# <aimachine/>

## Suite of Secure Communication Applications

User Manual and Security Overview

The AI Machine UG
D-66333 Völklingen, Germany

May 3, 2026

Legal company name: **The AI Machine UG (haftungsbeschraenkt)**.
Short name used in this document: **The AI Machine UG**.
Landing page: https://aimachine.io

## Contents

# 1 Introduction

## 1.1 Scope of This Document

This document provides a unified user manual and security overview for the four browser-based applications that together form the <aimachine/> suite:

- <magiccap/> – secure, ephemeral peer-to-peer messaging.
- <infinity/> – secure peer-to-peer messaging with cryptographically verifiable session logs.
- <armoredfile/> – secure end-to-end encrypted file transfer.
- <ephcall/> – secure, ephemeral peer-to-peer voice calls.

The focus is twofold:

1. **User perspective**: how to use each application safely and effectively, including operational best practices.
2. **Technical perspective**: how the underlying cryptographic and networking mechanisms work and how they relate across the suite.

The intended audience is users with a basic understanding of cryptographic concepts (keys, encryption, MACs, key exchange) who want to understand what exactly the applications do and do not protect.

Formal legal terms (e.g. warranty disclaimer, limitation of liability, demonstration-only nature of the services) are described in the separate Terms and Conditions document located at `landing/docs/aimachine_tc.pdf`. This document concentrates on technical and operational aspects.

## 1.2 Applications in the Suite

All four applications are accessed via modern browsers and are delivered as static browser applications, ranging from single-file HTML apps to small HTML/JavaScript bundles. At a high level:

- All applications use `WebRTC` data and/or media channels, typically via `PeerJS`, for peer-to-peer connectivity.
- All applications use the browser `WebCryptoAPI` for cryptographic operations.
- The current applications use ECDH over P-256 and HKDF-based key derivation; <infinity/>, <armoredfile/>, and <ephcall/> require a user-provided shared secret for authenticated sessions, while <magiccap/> supports shared-secret authentication but can still be used without it.
- All applications are designed to be *ephemeral*: session state and contents are not stored on the server, and in many cases not stored persistently on the client either.

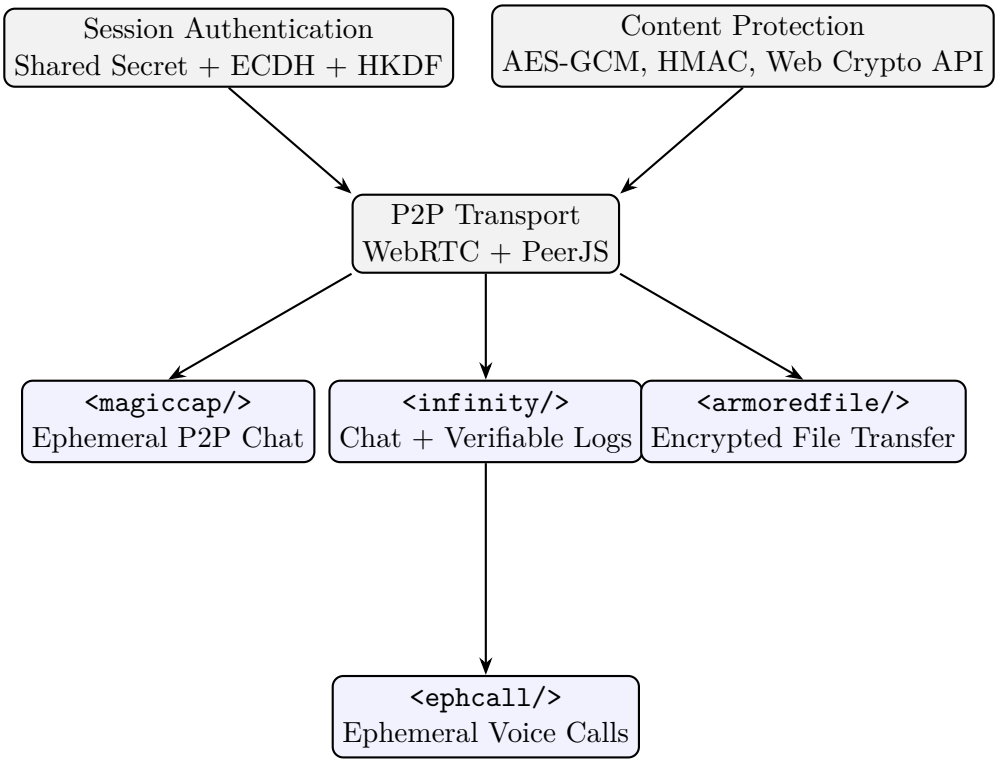


Figure 1: Overview of the <aimachine/> application suite and the shared security building blocks used across all four applications. All applications share the same core building blocks and differ in user interface and payload type (text, files, voice, logs).

# 2 Shared Architecture and Security Model

## 2.1 High-Level Architecture

Each application follows the same high-level architectural pattern:

1. The application HTML/JS bundle is loaded from the respective .aimachine.io domain.
2. The application connects to signaling infrastructure (including PeerJS and STUN servers, with TURN relays possible where configured) to discover and establish a WebRTC peer-to-peer connection.
3. A key establishment protocol is executed between peers, combining ECDH key agreement with a user-supplied shared secret where the application requires or is configured to use one.
4. After successful key confirmation, application-specific encrypted communication starts:
   - encrypted text messages (<magiccap/>, <infinity/>),
   - encrypted file chunks (<armoredfile/>),
   - voice media protected by WebRTC DTLS-SRTP, with optional insertable-stream frame encryption where supported (<ephcall/>).
5. Sessions are intentionally short-lived; application state is cleared on disconnect, reset or timeout.

