Secure Video Streaming Project Report

Course: ECE 4301 - Cryptographic Algorithms on Reconfigurable Hardware
Project: Secure Real-Time Video Streaming on Raspberry Pi 5
Team: Group C
Date: 11/1/2025

---

1. Executive Summary

We implemented a hardware accelerated secure video streaming system for Raspberry Pi 5 that demonstrated real time video encryption and transmission using modern cryptographic protocols like RSA and ECC and Diffie-Hellman key exchange. This was extended to work with three nodes. Our results show that this can be done at low power on embedded or inexpensive SBC platforms such as the Raspberry Pi 5.

2. System Design

2.1 Architecture Overview

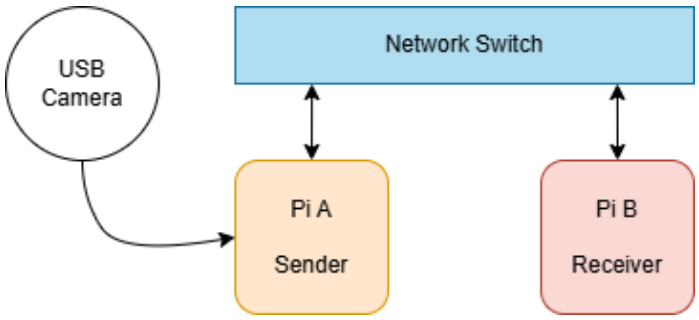


Fig 2A: The basic setup for two Pis to securely stream video. This can be duplicated to stream and receive bidirectionally.

2.2 Cryptographic Design

Key Establishment:

- **RSA-2048 OAEP-SHA256:** Using point to point communication where the receiver has a RSA public key. The sender generates a 24 byte session key consisting of a 16 byte AES-128 key and an 8 byte nonce base. The sender encrypts the receiver's RSA-2048 public key using OAEP-SHA256. Receiver decrypts with their private key.
- **ECDH P-256 + HKDF-SHA256:** Both sides generate fresh short term P-256 key pairs per session and then they compute the shared secret. The keys are derived from

HKDF-SHA256 where two short lived public keys are concatenated. The reason for the short lived keys is so there is no trace to reveal past sessions.

- **Context strings, salt handling, key derivation:** The context string is a short ASCII tag which includes the HKDF info to help rotate formats. Salt handling gives domain separation and resists key compromise impersonation style bad randomness by a handshake via SHA-256.

Session Encryption:

- **Cipher:** AES-128-GCM
- **Nonce construction:** 64-bit random base + 32-bit counter
- **AAD:** Frame header (flags, timestamp, counter, length)
- **Rekey policy**: Every 600s OR 2^{20} frames, whichever first. It keeps nonce usage far from the 2 limit and bounds key lifetime for GCM's security margins.

2.3 Transport Protocol

- **Why TCP vs UDP:** The application benefits from ordered, loss free delivery. TCP gives built in retransmission, rate limit and path fairness without reimplementation.
- **Reliability vs latency tradeoffs:** We accepted head of line blocking in exchange for reliability and simpler congestion behavior.

2.4 Implementation Details

- **Language:** Rust (better memory protection than C, but suited to C-like applications)
- **Crypto crates**: `aes-gcm`, `p256`, `rsa`, `hkdf`, `sha2`
- **Hardware acceleration:** The aes-gcm backend uses AES and PMULL via the crypto extension when the binary is built which improves GCM throughput.
- **Async runtime:** Tokio for TCP I/O and timers. All handshakes and streams are fully asynchronous.
- **Metrics:** CSV logging with sysinfo + manual power sampling with

3. Hardware Acceleration

3.1 CPU Feature Verification and Runtime Detection

```
miad@miapi5:~/midterm1_display $ grep Features /proc/cpuinfo | head -1
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdh
```

```
miad@miapi5:~/midterm1_display $ ./target/release/stream --mode sender --print-c
onfig
=== Configuration ===
Mode: Sender
Mechanism: Ecdh
Host: 0.0.0.0
Port: 8443
Node ID: node-1
Video Source: camera
Video Device: /dev/video0
Resolution: 640x480 @ 15 fps
Rekey interval: 600s
Simulate: false
Display: false

=== ARMv8 Crypto Extensions Detection ===
AES:   ACTIVE ✓
PMULL: ACTIVE ✓
SHA1:  ACTIVE ✓
SHA2:  ACTIVE ✓
=======================================
```

