Security Review Report — Revision 2

Keylay

Protocol Implementation Review: Cryptographic Design, Transport Security, Relay Trust Claims, and Annotated Code Map

Subject:	**Keylay Encrypted QR Relay — index.html, server.js**
Security page:	**security.html (keylay_site_updated)**
Audit scope:	**Cryptographic correctness, protocol claims, relay trust model, code structure**
Methodology:	**Manual source review, claim-vs-code differential analysis**
Original review:	**April 2, 2026**
This revision:	**April 02, 2026**
Changes since v1:	**Three findings resolved (F-01, F-03, F-07); security.html updated; code map appendix added**
Status:	**Internal draft — not a formal third-party security review**

Revision 2 reflects three fixes applied to the codebase following the initial review: counter-before-decryption ordering (F-01), server-side channel-hash logging (F-03), and modular bias in code generation (F-07). The security.html documentation page was also updated to correct the three discrepancies identified in Revision 1. All other analysis, findings, and recommendations are carried forward unchanged.

Table of Contents

1. Executive Summary

This revision updates the April 2026 review of the Keylay encrypted QR relay to reflect three code fixes applied in response to findings F-01, F-03, and F-07, and to document the corresponding update to the security.html page. The fixes are verified against the current source.

F-01 (counter advanced before decryption) — resolved. In `processDataMessage()`, the assignment `recvCounter = msg.counter` now occurs inside the `try` block, after `await decrypt(...)` returns successfully. A failed decryption no longer consumes the counter slot, closing the relay-injectable denial-of-service vector.

F-03 (server channel-hash logging) — resolved. The relay server no longer logs the `code` field (channel hash) or full message objects. Log output is now limited to message type and peer count.

F-07 (modular bias in code generator) — resolved. `generateRandomCode()` now uses rejection sampling with a threshold of 248, discarding any byte value ≥ 248 before applying `% 31`. The generator now produces a provably uniform distribution across the 31-character alphabet.

Four findings remain open: no server-side rate limiting or message size enforcement (F-02), unauthenticated sender role claiming (F-04), absent Content Security Policy (F-05), and no server-side session expiry (F-06). None of these are critical within the stated threat model, but each is straightforward to address.

The three documentation discrepancies identified in Revision 1 — PBKDF2 iteration count, replay protection status, and hello wire format — have been corrected in security.html. The page now accurately describes the deployed implementation.

2. Scope and Methodology

Files Reviewed

File	Role	Lines
index.html	Client application — all cryptographic logic, UI, WebSocket client	~12,720
server.js	WebSocket relay server — session routing, role management	146
security.html	Published security claims page — basis for claim comparison	~415

Methodology

The original review traced every specific, testable claim in security.html to the corresponding code path, evaluated each for accuracy, and audited the server independently. This revision re-examined each of the three affected code locations to confirm that the fixes address the identified problems without introducing new issues, and re-read the updated security.html to confirm it now correctly reflects the deployed state.

3. Protocol Architecture

Keylay routes coordination data (PSBTs, public keys, UR sequences) between two peers through a WebSocket relay. The relay is kept blind: it routes ciphertext and sees only a derived channel identifier, never the raw session code or plaintext payload. The security model depends on this relay-blindness being cryptographically enforced, not merely trusted.

Protocol Steps (as implemented)

1	**Code entry**	User enters or generates a session code. The generator uses a 31-character unambiguous alphabet (no 0/O/1/I/l) at 10 characters, producing a uniform distribution via rejection sampling. Effective entropy: $\log_2(31^{10})$ ~ 49.84 bits. Manual entry accepted up to 32 characters.
2	**Channel derivation**	SHA-256(code) is computed; the first 32 hex characters (128 bits) are used as the WebSocket channel name. The relay sees this hash, not the raw code.
3	**HMAC key derivation**	The raw code is stretched via PBKDF2-SHA256, salt ‘keylay-v1-hmac’, 300,000 iterations, producing an HMAC-SHA256 signing key that authenticates hello messages without revealing the code.
4	**Ephemeral X25519 key generation**	An ephemeral X25519 key pair is generated per session via crypto.subtle.generateKey. Private key is set to null after handshake.
5	**Signed public-key exchange**	Each peer HMAC-signs its base64-encoded ephemeral public key and sends {type:'hello', pubkey, sig}. The relay forwards it. The receiver verifies the HMAC before accepting the key. Without the session code, the relay cannot forge a valid signature.
6	**HKDF session key derivation**	Both peers compute X25519(myPrivate, theirPublic). Shared bits are imported as HKDF key material. HKDF info encodes 'keylay-v1-session|' + sorted public key pair, binding the AES-256-GCM key to the exact handshake participants. No key material transits the wire.
7	**AES-GCM message encryption**	Each message is encrypted with a fresh 12-byte random IV and AAD 'keylay-v1|' + counter. The receiver enforces strictly monotonic counter ordering; the counter is advanced only after successful decryption and GCM tag verification.
8	**Handshake state machine**	States: IDLE → HELLO_SENT → ACTIVE. A concurrency lock prevents simultaneous hello processing. A different incoming peer key while ACTIVE forces a full handshake reset.

