WebRTC Conferencing Framework with End-to-End Encryption: Architecture, Deployment, and Operations

Sergio Emiliano Mancinas Fontalvo
Advanced Multimedia Communication Lab
Technische Hochschule Köln / University of Applied Sciences
Supervisor: Prof. Dr.-Ing. Andreas Grebe
Email: sergio_emiliano.mancinas_fontalvo@smail.th-koeln.de

***Abstract*—This report describes a self hosted video conferencing system I built using WebRTC that keeps all communication encrypted from end to end, even when using a server to route the media. The system uses LiveKit as the media server, Coturn to help users connect through firewalls and NAT, and a Next.js web application for the frontend. I also set up monitoring with Prometheus and Grafana to track call quality. Everything runs in Docker containers on a Debian virtual machine with Nginx handling HTTPS. This report covers how the system works, how I deployed it, and what I learned about keeping it secure.**

***Index Terms*—WebRTC, end-to-end encryption, Selective Forwarding Unit, LiveKit, Docker, TURN, DTLS-SRTP**

I. Introduction

Video conferencing is now a core part of everyday communication, from professional meetings to online education and personal conversations. Modern web browsers rely on WebRTC to enable real-time audio and video communication without additional plugins [1]. However, supporting multi-party calls typically requires a centralized server to route media streams, which introduces privacy concerns because this server may have access to the media content.

This report presents a conferencing system I built that solves this problem by keeping the video and audio encrypted so that even the server cannot see or hear anything. The main goal was to make a system where users can have private conversations without trusting the server [6]. I used Docker containers to package everything together so it can be easily deployed on any Linux server.

The main things I accomplished with this project are: (1) getting end to end encryption to work with a media server using the insertable streams API, (2) building a monitoring system to track call quality metrics like latency and packet loss, and (3) creating a deployment setup that can be reproduced using Docker Compose.

II. Background and Related Work

A. WebRTC Security Architecture

WebRTC mandates that all audio and video traffic be encrypted during transmission using DTLS-SRTP [2]. During session establishment, the browser negotiates a secure transport channel, which is then used to protect all media packets exchanged between endpoints. The WebRTC specification requires support for modern cipher suites, including AES-128-GCM [3].

This transport-layer encryption ensures confidentiality and integrity of media while in transit across the network. However, in conferencing architectures that rely on a media server to forward streams, the server must terminate the DTLS-SRTP sessions in order to route the media. As a consequence, the server gains access to the decrypted media content.

B. End to End Encryption with Insertable Streams

To fix the problem mentioned above, there is a newer browser API called Insertable Streams that lets allows encrypt the video frames before they even get to the normal WebRTC encryption layer [6]. This way, the server only sees encrypted blobs and cannot actually watch or listen to the call. The IETF is working on a standard format for this called SFrame [7].

C. Selective Forwarding Units

A Selective Forwarding Unit or SFU is a type of video server that just forwards packets without decoding them, unlike older systems that would mix all the video together [8]. LiveKit is an open source SFU written in Go that I chose for this project because it already has E2EE support built in [9].

D. NAT Traversal

One of the trickiest parts of WebRTC is getting connections to work when users are behind firewalls or home routers. The ICE protocol coordinates STUN and TURN to punch through NAT [4]. I used Coturn which is a well known open source TURN server [10].

III. System Architecture

A. Media Flow Architecture

Figure 1 shows how the different parts of the system connect together. When someone opens the website, their browser connects to Nginx which handles all the HTTPS stuff and then passes requests to the right backend service.

