than N, reaching 75 at $N{=}32$, and throughput settles near 50 pushes/s for $N \geq 16$. Below that the whole sweep finishes in under 300 ms, so the derived rate is dominated by client start-up rather than contention and ranges from 11 to 58 pushes/s without a trend. A starvation configuration—one 8 MiB push against four concurrent small pushes—completed the large push in 1.02 s (fig. 15). Of the eleven reconstructed scenarios, five are measured: a malicious-host tip fork is detectable, a removed member colluding with the server is prevented, forward-only pre-join ciphertext stays contained, a force-push without admin co-signature is rejected, and a concurrent head-log race recovers. Full table: table XI and appendix F-D.

C. Deployment model

Operators self-host `safehub-server` (single binary, filesystem backend today), place `sit` and `shub` on `PATH`, and point remotes at `sit://owner/name`. Plaintext browse runs on member machines via `safehub-local-ui`, never in the untrusted host binary. Identity must come from an external IdP, a KT log, or manual safety-number pins, since host-as-IdP KeyPackage substitution lies outside $\mathcal{F}_{\mathsf{ca}}$ (appendix H). This is a research prototype: the OpenMLS Category 5 ciphersuites remain draft pending IANA registration [9], and there is no production SLA comparable to gittuf [18]. Operational ceilings: table X.

Encryption also removes capabilities from the operator. A host that cannot read a repository cannot scan for malware or leaked credentials, serve hosted search, run CI on its own fleet, deduplicate across repositories, answer a content-based takedown by inspection, or enforce a quota finer than ciphertext bytes. Each relocates rather than vanishing: scanning and CI become organization-trusted forward-only members (table III), compaction becomes admin-authorized consolidation, and content-based legal process reaches the organization holding the keys—the allocation an on-premises

forge already implies. The host retains sequencing, sizes, timing, and compare-and-swap outcomes (section IV-H).

D. Attack scenarios (full table)

TABLE XI: Reconstructed attack scenarios and their outcomes. *Meas.* denotes an outcome observed in the harness, *design* one enforced by the protocol whether or not it was timed, and *analytic* a threat analysis or deployment requirement for which no end-to-end measurement exists.

ID	Scenario	Mechanism	Outcome
S1	Malicious-host tip fork	Checkpoint Compare	Detectable (meas.)
S2	Removed member + server ciphertext	Rotate forward-block	Prevented (meas.)
S3	Forward-only pre-join ciphertext	History window + join shallow	Contained (meas.)
S4	Force-push w/o cosig / fake FF	Verifier FF + ML-DSA-87 cosig	Rejected (meas.)
S5	Concurrent head-log race	CAS + retry ≤ 8	Recovered (meas.)
S6	Post-Remove storage rewrite	Rotate + admin receipt	Lags consolidate (design)
S7	Stale tip / rollback	Hash chain + anchors	Detectable (meas.)
S8	KeyPackage substitution	Pinned digest / KT / IdP	Needs KT (analytic)
S9	MLS Commit partition	Epoch-witness Compare	Detectable (design)
S10	Malicious-member consolidation	Admin-gated CONSOLIDATE, Merkle receipt	Rejected or unverified (analytic)
S11	Join-shallow unverifiable FF	Must-reject unless cosig	Rejected (design)

E. PQ object sizes

TABLE XII: Sizes in bytes of the NIST PQ Category-5 objects, and where each is used in SAFEHUB.

Object	Size	Where it appears
ML-KEM-1024 encaps. key	1,568	ratchet-tree nodes, KeyPackages
ML-KEM-1024 ciphertext	1,568	per copath node per commit
ML-KEM-1024 decaps. key	3,168	member secret state
ML-DSA-87 public key	2,592	credentials, admin roster
ML-DSA-87 signature	4,627	commits, heads, admin co-sigs
KeyPackage (total)	≈15,000	invitations
Commit (n=100, clean path)	≈40,000	membership events
RefHead overhead	≈5,600	every push (sig, tags, wrap)
RO-pad AEAD seal	92	per sealed chunk
SHA-512 digest	64	CAS roots, chain links

F. Secondary evaluation plots

Three size-axis clone figures on deliberately different corpora: fig. 8 (incompressible synthetic; upper-bounds transfer overhead by removing Git delta/`zlib` advantage); fig. 16 (compressible source-shaped trees; exposes storage ratio against `gc`-packed Git); fig. 17 (additive workflow at fixed push count; isolates consolidation costs with no Git analogue). Not interchangeable.

