Considus

TECHNICAL WHITE PAPER

Catchlight for iPhone

The Catch*light* *encryption architecture*

How Catchlight seals a note, where the keys live, and what the design deliberately does not protect you against. Published before launch so it can be checked rather than trusted.

Applies to Catchlight 1.0

Introduction

I wrote a line on Reddit a while back that I keep coming back to. "AES-256" on its own doesn't tell you whether the key came from your passcode or whether it's just sitting next to the data in the same database, and the people who actually pay for privacy will ask. It's a fair thing to ask of me too, so this is the answer, in full, with nothing left in the drawer.

Every app in this category says it's encrypted. Most of them are telling the truth and it still doesn't settle anything, because encrypted is a brochure word. A brochure tells you the house has a new roof. A survey tells you where the damp is. What follows is the survey, and I'd rather you read it now, before the app ships, than take my word for it afterwards.

Most of the weight sits on a handful of things.

Twelve words are the only root secret in the whole system, and everything else is derived from them. Each note gets its own key, so the master key never touches your writing directly. Your encrypted data goes into a cloud folder that you pick and you control, because I run no server for it. And if you lose every device and the words, it's gone, and I mean gone, including from me.

The damp is in here too, and it has a section to itself. There's no live pairing between two phones at launch, so a phone and an iPad on day one is a v1.1 thing and section 6 says so before it says anything flattering. Whoever hosts your folder can still count your files and see when you last wrote, even though they can't read a word inside them. And an unlocked phone in the wrong hand is an unlocked phone, whatever the storage layer is doing underneath. None of that is comfortable to put in a document I'm hoping will impress you, which is more or less why it's there.

Catchlight hasn't launched yet. This describes the architecture as built and merged, not as hoped for, and I'm publishing it before launch on purpose. If something in here is wrong, I'd much rather hear it now.

What this covers, and what it doesn't

This is about the encryption. Key generation, key derivation, where keys live, how notes are sealed, how the device protects them at rest, how two devices agree without a server in the middle, how recovery works, and exactly what I hold. Sections 7 and 8 give the reasoning behind each choice and the published guidance the design was checked against, because a specification without its reasoning is just a list.

It's written for two readers who won't read it the same way. If you build this sort of thing, sections 2 to 8 are the specification and you can skip the rest without missing anything load-bearing. If you don't, sections 9 to 12 are the ones that answer the question you actually came with, which is whether I can read your notes. I've tried not to make either of you wade through the other's half, though the middle is unavoidably technical and I'd take that over vague.

Everything here describes code that's written and merged rather than planned, and where the app is behind the cryptography I've said so in the section itself rather than in a footnote at the back. The version on the front cover is the version this describes. When the app changes this changes with it, and the version history at the end says what moved and when.

Some things aren't here. Catchlight is a single-user app with no sharing, so there's no collaboration model to describe. There's no server-side security section because there's no server holding your data. Transport security between your phone and your cloud provider is that provider's TLS and I neither control it nor audit it. App Store signing, the internals of iCloud Drive or Dropbox, all out of scope.

Section 10 is the one where I list what this design doesn't protect you against. Read it with this one. Either half on its own gives you a false picture, and the second half is the one that's usually missing.

1 The principles the design has to satisfy

These are constraints rather than preferences. A design that breaks one doesn't ship, which has already killed a couple of ideas that would otherwise have been convenient.

PRINCIPLE	WHAT IT MEANS IN PRACTICE
Zero knowledge	I hold no user data, no credential, no email address and no key. An order served on Considus produces nothing about you, because there's nothing to produce
Storage you control	Your encrypted files go to a cloud folder you choose. Catchlight is cloud-agnostic and runs no sync infrastructure of its own
Invisible encryption	It's always on. No toggle, no setup step, no settings screen implying it could work another way. Infrastructure, not a feature
Verifiable	The design has to be documentable and checkable. This paper exists because of this principle, not despite it
The phrase is yours	Zero knowledge has a price and you pay it. You hold the only copy of the root secret and I can't help you if it goes
Apple's frameworks where Apple has one	CryptoKit for the cryptography, Keychain and the Secure Enclave for key storage, NSFileCoordinator for file access. Fewer third-party parts, less to go wrong

2 The key hierarchy

One phrase produces every key in the system. No other root secret exists. Figure 1 has the whole chain on one page, and the rest of this section is that picture in words.