The important thing about E2EE is that LiveKit only ever sees encrypted video frames. It can look at the packet headers

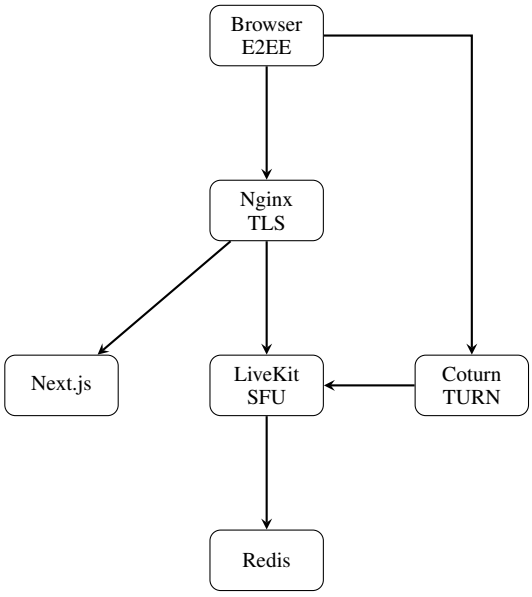

Fig. 1. System architecture showing how browsers connect through Nginx to reach the different services.

to figure out where to send things, but it cannot actually see or hear the call content [6].

B. Monitoring Pipeline

To evaluate call performance and operational stability, a monitoring pipeline was implemented using Prometheus and Grafana [13]. Client-side quality metrics are collected in the browser via the WebRTC `getStats` API, providing measurements such as round-trip time, jitter, and packet loss [5]. These metrics are transmitted to a FastAPI-based backend service and exposed in Prometheus format for time-series storage.

In addition to performance metrics, logs from all Docker containers are aggregated using Promtail and stored in Loki. Grafana is then used to visualize both metrics and logs through unified dashboards, enabling continuous observation of call quality and system behavior.

C. Network Security Boundaries

Table I shows which services are exposed to the internet and which ones are only accessible internally. I tried to minimize what is publicly accessible.

TABLE I
NETWORK PORT EXPOSURE

Service	Ports	Binding
Nginx	80, 443/tcp	Public
Coturn	3478, 5349/tcp	Public
Coturn relay	49160-49200/udp	Public
LiveKit media	7881, 50200-50300/udp	Public
Application	3001/tcp	127.0.0.1
API	8000/tcp	127.0.0.1
Prometheus	9090/tcp	127.0.0.1
Grafana	3000/tcp	127.0.0.1

IV. Implementation Details

A. LiveKit Room Configuration and E2EE

When a user joins a room, the client code needs to set up the LiveKit connection with E2EE enabled. Listing 1 shows how this works. The key thing is creating an ExternalE2EEKeyProvider and giving it the passphrase that all participants need to share.

Listing 1. Setting up a LiveKit room with encryption

```
const roomOptions: RoomOptions = {
  adaptiveStream: true,
  dynacast: true,
  publishDefaults: { simulcast: true, videoCodec: 'vp8' },
};
if (e2ee.enabled && e2ee.passphrase) {
  keyProvider = new ExternalE2EEKeyProvider();
  roomOptions.e2ee = {
    keyProvider,
    worker: new Worker('/livekit-e2ee-worker.js'),
  };
  await keyProvider.setKey(e2ee.passphrase);
}
```

B. Token Authentication

Before a user can join a room, they need to get a token from the server. The server checks if they know the right passphrase by comparing hashes. I made sure to never store or log the actual passphrase, only a hash of it.

Listing 2. How the server checks the passphrase and creates tokens

```
if (e2ee) {
  const result = await validateRoomKey(room, keyHash ?? "")
      ;
  if (!result.ok) {
    return NextResponse.json(
      { error: result.error }, { status: result.status });
  }
}
const at = new AccessToken(apiKey, apiSecret, { identity })
    ;
at.addGrant({ roomJoin: true, room,
              canPublish: true, canSubscribe: true });
const token = await at.toJwt();
```

C. API Surface

Table II lists the main API endpoints that the web application uses.

TABLE II
PRIMARY API ENDPOINTS

Endpoint	Purpose
`/api/token`	Issue LiveKit access tokens
`/api/validate-room`	Validate E2EE passphrase hash
`/api/ingest`	Receive browser QoE metrics
`/api/metrics`	Prometheus format metrics
`/api/health`	Health status check

D. AI-assisted Development (Codex)

Parts of the frontend and backend implementation were developed with assistance from the OpenAI Codex coding agent (latest version at the time of writing). Codex was used to

draft boilerplate code, suggest refactorings, and generate initial versions of configuration and API handler code. All generated code was reviewed, adapted, and tested by the author before deployment.