3.2 AES-128-GCM Encryption Benchmark Results

```
miad@miapi5:~/midterm1_display $ ./target/release/bench
=== AES-128-GCM Hardware Acceleration Benchmark ===

CPU Features:
  AES:   ✓ DETECTED
  PMULL: ✓ DETECTED

Generating 256 MB of random test data...
Starting encryption benchmark...

=== Results ===
Total encrypted: 256.00 MB
Time elapsed:    3.965 seconds
Throughput:      64.56 MB/s
Frames/sec:      147.28 fps (@ 460KB/frame)

=== Performance Analysis ===
⚠ WARNING: Low throughput despite HW support detected
  Expected: >200 MB/s with ARMv8 Crypto Extensions
  Check: Compilation flags, CPU governor, thermal throttling

To build control (software-only) version:
  RUSTFLAGS='-C target-feature=-aes,-pmull' cargo build --release
miad@miapi5:~/midterm1_display $
```

4. Streaming Performance

4.1 Latency Distribution

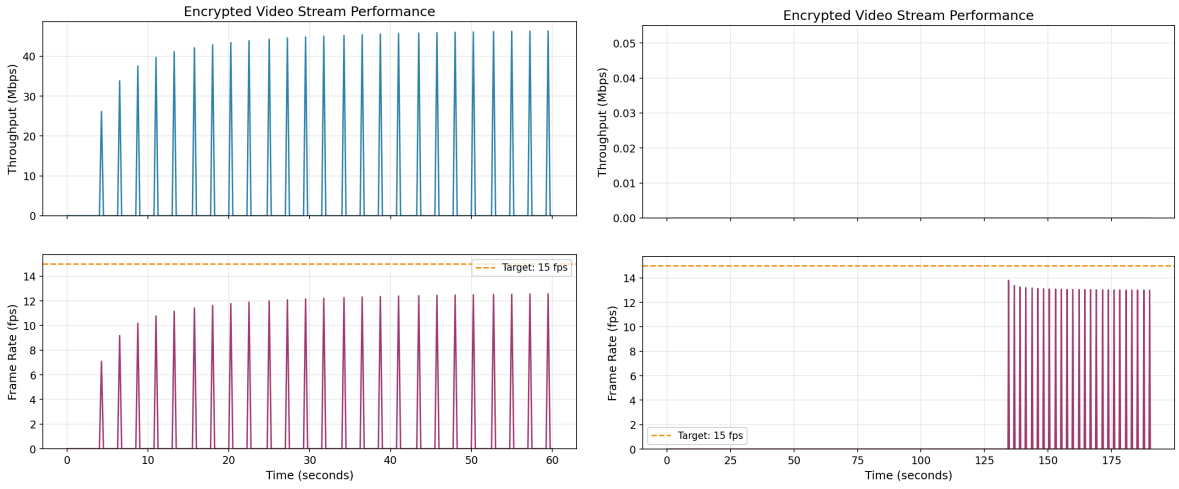


Fig 4.1A/B: Throughput (group leader Pi-1 on left; receiver Pi-2 on right)

There was a problem in measuring or recording throughput on the receiver with this script, but we can see the point at which the receiver starts to record receiving video is around 130 seconds.

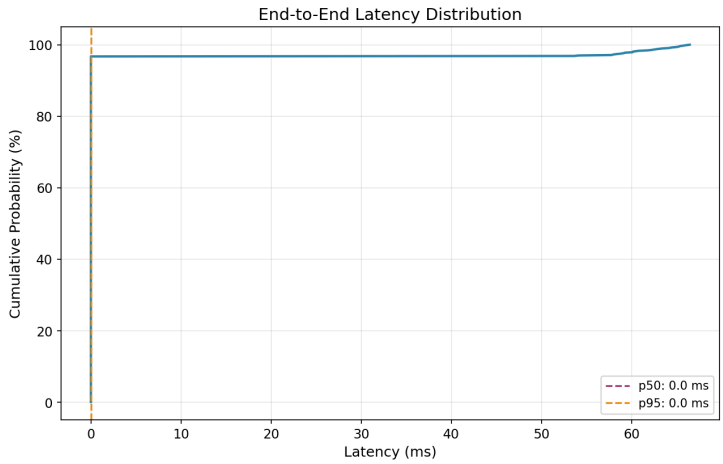


Fig 4.1C: Cumulative distribution function for the latency, recorded from the receiver Pi-2.

