ZestSSH

SECURITY WHITEPAPER

Security architecture for an SSH client that can't read your data.

A technical overview of how ZestSSH encrypts, synchronizes, and protects your SSH credentials — so that even a full compromise of our infrastructure leaves your data cryptographically unreadable.

DOCUMENT	ZestSSH Security Whitepaper
PUBLISHED	August 2026
APPLIES TO	ZestSSH v1.7.0 and later
PUBLISHER	Affluent Labs
CONTACT	support@zestssh.com

Contents

01

Executive summary

ZestSSH is a cross-platform SSH client — shipping on Windows, macOS, and Android today, with iOS and Linux planned — built on a single codebase. It lets users connect to remote servers, manage SSH keys, run port forwarding, and synchronize their entire configuration across devices through end-to-end encrypted sync.

This whitepaper documents how ZestSSH handles cryptographic operations: how credentials are encrypted before they leave a user's device, how synchronization occurs without the server ever seeing plaintext, how local data is protected on each platform, and how the Automation Engine safely exposes an execution path to external apps without giving them unrestricted access.

The document is intended for security reviewers, compliance teams, and technically-oriented users who want to verify the claims made on zestssh.com/security. Nothing here is proprietary. Every primitive is a standard, well-audited algorithm. The design is publicly documented because a secret security design is not a secure one.

Core claims

Claim 01 — Zero-knowledge sync

The sync server stores only opaque ciphertext. It never receives your password, your keys, or your plaintext data. Not in aggregate, not under any circumstance. The guarantee holds identically whether you sync to hosted ZestSSH Cloud or to a backend you own — WebDAV, S3-compatible, or Google Drive — because every blob is encrypted on-device before any backend receives it.

Claim 02 — Standard primitives

Every cryptographic primitive is an industry standard: AES-256-GCM and Argon2id for the vault and backups, HKDF-SHA256 for key separation, PBKDF2-HMAC-SHA256 for connection sharing, and ChaCha20-Poly1305 for SSH transport. The one non-standard construction — an authenticated HMAC-SHA256 keystream used for local session transcripts — is disclosed in §9. No proprietary or unreviewed vault cryptography.

Claim 03 — Zero telemetry

No analytics. No crash reports. No usage tracking. There is an **identity plane** (Google/Apple + Firebase Authentication) kept separate from the zero-knowledge data plane, plus a short, disclosed set of user-initiated network calls — purchase/entitlement verification, an optional avatar upload, automation webhooks and Prometheus monitoring you configure, and an anonymous update check. None of it carries vault plaintext. The full egress inventory is in §14.

Summary of guarantees

The following properties are consequences of the architecture, not policy statements. An attacker who fully compromised ZestSSH infrastructure would still face the cost of breaking AES-256-GCM and Argon2id per user.

- Your sync password is never transmitted.
- The server sees only encrypted blobs, wrapped keys, and salts.
- Changing your password does not require re-encrypting data.
- Tampering with server-stored ciphertext is detectable on decryption.
- The local database and OS-native secure storage are both required to read on-device data.
- We cannot recover your data if you lose both your password and your recovery key.
- We cannot assist an attacker who steals your session token in deleting your data.

02

Design principles

ZestSSH's security design follows a handful of rules. They are not novel. They are the basic rules any serious modern E2EE system follows, and they are listed here so reviewers can check the rest of the document against them.

Principle 1 — Encrypt on the device, always

Any data that leaves a user's device, whether to our sync server, to a backup file, or to a cloud provider, is encrypted on-device first, before any network call. There is no mode in which ZestSSH transmits plaintext credentials, connection details, or private keys.

Principle 2 — The server is untrusted

We treat our own server as hostile infrastructure. Its compromise must not produce a cryptographically meaningful leak. The server's role is reduced to three functions: store opaque ciphertext, hand it back on authenticated request, and resolve push/pull conflicts via monotonic versioning. It holds no secret material capable of decrypting anything.

Principle 3 — Use boring cryptography

The vault, sync, and backups use only published, peer-reviewed primitives from maintained libraries — AES-256-GCM, Argon2id, HKDF, PBKDF2 — combined in standard ways, with authentication tags always on. There is one deliberate exception, disclosed rather than hidden: local session transcripts use an authenticated HMAC-SHA256 keystream (encrypt-then-MAC) instead of AES-GCM (§9). It is a standard construction pattern but not a standard cipher, and we call it out so the primitives table stays honest.

Principle 4 — Separate keys by domain

Keys that serve different purposes are derived with domain separation. A verification value used to prove knowledge of a password to the server is derived with HKDF using a different domain string than the key used to encrypt data. Intercepting one does not help an attacker reconstruct the other.

Principle 5 — Defense in depth on the local device

Local data is protected by two independent layers: the ZestSSH database is encrypted at rest with SQLCipher, and the database's master key is itself held in the operating system's native secure storage (Keychain, Keystore, DPAPI, libsecret). Both layers must be defeated for local data to be readable.

Principle 6 — No telemetry means no telemetry

ZestSSH contains zero analytics, crash reporting, or usage tracking. Not opt-in. Not anonymized. This constrains what we can ever know about our users, which is the point.

Principle 7 — State the limits

Where a design choice has a known cost, this document names it rather than omitting it. The compression side channel, the use of plain SQLite in debug builds, and certificate validation rather than SPKI pinning are all disclosed in the sections that follow, not omitted.

03

Threat model

A security system is only as meaningful as the threats it claims to defeat. This section lists the adversaries ZestSSH is designed to withstand, and the ones it is not.

Adversaries in scope

ADVERSARY	CAPABILITY ASSUMED	PROTECTION
Passive network observer	Reads traffic in transit	TLS 1.2+ with certificate validation; payloads are already E2E-encrypted, so TLS protects metadata only
Active network attacker (MITM)	Intercepts or alters SSH connections	Host-key Trust-On-First-Use; a changed host key is surfaced, and rejected outright for automations
Compromised sync server or server-DB leak	Full read of stored blobs and verification values	Blobs are AES-256-GCM encrypted under keys derived on-device; the stored verification value is derived from an Argon2id-hardened master key and grants no data access
Lost or stolen unlocked device	Physical access to a running device	App Lock (PIN everywhere, biometric where supported); SSH credentials and DB key held in OS secure storage
Lost or stolen device at rest	Physical access to storage	OS-native secure storage plus SQLCipher-encrypted local database, key held in secure storage
Malicious app triggering automations	Invokes the automation API	Default-off engine, API-key auth (hashed, constant-time compare, rate-limited), credential isolation, host-key verification
Hostile or compromised sync/backup store	Serves oversized or malicious blobs to exhaust the client (availability)	The `pull` path caps the blob (16 MiB), response (32 MiB, streaming abort), and zlib inflation (64 MiB) and clamps downloaded KDF params; a bring-your-own store is an untrusted boundary (see §11 for the current non-`pull` limitation)

A. Compromised sync server

An attacker obtains full read/write access to the sync server: the database, file storage, and operator credentials. They can view every encrypted blob, every salt, every verification value, and every wrapped DEK.