The server-side components are limited to:

- static hosting of the application bundle(s),
- signaling and rendezvous services for WebRTC,
- STUN infrastructure, and TURN relay infrastructure where configured, to reach peers behind NAT/firewalls.

Servers do *not* see encrypted payload content and are not used to store chat messages, files, or voice recordings.

## 2.2 Transport and Connectivity

All four applications use WebRTC as the underlying transport technology:

- **Data channels** (SCTP over DTLS) are used for text messages, file metadata, and encrypted file chunks.
- **Media channels** (RTP over DTLS-SRTP) are used for voice in <ephcall/>.

Connection establishment uses:

- PeerJS as a convenience layer for peer ID management and signaling.
- STUN (Session Traversal Utilities for NAT) servers to discover external addresses.
- TURN (Traversal Using Relays around NAT) servers when configured and when direct P2P connectivity is not possible.

The signaling path is not assumed to be confidential or authenticated beyond transport-level protections. Confidentiality of application content is provided by application-level encryption where implemented, and peer authentication depends on the shared-secret-bound handshake when that handshake is required or used.

## 2.3 Core Cryptographic Building Blocks

Across the suite, the following cryptographic primitives are used (details vary slightly per application):

- ECDH over NIST P-256 for ephemeral key agreement.
- HKDF-SHA-256 for deriving session keys from ECDH shared secrets and user-supplied shared secrets.
- AES-GCM (typically 256-bit keys) for authenticated encryption of messages, file chunks, and exported logs.
- HMAC-SHA-256 for challenge/response key confirmation in the newer protocols and, in some places, additional integrity checks.
- PBKDF2 for password-based key derivation in <infinity/> when encrypting exported logs.
- Ed25519 for signing log events in <infinity/>.
- SHA-256 for hashing events, building hash chains, and Merkle trees; browser CSPRNG output is used for generated handles and share codes.

Depending on the application, per-session keys are derived as needed:

- An *encryption key* for AES-GCM content encryption.
- An *auth key* for HMAC-based mutual authentication and key confirmation.
- Additional *wrap keys* for transporting file keys (<armoredfile/>) or other secondary materials.

## 2.4 Shared-Secret-Bound Session Authentication

The central authentication idea is that peers who want an authenticated session must know a common *shared secret*. This secret is not transmitted; instead it is used as an input to key derivation and/or challenge/response messages.

At a high level, the newer handshakes across the applications follow the pattern shown in Figure 2. The exact transcript differs by application. The common steps are:

1. Each peer generates an ephemeral ECDH key pair in the browser.
2. Peers exchange public keys (and auxiliary nonces/challenges) over the signaling channel.
3. Each side computes the ECDH shared secret and combines it with the user-supplied shared secret via HKDF to derive session keys.
4. A mutual challenge/response sequence proves that both sides know the shared secret and the derived keys.
5. Only after successful symmetric key confirmation is the session marked as authenticated and subsequent messages/traffic accepted.

If the user omits a shared secret where the application allows this, the handshake degenerates to unauthenticated ECDH: the connection is still encrypted, but a man-in-the-middle (MITM) attacker who can intercept signaling and connection setup could potentially impersonate peers. Using a strong shared secret is therefore required in some apps and strongly recommended everywhere.

## 2.5 Ephemerality and Data Minimization

The suite is designed with strong ephemerality:

- Servers do not store chat histories, files, or voice data.
- <magiccap/> and <ephcall/> do not provide persistence for content beyond what exists in browser memory during the session.
- <magiccap/> removes messages from the visible UI after roughly 60 seconds.
- <armoredfile/> discards keys and state once a transfer is completed, reset, or timed out; share codes expire after inactivity.
- <infinity/> persists content only in exported logs under the users control. The current encrypted export uses the configured shared secret as the export passphrase; the application itself does not keep server-side copies.

Some limited browser-side persistence may exist for usability (e.g. remembering user handles in `sessionStorage`), but no sensitive content such as full chat histories or decrypted files is intentionally stored persistently by the applications.