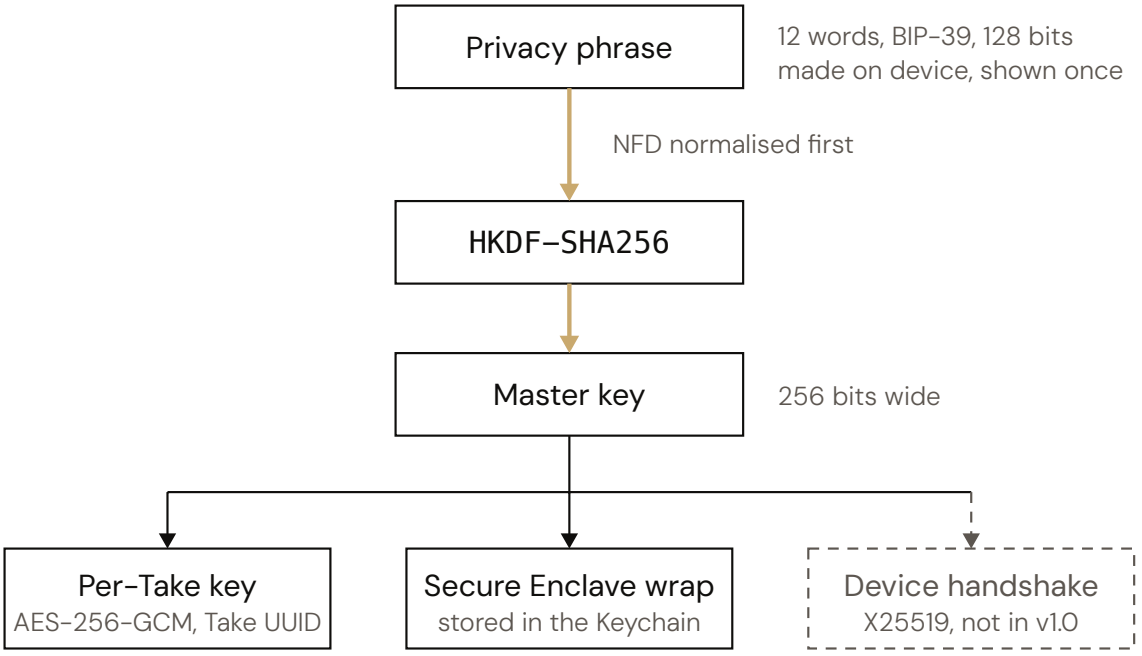

Figure 1. The key hierarchy. The phrase is normalised to Unicode NFD so the same words typed on any device produce identical input. One key per Take, derived with the Take's UUID as the context value, so a compromised key costs one note rather than all of them. The master key is 256 bits wide and carries the phrase's 128 bits of entropy. Dashed: built but not reachable in v1.0.

2.1 The Privacy phrase

At first launch the app generates a random 12-word phrase from the BIP-39 word list. Twelve words carry 128 bits of entropy. The phrase is displayed once. Considus never receives it, and no copy exists anywhere except the one the user makes.

2.2 The master key

The app normalises the phrase to Unicode NFD form and passes it to HKDF-SHA256 through CryptoKit, producing a 256-bit master key. No separate key pair is used.

The derivation is deterministic. The same phrase always produces the same master key, which is what makes recovery possible with no server.

The Secure Enclave wraps the master key. The wrapped key is stored in the iOS Keychain under two attributes.

`kSecAttrAccessibleWhenUnlockedThisDeviceOnly` makes the key unavailable while the device is locked, excludes it from device backups, and prevents it being restored onto another device.

An access-control flag requiring user presence before the app will reveal the Privacy phrase. Face ID, Touch ID or the device passcode must succeed first, every time, including inside an already-unlocked app.

Failed PIN attempts are counted in the Keychain rather than in memory, so the counter survives an app restart and a device restart.

2.3 Per-item keys

The master key does not encrypt user data. For each note, the app derives a dedicated key with HKDF-SHA256, using the master key as input keying material and the note's UUID as the context value.

Two properties follow. A compromised per-item key exposes one note rather than the whole store. And decrypting one note requires no other note's key material to be resident in memory.

2.4 Ciphers

USE	CONSTRUCTION
Note content	AES-256-GCM, fresh random nonce per operation
Device-to-device handshake	X25519 key agreement, ChaCha20-Poly1305 wrap

Both are authenticated constructions. A note whose ciphertext has been altered fails to open rather than decrypting to something wrong.

A sealed note carries three parts. The nonce, the ciphertext and the authentication tag. Neither the nonce nor the tag is secret, and neither needs to be. The nonce is what stops identical content producing identical ciphertext. The tag is what makes any later change to the stored bytes detectable.

Opening a note is therefore a check as much as a decryption. The tag is verified before any plaintext is returned, so a note whose stored bytes have been altered, truncated, or swapped for a different note's, fails closed and is reported as unreadable. There is no partial result and no fallback to an unauthenticated read. A fallback would remove the property the tag exists to provide.