V. Deployment Configuration

A. Container Orchestration

I used Docker Compose to manage all the services because it makes it easy to start everything with one command [11]. The internal services like the web app and Prometheus only listen on localhost (127.0.0.1) so they cannot be accessed from outside. LiveKit and Coturn need to be reachable from the internet for the media to flow.

B. Reverse Proxy and TLS

Nginx sits in front of everything and handles HTTPS [12]. It also upgrades connections to WebSocket when needed for the LiveKit signaling. I used acme.sh to automatically get and renew TLS certificates from Let's Encrypt.

C. Host Specifications

Table III shows the specs of the virtual machine I deployed everything on.

TABLE III
Virtual Machine Specifications

Component	Specification
Operating System	Debian GNU/Linux 13 (trixie)
CPU	8 vCPU, Intel Xeon E312xx (KVM)
Memory	31 GiB RAM, 12 GiB swap
Storage	239 GB root disk
Public Domain	sergio-webrtc-app.duckdns.org

VI. Security Incident and Hardening

During the early stages of deployment, a misconfiguration was introduced at the network level, affecting both the DMZ firewall rules and the Docker Compose configuration. Several internal services were unintentionally exposed by binding to all network interfaces (0.0.0.0) instead of being restricted to localhost. In addition, the Coturn TURN server was initially configured as an open relay without proper access controls.

As a result of these issues, the server was compromised by an unauthorized actor who deployed cryptocurrency mining software. The incident highlighted the risks of improper service exposure and insufficient network hardening during initial deployment phases.

A. Mitigations Applied

After detecting the intrusion, the system configuration was corrected and the server was comprehensively hardened.

At the container level, all internal services defined in Docker Compose were reconfigured to bind exclusively to the loopback interface (127.0.0.1) instead of all network interfaces (0.0.0.0). The Coturn TURN server was also corrected to bind only to the public IP address, and all settings that allowed it to operate as an open relay were removed.

At the network level, a host-based firewall was configured using UFW to strictly limit inbound traffic. Only the required ports were permitted: SSH (22/tcp), HTTP and HTTPS (80 and 443/tcp), TURN signaling ports (3478 and 5349/tcp), and the UDP port ranges required for LiveKit media transport.

As a preventive measure, `rkhunter` and `chkrootkit` had already been installed on the host prior to deploying Docker and any application services. Following the incident, both tools were executed again in monitoring mode to assess system integrity. No evidence of persistent malware or backdoors was detected. An external port scan was also performed to verify that internal services were no longer reachable from outside the host.

VII. Security and Cryptography

A. Transport vs. End-to-End Encryption

WebRTC mandates encryption of all media traffic using DTLS-SRTP [2], which ensures confidentiality and integrity while data is in transit across the network. This transport-layer encryption, however, only protects media between endpoints and the immediate transport hops.

In typical server-assisted conferencing architectures, such as those based on SFUs, the media server must terminate DTLS-SRTP in order to process, route, or adapt media streams. As a result, the server gains access to the decrypted media content.

E2EE addresses this limitation by introducing an additional encryption layer applied at the application level prior to transport encryption. This second layer ensures that media content remains confidential not only on the network but also at intermediate servers, which therefore cannot decrypt or access the underlying audio or video data.

B. E2EE Implementation

Figure 2 shows how the encryption works step by step. The browser encrypts the video before it even gets to the normal WebRTC encryption.

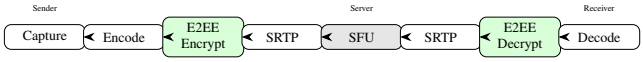

Fig. 2. End-to-end encryption workflow: media is encrypted at the sender before transport, ensuring that intermediate servers can only forward opaque, encrypted payloads without access to plaintext content.

As shown below:

1) Media is captured at the sender and encoded using VP8 for video or Opus for audio
2) An application-layer E2EE mechanism encrypts the encoded frames using AES-GCM
3) Standard SRTP encryption is subsequently applied for transport-level protection
4) The SFU forwards the encrypted media without access to the plaintext content
5) The receiving endpoint removes both encryption layers and decodes the media

The server never gets to see the actual video or audio content.