4. Verified Security Claims

Each security property described in security.html was traced to a specific code path and evaluated for correctness. No claims changed status between Revision 1 and Revision 2.

Status	Claim (from security.html)	Finding
PASS	Relay sees only ciphertext and channel hash, not plaintext	Confirmed. Payload is AES-GCM ciphertext; channel name is SHA-256(code)[:32].
PASS	Relay MITM resistance without code knowledge	Confirmed. HMAC over ephemeral public key; forgery requires PBKDF2 derivation from code.
PASS	Forward secrecy: past sessions not decryptable after code disclosure	Confirmed. Session key derived from ephemeral X25519; private key nulled post-handshake.
PASS	No session key transmitted	Confirmed. Both peers derive AES key locally from shared DH bits via HKDF.
PASS	HKDF info binds key to sorted public key pair	Confirmed. info = "keylay-v1-session|" + [myPub,theirPub].sort().join("|").
PASS	AES-GCM with 12-byte random IV per message	Confirmed. crypto.getRandomValues(new Uint8Array(12)) per encrypt() call.
PASS	Counter-bound AAD prevents replay; counter enforced on receipt	Confirmed. AAD = "keylay-v1|"+counter; recvCounter advanced only after successful decrypt.
PASS	Unencrypted data messages rejected	Confirmed. msg.encrypted === false causes immediate rejection.
PASS	Peer-key change triggers full handshake reset	Confirmed. New ephemeral key pair generated; counters reset; sessionKey nulled.
PASS	Session limited to exactly two peers	Confirmed. server.js refuses third connection and closes socket.
PASS	Ephemeral private key discarded after handshake	Confirmed. ephemeralKeyPair = null immediately after key derivation.
PASS	HMAC verification uses Web Crypto constant-time verify	Confirmed. crypto.subtle.verify() used — not a string comparison.
PASS	No custody of wallet private keys	Confirmed. App handles only coordination data. No signing occurs.
NOTE	Code knowledge treated as session membership	Correct as stated. Any party with the code can join — by design.

5. Discrepancies: Security Page vs. Deployed Code (Resolved)

Revision 1 identified three cases where security.html described the implementation as less capable than the deployed code actually was. All three have been corrected in the current version of security.html. They are preserved here for the review record.

RESOLVED	D-01 · PBKDF2 Iteration Count
Was:	Page claimed: current deployment uses 100,000 iterations; target V1 is 300,000.
Now:	Code used 300,000. Security page has been updated to reflect this.

RESOLVED	D-02 · Replay Protection Status
Was:	Page claimed: "Replay protection is not yet deployed."
Now:	Counter-bound AAD and strict monotonic counter enforcement were already deployed. Security page now accurately describes this.

RESOLVED	D-03 · Hello Message Wire Format
Was:	Page claimed: hello carried inside a data envelope with JSON-stringified payload.
Now:	Code sends type:"hello" messages directly with pubkey and sig fields. Security page updated to match.

6. Identified Vulnerabilities and Weaknesses

F-01, F-03, and F-07 were resolved between Revision 1 and this revision and are shown in resolved state. F-02, F-04, F-05, and F-06 remain open.

RESOLVED F-01 · Counter Advanced Before Decryption Verification

RESOLVED. In processDataMessage(), recvCounter = msg.counter now occurs inside the try block, after await decrypt(...) returns. A relay-injected message with a fabricated counter value now fails decryption without consuming the counter slot. Subsequent legitimate messages are not affected.

Verified fix (lines 12357–12368):

async function processDataMessage(msg) { if (typeof msg.counter !== 'number' || msg.counter <= recvCounter) { console.warn(...); return; } let text; try { text = await decrypt(msg.payload, sessionKey, msg.counter); recvCounter = msg.counter; // ← advanced only on success ... } catch (error) { ... }

MODERATE F-02 · Server: No Rate Limiting or Message Size Enforcement

Open. The relay server imposes no restrictions on connection rate, message frequency, or payload size. Any client that can reach the WebSocket endpoint can join sessions and transmit arbitrarily large or frequent messages.

The client-side messageBuffer has no size cap, creating a secondary memory exhaustion vector if a peer floods messages before the handshake completes.