The handshake pair does a different job. X25519 lets two devices agree a shared secret across a channel anyone can read, and ChaCha20-Poly1305 uses a key derived from that secret to wrap the master key for the crossing. Poly1305 is the authentication half, so the wrapped key fails closed in the same way a note does. Section 6 has the sequence, and the reason the interface that drives it is not in version 1.0.

3 Storage on the device

Two independent layers protect data at rest.

LAYER	MECHANISM	COVERAGE
File	SQLite3 database under `NSFileProtectionCompleteUntilFirstUserAuthentication`	The whole database file including metadata, keyed to the device passcode and the Secure Enclave
Column	AES-256-GCM sealed payload columns	Note content, under its own per-item key, independent of the layer above

The database and its `-wal`, `-shm` and `-journal` sidecar files are all marked as excluded from device backup, and the sidecars carry the same file-protection class as the database itself. Figure 2 shows how the two layers nest, which is the part a table cannot show.

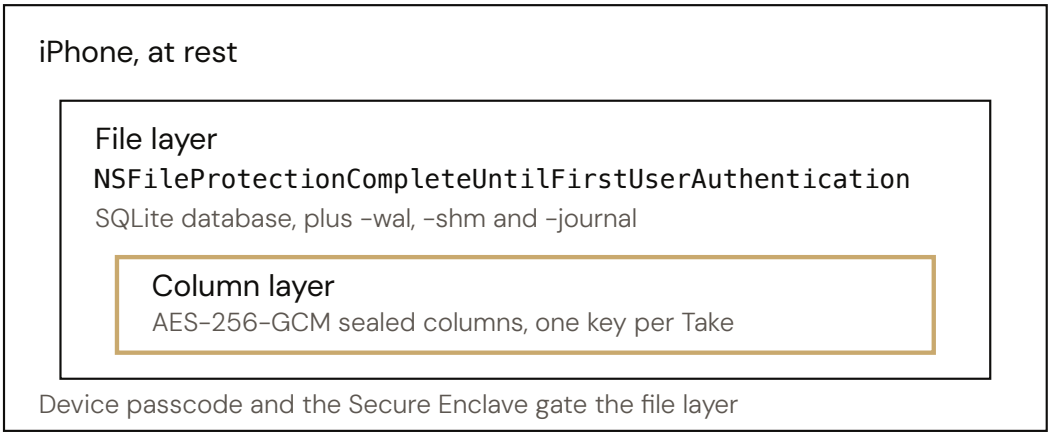

Figure 2. Two layers, independently sufficient. The database and all three sidecars are excluded from the device backup, so restoring an iPhone restores no Takes. Either layer alone would be a defensible design. The file layer ties the whole database to the device passcode and the Secure Enclave, and the column layer seals each Take's content under its own key, so the content stays sealed even to something that has already opened the file. NSFileProtection behaviour was measured on physical hardware on 6th June 2026. The iOS Simulator does not model file protection faithfully, so a passing simulator test demonstrates nothing here.

4 Storage in the cloud

Catchlight writes encrypted files to a folder the user selects. Version 1.0 supports iCloud Drive and Dropbox. The app holds a security-scoped bookmark to that folder locally, and holds no provider credential and no OAuth token.

Encryption happens on the device before any file is written. The provider receives ciphertext only.

The folder index is encrypted as well. An earlier design left a manifest in the clear, which allowed anyone reading the folder to count the notes held, observe when one last changed, and watch deletions occur. That manifest is now sealed inside an encrypted envelope. Section 10 covers what a provider can still observe, which is not nothing.

Sync coordination. Concurrent writes are serialised behind a lock file, `catchlight.lock`, in the cloud folder, with file-level access coordinated through NSFileCoordinator. A device finding the lock held waits and retries. Sync runs on foreground entry and on a scheduled background task.

Deletion. Deletion is tombstone-only. A deleted note receives a deletion timestamp and is excluded from display. Tombstones are pruned after 180 days, which is the only retention period in the product. It was 30 until July 2026, raised because a device offline past the shorter window would have re-uploaded its own deleted notes on reconnecting. An absent file is never interpreted as a deletion. A sync timing window, a provider outage or a partially downloaded folder all make a file appear absent when it is not, and deletion-by-absence would convert any of those into permanent data loss.

> NOTE
>
> A provider that retains file versions retains versions of ciphertext. Old versions remain encrypted under the same key hierarchy.

5 First device setup

STEP	ACTION
1	The user sets a local PIN. The failure counter is written to the Keychain
2	The user chooses whether to sync at all. Everything can stay on the device. If a cloud folder is chosen for backup and sync, the app stores a security-scoped bookmark to it and nothing else
3	The app generates a random 12-word phrase and normalises it to NFD
4	HKDF derives the 256-bit master key. The Secure Enclave wraps it, the Keychain stores it
5	The phrase is displayed and the user confirms they have recorded it
6	Setup complete. Content is encrypted before it is written. What crosses that boundary is set out under the table, and none of it is the body of a Take

The whole sequence takes under a minute. There is no account to create, no email address to give and no server to reach, so nothing in it phones home. Steps 3 and 4 happen entirely on the device. The phrase in step 5 is shown once and is never transmitted anywhere.

