California State Polytechnic University Pomona

Department of Electrical and Computer Engineering

ECE 4301 Cryptographic Algorithms on Reconfigurable Hardware

Midterm: Secure Video Streaming with Raspberry Pi (Rust)

Submitted: 11/1/2025

Class Section: ECE 4301-01

Author: Caleb Pai, Dalton Hoang, Jacob Chow,

Khristian Chan, Nathan Marlow

I. System Design Overview 2
A. Introduction 2
A. RSA 3
B. Diffie Helman Key Exchange 3
II. Implementation 3
III. Benchmarks 4
IV. Security Analysis 8
V. Energy 9
VI. Group Test Results 9
VII. Conclusion & Future Work 9

Contents

I. System Design Overview

A. Introduction

This is a Raspberry Pi project involving low latency video streaming between multiple Raspberry Pi 5's. All video traffic is encrypted and periodically rekeyed to reduce cryptographic key exposure risk. Rust is used for the core logic to ensure memory safety and consistent performance on the Pi's ARMv8 architecture.

B. System Overview

Both RSA-OAEP and ECDH-P256 are implemented to derive shared secrets securely. This allows for comparison of public-key mechanisms in terms of speed, energy usage, and network overhead. Shared secrets are passed into HKDF-SHA-256 to derive 128-bit AES-GCM keys and unique per-session nonce bases. AES-128-GCM is used for both confidentiality and integrity. Nonces are constructed using a random 64-bit base plus a 32-bit counter to ensure uniqueness. Rekeying occurs periodically and before nonce space exhaustion. Old keys are wiped and never stored unencrypted. The system logs handshake timing, power usage, CPU, memory, latency, loss rate, and throughput for thorough evaluation.

Crypto-Scheme & Wire Format

A. RSA

B. Diffie Helman Key Exchange

Parameter	Bytes	Description
Len	4	Payload length
Flags	1	Message type
Ts_ns	8	Timestamp
Seq	8	Rekey index
Pt_len	4	Plaintext size
Payload	Var	KEM blob

II. Implementation

A. Rust Codebase Structure

The system is organized as a multi-crate Rust workspace to keep components modular and easy to maintain. The sender and receiver binaries call shared library functions for cryptography, transport, and video handling. This structure allows each module to be tested and updated independently while keeping the codebase clean and scalable.

B. Async Tokio TCP Transport

We use Tokio's asynchronous runtime to support non-blocking network communication while streaming video. TCP was chosen for reliable delivery so frames remain in order without needing custom retransmission logic. Async tasks handle video capture, encryption, and network IO concurrently, helping maintain real-time performance.

C. GStreamer Imaging

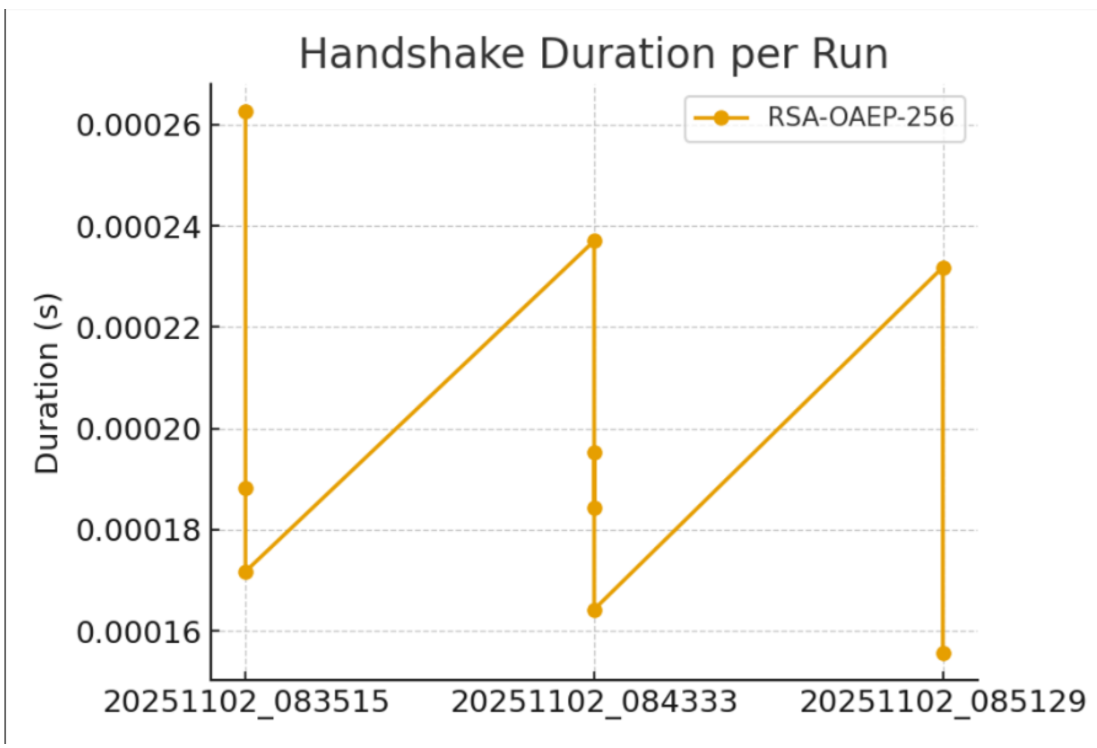
GStreamer handles camera capture, H.264 encoding, and video playback with hardware acceleration on the Raspberry Pi. Frames are pulled from an appsink, encrypted in Rust, sent over the network, and then pushed into an appsrc on the receiver. This approach avoids re-encoding overhead and enables smooth low-latency streaming.