C. Passphrase Handling

One thing I was careful about is making sure the server never sees the actual passphrase. The browser hashes it first using SHA-256:

Listing 3. Hashing the passphrase in the browser

```
// Client: hash passphrase with SHA-256
const data = new TextEncoder().encode(passphrase);
const hash = await crypto.subtle.digest("SHA-256", data);
const keyHash = Array.from(new Uint8Array(hash))
  .map(b => b.toString(16).padStart(2, "0")).join("");
// Only keyHash sent to server
```

The server then checks if this hash matches what it expects. I used a timing safe comparison to prevent timing attacks:

Listing 4. Checking the hash on the server side

```
const ok = crypto.timingSafeEqual(
  Buffer.from(configuredHash, "hex"),
  Buffer.from(hashFromClient, "hex"));
if (!ok) return { ok: false, status: 401 };
```

D. Integrity and Error Handling

The E2EE encryption uses AES-GCM which also checks that nobody tampered with the data. If someone joins with the wrong passphrase, the decryption will fail and after a few errors the client automatically disconnects. This prevents awkward situations where someone is in the call but cannot actually see or hear anything.

E. Operational Controls

I added a few other security measures: there is rate limiting on the token endpoint so someone cannot spam it, the metrics endpoint can optionally require a password, and the firewall blocks access to TURN metrics from outside the server.

VIII. User Interface

The web application is built with Next.js and provides a simple interface for joining E2EE protected video rooms. The system supports two types of users: administrators who can create and manage rooms, and guests who join existing rooms with a passphrase.

A. Admin Console

Administrators access the system through a separate login flow. The admin login page requires credentials that are validated against the members database. Once authenticated, admins get access to management pages that regular guests cannot see.

The members page at `/members` lets admins manage all users in the system. They can search and filter members, change roles between admin and guest, edit display names and emails, reset passwords, or delete accounts. New users can be added through the registration form which calls `/api/register`.

The LiveKit console at `/console/livekit` provides control over rooms, ingress, and egress. It is protected by session cookies and redirects unauthorized users to the login

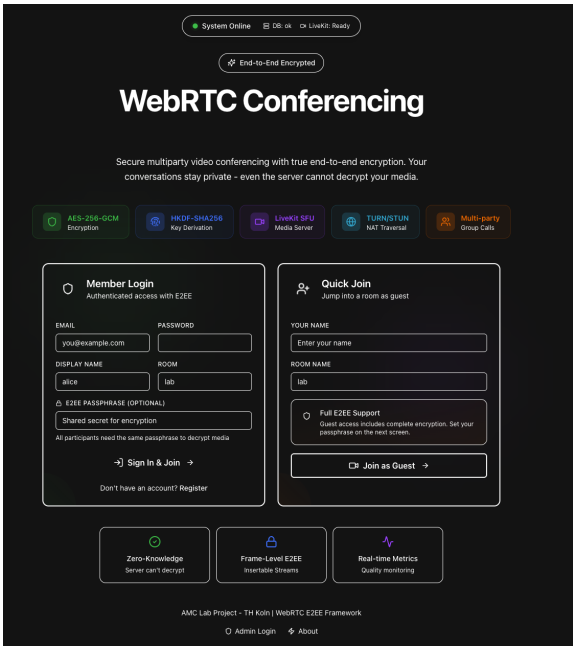

Fig. 3. The main login page where users choose to enter as admin or guest.

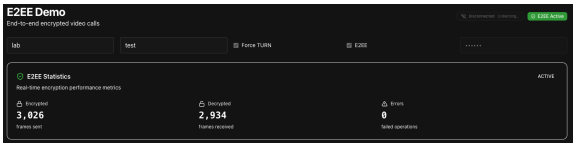

Fig. 4. The join page where users enter their name and passphrase.

Fig. 5. The main room view showing an active E2EE session.