What the app sends across that boundary. A Markdown export, which only ever happens because you ask for it. Up to 100 characters of a reminder's text, which iOS may show on your lock screen. The address of a web link you share into the app, fetched once to build its preview. And if you set a location reminder, the place you type goes to Apple to be matched against real ones, while a set of coordinates goes to Apple to be turned into a street name. A place you typed is still something you wrote, so it belongs here rather than in a footnote about maps.

The app also talks to the App Store about your subscription and asks Apple for map tiles. Neither carries a word of what you wrote, so neither crosses this boundary.

Another path is not the app's at all, and opening it is your call. The editor is an ordinary iOS text view, so anything iOS offers wherever you type works inside it, and that includes Writing Tools. Select part of a Take, ask for a proofread or a

rewrite, and that text goes where Apple sends it, which for some requests is their servers rather than your phone. Catchlight has no AI, asks for none and adds none. There is a switch in Settings, under Security, with three positions, off, panel and inline, and **off is the default**, so these tools reach into a Take only if you tell them they can. A third-party keyboard with full access can read the field you are typing in and pass it on, and that is a separate mechanism which is deliberately not blocked, because taking away a keyboard you chose to install is not mine to do. The choice stays yours either way, and what I owe you is saying plainly that these tools reach in here at all.

Step 5 is also the only step that can go wrong in a way nothing later can repair. The phrase is the root of the whole hierarchy, so a phrase nobody wrote down is a hierarchy with no root the moment the device is gone. The app asks the user to confirm they have recorded it, and that confirmation is a prompt rather than a guarantee. I would rather say that plainly than let the confirmation imply the app can check. Section 6.1 covers what recovery does, and section 10 covers what it cannot do.

6 A second device, and the part that isn't wired up yet

Start here, because this is the one place where the cryptography is ahead of the app. **Version 1.0 syncs a single device to your folder.** The handshake below is built and tested, and the interface that would let you drive it is not, so live pairing between two phones ships in v1.1 rather than at launch.

I'll say why, because the reason is unglamorous. I don't own a second iPhone, and the concurrent-edit validation genuinely cannot be done without one. Shipping a pairing flow I've never watched two real devices perform would be the exact thing this paper is supposed to be an argument against.

What follows is therefore how the handshake works, not a feature you can use on day one. Figure 3 has the sequence, drawn dashed throughout for the same reason.

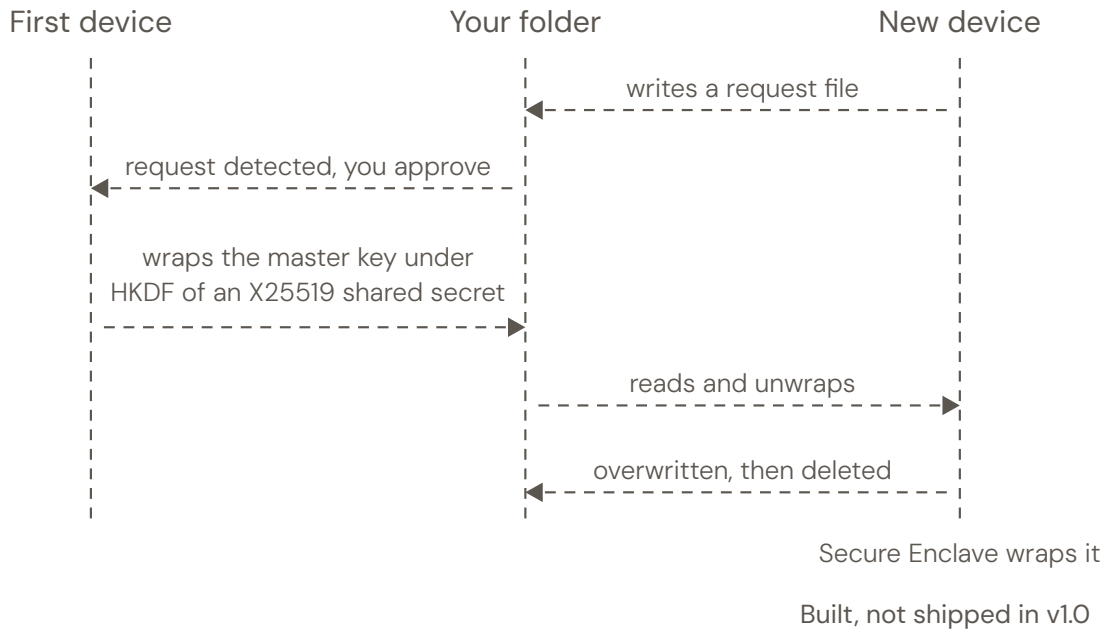

Figure 3. The second-device handshake, drawn provisionally. A 15-minute expiry, the ephemeral X25519 binding and a secure-delete overwrite together answer a provider that retains old file versions. Everything here is dashed because the cryptography is built and tested while the interface that would drive it is not. Live pairing ships in v1.1. No Considus infrastructure appears in this diagram because none takes part.