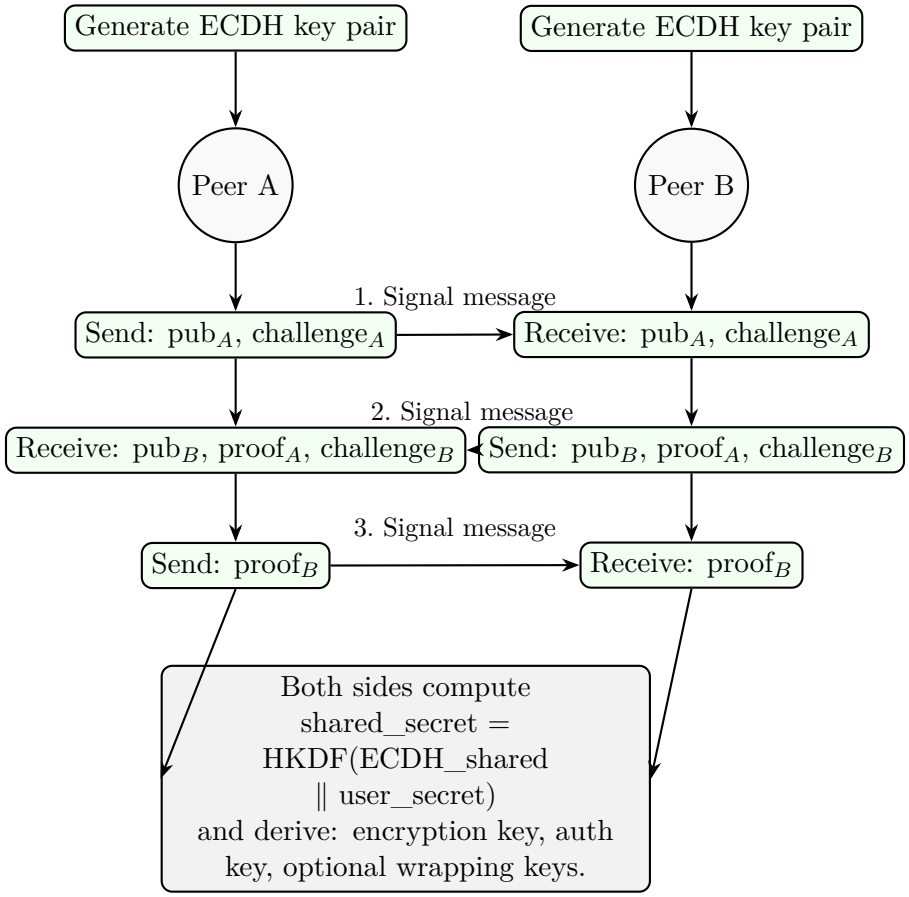


Figure 2: Shared-secret-bound ECDH handshake pattern used (with minor variations) in all <aimachine/> applications.

### 2.6 Threat Model and Non-Goals

The main security goals are:

- Confidentiality and integrity of content between peers (messages, files, voice).
- Mutual authentication of peers based on a shared secret and key-confirmation handshake where configured or required.
- Detectability of tampering in exported logs in <infinity/>.
- Minimized reliance on servers and centralized storage.

Important non-goals and limitations:

- **Endpoint security is assumed.** If a device is compromised (malware, keyloggers, remote-access Trojans), an attacker can bypass all application-level protections.
- **Network metadata is only partially protected.** IP addresses, timing patterns, and approximate traffic volumes may still be visible to network observers and relay infrastructure. A VPN can help but does not completely eliminate metadata.
- **No long-term identity infrastructure.** The applications do not implement complex public-key infrastructures or long-term identity bindings; identity assurance is primarily based on shared secrets and out-of-band verification.

- **Demonstration nature.** The suite is built as a demonstration and experimental platform, not as a formally verified system with strong security guarantees under all operating conditions. This is reflected in the Terms and Conditions.

# 3 General Usage and Operational Guidance

## 3.1 Environment and Browser Requirements

For all applications:

- Use a modern, up-to-date browser with `WebRTC` and `WebCryptoAPI` support (current versions of Chromium-based browsers and Firefox are typically suitable).
- Ensure JavaScript is enabled and no browser extensions interfere with WebRTC or cryptographic APIs.
- Use a reasonably reliable network connection; high packet loss or unstable connections can cause call drops or file transfer failures.

For `<ephcall/>`, insertable-stream E2EE depends on browser support. If your browser does not support insertable streams, only the transport-level DTLS-SRTP encryption layer will be active; the user interface indicates this.

## 3.2 Verifying Application Authenticity

To reduce supply-chain and phishing risks:

- Always access the applications via their official URLs:
  - `<magiccap/>`: https://magiccap.aimachine.io
  - `<infinity/>`: https://infinity.aimachine.io
  - `<armoredfile/>`: https://armoredfile.aimachine.io
  - `<ephcall/>`: https://ephcall.aimachine.io
- Verify that the browser shows a valid HTTPS connection with the expected domain.
- Avoid following random links to similar-looking domains; bookmark the official URLs instead.

The applications ship their PeerJS client code locally or inline and do not execute remote CDN fallback code for that dependency. Users should still verify the official HTTPS domains and the signed release manifests.

## 3.3 Choosing and Managing User Handles

Several applications use user handles, peer IDs, or share codes:

- `<magiccap/>`, `<infinity/>`, and `<ephcall/>` can generate pseudonymous handles or IDs from browser CSPRNG output.
- `<armoredfile/>` generates a temporary share code for each sender session.
- Users can also specify custom handles, which may be more memorable but less privacy-preserving.

Recommendations:

- Prefer randomly generated handles over personal identifiers.
- When sharing handles with peers, always share the *full* handle/ID, not a shortened display version.
- Treat handles as connection identifiers only; they do not by themselves authenticate identity.

### 3.4 Shared Secrets: Selection and Exchange

The security of all applications heavily depends on the quality and secrecy of the user-chosen shared secrets:

- Use long, high-entropy secrets (e.g. random passwords generated by a password manager).
- Avoid short, guessable secrets or reused passwords from other services.
- Use a fresh shared secret for each independent session or group of sessions that should be isolated.

Exchange shared secrets only over trusted, pre-existing secure channels:

- In person (face-to-face exchange).
- Through existing end-to-end encrypted channels that you trust.

Avoid sending shared secrets over unencrypted email, SMS, or other insecure channels. Anyone learning the shared secret can join or impersonate participants in sessions.

### 3.5 Out-of-Band Verification and Peer IDs

Because peers are identified by handles/IDs and the shared secret, robust authentication depends on:

- verifying that you have the correct handle/ID for the counterparty, and
- verifying that both sides have configured the same shared secret.

Recommended practices:

- Use a separate trusted channel to exchange both the full peer IDs and the shared secret.
- If possible, read IDs aloud or compare them character by character to detect typos or MITM attacks.
- In repeated collaborations, establish stable operational procedures for exchanging and rotating secrets.