Outcome: the attacker holds opaque ciphertext. To recover any user's data, they must break AES-256-GCM (infeasible) or brute-force the Argon2id-derived key for a specific user (infeasible at the configured parameters). Per-user effort, with no amortization across users.

B. Network adversary

An attacker sits between the user's device and our servers. They can observe, modify, drop, or replay traffic.

Outcome: the attacker sees TLS-protected traffic. If TLS is defeated, they observe already-encrypted blobs. They cannot decrypt blobs and cannot forge verification values without the master key.

C. Lost or stolen device (locked)

Outcome: the attacker cannot access the app's data without first defeating the OS lock. ZestSSH's own App Lock adds a second gate. The database remains encrypted at rest, and the OS-native secure storage holding its key remains inaccessible while the device is locked.

D. Malicious third-party app on the user's device

Outcome: ZestSSH stores secrets in OS-native secure storage with app-scoped access. On Android, private keys sit behind Keystore plus the app sandbox. On iOS, Keychain entries are scoped to the app bundle identifier. Automations require an API key which itself sits in secure storage and cannot be read by other apps.

E. SSH server with a changed host key

Outcome: ZestSSH uses Trust-On-First-Use. A changed host key produces an explicit warning, not silent acceptance. Automations fail closed on host key mismatch.

Adversaries out of scope

Compromised user device, unlocked, with the vault already unlocked by the legitimate user. No user-space application can provide meaningful protection here. The attacker can read clipboard, screen, and memory of running apps.

Compromised password and compromised recovery key. If both are known to an attacker, the architecture provides no defense. They are, by design, the two valid decryption paths.

Supply-chain attacks on dependencies. ZestSSH relies on underlying libraries: Flutter, zest_ssh_core, SQLCipher, and OS secure storage APIs. Malicious modification to these is outside ZestSSH's direct control, although we pin dependency versions and review updates.

Quantum adversaries. ZestSSH does not currently use post-quantum-resistant primitives. Harvest-now-decrypt-later is a concern the industry is tracking. We will transition to post-quantum algorithms as they mature and are standardized.

04

Cryptographic primitives

Almost every cryptographic operation uses a standard algorithm from this list. The one exception — a custom authenticated keystream for local session transcripts — is called out in the table and in §9.

PRIMITIVE	ALGORITHM	PURPOSE AND PROPERTIES
Symmetric AEAD	AES-256-GCM	Authenticated encryption for all at-rest data: sync blobs, backups, DEK wrapping. 256-bit key, fresh random 96-bit nonce per blob.
Password-based KDF	Argon2id	Memory-hard key derivation. Resists GPU and ASIC brute-force. OWASP-recommended.
Connection-share KDF	PBKDF2-HMAC-SHA256	Key derivation for the connection-sharing / QR `.zest` export only (600,000 iterations; 100,000 legacy). Not memory-hard, so weaker against GPU/ASIC than the 128 MB Argon2id used everywhere else — an accepted tradeoff for small single-connection share files.
App-lock KDF	Argon2id (light profile)	Hashes the App Lock PIN under a smaller profile than sync (about 4 MB / 4 iterations / 2 lanes) — sized for a short PIN checked interactively on-device.
Key derivation	HKDF-SHA256	Domain-separated derivation of the verification value from the master key.
Hash	SHA-256	Verification value computation, API key hashing for storage comparison.
Local DB encryption	SQLCipher 4	AES-256-CBC with per-page HMAC-SHA512 authentication. Default config, HMAC not disabled.
Transcript cipher	HMAC-SHA256 keystream + EtM	Non-AES authenticated construction for local session-transcript files: an HMAC-SHA256 counter-mode keystream XORed with the plaintext, then encrypt-then-MAC under an independent HMAC-SHA256 key. The one custom construction in the app; see §9.
Transport	TLS 1.2+	Certificate validation for all sync server communication. Protects metadata and transport, not data confidentiality.
SSH AEAD	ChaCha20-Poly1305	Recommended cipher for SSH transport. Fast in software, no timing-channel concerns.
SSH key exchange	curve25519-sha256	Modern elliptic-curve Diffie-Hellman for SSH session key establishment.
SSH host and client keys	Ed25519	Modern signature scheme. Short keys, fast verification, no weak-curve concerns.
Secret storage	OS-native	Keychain (iOS/macOS), Android Keystore, Windows DPAPI, libsecret (Linux).

Nonce uniqueness (GCM)

Each blob is encrypted with a fresh random 96-bit nonce. Under a single long-lived data key, nonce uniqueness is birthday-bounded, roughly a 2^-32 collision probability near 2^32 encryptions. A sync performs single-digit encryptions per cycle, so operation stays dozens of orders of magnitude inside the safe envelope.

Compression

Payloads are zlib-compressed before encryption to reduce size. Compression is a size optimization, not a confidentiality measure. It introduces a ciphertext-length side channel (the CRIME/BREACH family). This is accepted because the threat model has no adversary who can both inject chosen content into a user's data and observe the resulting blob sizes.

Why no custom cryptography

Every widely-exploited cryptographic vulnerability of the last two decades has come from one of: implementing a standard primitive incorrectly, combining primitives in an unreviewed way, or rolling a new primitive without peer review. The vault, sync, and backup paths avoid all three by using well-audited library implementations of well-specified standard algorithms. The single deliberate deviation — the session-transcript keystream (§9) — is an encrypt-then-MAC composition of a standard primitive (HMAC-SHA256), not a new primitive, and is disclosed here rather than buried.

05

Key hierarchy and derivation

Sync uses a three-layer key hierarchy with indirection through a randomly generated Data Encryption Key (DEK). This section describes each key, how it is derived, and what it protects.

The three keys

KEY	FULL NAME	ROLE
DEK	Data Encryption Key	Random 256-bit key, generated once on-device during sync setup. Encrypts all actual user data. Never changes over the lifetime of a sync account. Never transmitted in plaintext.
MEK	Master Encryption Key	Derived from the user's sync password via Argon2id. Wraps the DEK. Also used to derive the verification value that authenticates sync requests.
KEK	Key Encryption Key	Derived from the user's recovery key via Argon2id with a distinct salt. Wraps the DEK independently of the password, providing an alternative decryption path if the password is lost.

Argon2id parameters

Both the MEK and KEK are derived with Argon2id using the following parameters. These are conservative defaults that exceed OWASP's minimum recommendations and are forward-compatible via version bumping.