Recommended: per-IP connection limits, per-session message rate limits, maximum payload size (e.g. 64 KB) at the server, and a client-side messageBuffer cap.

RESOLVED F-03 · Server: Full Message Logging Including Channel Hash

RESOLVED. The relay server no longer logs the code field (channel hash) or full message objects. Log output is now limited to message type and peer count: e.g., "Received message type: data", "Routed data to 1 peer(s)", "Forwarded hello to 1 peer(s)".

LOW F-04 · Unauthenticated Role Claiming

Open. Any connected peer can send {type: "claim"} at any time to claim the sender role, demoting the current sender to receiver. No additional authentication is required beyond channel membership.

This is within the stated threat model but is not documented in security.html. A code-knowing adversary could disrupt session workflow by continuously cycling the sender role.

Recommended: document this behavior explicitly. Consider whether the sender role should be permanently bound to the first joiner.

LOW F-05 · Missing Content Security Policy

Open. The page sets no Content-Security-Policy header or meta tag. A successful XSS injection would have unrestricted access to session keys, decrypted payloads, and the WebSocket connection.

Recommended: add a restrictive CSP. At minimum: default-src 'self'; script-src 'self'; connect-src wss: ws:

INFO

F-06 · No Server-Side Session Expiry

Open. Sessions in the server's Map are removed only when all peers disconnect. No maximum session lifetime or idle timeout is enforced. Sessions whose peers become unreachable persist indefinitely.

Recommended: maximum session age (e.g. 24 hours) and idle timeout (e.g. 30 minutes). Clean up stale sessions on a periodic interval.

RESOLVED

F-07 · Modular Bias in Code Generator

RESOLVED. generateRandomCode() now uses rejection sampling. Byte values ≥ 248 are discarded and redrawn before applying % 31. The generator now produces a provably uniform distribution across the 31-character alphabet. Expected redraws per character: ~1.03.

Verified fix (lines 10789–10804):

function generateRandomCode() { const alphabetLen = alphabet.length; // 31 const threshold = 256 - (256 % alphabetLen); // 248 let code = ''; while (code.length < CHANNEL_CODE_LENGTH) { const arr = new Uint8Array(CHANNEL_CODE_LENGTH - code.length); crypto.getRandomValues(arr); for (let i = 0; i < arr.length ...) { if (arr[i] < threshold) { code += alphabet.charAt(arr[i] % alphabetLen); } } } return code; }

7. Entropy and Code-Strength Analysis

Generated Code Parameters

CHANNEL_CODE_CHARS = `'abcdefghjkmnpqrstuvwxyz23456789'` (31 characters; no ambiguous glyphs 0/O/1/I/l). CHANNEL_CODE_LENGTH = 10. With the corrected uniform generator, effective entropy equals the theoretical maximum: $\log_2(31^{10}) \sim 49.84$ bits.

Attack Cost Analysis

At 49.84 bits, online brute-force requires on the order of $2^{49} \sim 562$ trillion guesses worst-case. Each guess requires a PBKDF2 operation at 300,000 iterations (~300-500 ms on modern hardware), making online guessing computationally prohibitive even against weak codes.

Offline attack against recorded traffic is blocked by forward secrecy: the session key is derived from ephemeral X25519, not from the code. Learning the code after a session ends does not decrypt past ciphertext. An attacker must be present at session time to substitute a key, which requires forging the HMAC — which requires the code.

For manually entered codes, security is parameterized on code quality. A 4-character manual code yields ~24 bits — trivially brute-forceable offline against a recorded handshake. The security page correctly notes this.

8. Server-Side Security Review

Session Isolation

Sessions are keyed by channel hash and stored in a server-side Map. Each session holds exactly one sender and one receiver WebSocket reference. Third connections are refused and the socket closed immediately. Cross-session isolation relies on 128-bit channel identifier collision resistance; accidental collision under normal usage is negligible.

Role Assignment and Transfer

The first client to a session hash becomes sender. Role transfer via "claim" is unauthenticated beyond channel membership (see F-04). The disconnect handler promotes the first available receiver to sender if the sender drops, which allows sessions to continue without requiring a full rejoin. This is reasonable behavior but the automatic promotion is also not documented in the security page.

Logging

Following the F-03 fix, production log output contains only message type and peer count. Channel hashes, public keys, and HMAC signatures no longer appear in log output.

Input Validation

Malformed JSON is caught and silently discarded. Field-level validation remains absent — the code, payload, pubkey, and sig fields are not bounded in length. The F-02 finding (no payload size limit) is the primary unresolved concern here.