### 3.6 Network-Level Privacy and VPN Use

WebRTC exposes your IP address (and possibly some network topology information) to the peer and potentially to relay infrastructure. To improve network-level privacy:

- Consider using a reputable VPN to obfuscate your real IP address.
- Be aware that VPNs add latency, which may impact call quality or file transfer speed.
- Understand that even with VPNs, timing and volume metadata remain observable to some extent.

### 3.7 Endpoint Hygiene

No end-to-end secure protocol can protect against a compromised endpoint. Maintain:

- up-to-date operating systems and browsers,
- regular security updates,
- careful control over installed software and browser extensions,
- secure physical access to devices.

## 4 <magiccap/> – Ephemeral P2P Messaging

### 4.1 Purpose and Core Features

<magiccap/> is a browser-based peer-to-peer chat application focused on:

- end-to-end encrypted messaging between participants,
- minimal reliance on central servers,
- ephemeral message display (messages disappear from the UI after roughly one minute),
- pseudonymous handles and minimal stored metadata.

### 4.2 User Workflow

**Logging in and choosing a handle.**

1. Navigate to https://magiccap.aimachine.io in a modern browser.
2. Choose a user handle:
   - Recommended: click `Generate Random Handle` to create a long random identifier; an abbreviated version is shown but the full ID is used internally.
   - Alternatively: type a custom handle; for privacy, avoid personally identifying names.
3. In the `> shared_secret` field, enter a strong shared secret that all intended peers will use for this session. The field is technically optional in the current app, but unauthenticated sessions should be avoided.
4. Click `ENTER` to log in and register your presence with the signaling infrastructure.

**Connecting to peers.**

1. After logging in, copy your full connection ID using the clipboard icon in the status bar.
2. Share your full ID and the shared secret with peers via a trusted out-of-band channel.
3. To connect to a peer, paste their full ID into the `> target_full_id_` field and click `CONNECT`.
4. The application performs the shared-secret-bound handshake; if secrets match, the link is authenticated and encrypted.

**Messaging.**

1. Once at least one authenticated encrypted link is active, the message input field is enabled.
2. Type a message and press Enter or click `SEND`.
3. Messages appear briefly in the chat view and are then removed after approximately 60 seconds.

**Ending a session.**

- Close the tab or window, or use application controls to disconnect.
- An inactivity timer may automatically log you out after a period with no activity.

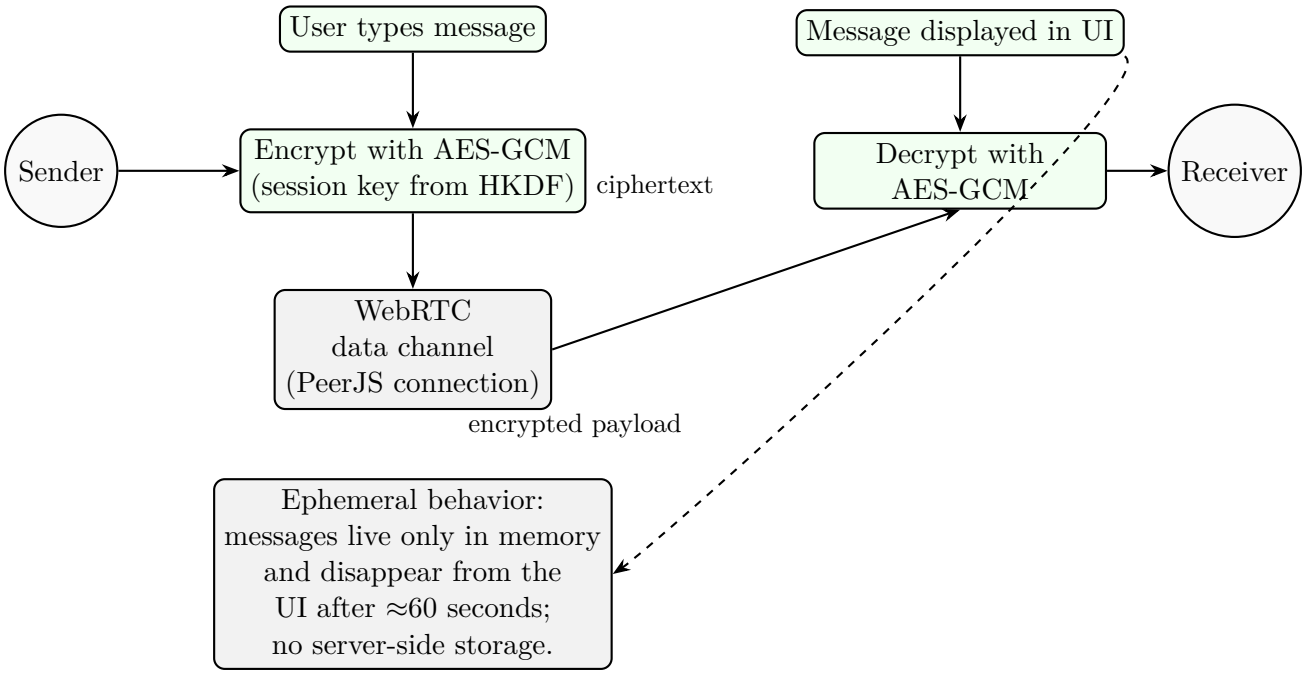


Figure 3: High-level message flow in `<magiccap/>`: user messages are encrypted locally with a session key derived from the shared-secret-bound handshake, then sent as ciphertext (with IV and AAD) over a WebRTC data channel and decrypted on the peer side, where they are displayed only ephemerally in the UI.