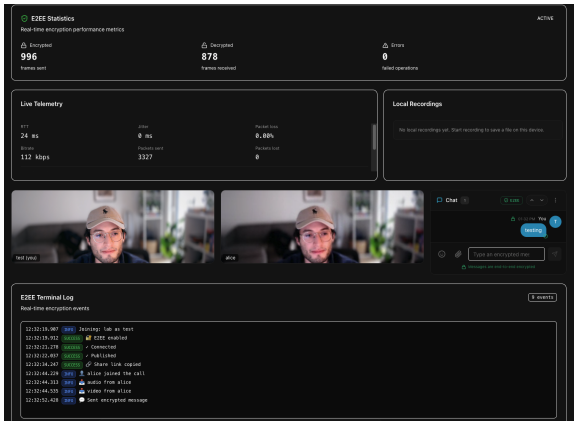

Fig. 6. Grafana showing real time metrics like RTT, jitter, and packet loss.

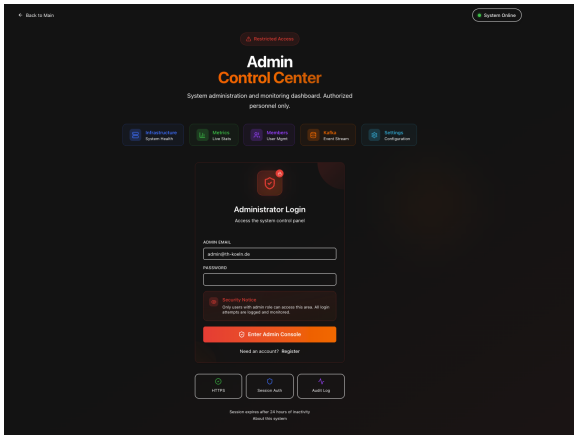

Fig. 7. The admin login page for accessing management features.

page. The rooms tab lets admins list all rooms, create rooms with custom settings, manage participants, and send data messages. The ingress tab handles external stream inputs like RTMP, WHIP, or URL pulls. The egress tab manages recordings and outbound streams with different layouts and formats like MP4 or WebM.

The overview page at `/console` shows a high level dashboard with service health cards for FastAPI, Next.js, Kafka, RabbitMQ, and Redis. It also displays connection metrics, message queue statistics, and real time charts for

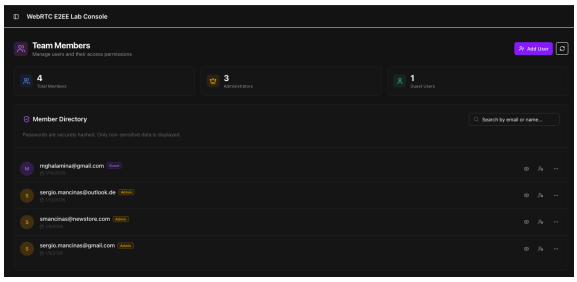

Fig. 8. The members management page where admins can add, edit, and remove users.

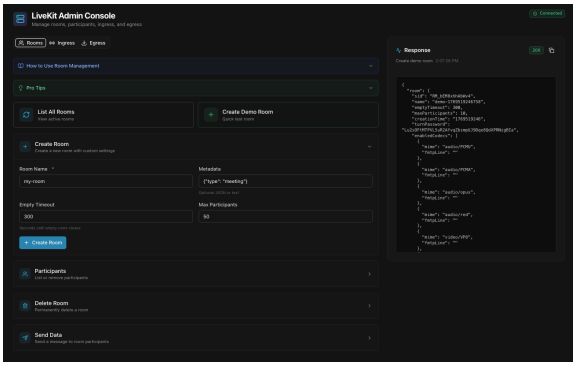

Fig. 9. The LiveKit console showing room management, ingress, and egress controls.

latency and packet loss. The data comes from polling `/api/metrics-json` and `/api/infrastructure`.

The metrics page at `/metrics` embeds Grafana panels as iframes showing Prometheus scrape duration, VM specs, server health, container resources, TURN bandwidth, and Loki log volumes.

The settings page at `/settings` provides configuration options including passphrase hashing and validation, force TURN toggle, debug mode, and auto reconnect settings. It also shows system health and explains how to rotate room keys.