9. Positive Findings

✓ **Web Crypto API exclusively**	All cryptographic operations use crypto.subtle. No third-party cryptographic primitives are in the protocol path.
✓ **Ephemeral private key discarded post-handshake**	ephemeralKeyPair = null immediately after deriveSessionKey(). Forward secrecy is correctly implemented.
✓ **Constant-time HMAC verification**	crypto.subtle.verify() is used — timing side-channel attacks cannot leak information about the expected HMAC value.
✓ **Counter-bound AAD with correct ordering (fixed F-01)**	recvCounter is now advanced only after successful AES-GCM decryption. The counter slot is no longer consumable by a failing injection.
✓ **Uniform code generation (fixed F-07)**	Rejection sampling ensures each character is drawn from a provably uniform distribution over the 31-character alphabet.
✓ **HKDF info binds key to session participants**	The sorted public key pair in HKDF info means two different handshakes involving the same code but different ephemeral keys derive different session keys.
✓ **Handshake concurrency lock**	helloProcessing flag prevents concurrent handleHelloMessage executions from racing on ephemeralKeyPair.
✓ **Peer key change detection forces reset**	A different incoming public key while ACTIVE clears sessionKey, resets counters, regenerates the ephemeral key pair, and restarts the handshake.
✓ **X25519 browser capability check**	The app detects missing X25519 support before enabling the join flow and displays a clear warning. No silent fallback to weaker cryptography.
✓ **Minimal server logging (fixed F-03)**	Log output is now bounded to message type and peer count. No session identifiers or key material appear in server logs.

10. Summary Table

ID	Finding	Severity	Status
D-01	PBKDF2 iterations: page said 100k, code used 300k	Discrepancy	Resolved
D-02	Replay protection: page said absent, code had it	Discrepancy	Resolved
D-03	Hello wire format described incorrectly	Discrepancy	Resolved
F-01	Counter advanced before decryption — DoS vector	Moderate	Resolved
F-02	No rate limiting or message size enforcement (server)	Moderate	Open
F-03	Channel hash logged in server output	Low	Resolved
F-04	Unauthenticated role claiming, undocumented	Low	Open
F-05	No Content Security Policy	Low	Open
F-06	No server-side session expiry	Info	Open
F-07	Modular bias in code generator	Info	Resolved

11. Conclusions and Recommendations

The three issues that posed the most concrete risk — a counter-ordering DoS vector, server-side session correlation through logging, and a biased random generator — have been fixed and verified. The security documentation page has been updated to match the deployed code. The implementation is now in substantially better alignment between what it claims and what it does.

Remaining Open Recommendations

High	Add server-side rate limiting, per-IP connection caps, and a maximum message payload size. Add a client-side cap on messageBuffer.
Medium	Add a Content Security Policy header or meta tag to index.html.
Low	Document the unauthenticated role-claim behavior in security.html.
Low	Add server-side session expiry (maximum age and idle timeout).
Low	Pin the security page to a specific commit or version tag.

This report was produced through manual source review with AI-assisted analysis. It constitutes a detailed technical review but not a formal third-party security audit.

Appendix A: Annotated Code Map

The following tables map every significant block in index.html and server.js to its function in the execution and security model. For third-party libraries, the source and version are noted and the contents are not analyzed beyond identity verification — their security properties are those of the published packages. For application code, key lines are quoted to anchor each annotation to the specific code path.

A.1 index.html — Structure Overview

Lines	Block	Role in execution / security model
1–4	**Document bootstrap**	DOCTYPE, html, head — no security relevance.
5–13	**Meta tags**	charset=UTF-8, viewport. No security relevance.
7–394	**CSS <style> block**	All UI styling. No cryptographic or security relevance.
396–495	**HTML body / UI markup**	Channel setup form, main app container, save-format modal, QR display canvas, video element for camera. maxlength="32" on the code input is the only security-relevant attribute.
496–10599	**jsQR v1.4.0 (inlined)**	Third-party library — see §A.2.
10600–10609	**qrcode v1.5.1 (inlined)**	Third-party library — see §A.2.
10610–10649	**Application script / constants**	Version string and all configuration constants. Security-relevant: GCM_IV_LENGTH, CHANNEL_CODE_LENGTH, CHANNEL_CODE_CHARS, WS_LOCAL_URL, WS_CLOUD_URL, SESSION_LIMIT_MS.
10653–10804	**Cryptographic functions**	All crypto primitives — see §A.3.
10806–10960	**BBQr utility functions**	Base36, hex, Base32 encode/decode; BBQr frame packing and unpacking. No cryptographic role.
10961–11300	**Format helpers / file handling**	UR frame parsing, filename generation, file upload processing, format detection, BBQr/UR send paths. Data enters encryption at sendEncrypted() — these helpers prepare it before that call.
11301–11320	**Server selection**	Tries ws://localhost:8080 first (3 s timeout), falls back to wss://app.keylay.org/ws. Local connection is shown with a green indicator; cloud connection with a grey one.
11321–11375	**State variable declarations**	All mutable session and handshake state — see §A.4.
11376–11398	**sendEncrypted(), helper**	The encrypt-and-send entry point. Enforces handshakeState === ACTIVE before any send. Increments sendCounter, calls encrypt(), writes to WebSocket.
11400–11435	**updateConnectionStatus()**	UI only — no security relevance.