## 4.3 Security Internals

`<magiccap/>` uses a simpler variant of the shared handshake pattern described in Section 2.4:

- ECDH (P-256) per peer link.
- HKDF-SHA-256 over the ECDH shared secret and the user-provided shared secret.
- Derived session material:
  - an AES-GCM key for message encryption,
  - in the current implementation, shared-secret proofing based on SHA-256 over the secret and nonce values.

- Per-message 12-byte IVs with AES-GCM authenticated ciphertext. The current <magiccap/> message format does not bind sender/receiver IDs or sequence numbers as AES-GCM AAD.

Messages are not sent in plaintext over the data channel once an encrypted key has been established. If no shared secret is configured, the app can still establish an encrypted but unauthenticated ECDH session; users should treat that mode as vulnerable to MITM attacks.

### 4.4 Best Practices and Common Pitfalls

Recommended practices specific to <magiccap/>:

- Always use the shared-secret functionality; operating without a shared secret exposes you to MITM attacks.
- Use the random handle generator instead of custom short names; this reduces the chance of linking identities across contexts.
- Remember that message disappearance in the UI does not prevent screenshots or manual copying by peers; operational trust between participants remains essential.
- For longer or more structured conversations that may require later verification, consider using <infinity/> instead, which supports verifiable session logs.

## 5 <infinity/> – Messaging with Verifiable Session Logs

### 5.1 Purpose and Core Features

<infinity/> is a peer-to-peer chat application similar to <magiccap/> but adds:

- encrypted and signed session log export,
- support for both a current encrypted-and-signed export format and a legacy plaintext format,
- a verification tool that can check exported logs for internal integrity and signatures.

Its core transport model is similar to <magiccap/>, while its current authenticated handshake and message metadata binding are stricter.

### 5.2 User Workflow

**Starting a session.** The steps mirror <magiccap/>:

1. Open https://infinity.aimachine.io in a modern browser.
2. Choose or generate a handle.
3. Enter a shared secret.
4. Obtain your active peer ID and connect to a target peer using their full ID.

**Messaging.** Messaging works like <magiccap/>: once an authenticated link is established, messages are encrypted with AES-GCM and transmitted over the WebRTC data channel.

**Exporting logs.**

1. At any time, you may choose to export the current session log.
2. The current export feature requires the shared-secret field to be populated; that value is used with PBKDF2 to derive an AES-GCM key for encrypting the log bundle.
3. The resulting file is stored locally by the browser (e.g. downloaded to your file system).

**Verifying logs.**

1. Open the verification tool within `<infinity/>`.
2. Provide the exported log file.
3. For encrypted signed logs, provide the same shared secret/export passphrase used at export time.
4. The tool decrypts (if necessary) and verifies:
   - the hash chain integrity,
   - the event signatures,
   - the Merkle root over all events.
5. The tool indicates success or failure; on failure, it should provide information about which checks failed.

## 5.3 Session Log Model

Figure 4 illustrates the export pipeline. The key ideas:

- Each event (message, connection change, etc.) is represented as a structured object.
- A cryptographic hash (SHA-256) is computed over a canonicalized representation of each event.
- Events are chained via a `prevEventHash` field, forming a linear hash chain that makes deletions or reorderings detectable.
- Each event hash is signed using Ed25519 with a signer key generated for the session.
- A Merkle tree is constructed over all event hashes, and the root acts as a compact commitment to the entire log.
- The log payload (events, signatures, Merkle root, and signer metadata) is encrypted with AES-GCM using a key derived from the shared secret/export passphrase via PBKDF2.

## 5.4 Interpreting Verification Results

When verifying an exported log:

- Successful decryption plus successful validation of the hash chain and Merkle root indicates that the encrypted payload is internally consistent and has not changed without detection.
- Successful signature verification indicates that all events match the embedded signer public key; it does not by itself tell you who the signer is. Identity still depends on how you obtained or trust the signer key.
- For legacy unsigned exports, verification can only check hash-chain and Merkle integrity, not the identity of the signer.