STEP	WHAT THE HANDSHAKE DOES
1	The new device is pointed at the folder and writes a request file
2	The first device detects the request and prompts for approval
3	The first device generates a random one-time value and an ephemeral X25519 key pair, then wraps the master key under a key derived from the X25519 shared secret combined with that one-time value. The ephemeral private key never leaves that device's memory. The envelope carries a 15-minute expiry
4	The new device reads the wrapped key and the one-time value, derives the same key, and unwraps the master key
5	Both devices overwrite the transfer files and delete them
6	The Secure Enclave on the new device wraps the master key, and its Keychain stores it

The threat here is a cloud provider retaining an old version of a transfer file, so three controls apply rather than one. The 15-minute expiry bounds the window. The ephemeral X25519 binding means a recovered one-time value alone is insufficient. And the files are overwritten before deletion rather than unlinked.

6.1 Recovery when every device is gone, which does ship

You install the app somewhere new, point it at your folder, type your 12 words. HKDF derives the identical master key and your notes decrypt.

No server is involved. There's no account to reset and nobody to ask, and nothing about this flow needs Considus to be reachable, trading, or even still in business. If I disappeared tomorrow your notes wouldn't notice.

WARNING, IRREVERSIBLE

Do not lose your devices and your Privacy phrase together. If both go, the data is unrecoverable. There's no support process behind this, no override, no key escrow, nothing. I'm not saying I won't help, I'm saying I can't.

7 Why these choices and not others

A specification tells you what was built. This section tells you why, because every decision below had a plausible alternative and a reviewer is entitled to know why it lost.

7.1 Why HKDF, and not Argon2id or PBKDF2

This is the question a cryptographer asks first, and asked cold it looks like a mistake, so here is the full answer.

Argon2id, scrypt and PBKDF2 are password-based key derivation functions. They are deliberately slow and, in Argon2's case, deliberately memory-hungry. That cost exists for one reason, which is to make guessing expensive. They protect a **low-entropy secret chosen by a human**, where an attacker can plausibly work through the candidate space.

The Privacy phrase is not chosen by a human. It is generated by the device from a cryptographically secure random source, and it carries 128 bits of entropy. There is no candidate space to work through. Adding a work factor to a 128-bit random secret slows down the legitimate user on every launch and costs an attacker nothing they were ever going to attempt.

HKDF is the correct tool for this input. It is designed for exactly this case, extracting and then expanding **existing high-entropy keying material** into one or many keys, which is precisely what the key hierarchy in section 2 needs.

There is a second reason and it is about the future rather than the maths. HKDF-SHA256 is available natively in WebCrypto and in Tink. If Catchlight ever ships a web or Android client, that client re-derives byte-identical keys with no third-party cryptographic library and no reimplementation of a memory-hard function in JavaScript. Argon2id would have required shipping and trusting a library on every platform, forever.

One honest caveat, because someone will notice it and I would rather they read it here. The master key is 256 bits wide and carries 128 bits of entropy, because a derivation function cannot manufacture entropy its input did not have. That is deliberate, and 128 bits remains far beyond any feasible search.

7.2 Why 12 words and not 24

Twenty-four words carry 256 bits of entropy instead of 128. Neither is reachable by any attack that will exist in your lifetime, so the extra 128 bits buy nothing real.

What they cost is transcription. This phrase is the single thing standing between a user and permanent loss of their writing, it gets copied onto paper by hand, and every additional word is another chance to write one down wrong. Doubling the length doubles that risk to defend against a search nobody can run. The specialist review confirmed 12 as sufficient.

7.3 Why AES-256-GCM for note content

GCM is an authenticated mode. It provides confidentiality and integrity in one construction, so a note whose ciphertext has been altered fails to open rather than decrypting into plausible nonsense. Using an unauthenticated mode such as CBC would have meant bolting on a separate MAC and getting the composition right, which is a well-known way to introduce a subtle flaw.

AES-GCM is also hardware-accelerated on Apple silicon and is the mode CryptoKit is built around, so the fast path and the well-tested path are the same path.