Figure 16 reports the compressible-repository series of `realrepo-scale-latest.json` over a dense working-tree grid (4→64 MiB). Figure 17 overlays the additive size sweep's plain-Git N=8 incremental push and clone baselines; Rotate and tip-rewrite consolidation are plotted as SAFEHUB-only series. WAN validation remains model-only (appendix H-B).

APPENDIX G
STANDARD-AEAD TRANSPORT REFERENCE

SAFEHUB's deployed transport is not a standard AEAD, for proof reasons rather than engineering ones; this appendix separates the cost of the security argument from the cost of encryption.

A. Why the deployed transport is not AES-256-GCM

Lemma 1(a) needs the simulator to reopen an *already-published* ciphertext to the plaintext $\mathcal{F}_{\mathsf{safehub}}$ discloses on adaptive corruption, and AES-256-GCM, being committing on the message, cannot do it. The deployed transport can: an RO pad XORed with $(\mathsf{cblock} \,\|\, \mathsf{pt})$ under an outer HMAC-SHA-512-256 lets one programmed pad point reopen a published ciphertext to any equal-length plaintext with the tag unchanged, while remaining committing on the *key* (theorem 6(ii)). The substitution is load-bearing for theorem 1, and its cost is that the pad is SHA-512/HMAC work, which takes hardware AES off the application transport path. AES-256-GCM is retained inside the MLS ciphersuite, where no equivocation is needed.

B. Measured cost of the choice

Table XIII reports three transports under one methodology (3 warmups, median of 25, release build). Two are measured in-process on the current build—deployed RO-pad and software AES-256-GCM (`aes-gcm` crate; regenerable with `cargo run -p safehub-crypto --release --example aead_backend_compare`). The third is hardware-accelerated AES-256-GCM retained from the pre-migration run (`fullstack-latest.json`, `aead_backend=aws-lc-rs/AES-256-GCM`).

C. Interpretation

On the same core the equivocable construction is free in bulk—0.99× software AES-256-GCM at 1 MiB, 0.92× on open—and costs 1.41× at 1 KiB, where the fixed pad and commit block are not yet amortized. Bulk AEAD is anyway a minority of a `push`, since ML-DSA-87, epoch MAC, bundling, chunk upload, and the CAS round trip are all transport-independent; the migration accordingly did not move the fixed-cost floors reported in table VII. The choice is therefore not a trade-off between speed and security. AES-256-GCM does not yield a weaker form of lemma 1(a); it yields none at all. Preferring the standard primitive would obtain the figures in tables XIII and XIV at the cost of a strictly weaker theorem, with adaptive corruption falling back

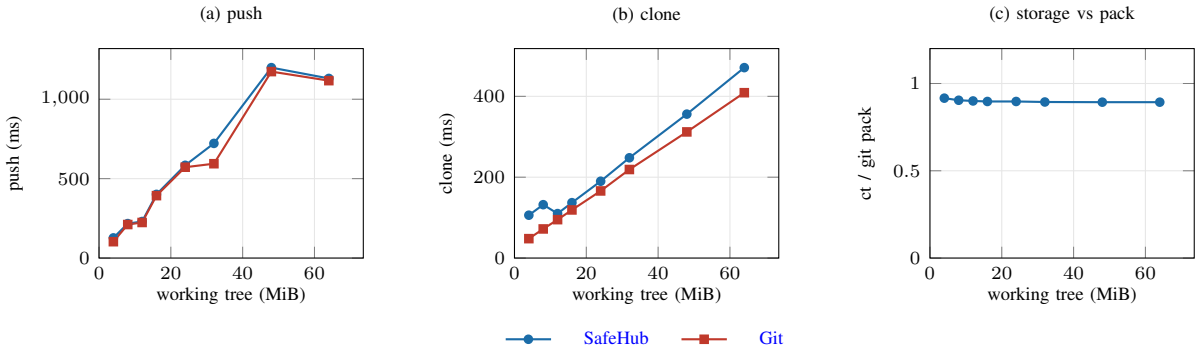

Fig. 16: Push, clone, and stored size on source-code repositories that compress well, against plain Git on the same trees. Panel (c) is server ciphertext against `gc`-packed Git. Client and server share one host.