Lines	Block	Role in execution / security model
11435–11570	**joinChannel() / leaveChannel()**	WebSocket lifecycle — see §A.5.
11571–11910	**QR scan loop and dispatch**	Camera frame processing, BBQr and UR multi-frame reassembly state machines, QR result routing. Scanned data is passed to sendEncrypted() — these functions sit upstream of encryption.
11910–12180	**Sender display / file output**	QR animation display, file download, received-data UI. Downstream of decryption — handles plaintext after processDataMessage() resolves.
12181–12299	**Handshake helpers**	broadcastHello(), handleHelloMessage() — see §A.6.
12301–12354	**handleWebSocketMessage()**	WebSocket event dispatch — see §A.7.
12355–12410	**processDataMessage()**	Replay protection and decryption dispatch — see §A.8.
12411–12640	**UI update functions**	Role display, scan status, modal management, received-data type label. No cryptographic role.
12641–12720	**Event setup / DOMContentLoaded**	Button handlers (join, generate, claim, leave, scan), X25519 browser capability check on load.

A.2 Third-Party Inlined Libraries

Two libraries are inlined directly into the HTML rather than loaded from a CDN. Inlining eliminates CDN availability as a dependency and removes the risk of CDN-level compromise for these specific packages, but it means updates require manual replacement of the inlined block.

Library	Lines	Notes
jsQR v1.4.0	`496–10599`	QR code image decoder by Cosmo Wolfe. Used solely to decode frames from the device camera feed via jsQR(imageData, width, height). The library is not in the encryption or handshake path; it only processes inbound camera pixels before the scanned string is passed to sendEncrypted(). The inlined block opens with the webpack bundle header (function webpackUniversalModuleDefinition(...)) and closes with exports.default = jsQR at line 876, consistent with the published npm package structure for jsQR@1.4.0, the last published version. v1.4.0 has no known security advisories.
qrcode v1.5.1	`10600–10609`	QR code canvas renderer by soldair (node-qrcode project). Used solely to render QR codes to the display canvas via QRCode.toCanvas(...). Not in the encryption or handshake path; it only converts already-encrypted or already-plaintext strings into a visual QR image for display. The comment block references the jsdelivr CDN source path /npm/qrcode@1.5.1/build/qrcode.js and notes the SRI advisory. The library body is a single minified line (line 10607). v1.5.1 has no known security advisories.

A.3 Cryptographic Functions (lines 10653–10804)

All cryptographic operations are grouped in one section of the application script, immediately after the constants block. Every function uses `crypto.subtle` exclusively. No custom algorithms, no third-party crypto libraries.

checkX25519Support() (lines 10658–10665)

Attempts crypto.subtle.generateKey({name:'X25519',...}) and returns true/false. Called on DOMContentLoaded to gate the join button.

Security role: prevents silent fallback to missing cryptography on unsupported browsers.

deriveChannelName(code) (lines 10669–10676)

Computes SHA-256 of the raw code string and returns the first 32 hex characters.

```
const hashBuffer = await crypto.subtle.digest('SHA-256', encoded);
return Array.from(new Uint8Array(hashBuffer))
  .map(b => b.toString(16).padStart(2,'0'))
  .join('').slice(0, 32);
```

Security role: the relay receives this hash as the room identifier, never the raw code. 128-bit channel name provides negligible collision probability under normal usage.

generateEphemeralKeyPair() (line 10680)

Wraps crypto.subtle.generateKey({name:'X25519'}, true, ['deriveBits']).

```
return crypto.subtle.generateKey({ name: 'X25519' }, true, ['deriveBits']);
```

extractable:true is required to export the public key for the handshake. The private key is never exported by the application; ephemeralKeyPair is set to null after use, making private key material inaccessible from JS context.

deriveSessionKey(myKeyPair, myPubB64, theirPubB64) (lines 10691–10718)