The nonce is randomly generated per operation. Nonce reuse under the same key is the one catastrophic failure mode of GCM, and per-item keys narrow the exposure further, since no two notes share a key in the first place.

7.4 Why NSFileProtection and not SQLCipher

SQLCipher was evaluated and rejected.

It is a third-party dependency doing a job the platform already does. NSFileProtection ties file access to the device passcode and the Secure Enclave, which is a hardware-backed guarantee that a userspace library cannot offer. SQLCipher would also have introduced a second key-management problem, with its own key needing derivation, storage and rotation alongside the hierarchy in section 2.

The two are not equivalent and the choice is not only about dependencies. Application-layer AES-256-GCM on the payload columns covers the note content, and NSFileProtection covers the database file and its metadata. Both are in place, which is why section 3 has two rows.

7.5 Why Apple's frameworks throughout

This was the specialist reviewer's overarching recommendation and it became a design principle.

Every third-party cryptographic library is code that has to be audited, updated and trusted, and it is code that Apple's own hardware acceleration and platform hardening do not cover. CryptoKit, Keychain Services and NSFileCoordinator are maintained by the platform vendor, receive security updates through the OS, and have a vastly larger installed base than any library Catchlight could pick.

The trade is real and worth stating. It ties the implementation to Apple's platforms, and it means a future Android client is a reimplementation rather than a port. Choosing standard, widely implemented primitives, as section 7.1 describes, is what keeps that reimplementation possible.

7.6 Deletion as tombstones

Covered mechanically in section 4. The reasoning belongs here. In a folder-synced system with no authoritative server, "the file is missing" and "the file has not arrived yet" are indistinguishable. Any design that treats absence as intent will eventually delete someone's writing during a network partition. Tombstones cost storage. The alternative costs data.

8 What this design was checked against

I am not claiming certification, an audit or formal verification, and section 11 is explicit about what the review actually was. What this section lists is the published material the design was built and checked against, so you can go and read the same documents.

8.1 Primitives, as specified

PRIMITIVE	SPECIFICATION
HKDF	RFC 5869, HMAC-based Extract-and-Expand Key Derivation Function
AES	FIPS 197
AES-GCM	NIST SP 800-38D
X25519	RFC 7748
ChaCha20-Poly1305	RFC 8439
Mnemonic phrase	BIP-39
SHA-256	FIPS 180-4

Every one is a published, widely implemented standard with independent implementations available for cross-checking. Nothing in Catchlight is a bespoke cryptographic construction, and that is a deliberate decision rather than a lack of ambition. Novel cryptography in a notes app is a warning sign, not a feature.

8.2 Mobile-specific guidance

The **OWASP Mobile Application Security Verification Standard** is the relevant framework for an app of this kind, and two of its categories bear directly on this design. **MASVS-CRYPTO** covers correct use of cryptography, key management, and the absence of hardcoded keys. **MASVS-STORAGE** covers secure on-device storage and, importantly for section 3, whether sensitive data leaks into logs, caches or backups. The associated **Mobile Application Security Testing Guide** supplies the test procedures behind those categories.

Three consequences of reading MASVS-STORAGE seriously are visible in the app. The database and its sidecars are excluded from device backup. The diagnostics log is content-free by construction, carrying timestamps, categories and event messages but never note text and never key material. And there is no analytics or third-party crash reporting for anything to leak into either.

To be precise about the claim. The design was built against this guidance and I have reviewed it against those categories myself. No independent MASVS verification has been carried out, at any level, and this document does not assert one.

8.3 Apple's own guidance

The **Apple Platform Security** guide is the authoritative reference for everything in sections 2.2 and 3, and it is worth reading if you want to understand what the Secure Enclave and the Keychain actually guarantee. Keychain items are themselves encrypted under AES-256-GCM, with the secret value requiring a round trip through the Secure Enclave, and access-control lists evaluated inside the Enclave rather than by the app. The choice of `kSecAttrAccessibleWhenUnlockedThisDeviceOnly` and the user-presence flag in section 2.2 both come straight from that model.

Apple's **CryptoKit** and **Keychain Services** documentation covers the API-level choices. Where Apple documents a preferred approach, Catchlight uses it, which is the sixth design principle stated as a rule.

8.4 Where the guidance runs out

Two decisions in this document are not settled by any of the above, and they are mine.

Using HKDF rather than a password-based KDF is correct for a high-entropy generated secret, and most general-purpose guidance assumes a human-chosen password and recommends Argon2id accordingly. Section 7.1 is the reasoning, and it is the paragraph to attack if you think I have this wrong.

Choosing no account recovery at all is a product decision that no security standard makes for you. Standards will tell you how to protect a recovery mechanism. They will not tell you whether to have one. Section 10 states what that costs.

9 What I actually hold

The whole list.