4.2 System Resource Usage

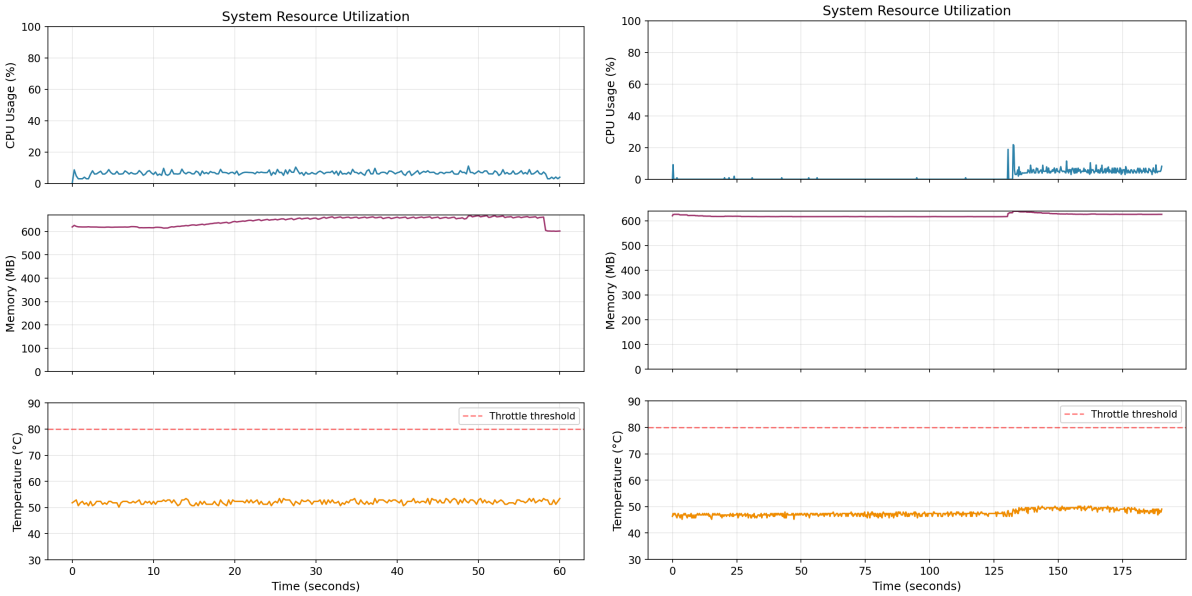


Fig 4.2A: Resource use (group leader Pi-1 on left; receiver Pi-2 on right)

5. Energy Analysis

5.1 Measurement Setup

We had USB-C connectors that would display live power data like voltage, amperage and wattage which can help derive the total power consumption during the stages of communication. This data was collected in a video and manually recorded.

Fig. 5.1A/B:A: left is a close up of the USB power meter; right, an example of the setup for measuring power in real time

5.2 Results

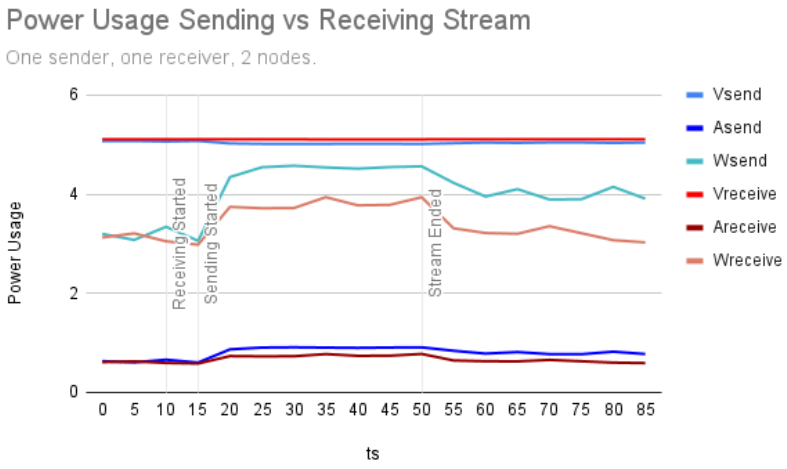


Fig 5.2A: Power usage for two node video streaming.