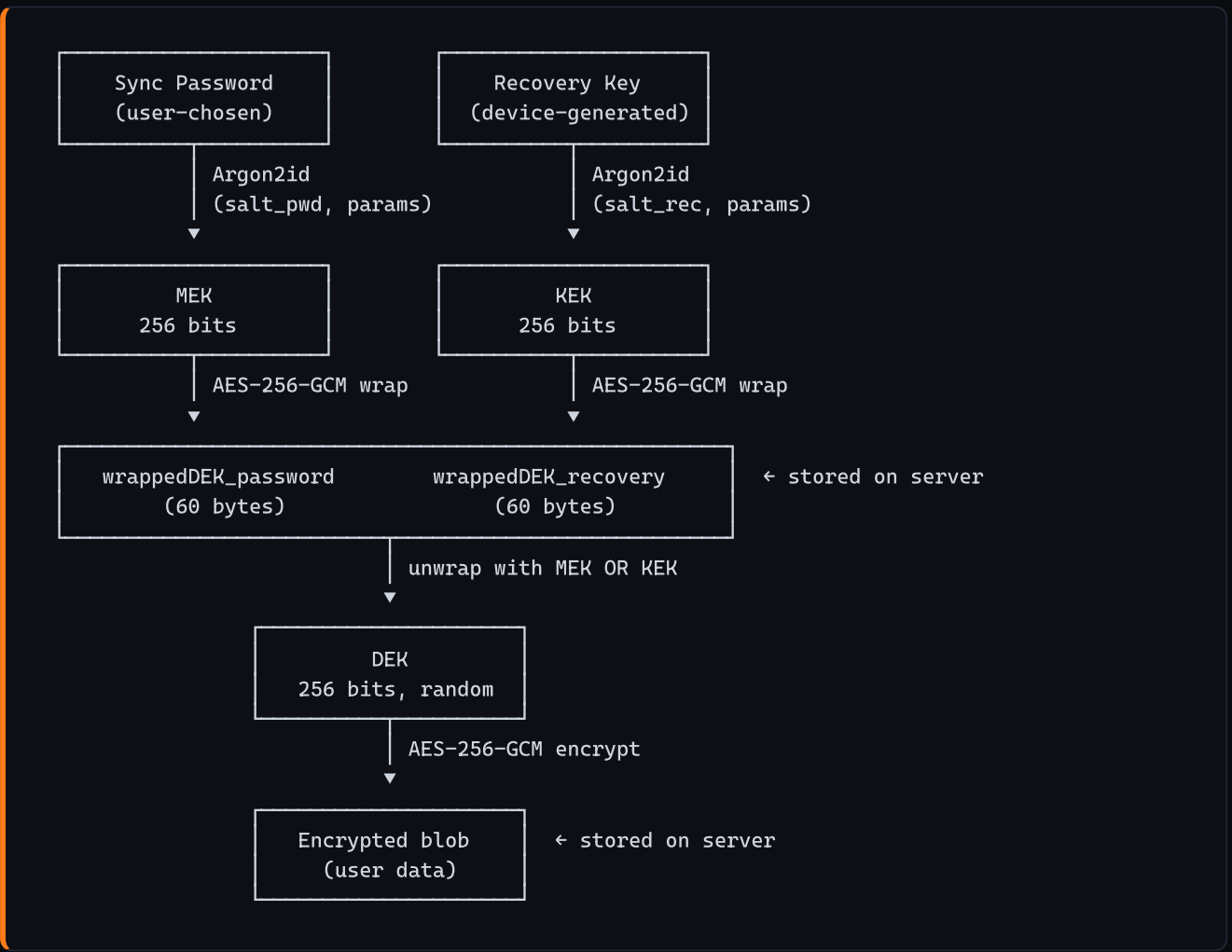
- **Memory cost:** 128 MB
- **Iterations:** 3 passes
- **Parallelism:** 4 lanes
- **Output length:** 32 bytes (256 bits)
- **Salt:** 32 random bytes per user, via `Random.secure()`
- **Parameter-set version:** `argon2Version = 2` — ZestSSH's own tag for this parameter set (v2 is the current 128 MB profile; v1 was an earlier, lower-memory profile), distinct from the Argon2 algorithm version. Each blob records the tag it was created under.

The memory cost forces an attacker to allocate the full working set per brute-force attempt. On a modern GPU this reduces parallelism per card by several orders of magnitude compared to memory-light hash functions. The iteration and parallelism parameters add further linear-time cost.

Parameter versioning

Argon2id parameters are stored alongside the encrypted blob on the server as a version number. If we raise the memory cost in a future release, existing users can still derive their keys using the parameters active when their blob was created. Re-derivation with new parameters happens on the next sync push, invisibly.

The hierarchy



Why indirection through a DEK

Encrypting user data directly with a password-derived key would force a full re-encryption of every byte whenever the user changes their password. For users with large sync profiles this would be slow and risky, since a failure mid-migration leaves data in an inconsistent state.

By wrapping a random DEK with the password-derived MEK, password changes become a single operation: re-derive a new MEK from the new password, re-wrap the same DEK. The actual data blob is untouched. Fast, atomic, low-risk.

The recovery key provides a second, independent wrap of the same DEK. Using the recovery key is not a bypass. It is a fully cryptographic decryption path that never depended on the password in the first place.

Verification value

Account sign-in is handled by the user's identity provider (Google or Apple) plus a backend-issued token. The sync password is never the account credential.

The sync-password verification value is a 32-byte tag derived from the master key, not from the password directly. HKDF-SHA256 is applied to the MEK using a dedicated domain string, and the server stores the result. The client presents it on every push, and the server enforces it on the destructive operations — purging sync data and deleting the account — refusing them unless the presented value matches its stored copy in constant time. On a password or recovery-key change, the client additionally proves knowledge of the *old* secret (an `authorize_hash` / `authorize_recovery_hash`) so the rotation of the stored value is itself authenticated. On its own the value grants no account access and no read access to data.

An attacker who intercepts the verification value cannot derive the MEK. HKDF is one-way, and domain separation ensures that a successful attack on the verification context would not help compromise any other key derived from the same MEK. Recovery-key verification uses a parallel, cryptographically independent domain.

DEK wrapping format

The wrapped DEK, stored server-side in two forms (password-wrapped and recovery-wrapped), is a fixed 60-byte structure:

```
wrappedDEK = nonce(12) + gcm_tag(16) + ciphertext(32)
           = 60 bytes total
```

Wrapping uses AES-256-GCM with the MEK or KEK as the wrapping key and a fresh 12-byte nonce per wrap. Rewrapping on password change generates a new nonce.

06

Cloud sync architecture

Cloud sync is a product-wide feature. This section describes what happens when a user presses sync, from the plaintext object graph in the app's memory to the encrypted blob at rest.

Where the encrypted blob is stored is a separate choice: hosted ZestSSH Cloud (the Juiced tier) or a backend you own — WebDAV, S3-compatible, Google Drive, or a self-hosted ZestSSH server. The zero-knowledge guarantee holds identically in every case, because every blob is encrypted on-device before any backend receives it. Juiced pays for the hosted storage, not for the encryption.

What gets synced

- SSH connection profiles (hostname, port, user, auth preferences)
- SSH identities including private keys
- Stored passwords associated with identities
- Connection groups and tags
- Command snippets and snippet categories
- Port-forwarding rules
- Networks and per-network overrides — the Wi-Fi/subnet/CIDR triggers and the per-network address, port, user, identity, and jump-host rewrites (internal-topology metadata)
- Deletion tombstones (see Conflict resolution)

Application settings (theme, font size, UI preferences) are not synced. They are device-local because they reflect per-device ergonomics rather than shared state.