	DO I HOLD IT?
Your name or email address	**No.** The pre-launch mailing list is opt-in and isn't linked to the app
Your password, or a hash of one	**No.** There's no password and no account
Your Privacy phrase, or any key	**No**
Where your cloud folder is	**No**
Your encrypted data	**No.** It's in your folder, not mine
A purchase record	**Yes**, through the App Store. Apple's standard transaction data, and I see what any developer sees

Not so much as an email address. That isn't a promise about my future behaviour, which would be worth very little, it's a description of what the architecture makes possible. I can't be leaned on for a key I never had.

10 What this doesn't protect you against

Here's what the survey found.

Your unlocked phone in someone else's hand. If an attacker is holding your unlocked device, they see what you see. Encryption protects data at rest and in transit, not a screen that's already open. A per-note lock is planned for a later release and narrows this, and it is not in v1.0, so don't count on it.

A compromised operating system. All of this is built on iOS security primitives. Beat those and you've beaten me too. I claim no defence against a hostile OS update, a jailbroken device or an exploit in iOS itself, and anyone who does claim that is overselling.

Note identifiers aren't encrypted. Each note's UUID is the context value for deriving its key, so it exists in the clear. It's a random identifier carrying no content, but it isn't hidden and I'm not going to pretend otherwise.

Your folder still leaks some metadata. The index is encrypted, so the folder no longer tells anyone how many notes you hold or what you deleted. What it cannot hide is the shape of the files themselves. Anyone who can read the folder sees the file count, the size of each file and the provider's own timestamps on them. They can't read a word of any of it. They can tell you wrote something on Tuesday night, and for some people that alone matters.

Transport is your provider's problem. The connection between your phone and your cloud provider uses that provider's TLS. I assume it and I don't verify it. Your content is already encrypted before it gets near that channel, so a failure there exposes ciphertext rather than notes, but I'd rather say it than let you assume I'd checked.

One memory path isn't wiped. The Secure Enclave decryption path manages its own memory and the app doesn't attempt to zero the plaintext buffer afterwards. It's a known limitation, it's documented in the code, and it isn't an oversight I've just discovered while writing this.

Loss really is permanent. It's in section 6 and it belongs here too. Zero knowledge and account recovery can't both be true. I picked zero knowledge and this is the cost.

One device at launch. Section 6 says it and it belongs in this list too. v1.0 is single-device. If you want your notes on a phone and an iPad on day one, Catchlight doesn't do that yet.

No collaboration, but not quite no sharing. There are no shared notes in Catchlight, no permissions and no second author. A collaboration feature needs a server and an entirely different trust model, so it isn't something I've deferred, it's something I've refused.

What does exist is subtler and worth stating. Your Privacy phrase is the whole account, so anyone you give it to can enter it on their own phone, pull the folder, and hold everything you have written. That is second-device restore working exactly as designed, and it does not care whether the second device is yours. Two people sharing one phrase is a shared notebook in every practical sense, with no private corner for either of them and no way to take it back afterwards.

Version 1.0 was not built for that, and the concurrent-edit path is the v1.1 gap section 6 describes. What is built and shipping is conflict resolution: when two devices change the same note, the app surfaces both versions and asks, rather than silently keeping one. **Treat your Privacy phrase as the account itself, because that is what it is.**

11 About the review

An independent cryptography specialist went through this design with me on 5th June 2026.

He looked at the phrase length, the choice of key derivation function, the cipher, the per-note key hierarchy, the device handshake, the three controls on file versioning, the Keychain attributes, the sync serialisation and the local database approach. He confirmed each decision, told me to stay on Apple's own frameworks throughout rather than reaching for third-party crypto, and every recommendation he made went in.

Now the part that matters more. It was a conversation, not a report. There's no written sign-off and no signature anywhere. He reviewed the design and not the code. And it happened before the implementation was finished, so the remediation pass on 10th June that added the Secure Enclave wrapping, the user-presence check on the phrase, the persistent PIN lockout, the sealed payload columns, the tombstone deletion model and the fail-closed push handlers, all of that came after he'd looked and he hasn't seen any of it.

So Catchlight is not independently audited and this document doesn't claim it is. The offer of a proper code review before launch is still open, and if it happens this paper will say so and carry the result, including anything in it that doesn't flatter me.

I'm spelling this out because the alternative is to write "reviewed by a cryptography specialist" and let you fill in the blanks. Plenty of products do exactly that. If you found out later that the review was a chat over a design document, you'd stop believing the other ten sections too, and you'd be right to.

12 Don't take my word for any of it

Four checks, none of which need any expertise.

Try to reset your password. There isn't one, so there's nothing to reset and nobody to email. An app that can send you a recovery link can also reach your writing, every time, without exception.