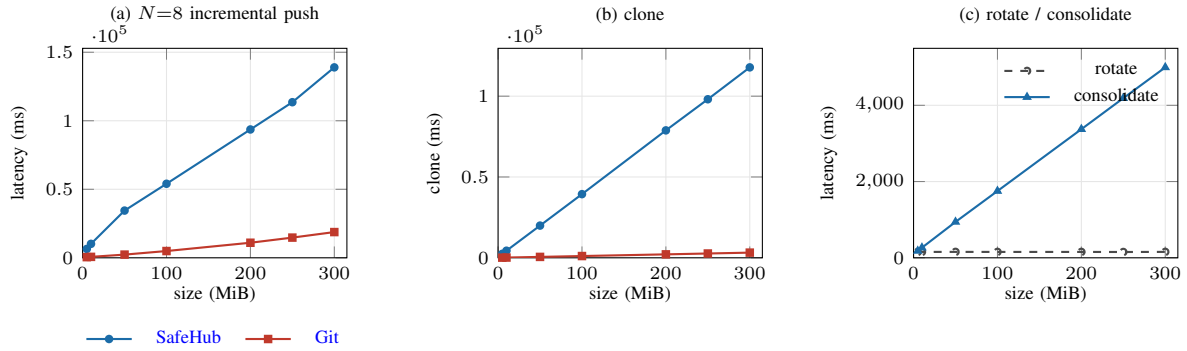

Fig. 17: Push and clone as a repository grows by eight successive pushes, against plain Git on the same trees. Panel (c) has no plain-Git counterpart: it reports MLS epoch rotation, and a consolidation series that measures transport throughput on a synthetic payload and is withdrawn as a compaction measurement (appendix F-A). Client and server share one host.

TABLE XIII: Three reference points for the transport AEAD. The deployed random-oracle pad is compared with software AES-256-GCM in the same process and build, so the ratio between them isolates the construction. The AES-NI column is retained from the pre-migration run on different hardware and therefore measures the hardware path rather than the cost of equivocability. The first two columns are the median of 25 runs after three warm-up runs. The last two rows are SHA-512 and ML-DSA-87 work on RefHead, which lies outside the AEAD and is unaffected by the choice of backend.

Operation	RO-pad (deployed)	AES-256-GCM (software)	AES-256-GCM (AES-NI*)
AEAD seal 1 KiB	11.8 µs	8.3 µs	3.7 µs
AEAD seal 1 MiB	7.92 ms	8.00 ms	1.72 ms
AEAD open 1 MiB	7.31 ms	7.98 ms	1.70 ms
Bulk seal throughput	132 MB/s	131 MB/s	610 MB/s
Ciphertext expansion	92 B	16 B	16 B
Deployed ÷ column, 1 MiB	1.00×	0.99×	4.60×*
Backend-independent RefHead *work*			
RefHead chain hash			1.04 µs
Head verify (100)			0.198 ms

*Different hardware, retained; not a same-machine ratio.

TABLE XIV: Full-stack `sit://` measurements under standard AES-256-GCM, retained from the pre-migration run over repositories of 8, 10, and 12 MiB containing 204–206 files. The measurements in fig. 8 use the deployed transport instead.

	push (ms)		fetch (ms)		clone (ms)	
fixture	sit	git	sit	git	sit	git
8 MiB	331	295	134	55	287	82
10 MiB	372	334	83	51	205	78
12 MiB	436	399	87	54	207	95

to a selective residual (appendix H). Table IX carries RO-pad commit-block margins for the transport row and the AES-256 Grover floor for the MLS-ciphersuite row. Regeneration: `aead-backend-compare-latest.json`.

Appendix H
Limitations and Artifact Scope

What the guarantees exclude, and what the measurements do not establish.

A. System and security limitations

Theorem 1 holds within the following bounds.