Host-key trust is not synced. The known-hosts database is serialized into the encrypted blob — so it is uploaded and stored, as ciphertext — but a restore intentionally never applies it. Each device re-learns host keys through its own Trust-On-First-Use, so a malicious or stale blob cannot pre-seed a forged host key. `.zest` backups behave the same way.

The encryption flow

1. **Serialize** — the application's sync object graph is serialized to JSON.
2. **Compress** — the JSON is zlib-compressed. See the compression note in the primitives section for the accepted side-channel tradeoff.
3. **Generate IV** — a cryptographically secure 12-byte random IV is generated via `Random.secure()`. IVs are never reused across encryptions with the same key.
4. **Encrypt** — AES-256-GCM encrypts the compressed plaintext using the DEK. The output includes a 16-byte authentication tag.

5. **Package** — the output is concatenated: `salt(32) + IV(12) + tag(16) + ciphertext`.
6. **Transmit** — the package is sent to the sync server over TLS along with the user's verification value.

```
Your device (plaintext) → serialize → zlib compress → AES-256-GCM encrypt
  → encrypted blob (unreadable) → TLS 1.2+ → server (stores ciphertext, cannot decrypt)
```

The decryption flow

1. **Pull** — the app requests the current blob. The server responds with the package plus metadata (current version, Argon2id parameters, wrapped DEK).
2. **Parse** — the package is split into its four components.
3. **Derive MEK** — the password, salt, and Argon2id parameters are fed into Argon2id to reproduce the MEK.
4. **Unwrap DEK** — the server-stored `wrappedDekPassword` is decrypted using the MEK, yielding the DEK.
5. **Decrypt** — AES-256-GCM decrypts the ciphertext and verifies the authentication tag. A mismatched tag produces `SyncDecryptionException`, which the app handles as wrong password or corrupted blob.
6. **Decompress** — zlib-decompress.
7. **Deserialize** — parse JSON back into the application's object graph.

Conflict resolution

Two mechanisms operate at different layers.

Push lock (server-side). The server tracks a monotonically increasing version counter per user. Every push includes an `expected_version` matching the last version the client pulled. If another device pushed in the interim, the server rejects the push with a conflict and returns the current server version; the client must pull, merge locally, and retry. This is optimistic concurrency control — it serializes writes. It is not the merge.

Merge (client-side). When a client pulls, it applies the incoming blob as an upsert: each incoming record overwrites the matching local record wholesale, and records not present locally are inserted. There is no field-level merge, and — importantly — no `updatedAt` comparison for edit-versus-edit conflicts. Whichever state was most recently pushed to the server wins; the push lock above is what serializes those pushes. `updatedAt` is consulted only for **deletions**: a delete syncs a tombstone carrying its `deletedAt`, and on merge a timestamped record survives a matching tombstone only if its `updatedAt` is newer than the `deletedAt` (timestamp-less configuration tables are delete-wins). Tombstone propagation covers connections, connection groups, identities, snippets, snippet categories, networks, port-forward rules, and network overrides.

This is deliberately simple: a last-writer-to-the-server upsert with tombstoned deletes, avoiding the complexity class of operational-transform or CRDT systems. The tradeoff is explicit — a local edit made

against a stale pull can be overwritten by the next pull, so the client pulls before it pushes.

What the server sees and stores

DATA	STORED?	NOTES
Encrypted blob	Yes	Opaque ciphertext. No structure visible.
Salt (password)	Yes	32 random bytes. Needed to re-derive MEK.
Salt (recovery)	Yes	32 random bytes. Needed to re-derive KEK.
Argon2id parameters	Yes	Memory, iterations, parallelism, version.
Wrapped DEK (password)	Yes	Useless without MEK.
Wrapped DEK (recovery)	Yes	Useless without KEK.
Verification value	Yes	Auth only. HKDF-derived, not the MEK.
Version counter	Yes	Monotonic. For conflict resolution.
Operational metadata (hosted backend)	Yes	Account email/ID, a non-identifying device id, blob size, and timestamps — never the contents of your data. A user-owned backend (WebDAV, S3, Drive) stores only the encrypted blob.
User password	Never	Not in any form.
Recovery key	Never	Generated on device, shown once.
MEK / KEK / DEK	Never	All live on-device only.
Plaintext sync data	Never	Encrypted before transmission.

07

Encrypted backups (.zest)

ZestSSH's manual backup format, `.zest`, uses the same cryptographic primitives as sync. Users can export their configuration to a local file, store it wherever they like including third-party cloud services, and restore it on another device.

Because backups are encrypted with the user's password and not a ZestSSH-held key, they can be stored in third-party cloud drives without exposing data to those providers.

File format

OFFSET	SIZE	CONTENT
0	4 bytes	Magic bytes: `ZEST` (ASCII)
4	1 byte	Version byte — current format is `3`; the importer also accepts legacy `1` and `2`
5	variable	Encrypted payload

The version-3 payload prepends a small header recording the Argon2 parameters the backup was written under, so it still decrypts after the app's default parameters change:

```
payload = header_length(2) + argon2_params_header(JSON) + salt(32) + IV(12) + gcm_tag(16) + ciphertext
```

The encrypted body is the same AES-256-GCM structure as a sync blob. Legacy `1` files omit the parameter header (the app derives with the current default and falls back to the older parameters); legacy `2` files are unencrypted JSON and are refused unless the user explicitly confirms an unprotected import.

A second, unrelated `.zest` variant exists: **connection sharing / QR export** produces a per-connection JSON file (no `ZEST` magic bytes) encrypted with AES-256-GCM under **PBKDF2-HMAC-SHA256**, not Argon2id. It is a different format for a different purpose — sharing a single connection — and is documented under Cryptographic primitives. The two `.zest` formats are not interchangeable.

There is no plaintext export option. Every backup file is encrypted. A user-accessible plaintext dump of SSH credentials and private keys would be a catastrophic security failure by design.

Password requirements

Backup passwords must be at least 12 characters, enforced in the export flow before any encryption occurs. The floor reflects the reality that backup files land in user-accessible storage where an offline attacker who obtains the file has unlimited time to attack.

We recommend a high-entropy passphrase. Short numeric PINs, dictionary words, and reused passwords are all vulnerable to offline brute force regardless of Argon2id's cost parameters, because password entropy is the limiting factor.

What a backup contains

DATA CATEGORY	INCLUDED
SSH connection profiles	Yes
SSH identities and private key material (encrypted)	Yes
Stored passwords	Yes
Known-hosts database	In the file, but not applied on restore (host keys re-learned via TOFU)
Connection groups and tags	Yes
Snippets	Yes
Port-forwarding rules	Yes
Session transcripts	No (too large, device-local)
Automation audit logs	No (device-local)
App settings and theme preferences	No (device-ergonomic)

Import behavior

Imports use the same restore path as sync: incoming records overwrite matching local records, new records are inserted, and the imported blob's **tombstones are applied** — so an import can also *remove* local records whose deletion it carries. Timestamped records are protected (a record survives an imported tombstone when the local copy is newer than the tombstone's `deletedAt`), but timestamp-less configuration records are delete-wins. Importing an older backup can therefore roll back or delete newer local data, not only add to it.