Performs X25519 Diffie-Hellman, imports the shared bits as HKDF material, and derives an AES-256-GCM key. The HKDF info field encodes the sorted pair of base64 public keys.

```
const sharedBits = await crypto.subtle.deriveBits(
  { name: 'X25519', public: theirPublicKey },
  myKeyPair.privateKey, 256);
const keys = [myPublicKeyB64, theirPublicKeyB64].sort();
const info = new TextEncoder().encode('keylay-v1-session|' + keys.join('|'));
return crypto.subtle.deriveKey(
  { name: 'HKDF', hash: 'SHA-256', salt: new Uint8Array(32), info },
  hkdfKey, { name: 'AES-GCM', length: 256 }, false, ['encrypt','decrypt']);
```

Security role: no session key transits the wire. Sorting the public keys ensures both peers derive the identical key regardless of join order. The HKDF salt is a 32-byte zero array — not secret, but its domain-separation role is covered by the info field.

deriveHmacKey(code) (lines 10723–10734)

Derives an HMAC-SHA256 key from the session code using PBKDF2-SHA256, salt='keylay-v1-hmac', iterations=300,000.

```
return crypto.subtle.deriveKey(
  { name: 'PBKDF2', hash: 'SHA-256',
    salt: new TextEncoder().encode('keylay-v1-hmac'), iterations: 300000 },
  keyMaterial, { name: 'HMAC', hash: 'SHA-256' }, false, ['sign','verify']);
```

Security role: makes brute-forcing the code expensive (~300–500 ms per attempt). PBKDF2 is not memory-hard; Argon2 would be stronger for offline attack resistance. Salt diverges from target V1 value ('keylay-v1-hmac-salt') — changing it would require a migration plan and is the one remaining V1 delta.

signPublicKey() / verifyPublicKey() (lines 10736–10748)

Sign: applies crypto.subtle.sign('HMAC', hmacKey, encoded_pubkey), returns base64. Verify: applies crypto.subtle.verify('HMAC', hmacKey, sig, encoded_pubkey), returns bool.

```
const sig = await crypto.subtle.sign(
  'HMAC', hmacKey, new TextEncoder().encode(pubkeyB64));
// verify path:
return await crypto.subtle.verify(
  'HMAC', hmacKey, sig, new TextEncoder().encode(pubkeyB64));
```

Security role: verify() uses a constant-time comparison — no timing side-channel. Any peer that does not know the session code cannot produce a valid signature and will be rejected by the receiving peer's verifyPublicKey() call.

encrypt(plaintext, key, counter) (lines 10751–10765)

Generates a fresh 12-byte IV, constructs AAD as 'keylay-v1|'+counter, calls AES-GCM encrypt, and prepends the IV to the ciphertext before returning as base64.

```
const iv = crypto.getRandomValues(new Uint8Array(12));
const aad = new TextEncoder().encode('keylay-v1|' + counter);
const ciphertext = await crypto.subtle.encrypt(
  { name: 'AES-GCM', iv, additionalData: aad }, key, plaintext_bytes);
// iv prepended: combined = [iv (12 bytes) | ciphertext + GCM tag]
```

Security role: per-message random IV ensures identical plaintexts produce different ciphertexts. Counter in AAD binds each ciphertext to a specific sequence position; modification of the counter field in transit causes GCM tag failure. GCM tag (16 bytes) provides authenticated encryption.

decrypt(encryptedBase64, key, counter) (lines 10768–10786)

Decodes base64, slices out the IV (first 12 bytes) and ciphertext, reconstructs AAD from the counter, calls AES-GCM decrypt.

```
const iv = combined.slice(0, 12);
const ciphertext = combined.slice(12);
const aad = new TextEncoder().encode('keylay-v1|' + counter);
const decrypted = await crypto.subtle.decrypt(
  { name: 'AES-GCM', iv, additionalData: aad }, key, ciphertext);
// throws on tag mismatch — caught by caller
```

Security role: GCM decryption will throw if either the ciphertext or the AAD (including the counter) was altered. The counter the receiver reconstructs comes from msg.counter (plaintext on the wire); if the relay modifies it, decryption fails because the AAD won't match what was used during encryption.

generateRandomCode() (lines 10789–10804)

Fills Uint8Arrays from crypto.getRandomValues, discards bytes ≥ 248 (rejection sampling), maps remaining bytes via % 31 into CHANNEL_CODE_CHARS.

```
const threshold = 256 - (256 % alphabetLen); // 248
while (code.length < CHANNEL_CODE_LENGTH) {
  const arr = new Uint8Array(CHANNEL_CODE_LENGTH - code.length);
  crypto.getRandomValues(arr);
  for (let i = 0; i < arr.length ...) {
    if (arr[i] < threshold) {
      code += alphabet.charAt(arr[i] % alphabetLen);
    }
  }
}
```

Security role: produces a uniformly distributed code with $\log_2(31^{10})$ ~ 49.84 bits of entropy. Rejection rate per byte: 8/256 ~ 3%; expected draws per character: 1.03.