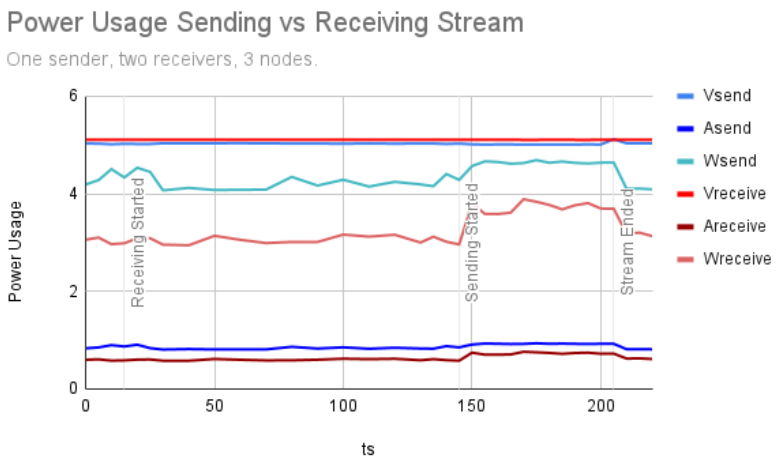
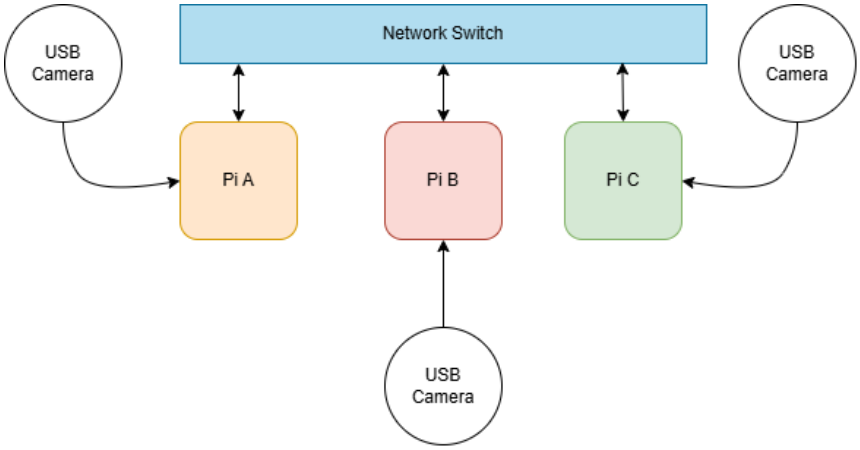


Fig 5.2B: Power usage for three node video streaming.

6. Group Extension (N≥3)

6.1 Protocol Design



Leader-Distributed (KEM-style)

Leader generates K_group
  ├─> Pairwise ECDH with Member 1 → Enc(K_group)
  ├─> Pairwise ECDH with Member 2 → Enc(K_group)
  └─> Pairwise ECDH with Member 3 → Enc(K_group)
All members verify HMAC(K_group) before streaming

This style of leader distributed key has the members listen to do a key exchange with the leader in a pairwise manner: the leader creates and distributes a key. This does not scale as well as the binary tree model, but is very simple to implement for a small multicast environment.

7.2 Three-Node Demonstration

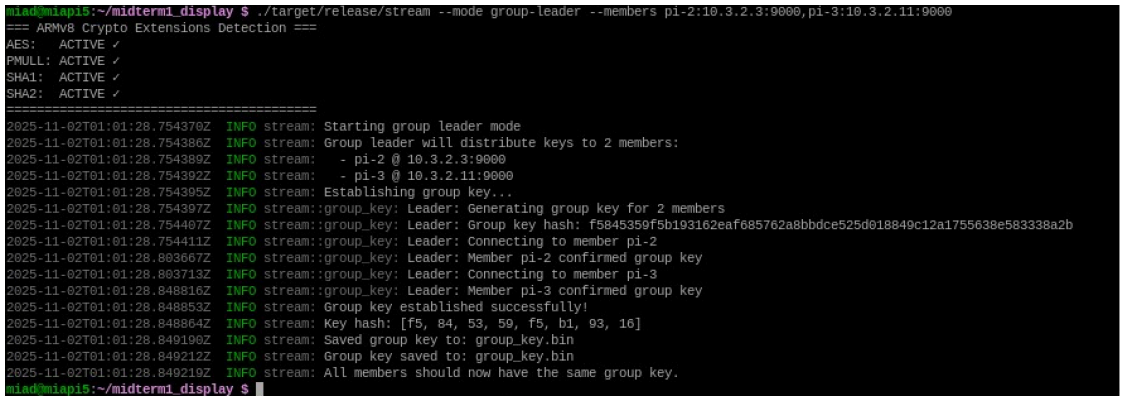
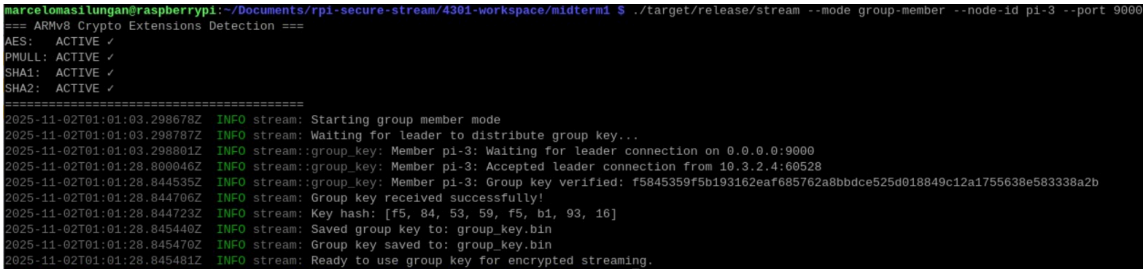