Identity secrets (private keys, stored passwords) are restored to OS-native secure storage only after the database transaction commits successfully. This prevents orphaned secrets in secure storage if the import fails mid-operation.

Import safety limits

Backup files larger than 50 MB are rejected before any content is loaded into memory. Malicious or corrupted `.zest` files claiming enormous sizes cannot crash the app with an out-of-memory error. For context, a typical ZestSSH configuration backup is under 100 KB even with hundreds of connections.

Auto-backups

ZestSSH creates automatic backups when the app backgrounds after data modification. They are stored on-device — in app-private storage on desktop, and on Android in the public `Download` folder (still encrypted, so only ciphertext is exposed there) — and are never uploaded or transmitted by ZestSSH. Retention is capped at three files, with the oldest deleted as new ones are created.

Auto-backups use the same AES-256-GCM-under-Argon2id encryption as manual backups, under a **separate auto-backup password the user sets**, held in OS secure storage — it is never derived from the sync password. If no auto-backup password has been set, auto-backup is silently skipped. Manual backups use a password you choose per export, can be saved anywhere, and are never auto-deleted.

08

SSH protocol support

This section documents the SSH protocol primitives ZestSSH offers on the wire when connecting to remote servers. These are separate from, and used alongside, the sync encryption described earlier. The lists below are ZestSSH's secure-by-default profile — the algorithms actually advertised during negotiation, in preference order. Weaker legacy algorithms remain implemented for interoperability but are excluded from the defaults and offered only when a user opts into the compatibility profile.

Key exchange algorithms

Offered in preference order.

ALGORITHM	NOTES
`curve25519-sha256@libssh.org`	Modern elliptic-curve Diffie-Hellman. Recommended default.
`ecdh-sha2-nistp521`	NIST P-521 curve.
`ecdh-sha2-nistp384`	NIST P-384 curve.
`ecdh-sha2-nistp256`	NIST P-256 curve.
`diffie-hellman-group-exchange-sha256`	DH group exchange with SHA-256.
`diffie-hellman-group14-sha256`	2048-bit DH group with SHA-256.

SHA-1 key exchanges (`diffie-hellman-group14-sha1`, `diffie-hellman-group-exchange-sha1`) and the 1024-bit Group 1 (`diffie-hellman-group1-sha1`) are implemented but removed from the defaults, because SHA-1 is collision-broken and the strong defaults must not be negotiable down to it. They are opt-in only, via the compatibility profile.

Ciphers (transport encryption)

Offered in preference order.

CIPHER	NOTES
`chacha20-poly1305@openssh.com`	AEAD. Recommended default. Fast in software, no timing-channel concerns.
`aes256-ctr`	AES-256 in CTR mode, paired with an Encrypt-then-MAC algorithm.
`aes128-ctr`	AES-128 in CTR mode.

CBC-mode ciphers (`aes256-cbc`, `aes192-cbc`, `aes128-cbc`) are implemented but excluded from the defaults, because of the CVE-2008-5161 plaintext-recovery weakness when CBC is not paired with an ETM MAC. They are opt-in only.

MAC algorithms

MACs authenticate non-AEAD ciphers (CTR mode). AEAD ciphers carry their own integrity tag and do not use a separate MAC. Offered in preference order.

ALGORITHM	NOTES
`hmac-sha2-512-etm@openssh.com`	Encrypt-then-MAC with SHA-512. Recommended.
`hmac-sha2-256-etm@openssh.com`	Encrypt-then-MAC with SHA-256.
`hmac-sha2-512`	Standard HMAC-SHA-512.
`hmac-sha2-256`	Standard HMAC-SHA-256.
`hmac-sha1`	Legacy, for older-server compatibility.
`hmac-sha2-512-96`	Truncated 96-bit tag. Weaker; offered last.
`hmac-sha2-256-96`	Truncated 96-bit tag. Weaker; offered last.

`hmac-md5` is implemented but removed from the defaults (MD5 is deprecated) and is opt-in only.

Host key types

Offered in preference order.

TYPE	NOTES
`ssh-ed25519`	Modern, fast, secure. Recommended.
`rsa-sha2-512`	RSA with SHA-512 signature.
`rsa-sha2-256`	RSA with SHA-256 signature.
`ecdsa-sha2-nistp521`	NIST P-521.
`ecdsa-sha2-nistp384`	NIST P-384.
`ecdsa-sha2-nistp256`	NIST P-256.

`ssh-rsa` (RSA with legacy SHA-1 signatures) is implemented but removed from the defaults; a downgrade to SHA-1 host-key signatures is opt-in only via the compatibility profile. `ssh-ed448` is

implemented but not offered by default.

ZestSSH verifies server host keys against a Trust-On-First-Use known-hosts database. On first connection the host key is recorded; subsequent connections verify the stored fingerprint against the presented key. A mismatch produces an explicit warning and requires user confirmation before proceeding, and automations fail closed on a mismatch.

Handshake integrity

ZestSSH advertises and enforces the strict key-exchange extension (`kex-strict-c-v00@openssh.com`) that mitigates the Terrapin prefix-truncation attack (CVE-2023-48795): when the peer supports it, message sequence numbers reset after the first key exchange and pre-handshake packet injection is rejected, closing the downgrade Terrapin relies on.

Jump hosts and proxy chains

A connection can route through a chain of jump hosts (bastions). Each hop is a full, independent SSH session: every host in the chain is host-key-verified against the same Trust-On-First-Use database, each hop uses its own credentials, and a broken or cyclic chain fails closed rather than falling through to a direct connection.

Client key types

Keys that ZestSSH can generate on-device or import from external formats.

TYPE	CAN GENERATE	CAN IMPORT
Ed25519	Yes	OpenSSH PEM
ECDSA (P-256/384/521)	Yes	OpenSSH PEM
RSA 4096	Yes	OpenSSH PEM, PuTTY `.ppk`
RSA 2048	Yes	OpenSSH PEM, PuTTY `.ppk`
FIDO2 / hardware-token keys (`sk-*`)	No — created on the authenticator	Attached via ssh-agent (`ssh-keygen -t ed25519-sk`)

SSH certificates

ZestSSH can act as a small certificate authority for OpenSSH user certificates. `SshCertSigner` issues `*-cert-v01@openssh.com` certificates — signing a public key with a CA key under a chosen set of principals, a validity window, a serial, and critical-options/extensions, with a fresh 32-byte `Random.secure()` nonce per certificate. The **CA private key never lives in the plaintext database:** it is held in OS-native secure storage (a `ca_key_{caId}` entry) and is loaded and decrypted only at signing time. Issued certificate

blobs are likewise kept out of the plaintext DB (`identity_cert_{identityId}`). A "certificate" authentication method presents the cert during public-key auth, and a certificate identity fails closed if its cert or CA material is missing. Because this is a credential-issuance and CA-key-custody surface, it is called out explicitly rather than folded into key generation.

Telnet and Mosh

ZestSSH also supports Telnet and Mosh connections, and the SSH guarantees in this section **do not apply to them.** Telnet is plaintext — no encryption, no host-key verification — so credentials and session data travel in the clear; use it only on a trusted local network or inside an already-encrypted tunnel. Mosh runs its own UDP protocol and uses SSH only to bootstrap the session. Both are offered for compatibility; SSH is the secure default.