A.4 State Variable Declarations (lines 11321–11375)

All mutable session state is declared as module-level `let` variables in a single block. The security-relevant variables are:

Variable	Init value	Security role
`handshakeState`	`'IDLE'`	Guards sendEncrypted() — messages cannot be sent until state reaches ACTIVE.
`ephemeralKeyPair`	`null`	Holds the X25519 CryptoKeyPair. Set to null after deriveSessionKey() completes, making private key inaccessible.
`myPublicKeyB64`	`''`	This peer's exported public key. Used in HMAC signing and HKDF info.
`peerPublicKeyB64`	`''`	Peer's public key from completed handshake. Used to detect peer reconnect with a different key.
`hmacKey`	`null`	HMAC-SHA256 CryptoKey. Used to sign and verify hello messages. Derived from session code; cleared on leaveChannel().
`sessionKey`	`null`	AES-256-GCM CryptoKey. Never transmitted. Cleared on leaveChannel() and on peer key change.
`messageBuffer`	`[]`	Holds data messages that arrived before ACTIVE. Flushed by handleHelloMessage(). No size cap (F-02).
`helloProcessing`	`false`	Concurrency lock. Prevents two simultaneous handleHelloMessage() executions racing on ephemeralKeyPair.
`sendCounter`	`0`	Monotonic send counter. Included in AES-GCM AAD. Reset to 0 on leaveChannel() and peer key change.
`recvCounter`	`-1`	Last accepted peer counter. Any incoming counter <= recvCounter is rejected. Reset to -1 on session reset.

A.5 joinChannel() / leaveChannel() (lines 11435–11570)

`joinChannel(code)` is the session bootstrap entry point. Its ordering is security-relevant: cryptographic setup runs before the WebSocket is opened, preventing a race where an incoming hello arrives before keys are ready.

1	Derive channel hash and ephemeral key pair (awaited before WebSocket is created)	`channelHash = await deriveChannelName(channelCode);`
		`ephemeralKeyPair = await generateEphemeralKeyPair();`
		`myPublicKeyB64 = await exportPublicKey(ephemeralKeyPair);`
		`hmacKey = await deriveHmacKey(channelCode);`

2	Open WebSocket. onopen sends join message and immediately calls broadcastHello()	`ws = new WebSocket(url);`
		`ws.onopen = async () => {`
		`  ws.send(JSON.stringify({ type: 'join', code: channelHash }));`
		`  broadcastHello();`
		`};`

3	Keepalive ping every 25 s prevents proxy idle timeouts	`setInterval(() => ws.send(JSON.stringify({type:'ping'})), 25000);`
4	Session hard limit: close after 30 min	`setTimeout(() => ws.close(1000,'Session timeout'), SESSION_LIMIT_MS);`

`leaveChannel()` clears all session state: closes the WebSocket, nulls `sessionKey`, `hmacKey`, `ephemeralKeyPair`, resets both counters to their initial values, empties the message buffer, and resets handshakeState to IDLE. This ensures no key material persists after a session ends.

A.6 Handshake Helpers (lines 12181–12299)

broadcastHello() — lines 12187–12212

Sets handshakeState to HELLO_SENT, then calls the inner `sendHello()` immediately and every 2 seconds until ACTIVE. Each send signs the current `myPublicKeyB64` with `hmacKey` and writes a type:'hello' message.

```
ws.send(JSON.stringify({ type: 'hello', code: channelHash, pubkey: myPublicKeyB64, sig
}));
```

The retry loop is necessary because the relay drops messages when the other peer is not yet connected. Retrying every 2 s is safe: hellos are idempotent and the receiving peer deduplicates on public key equality (handshakeState === ACTIVE && theirPublicKeyB64 === peerPublicKeyB64 → return).

handleHelloMessage(msg) — lines 12216–12299

The core handshake receiver. Runs under `helloProcessing` lock. Execution order:

Ignore own reflection	`if (theirPublicKeyB64 === myPublicKeyB64) return;`	The relay broadcasts hellos to all peers in the session including the sender; this guard discards reflected copies.
Verify HMAC signature	`if (!theirSig || !(await verifyPublicKey(theirPubKeyB64, theirSig, hmacKey)))` `  return;`	Rejects any peer that cannot sign with a key derived from the session code. Uses crypto.subtle.verify() (constant-time).