B. Connection Quality Indicator

The signal icon in the room view shows the current call quality based on live metrics. The browser collects WebRTC statistics using the `getStats()` API on the LiveKit peer connection every 5 seconds. These stats include RTT from remote inbound RTP reports, jitter averaged from inbound RTP values, packet loss percentage, and bitrate.

The metrics are sent to `POST /api/ingest` which stores them server side. The UI polls `/api/metrics-json` every 2 seconds to get the latest values. If metrics are older than 15 seconds, the UI shows a collecting status and treats the connection as inactive.

The quality score is computed in the `ConnectionQualityIndicator` component using three weighted factors. RTT contributes 40% of the score, where latency under 50ms scores above 90 and latency over 300ms scores zero. Packet loss also contributes 40%, with loss under 0.5% scoring high and loss over 10% scoring zero. Bitrate contributes the remaining 20%, where rates above 1000 kbps score well and rates below 50 kbps score poorly. The final score determines the label: 80 or above is Excellent, 60 to 79 is Good, 40 to 59 is Fair, and below 40 is Poor.

IX. Testing and Results

A. Codec Support

I tested the system with different video codecs to verify compatibility. The LiveKit logs confirmed that the following codecs work correctly:

For audio, Opus is always used since it is the standard WebRTC audio codec. For video, VP8, H264, and VP9 all worked in my tests. The default codec is VP8 because it has the widest browser support. H264 works well on devices with hardware acceleration, especially Chrome and Safari. VP9 and AV1 provide better compression but need more CPU power and newer browsers.

The UI lets users choose their preferred codec, but LiveKit negotiates the best option that both browsers support. If the selected codec is not available, it falls back automatically. Importantly, E2EE does not affect codec selection because encryption happens after encoding.

Simulcast was also working correctly. The logs showed three quality layers being transmitted: LOW at 320x180 pixels, MEDIUM at 640x360, and HIGH at 1280x720. This lets LiveKit adapt video quality based on each participant's available bandwidth.

B. Observed Metrics

I ran test calls with three participants (using the names jorgito, jammie, and david) and collected metrics through the monitoring pipeline. The results from `/api/metrics-json` showed:

- RTT: 32 to 68 ms
- Packet loss: 0%
- Bitrate: 831 to 2022 kbps
- Connection type: TURN relay
- E2EE: enabled with GCM frame encryption

All participants successfully connected through the TURN server and published both audio and video. The LiveKit logs confirmed media was flowing with encryption active. These numbers represent good call quality according to the scoring system described earlier.

C. Quality Estimation

To measure how well the system performs, I use the E-model from the ITU [14] which calculates a rating factor R on a scale from 0 to 100:

$$R = R_0 - I_s - I_d - I_e + A \tag{1}$$

The variables in this formula are:

- R_0: The base signal to noise ratio, usually around 94.77 for narrowband or 129 for wideband audio
- I_s: Impairments from loud connections or quantization noise
- I_d: Impairments caused by delay (mouth to ear latency)
- I_e: Impairments from the codec and packet loss
- A: Advantage factor that accounts for user expectations (for example, mobile users tolerate more issues)

The R value maps to a MOS score from 1 to 5, where 4.0 or higher means good quality. Based on the observed metrics with low RTT and zero packet loss, the estimated quality would be in the Excellent range.

X. Limitations

A. E2EE Trade-offs

Because encryption happens entirely in the browser, the server cannot access the plaintext media content. This means server side features like recording, transcription, or content analysis are not possible unless encryption keys are shared with a trusted component. This is an inherent trade-off of true end to end encryption.

B. Scale and Meeting Size

The system is designed for small to medium sized meetings. The practical limit depends on the VM resources, uplink bandwidth, and SFU load. I have not tested with very large meetings of hundreds of participants, and that would require additional scaling work like running multiple LiveKit instances with Redis coordination.

C. Browser Support

E2EE using Insertable Streams requires browser support for this API. Chromium based browsers like Chrome, Edge, and Brave are known to support it fully. Firefox has partial support and Safari's support varies by version. Users on unsupported browsers would not be able to use the E2EE features, though they could still join calls without encryption.