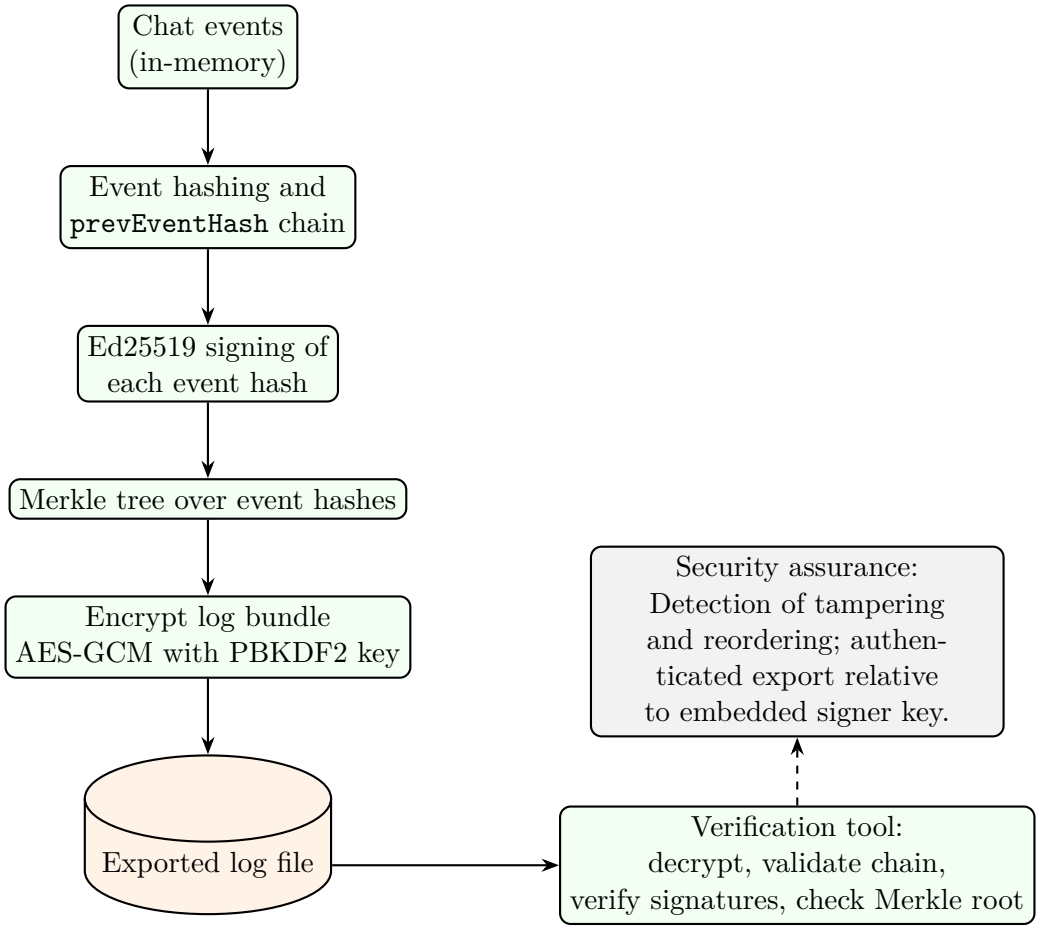


Figure 4: Export and verification pipeline in <infinity/>: chat events form a hash chain, are signed and aggregated into a Merkle tree, then encrypted with AES-GCM using a PBKDF2-derived key from a user-supplied password for export; the verification tool later decrypts the bundle and validates the chain, signatures, and Merkle root.

### 5.5 Best Practices

For <infinity/>:

- Use strong shared secrets for the session, as with <magiccap/>.
- Use a strong shared secret/export passphrase; do not reuse it for unrelated sessions or services.
- Store exported logs securely (e.g. encrypted disk, password manager file storage).
- Define clear procedures for how and when logs are exported and how they are verified, especially in environments where logs might be used as evidence or audit trails.

## 6 <armoredfile/> – Secure File Transfer

### 6.1 Purpose and Core Features

<armoredfile/> focuses on end-to-end encrypted file transfer between two peers with authenticated session setup:

- Files are encrypted in the browser before being sent.

- A per-file AES-GCM key is generated randomly for each transfer.
- The file is chunked and encrypted chunk-by-chunk; encrypted chunks are streamed over the WebRTC data channel.
- The per-file key is wrapped (encrypted) using session keys derived from the shared handshake.
- Share codes identify transfers and expire after inactivity.

## 6.2 User Workflow

**Sender.**

1. Open https://armoredfile.aimachine.io in a modern browser.
2. Under "1. SELECT FILE TO SHARE", choose the file to send (subject to configured size and chunk limits).
3. Under "3. ENTER SHARED SECRET", enter a strong secret agreed with the recipient.
4. Click `SHARE SELECTED FILE`.
5. The application generates a share code; send this share code to the recipient via a trusted out-of-band channel.

**Recipient.**

1. Open https://armoredfile.aimachine.io.
2. Paste the received share code into "2. ENTER SHARE CODE (IF RECEIVING)".
3. Enter the same shared secret as the sender.
4. Click `RECEIVE WITH CODE`.
5. After the authenticated handshake completes and metadata is validated, encrypted file chunks are streamed and a download is triggered once transfer and decryption succeed.

## 6.3 Security Internals

The file transfer pipeline is depicted in Figure 5:

- The sender generates a random AES-GCM file key for each transfer.
- The file is logically chunked; each chunk is encrypted with AES-GCM using:
  - the per-file key, and
  - an IV derived deterministically from a 12-byte base IV and the chunk index.
- The file key is wrapped (encrypted) using a wrap key derived from the shared-secret-bound ECDH handshake.
- Mutual authentication is performed with HMAC challenge/response proofs derived from the session auth key before file delivery begins.
- File metadata and chunk counts are checked by the receiver, while encrypted file-key transport and chunks rely on AES-GCM authentication for tamper detection.

- The recipient unwraps the file key, reassembles and decrypts chunks, and reconstructs the original file.

Integrity and abuse guards:

- Upper bounds on file size and chunk count.
- Checks for chunk overruns and mismatches between advertised and actual chunk counts.
- Size consistency checks after decryption.

### 6.4 Ephemerality and Operational Considerations

- Share codes expire after a period of inactivity (e.g. several minutes).
- Session keys and in-memory state are cleared on completion, timeout, or reset.
- On the receiver side, the reconstructed file is assembled in memory before being passed to the browser for download; extremely large files may stress browser memory despite chunked transfer.

### 6.5 Best Practices

For `<armoredfile/>`:

- Keep transferred file sizes within reasonable limits for the users devices and browsers.
- Use strong, unique shared secrets per transfer or per collaboration context.
- Transmit share codes and secrets via secure out-of-band channels; avoid sending both in the same channel if possible.
- After receiving sensitive files, store them securely on disk (e.g. encrypted volumes).

## 7 <ephcall/> – Secure Ephemeral Voice Calls

### 7.1 Purpose and Core Features

`<ephcall/>` provides ephemeral, browser-based peer-to-peer voice calls:

- authenticated session establishment using a shared-secret-bound ECDH handshake,
- voice transport over DTLS-SRTP (WebRTC media channels),
- optional application-level insertable-stream E2EE where supported,
- minimal state retention and no call recording by the application.