Open your cloud folder in the Files app or on a computer and look inside. Files, yes. Open one. You won't find a readable word in it.

Turn on airplane mode and use the app properly for a day. It all works, because none of it was ever going through me.

Read the code. Catchlight is open source under Apache 2.0, and the key hierarchy, the cipher choices and the storage layer are all in there for anyone who wants to check sections 2 to 7 against what actually runs.

If any of those comes out differently on your phone, then either I've made a mistake in the app or I've written something in here that isn't true, and I'd want to know which. There's a form at catchlight.app/support that doesn't ask who you are. Being corrected in public is a much better outcome for me than being quoted approvingly for something that turns out to be wrong.

What the four checks don't do is prove the app is secure, and I'd rather draw that line myself than have someone else draw it for me. Each one proves something narrow. No password reset means there's no route to your notes through me. Unreadable files mean the sealing is really happening. Airplane mode means the app isn't quietly leaning on something of mine. The code tells you what the design is. None of them tells you whether the implementation has a bug in it. That takes the proper code review section 11 says hasn't happened yet, and until it has, this paper is an argument you can check rather than a result you can rely on.

Most companies publish this sort of thing after launch, once there's something to defend. I'd rather put it out while I've still got time to be wrong about it.

Definitions

AEAD. Authenticated encryption with associated data. A construction that hides content and also detects tampering, rather than needing a separate integrity check bolted on.

AES-256-GCM. A symmetric AEAD cipher. Hides the content, and also notices if someone has tampered with it.

BIP-39. A published standard for turning randomness into memorable words and back again. It came out of cryptocurrency wallets and it's been hammered on for years.

End-to-end encryption. Encrypted on your device, decrypted on your device, unreadable to everyone in between.

Entropy. A measure of unpredictability. 128 bits means a guessing attack has 2^{128} possibilities to work through, which is not a number anyone is working through.

HKDF. A function that takes one high-entropy secret and produces one or many keys from it, giving away nothing about the input.

Keychain. Apple's encrypted store for secrets on iOS.

Password-based key derivation function. A deliberately slow function such as Argon2id, used to make guessing a human-chosen password expensive. Section 7.1 explains why Catchlight does not need one.

NFD normalisation. A rule for writing Unicode one consistent way, so the same phrase typed on two devices produces the same key.

Nonce. A number used once. Reuse one with the same key and you break the cipher, which is why a fresh random one is generated every single time.

Secure Enclave. A separate processor inside Apple devices that holds keys and does cryptography, and software on the main processor can't pull a key out of it.

Tombstone. A record saying an item was deleted, kept rather than just removing the item and hoping.

X25519. A key-agreement algorithm. Two devices use it to agree on a shared secret over a channel other people can read but cannot decipher.

Zero knowledge. The operator can prove nothing about your content, because the operator holds nothing.

References

Specifications

RFC 5869, HMAC-based Extract-and-Expand Key Derivation Function (HKDF)

RFC 7748, Elliptic Curves for Security

RFC 8439, ChaCha20 and Poly1305 for IETF Protocols

FIPS 197, Advanced Encryption Standard

FIPS 180-4, Secure Hash Standard

NIST SP 800-38D, Block Cipher Modes of Operation, Galois/Counter Mode

BIP-39, Mnemonic code for generating deterministic keys

Guidance

OWASP Mobile Application Security Verification Standard, MASVS-CRYPTO and MASVS-STORAGE

OWASP Mobile Application Security Testing Guide

Apple Platform Security

Apple CryptoKit

Companion documents

Who might read your notes, for who the design in this paper actually defends against, and the four exposures it does not

How to tell whether a notes app is actually private, for the seven tests a reader can run against any app, including this one

Sync with no server and no account, for how the folder format in section 4 behaves against a real cloud provider

Version history

VERSION	DATE	CHANGE
1.6	7 September 2026	Adds the location searches to what crosses the boundary, and stops stating a count
1.5	5 September 2026	Two prose semicolons replaced, the house style has none. No technical change
1.4	1 September 2026	States the tombstone retention period, 180 days. Paper 5 tells a reader to demand a retention number from any app and we had never published our own
1.3	1 September 2026	Added a Companion documents block, which this paper did not have, and linked every in-prose section reference to its heading
1.2	1 September 2026	iOS Writing Tools is a fourth thing crossing the encryption boundary, confirmed in the editor on device. Step 6 now scopes its count to what the app does, and a new paragraph covers the path the system opens
1.1	1 September 2026	Step 6 named a Markdown export as the only thing leaving unencrypted. A reminder's notification text and a shared link's address do too, both already published in the threat model. The count was written without enumerating, which is the fault rather than the number
1.0	31 August 2026	First release. Describes the architecture as built and merged, before launch