D. Mobile Considerations

Mobile devices may experience reduced performance because of limited CPU power and battery constraints. iOS is particularly restricted since all browsers on iOS use WebKit under the hood, which may not support all WebRTC features. I recommend testing on specific mobile devices before relying on mobile support for production use.

XI. Future Work

There are several things I want to work on next. First, I want to run proper experiments where I simulate bad network conditions and measure how the call quality changes. I also want to compare different video codecs like VP8, VP9, and AV1 to see which works best with E2EE. Another goal is to look into the SFrame standard [7] which is the official way to do E2EE that the IETF is developing. Finally, I would like to test horizontal scaling to see how well the system handles many simultaneous rooms.

XII. Conclusion

In this project I built a video conferencing system that keeps conversations private even when using a server to route the video streams. The main challenge was making end to end encryption work with a media server, and I solved this by using the insertable streams API that modern browsers support. The system also includes monitoring so I can track call quality and catch problems.

Everything runs in Docker containers which makes it easy to deploy on any Linux server. The security incident I experienced taught me important lessons about properly configuring

servers, and the hardened setup should be much more resistant to attacks.

Overall, the combination of LiveKit for media routing, Coturn for NAT traversal, and client side encryption provides a good foundation for private video calls.

Acknowledgment

I want to thank Prof. Dr.-Ing. Andreas Grebe for supervising this project and providing guidance throughout. Special thanks to Raphael Aerens who helped set up the virtual machine and was always available when I had infrastructure problems.

References

[1] H. Alvestrand, "Overview: Real-Time Protocols for Browser-Based Applications," *RFC 8825*, Internet Engineering Task Force (IETF), Jan. 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8825

[2] E. Rescorla, "WebRTC Security Architecture," *RFC 8827*, Internet Engineering Task Force (IETF), Jan. 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8827

[3] H. Alvestrand, "Transports for WebRTC," *RFC 8835*, Internet Engineering Task Force (IETF), Jan. 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8835

[4] R. Mahy, P. Matthews, and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)," *RFC 5766*, Internet Engineering Task Force (IETF), Apr. 2010. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5766

[5] C. Jennings, H. Boström, and J.-I. Bruaroey, "WebRTC: Real-Time Communication in Browsers," W3C Recommendation, Jan. 2021. [Online]. Available: https://www.w3.org/TR/webrtc/

[6] C. Oström and E. Ivov, "True End-to-End Encryption with WebRTC Insertable Streams," *webrtcHacks*, Apr. 2020. [Online]. Available: https://webrtchacks.com/true-end-to-end-encryption-with-webrtc-insertable-streams/

[7] E. Omara *et al.*, "Secure Frame (SFrame)," *Internet-Draft draft-ietf-sframe-enc*, Internet Engineering Task Force (IETF), Work in Progress, 2024. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-sframe-enc/

[8] LiveKit, Inc., "LiveKit Documentation," 2024. [Online]. Available: https://docs.livekit.io/

[9] LiveKit, Inc., "livekit/livekit: End-to-end stack for WebRTC," GitHub Repository, 2024. [Online]. Available: https://github.com/livekit/livekit

[10] O. Moskalenko *et al.*, "coturn/coturn: coturn TURN server project," GitHub Repository, 2024. [Online]. Available: https://github.com/coturn/coturn

[11] Docker, Inc., "Docker Compose Overview," *Docker Documentation*, 2024. [Online]. Available: https://docs.docker.com/compose/

[12] F5, Inc., "NGINX Reverse Proxy," *NGINX Documentation*, 2024. [Online]. Available: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/

[13] Grafana Labs, "Introduction to Prometheus," *Grafana Documentation*, 2024. [Online]. Available: https://grafana.com/docs/grafana/latest/fundamentals/intro-to-prometheus/

[14] International Telecommunication Union, "The E-model: A computational model for use in transmission planning," *ITU-T Recommendation G.107*, Jun. 2015. [Online]. Available: https://www.itu.int/rec/T-REC-G.107