### 7.2 User Workflow

**Getting started.**

1. Open https://ephcall.aimachine.io in a modern browser that supports `WebRTC`.
2. Allow microphone access when prompted.
3. Generate a random handle (recommended) or enter a custom handle.
4. Enter a shared secret and click `Connect + Authenticate`.

**Placing a call.**

1. Exchange full peer IDs and shared secret with the intended counterpart via a trusted out-of-band channel.
2. After the handshake completes and the UI indicates an authenticated secure session, initiate a call to the peers ID.
3. Depending on configuration and browser support, the application negotiates DTLS-SRTP transport and, optionally, insertable-stream E2EE.

**Call teardown.**

- Hang up using the in-app controls, or close the tab.
- Session keys and runtime state are cleared on hangup/reset/disconnect.

## 7.3 Security Layers

Figure 6 summarizes the layered protections:

- **Application-layer E2EE (optional).** When insertable streams are available and negotiated, audio frames are encrypted/decrypted in JavaScript with keys derived from the authenticated handshake, before/after the WebRTC stack.
- **DTLS-SRTP transport encryption.** WebRTC media stacks always use DTLS-SRTP, providing transport encryption between WebRTC endpoints, including paths that may traverse TURN relays.
- **Network and relay layer.** STUN and, where configured, TURN infrastructure is used to traverse NATs and firewalls; this layer sees encrypted RTP packets but not plaintext audio.

When insertable-stream E2EE is unavailable or disabled:

- Only DTLS-SRTP is active; the UI should indicate a "TRANSPORT ONLY" mode.
- The call remains encrypted at the transport layer but not additionally protected by application-layer E2EE.

## 7.4 Best Practices

For `<ephcall/>`:

- Prefer browsers that support insertable-stream E2EE and enable it when higher assurance is needed.
- Always use strong shared secrets to authenticate sessions and protect against MITM attacks.
- Use random handles rather than identifiable names.
- Use a VPN when feasible to reduce exposure of IP addresses to signaling and relay infrastructure.
- Remember that the application does not record calls, but peers can still record audio locally using other tools.

# 8 Summary and References

The <aimachine/> suite demonstrates a coherent approach to secure communication across text chat, verifiable logs, file transfer, and voice:

- A common architecture based on browser-native WebRTC and the WebCryptoAPI.
- Consistent use of ECDH, HKDF, AES-GCM, and HMAC to build authenticated, encrypted sessions bound to user-chosen shared secrets.
- Ephemeral design and data minimization as guiding principles.
- An explicit, cryptographically verifiable log model in <infinity/> for scenarios that require auditability.

At the same time, all applications depend critically on:

- correct and secure operational procedures (secret exchange, endpoint hygiene, URL verification),
- realistic expectations about network metadata and endpoint risks,
- an understanding that these applications are experimental and demonstration-focused, as described in the Terms and Conditions.

For further reading on the underlying primitives and standards:

- NIST FIPS 197 – Advanced Encryption Standard (AES).
- NIST SP 800-38D – Recommendation for Galois/Counter Mode (GCM).
- NIST SP 800-56A Rev. 3 – Pair-Wise Key Establishment Schemes (ECDH).
- RFC 5869 – HMAC-based Extract-and-Expand Key Derivation Function (HKDF).
- RFC 8018 – PKCS#5 (PBKDF2).
- Web Crypto API documentation.
- WebRTC and PeerJS documentation.