1) **Adaptive corruption requires secure erasure.** Without definition 1, only selective adaptivity survives, at a loss of $O(q_{\mathsf{seg}})$ that erodes 2^{128} to roughly 2^{108} at $q_{\mathsf{seg}} \approx 2^{20}$.
2) **The $\mathcal{F}_{\mathsf{acgka}}$ and QROM residual.** Adaptive labeled-export and Welcome security under erasure is assumed for the Category-5 suite, not proved (theorem 9); that suite is draft private-use in OpenMLS pending IANA registration. If it fails, epoch keys are compromised but device ML-DSA-87 signatures still detect forged heads. Losing an administrator key forfeits policy and membership control for that epoch, not the confidentiality of sealed history (appendices E-A and G and table IX).
3) **Availability and metadata.** Withholding and delay are detectable, not preventable. Sizes, timing, MLS framing, and compare-and-swap sequence metadata stay visible; ORAM-grade privacy is out of scope.
4) **Administrator power and path policy.** An authorized administrator holds ideal administrative power; path-level ACLs and gittuf-style path policy are not claimed (section II).
5) **Identity.** A host acting as identity provider can substitute a KeyPackage and obtain plaintext, so deployments need an external identity provider, a key-transparency log, or safety-number pins.
6) **Plaintext endpoints remain.** Member devices hold keys, and an organization-trusted CI, search, or consolidation fleet sees plaintext whenever it is used (section I). Corollary 2 excludes searchable encryption and path-level grants, because a searchable index would hand the host the access patterns and volumes that broke deployed systems [11]–[13]. Indexers are instead forward-only members decrypting locally.
7) **Consolidation authorizes a rewrite.** Receipt construction, signing, window-scoped verification, and the anchor check are implemented and reachable at 0.19 ms per head, but no server path applies the receipt, so no storage is reclaimed and the clone cost of § VI-A0d stands unmitigated. Even once implemented the rewrite compacts honest storage within one epoch and erases nothing a host has retained (appendix F-A). The epoch advance Remove performs is measured at 25–29 ms (section VI-A).
8) **Forks are detected, not prevented.** A client rejects unilaterally any tip that rolls back or contradicts its own anchor, but two clients that have each accepted a divergent branch learn of it only on Compare (lemmas 8 and 9). The host cannot forge a checkpoint, only delay one. Barring the second acceptance would need witness receipts or an external log, which $\mathcal{F}_{\mathsf{safehub}}$ does not assume.
9) **Window endpoints are not cryptographic within a segment.** A DKR token evaluates its derivation at any epoch, so the declared interval $[from, to]$ constrains an honest holder and not an adversarial one: whoever holds a segment root can derive every epoch of that segment. Only the segment boundary is enforced cryptographically. That suffices for the windows the protocol creates, because every event which opens or narrows one begins a new segment (appendix C-C), but a narrower interval imposed inside a segment is policy rather than a guarantee.
10) **Prototype ceilings.** No large-file storage, partial clone, hosted CI, or production service level, and encryption applies to a whole repository or not at all (table X). Device loss, key custody, and legal-hold escrow are operational concerns (section IV-J), not UC events.

B. *Artifact and evaluation limitations*

Four qualifications apply to section VI.

1) **Not every measurement crossed a network.** The six-system comparison and the repository-size sweep put client and server on separate hosts in one availability zone. The depth, delta, concurrency, additive, and compressible-corpus measurements run the server on the client machine, so they report computation rather than network transfer.
2) **Wide-area figures are calculated, not measured.** Where a wide-area round trip appears, the harness waits for the modelled duration; the total is $\lceil n/P \rceil$ round trips of arithmetic, with no wide-area protocol run behind it.
3) **Two kinds of repository.** Most sweeps use synthetic content that does not compress, which is the conservative choice because it denies Git its delta and compression advantage. Real source trees, which do compress, are measured separately and reported apart. The collaboration microbenchmark exercises the protocol path, not user-facing forge features.
4) **The 10^4-head clone.** End to end, with per-repetition samples and interquartile ranges, but on one repository lineage at a fixed 64 KiB delta. Git appears twice because its cost depends on remote packing: packed with `gc` before each checkpoint, and with `receive.autogc` enabled. Being independent runs, their difference also bounds reproducibility (§ VI-A0d).

Tables I, III, VI, XI and XII are the authoritative versions of their data.