09

Local data protection

On-device data protection uses two independent encryption layers. Both must be defeated for an attacker with file-system access to read any credential, key, or connection detail.

Layer 1 — OS-native secure storage

SSH private keys, stored passwords, sync keys, and API keys for the Automation Engine all live in the operating system's secure storage facility.

PLATFORM	BACKEND
iOS, iPadOS	Keychain; items accessible only after first device unlock
macOS	Keychain
Android	Android Keystore-backed encrypted storage (EncryptedSharedPreferences)
Windows	DPAPI (Data Protection API, user-scoped)
Linux	libsecret via Secret Service API, typically GNOME Keyring or KWallet

App-scoping is strong on **iOS and Android**, where the OS binds these entries to ZestSSH's app identity so another app cannot read them. On **desktop it is weaker, and worth stating plainly:** Windows DPAPI is scoped to your Windows *user account* with no additional app entropy, so any process running as the same user could call `CryptUnprotectData` on these entries; Linux libsecret is readable by any application in the unlocked login session. On desktop the meaningful boundary is your OS user account plus the Layer-2 database encryption, not per-app isolation. Unlock still requires the device's user authentication.

Layer 2 — Local database (SQLCipher)

The local database, which contains connection profiles, known hosts, snippets, port forwarding rules, and metadata about keys, is encrypted at rest using SQLCipher. SQLCipher provides transparent AES-256-CBC encryption with per-page HMAC-SHA512 authentication, so tampering with the database file is detectable.

SQLCipher's master key is held in OS-native secure storage (layer 1). It is never written to disk in plaintext, never printed to logs, and never transmitted. Opening the database requires layer 1 to be unlocked first.

On launch, the app verifies that SQLCipher and not plain SQLite is active, and refuses to open the database unencrypted. This is a fail-closed check.

An attacker with only raw file-system access sees an encrypted SQLCipher file and no way to open it. An attacker who extracts the SQLCipher key but not the file has a key to nothing. Both layers must be

compromised for local data to be exposed.

Known limitation: debug builds use plain SQLite. Only release builds are encrypted.

App Lock

Above the OS and database layers, ZestSSH offers App Lock. When enabled, a passcode is required before ZestSSH will open, and after the app has been backgrounded for a configurable interval. This defends against the scenario where the device is unlocked but the user walks away from it.

App Lock uses a PIN on every platform, plus biometric unlock (Face ID, Touch ID, fingerprint) on iOS, Android, and macOS where the device supports it. The PIN and lock settings live in the OS secure store, never in plaintext, and the lock is enforced entirely on-device.

App Lock does not replace OS-level protection, it supplements it. A device that is not locked at the OS level still has its data at rest encrypted, because SQLCipher and secure storage are always on.

Session transcripts

Session transcripts are a third at-rest data class that sits outside the two-layer model above, and it is worth being explicit about them. They are stored as standalone files (outside the SQLCipher database), and — as their own on-screen warning states — a transcript can contain anything printed at the terminal, including passwords, API keys, or tokens typed at a prompt.

Release-build transcript files are encrypted at rest, but with the non-AES construction noted in §4: an HMAC-SHA256 keystream (encrypt-then-MAC, independent HKDF subkeys) under a 256-bit key held in OS secure storage. A transcript file is therefore unreadable without the device's key, but it is protected by that single custom-cipher layer rather than the database's SQLCipher-plus-secure-storage pair. Debug builds additionally write plaintext `.txt` transcripts, which are never shipped in a release.

10

Automation security

ZestSSH's Automation Engine lets external apps (Tasker, MacroDroid, iOS Shortcuts, Siri, or any app capable of URL invocation) trigger pre-configured SSH commands on saved connections. Because this feature exposes a remote-execution surface, it is built with multiple layers of defense.

Layers of defense

1. Default OFF. The Automation Engine must be explicitly enabled in Settings. Enabling it shows a security warning describing what automation can and cannot do. No automation runs while the engine is disabled, and disabling it while a trigger is pending cancels the trigger.

2. API key authentication. Every automation request requires a valid API key. Keys are managed GitHub-style: each key has a human-readable label, creation date, last-used timestamp, and optional expiration. Keys can be revoked individually without affecting other keys.

3. Encrypted API key storage. API keys are stored in OS-native secure storage alongside SSH credentials. They are never written to the app database in plaintext and never logged.

4. Constant-time comparison. API key validation uses constant-time equality over the SHA-256 hashes of the submitted key and stored key. This defeats timing side channels that could otherwise let an attacker learn key prefixes by observing response time differences.

5. Rate limiting. Failed API key attempts trigger exponential lockout. Lockout cascades quickly render repeated guesses infeasible.

6. Credential isolation. The triggering app never gains direct access to SSH credentials. An external app can invoke a pre-configured automation, but it cannot read, export, or exfiltrate the SSH key or password that automation uses. Only the output of the SSH command is returned.

7. Host key verification. Automations use the same Trust-On-First-Use known-hosts verification as manual connections. A changed host key rejects the run. There is no flag that bypasses verification for automations.

8. No shell-string injection. Substituted values are escaped and commands are sent over the SSH exec channel, never concatenated into a shell string, so automation inputs cannot escape into unintended commands. This is not a claim that a legitimately configured command is sandboxed; the remote shell still executes the intended command.

Audit trail

Every automation run is logged locally: timestamp, API key label used, command, connection, exit code, and output. This local audit log is stored on-device only and viewable in Settings → Audit Logs, for your

own forensic review after a suspected credential compromise. The log itself is never uploaded — it is separate from the optional result webhook described next.

What automation cannot do

- Execute arbitrary shell commands not pre-configured in the automation library.
- Bypass host key verification on unknown or changed servers.
- Access credentials for servers not in the user's saved connections.
- Run while the Automation Engine is disabled.
- Be enabled silently. Every activation requires explicit user action in Settings.

Outbound webhook callbacks

Automation results stay on the device **unless you configure a result webhook.** If you do, each run POSTs a JSON body — connection name, command, and the full `stdout` / `stderr`, exit code, and timing — to the HTTPS URL you set (single runs and batch runs alike). This deserves clear statement: command output can contain secrets a command prints (for example, `cat ~/.ssh/id_rsa`), and if you configure a webhook, that output leaves the device to your endpoint.

The channel is constrained: HTTPS-only, redirects are not followed, and an SSRF guard refuses literal loopback, RFC-1918 private, link-local, and cloud-metadata (`169.254.169.254`) destinations. **One honest limitation:** the guard matches the literal host and does not resolve DNS, so a public hostname whose A record points at an internal address is not blocked at request time — disabling redirect-following closes the 302-to-internal vector, not the initial-DNS-to-internal one.

Deep links cannot nominate their own exfil

When automation is invoked by a `zestssh://` deep link, every connect / execute / snippet / batch action requires an explicit in-app confirmation, the URI is capped at 2048 bytes, and — importantly — a deep link **cannot supply its own callback webhook**: the result-webhook field is stripped from deep-link invocations. A drive-by link therefore cannot both run a command and choose where its output is sent; only a webhook you configured in Settings is ever used.