Fig 7.2A: Output for leader; key generation/distribution



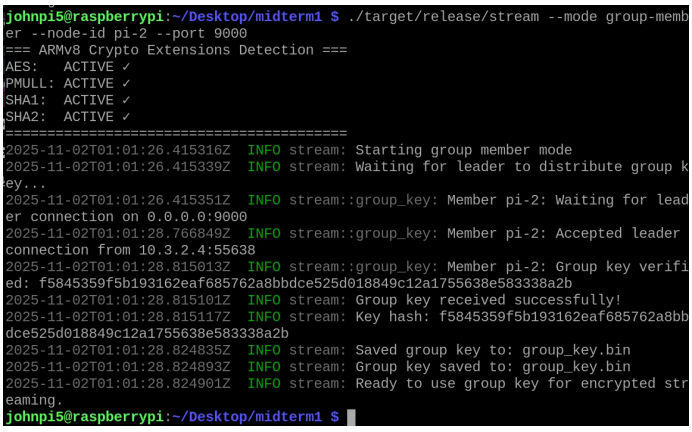


Fig 7.2 B/C: Output for the followers Pi-2 and Pi-3; group key distribution

Video demonstrations of the keying and streaming process from both the leader and follower ends are on the Github repo.

7. Discussion

Strengths	Limitations
• Hardware TRNG for entropy • Authenticated encryption (AES-GCM) • Unique nonces per frame • Forward secrecy (ephemeral keys) • Automatic rekeying • Key zeroization	• No authentication of endpoints (could add certificates) • Vulnerable to quantum attacks (both RSA and ECDH) • No protection against traffic analysis • Rekey visible in timing (opportunity to exploit) • Unable to get video to stream on both receivers in 3 node implementation

It was noted during testing that the benchmark result for a loopback/self-test showed reduced throughput compared to the expected results despite confirming the crypto extensions were working. This could be a limitation of the Rust implementation or a problem with the overall video ecosystem. For example, during the group video extension tests, significant frame drops or sync issues cause one feed to be a green screen, while the other displays fine.

Figure 4.1A, showing the group leader's metrics, also shows a lower frame rate than desired or expected, which could be related to the problem referenced above. There were some problems with specific webcams not working, such as one that could capture, but often dropped frames or lagged using the standard Video4Linux drivers/API.

8. Conclusion

In this project, we were able to implement working video streaming on RPI5. By implementing hardware acceleration, we were able to achieve the refresh rate requirement of approximately 15 Hz or frames per second (fps) with a latency of approximately 30 ms for a 2-node structure and approximately 60-80 ms for a 3-node structure. Additionally, we observe a linear increase in latency as more receiver nodes are added from a singular sender. Using the elliptic curve Diffie-Hellman (ECDH) key agreement protocol, we were able to generate group keys in 19 microseconds for the 3-node extension. We were able to implement the program with hardware TRNG usage, AES-128-GCM with nonce handling, and group keying protocols in rust.

Appendix

Github Repo

https://github.com/mealycpp/ECE4301_Fall2025/tree/main/Group%20C/midterm1

Hardware Specifications

- **Platform:** Raspberry Pi 5 (BCM2712, Cortex-A76 quad-core @ 2.4 GHz)
- **RAM:** 8GB LPDDR4X
- **OS:** Raspberry Pi OS 64-bit (Bookworm, kernel 6.1)
- **Network:** Gigabit Ethernet with switch
- **Camera:** USB webcam, different models
- **Power Supply:** 27W USB-C adapter

References

1. NIST SP 800-38D: *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM)*
2. RFC 7748: *Elliptic Curves for Security* (X25519, X448)
3. RFC 8017: *PKCS #1: RSA Cryptography Specifications Version 2.2*
4. ARM Architecture Reference Manual ARMv8 (Crypto Extensions)
5. *The Rust Programming Language* (https://doc.rust-lang.org/)