Handle peer reconnect	`if (handshakeState==='ACTIVE' && theirPubKeyB64 !== peerPublicKeyB64) {`
	`  sessionKey = null;  handshakeState = 'IDLE';`
	`  sendCounter = 0;  recvCounter = -1;`
	`  ephemeralKeyPair = await generateEphemeralKeyPair();`
	`}`
		If a different public key arrives while ACTIVE, all state is reset and a fresh handshake is forced. Prevents reuse of a session key with a new peer.

Derive session key	`sessionKey = await deriveSessionKey(` `  ephemeralKeyPair, myPublicKeyB64, theirPublicKeyB64);`	X25519 DH + HKDF. At this point ephemeralKeyPair still exists.
Advance to ACTIVE and discard private key	`handshakeState = 'ACTIVE';` `ephemeralKeyPair = null;`	Private key discarded immediately. Forward secrecy begins here.
Send final hello	`ws.send(JSON.stringify({type:'hello', code:channelHash,` `  pubkey:myPublicKeyB64, sig:finalSig}));`	One final hello ensures the peer can complete its own handshake if it joined after the initial hello was sent.

Flush message buffer	`const buffered = messageBuffer.splice(0);` `for (const msg of buffered) await processDataMessage(msg);`	Data messages buffered during the handshake are now decrypted in order.

A.7 handleWebSocketMessage() (lines 12305–12353)

Top-level WebSocket message dispatcher. All incoming messages arrive here. Routing logic:

`type:"role"`	Assigned by server on join. Sets isSender; calls updateRoleUI().
`type:"status"`	Server status message (e.g. "session full"). Shown in scan status UI.
`type:"hello"`	Routed to handleHelloMessage() if msg.code === channelHash. The channelHash check prevents processing hellos from other sessions that might share the same WebSocket connection (not currently possible by design, but a useful guard).
`type:"data"`	Requires msg.encrypted === true — unencrypted data messages are rejected with console.warn. If handshakeState !== ACTIVE, the message is pushed to messageBuffer. Otherwise routed to processDataMessage().

A.8 processDataMessage() (lines 12355–12410)

Replay protection and decryption. The function is called either directly from handleWebSocketMessage() (when ACTIVE) or by handleHelloMessage() when flushing the buffer.

```
async function processDataMessage(msg) { // 1. Replay gate if (typeof msg.counter !== 'number' || msg.counter <= recvCounter) { console.warn(`Replay detected...`); return; } // 2. Decrypt (counter advanced only on success) let text; try { text = await decrypt(msg.payload, sessionKey, msg.counter); recvCounter = msg.counter; // ← F-01 fix: after decrypt } catch (error) { showScanStatus('Failed to decrypt...'); return; } // 3. Dispatch by format (binary / ur / bbqr / text) ... }
```

Step 1 enforces strictly monotonic ordering: any counter that is not a number, or is ≤ the last accepted value, is silently dropped. Step 2 decrypts under the AES-GCM session key with the same counter-bound AAD used at encryption time. Step 3 routes the decrypted plaintext to format-specific display handlers (BBQR animation, UR sequence display, QR canvas render, or plain text display).

A.9 server.js — Complete Structure (146 lines)

The relay server is intentionally minimal. It routes messages between peers and manages role assignment. It has no knowledge of session keys, public keys (beyond forwarding them), or plaintext content.

Lines	Function	Role
1–3	**Module bootstrap**	require('ws'); create WebSocket.Server on port 8080.
5–6	**sessions Map**	Stores active sessions: code_hash → {sender: ws, receivers: Set(ws)}. All session state lives here. No persistence between restarts.
8–53	**connection handler**	Attaches message, close handlers per socket. Close handler removes the peer from its session and promotes a receiver to sender if the sender disconnects.

11–27	**message dispatch**	Parses JSON, routes to handleJoin / handleClaimSender / handleDataTransmission / handleHello by message.type. Malformed JSON is caught and discarded.
55–75	**handleJoin(ws, code)**	First joiner → sender role. Second joiner → receiver role. Third connection → refused, socket closed.
77–105	**handleClaimSender(ws, code)**	Any connected peer can claim sender role at any time. Current sender is demoted to receiver. See F-04.
107–128	**handleDataTransmission(ws, code, ...)**	Forwards encrypted payload to all other peers in the session. Does not inspect payload content. format, encrypted, counter fields are forwarded as-is.
130–143	**handleHello(ws, code, pubkey, sig)**	Forwards hello message to all other peers. Does not verify the HMAC signature — verification is the client's responsibility. Server is deliberately kept blind to key material.

This report was produced through manual source review with AI-assisted analysis. It constitutes a detailed technical review but not a formal third-party security audit.