11

Transport and network security

Transport security protects data in motion between the user's device and ZestSSH's sync server. It is distinct from, and layered underneath, the end-to-end encryption already described.

TLS 1.2+ with certificate validation

All sync API calls use TLS 1.2 or later with certificate validation.

Known limitation: this release performs certificate validation, not SPKI pinning.

Why transport security matters less than you would think

An intuitive question: if data is already encrypted with AES-256-GCM using a key the server never sees, why bother with TLS at all?

Two reasons. First, TLS protects metadata: request patterns, access frequency, approximate payload sizes. Unencrypted HTTP would reveal these to any network observer. Second, TLS protects the verification value used to authenticate sync requests. An attacker who captures it cannot decrypt data, but could potentially replay sync requests. TLS prevents that at the transport layer.

The important property is that compromising TLS is not catastrophic. An attacker who defeats TLS on a specific connection observes already-encrypted blobs. Data confidentiality does not depend on transport confidentiality.

Metadata minimization

The sync protocol is designed to reveal as little metadata as possible. Blob sizes are compressed, hiding exact plaintext sizes, and all sync users' blobs appear identically structured. The server cannot distinguish between a user with 5 connections and a user with 500 based on blob structure alone, though aggregate size is necessarily visible.

Download limits and the hostile-store boundary

A user-owned sync backend (WebDAV, S3, Drive) or a self-hosted server is a distinct trust boundary, and the app treats what it downloads as untrusted. The **pull** path is hardened against a hostile store: the blob is capped at 16 MiB, the overall response at 32 MiB with a streaming abort, and zlib inflation is bounded at 64 MiB, so a malicious or corrupt store cannot exhaust client memory. Downloaded Argon2 parameters are clamped to a safe floor and ceiling before use.

Honest limitation: only `pull` currently runs through this hardened helper. The other authenticated calls — `getStatus`, `push`, `verifyPassword`, `purge`, `deleteAccount`, and entitlement checks — use the

plain HTTP client, which applies no size cap and follows 3xx redirects that carry the `Authorization` bearer. Against the hosted ZestSSH backend this is not exploitable, but a compromised or custom server could use it for a token-replay redirect or a large-response memory exhaustion. Routing every authenticated request through the capped, no-redirect helper is the fix; until then the uniform "the server is hostile infrastructure" posture holds fully only for `pull`.

12

Account recovery

ZestSSH's recovery model is deliberately simple and deliberately unforgiving. It consists of a single mechanism: the recovery key, generated at sync setup and shown exactly once.

Recovery key format

During sync setup, ZestSSH generates a recovery key in the following format:

```
ZEST-XXXX-XXXX-XXXX-XXXX-XXXX
```

The five blocks are 20 random characters drawn from a 31-symbol alphabet — A–Z and 2–9, with 0, O, 1, I, and L removed to avoid visual ambiguity. That is $20 \times \log_2(31) \approx 99$ bits of entropy, far beyond brute-force reach even for an attacker who obtains the recovery-wrapped DEK.

The recovery key is generated on-device using `Random.secure()`. It is displayed to the user exactly once, at sync setup. It is never transmitted to ZestSSH's servers. The salt used to derive the KEK is stored server-side, since the client needs it to re-derive the KEK on recovery, but the recovery key itself is not.

How recovery works

1. The user has forgotten their sync password. They sign in on a fresh device and select "I forgot my password."
2. ZestSSH fetches the user's `salt_recovery` and Argon2id parameters from the server.
3. The user enters their recovery key.
4. Argon2id derives the KEK from recovery key plus salt.
5. The server-stored `wrappedDEK_recovery` is decrypted using the KEK, yielding the DEK.
6. The DEK decrypts the sync blob. Data access is restored.
7. The user is prompted to set a new sync password. A new MEK is derived from it, and `wrappedDEK_password` is rewritten with the new wrap.

Recovery does not re-encrypt data. Only the wrapping of the DEK under the MEK changes. The blob itself is untouched.

What happens if both are lost

If a user loses both their sync password and their recovery key, their data is permanently unrecoverable. There is no backdoor, no master key, no customer-support override. Our infrastructure holds only

ciphertext and salts. Without a user's password or their recovery key, we have no material capable of decrypting anything.

This is not a gap in the design, it is the design. The same property that prevents an attacker from decrypting your data after a server breach also prevents us from decrypting it at your request.

The user-facing implication: save the recovery key somewhere you can access but an attacker cannot. A password manager, a printed copy in a physical safe, or a trusted encrypted note.

Account deletion protection

Destructive operations, including deleting sync data and deleting the account, require proof that the requester holds the sync password, not merely a valid session token. Specifically, the verification value derived from the current session's master key must match the server's stored value.

This defeats a specific attack: an adversary who steals a user's session token cannot maliciously wipe the user's data to cause harm or extort them. They would still need the sync password, which is never in transit and is held only in memory on the user's unlocked device.

13

Hardened release builds

Every public release is compiled with Dart code obfuscation (`--obfuscate`, with debug symbols split out and kept private), stripping human-readable symbol names to raise the cost of reverse-engineering and tampering. Debug symbols are never shipped inside the app.

Symbols are split out with `--split-debug-info` and retained privately in the source repository, one folder per released version. This keeps a symbolication path for crash triage without exposing symbol names in the shipped binary.

14

Privacy and telemetry

ZestSSH contains no analytics, no crash reporting, no usage telemetry, no opt-in tracking, and no anonymized metrics. The absence of telemetry is not a privacy feature, it is the baseline.

What we do not collect

- How often you open the app.
- What features you use.
- Which servers you connect to.
- What commands you run — with the sole exception of voice dictation, which sends your spoken input to the platform speech recognizer while you actively dictate (see below).
- How long your sessions last.
- Your IP address, beyond transient TLS termination.
- Your device model, OS version, or locale.
- Crash stack traces.
- Performance metrics.
- A/B test bucket assignments.

Third-party services in the app

ZestSSH separates two planes. The **data plane** is zero-knowledge — your vault is encrypted on-device and no third party, us included, can read it. The **identity plane** exists only if you sign in for sync, and is handled by Google/Apple. Beyond the encrypted vault blobs, here is the complete list of what leaves the device, each only when you use the feature that needs it:

PATH	WHEN	DATA SENT
Purchase verification	On purchase / entitlement check	Mobile: RevenueCat receives an app-user ID and the store receipt token. Desktop: an entitlement request to `api.zestssh.com` under your Firebase token. Web: Stripe processes payment.
Sync sign-in	Only if you enable sync	Google/Apple + Firebase Authentication receive your email and an account ID to identify your sync account (authentication only — no analytics). Your ID token refreshes against Google roughly hourly while signed in.
Profile avatar	Only if you set a custom avatar	The image plus account metadata is uploaded to `api.zestssh.com` under your Firebase token.