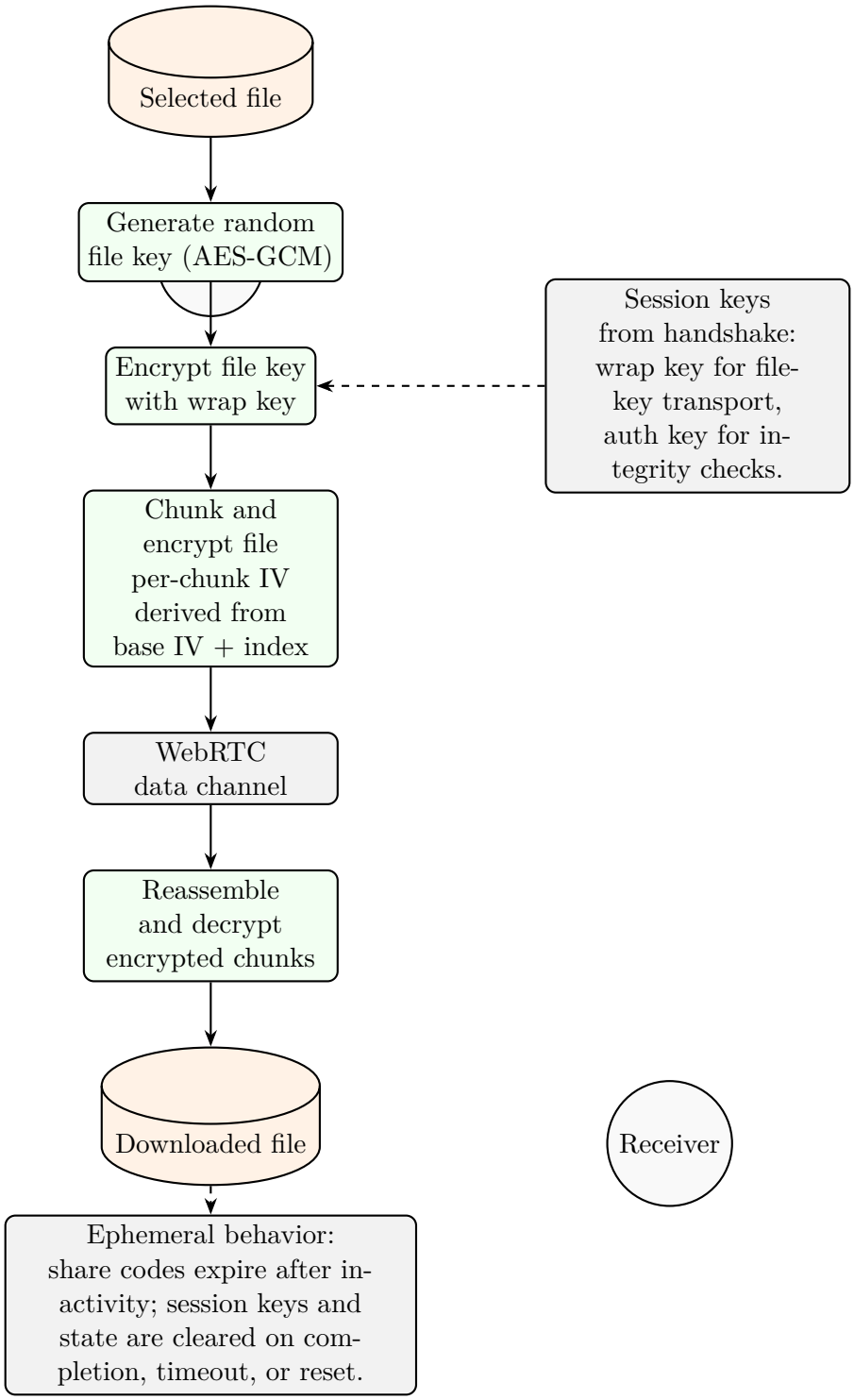


Figure 5: Encrypted file transfer pipeline in <armoredfile/>: a random file key is generated and wrapped using session keys from the authenticated handshake, and the file is encrypted and streamed chunk by chunk via WebRTC.

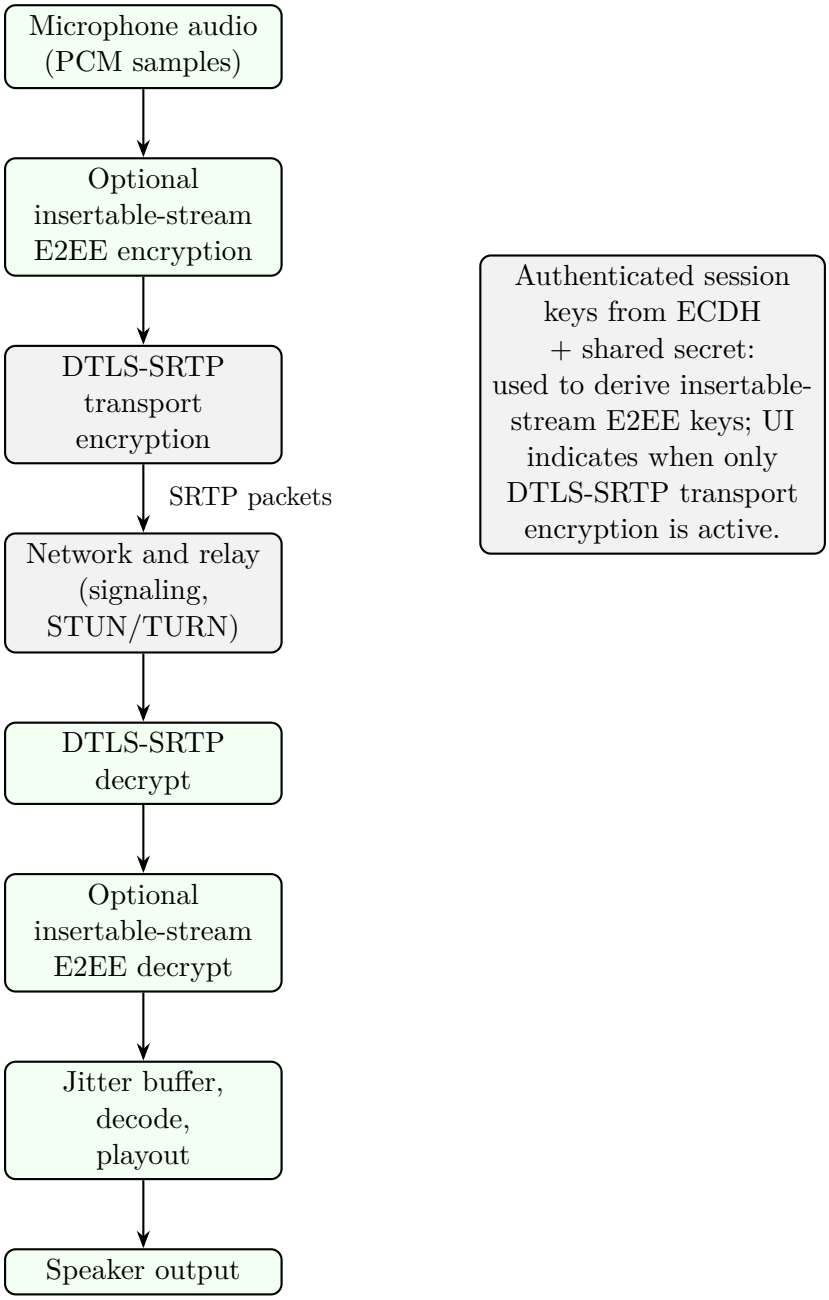


Figure 6: Security layers in <ephcall/>: application-level insertable-stream end-to-end encryption (when negotiated) sits on top of DTLS-SRTP transport encryption, with keys derived from an authenticated shared-secret-bound ECDH handshake.