Automation result webhook	Only if you configure one	The connection name, command, and full stdout/stderr of an automation run are POSTed to your chosen HTTPS URL (see Automation security).
Server monitoring (Prometheus)	Only if you configure an endpoint	Metrics requests — with the basic-auth or bearer credentials you entered — to the monitoring URL you chose.
Update check	Every non-iOS launch	A GET for a version file on `zestssh.com`. Nothing about you or your device is sent; the server sees only your IP and the time, as with any web request.
Voice-to-text (optional)	Only while you dictate	Spoken input goes to the platform speech recognizer, which on some Android devices is a cloud service. Off unless you tap the microphone.
Your connections	When you connect	SSH/Telnet/Mosh to the hosts you configure, and the BYO sync/backup stores you configure — user-directed, not telemetry.

The RevenueCat app-user ID is an anonymous UUID **before** you sign in; after sign-in it is set to your account ID so entitlements follow the account. It carries no usage data either way.

No analytics or crash-reporting SDK is embedded: no Firebase Analytics, no Crashlytics, no Sentry, no Mixpanel, no Amplitude, no Google Analytics, no Meta Pixel. Firebase is used for **Authentication only** — it never receives vault data.

Why this matters

An SSH client has access to your servers. The bar for trusting one should be high. We think the appropriate bar is this: you install ZestSSH, and we have no more information about you or your infrastructure than we did before you installed it.

15

What ZestSSH cannot do

This list exists to make our constraints explicit. These are properties of the system, not choices we can reverse with a policy change.

We cannot read your encrypted sync data

Our server holds an AES-256-GCM ciphertext. The key is derived from your password, which we never see. There is no copy of the key, plaintext, or derivation anywhere in our infrastructure.

We cannot decrypt your local backups

Backups are encrypted with a password you choose. We do not receive the password or the backup file. Even if we did receive a backup, we would hold opaque ciphertext.

We cannot recover your password or your recovery key

Neither secret is stored on our servers in any form. The password is not even in transit. The recovery key is generated on-device and shown once.

We cannot trigger automations on your behalf

API keys live in your device's secure storage. Our servers do not hold them, cannot derive them, and have no authenticated path into your Automation Engine.

We cannot selectively modify your sync data

AES-256-GCM's authentication tag detects any modification. A partial or targeted edit by us would produce a blob that fails decryption on your device. You would see it immediately, not silently accept tampered data.

We cannot delete your data without proof you hold the password

Destructive operations require the verification value, which requires the master key, which requires the password. A stolen session token is insufficient.

These are not promises. They are mathematical consequences of how the system is built. A policy change by Affluent Labs cannot grant us capabilities we have deliberately architected away from. If that ever changes, it will require a new major version with a different security model and explicit user consent, not a silent update.

16

Responsible disclosure

ZestSSH is built by one person. Security reports are taken seriously and responded to personally, not routed through a ticketing system or a bug bounty platform.

How to report

Email support@zestssh.com. Include:

- A description of the vulnerability
- Reproduction steps, ideally with a proof-of-concept
- Affected versions, platforms, and any relevant environment details
- Your preferred contact method for follow-up

If the issue is particularly sensitive, request a PGP key and we will exchange one for encrypted communication.

What happens next

1. **Acknowledgment** within 72 hours of receipt.
2. **Triage** within 7 days: assessment of severity and exploitability.
3. **Fix development** proportional to severity. Critical issues take priority over any other work.
4. **Coordinated disclosure.** We agree on a disclosure date before a fix ships, typically 14 to 90 days depending on severity and user exposure.
5. **Credit** in release notes and in this whitepaper's acknowledgments, if you want it.

What is in scope

- The ZestSSH mobile and desktop applications
- The sync server and its API
- The zestssh.com and docs.zestssh.com websites
- The Automation API
- Cryptographic design issues in this document

What is out of scope

- Third-party dependencies. Report upstream, and notify us so we can pin around the fix.
- Social engineering attacks against the developer.

- Physical attacks against the developer's hardware.
- Issues requiring a compromised user device as a precondition.
- Self-XSS and other issues requiring a user to paste attacker-supplied input into privileged contexts.

Safe harbor

Good-faith security research targeting ZestSSH is welcome. We will not pursue legal action against researchers who:

- Report findings promptly and give us a reasonable time to remediate before disclosure
- Avoid accessing, modifying, or destroying data that is not theirs
- Do not attack users, production infrastructure at scale, or adjacent services
- Operate within applicable law

Rewards

ZestSSH is a one-person project and cannot currently offer cash rewards. Validated security findings receive:

- A free Squeezed license or Juiced upgrade, your choice
- Public acknowledgment in the release notes, if you want it
- Direct communication with the developer during remediation

As the project grows, a formal bug bounty program is on the roadmap.

17

Appendix: algorithm references

For reviewers who want to verify the specific standards ZestSSH implements, the following references point to the original specifications and implementation guides.

ALGORITHM	REFERENCE
AES-256-GCM	NIST SP 800-38D, Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode
Argon2id	RFC 9106, Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications
HKDF-SHA256	RFC 5869, HMAC-based Extract-and-Expand Key Derivation Function
SHA-256	FIPS 180-4, Secure Hash Standard
ChaCha20-Poly1305	RFC 8439, ChaCha20 and Poly1305 for IETF Protocols
Ed25519	RFC 8032, Edwards-Curve Digital Signature Algorithm (EdDSA)
curve25519	RFC 7748, Elliptic Curves for Security
TLS 1.2	RFC 5246
TLS 1.3	RFC 8446
SSH-2	RFC 4251 to 4254, Secure Shell Protocol Architecture and Sub-Protocols
SQLCipher	Zetetic LLC, SQLCipher Design Specification (sqlcipher.net/design)

Library implementations

- `cryptography` **(Dart)** — AES-256-GCM, Argon2id, HKDF, SHA-256 — pure-Dart implementations; ZestSSH does not link OpenSSL/BoringSSL or libsodium for these
- `pointycastle` **(Dart)** — PBKDF2 and AES for the connection-share and app-lock paths
- `crypto` **(Dart)** — HMAC-SHA256 for the transcript keystream and hashing
- **SQLCipher** (`sqlcipher_flutter_libs`) — database-level encryption (the one native crypto dependency)
- **zest_ssh_core** — Affluent Labs' proprietary SSH library, forked from the MIT-licensed dartssh2 with the original license preserved (SSH protocol implementation)
- **flutter_secure_storage** — platform-native secure storage access

Document version history

VERSION	DATE	CHANGES
1.0	April 2026	Initial publication. Covers ZestSSH 1.4.x.
2.0	August 2026	Rebuilt from the source pipeline against ZestSSH 1.6.2. Corrects the Argon2 memory cost to 128 MB, states certificate validation rather than pinning, and documents tombstone-based deletion sync.
2.1	August 2026	Revised against a full source audit at ZestSSH 1.7.0. Corrected the third-party/egress disclosure (Firebase Authentication, avatar upload, automation webhook, Prometheus, voice-to-text, update check); added the session-transcript cipher and storage, PBKDF2 connection sharing, SSH certificate issuance, corrected sync-merge semantics, and known-hosts-not-restored; and documented Terrapin mitigation, jump-host chains, Telnet/Mosh, and client download limits.