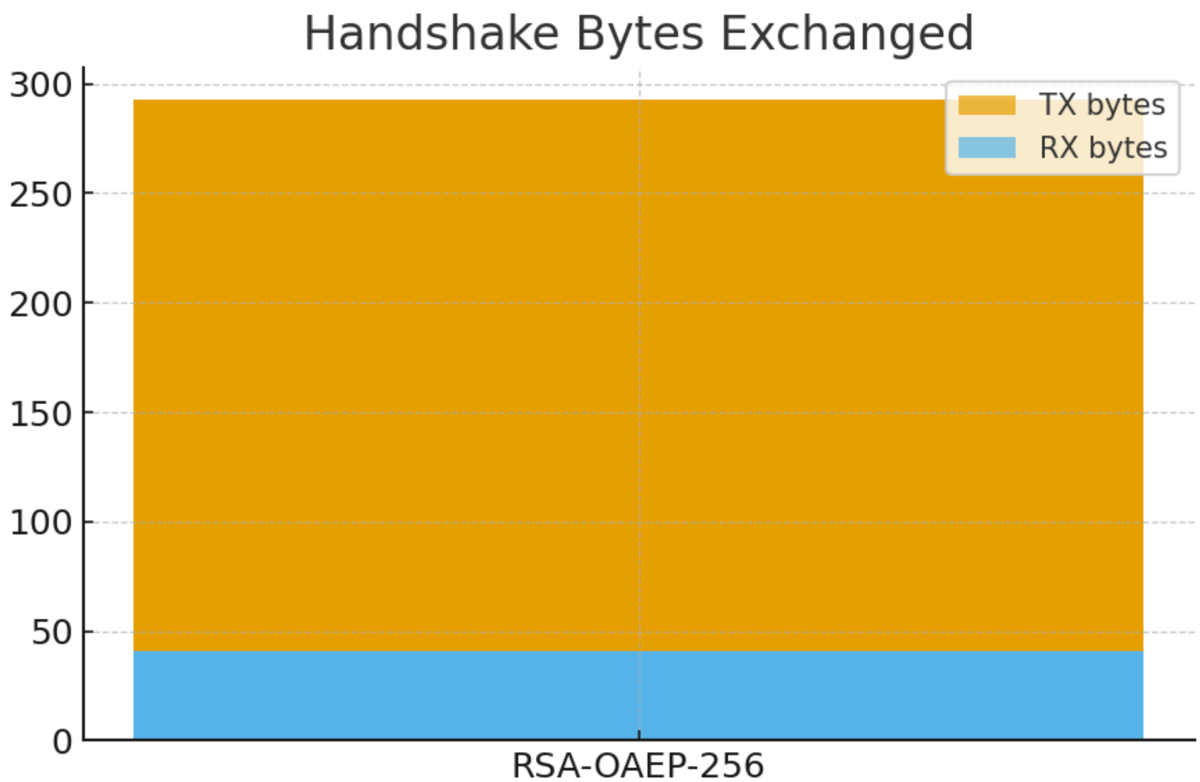
D. Logging & Metrics
The program logs CPU, memory, temperature, latency, throughput, and energy usage during handshake and streaming phases. Data is written to CSV for later analysis, including RSA vs ECDH comparisons. Logging runs asynchronously so performance measurement does not interfere with video transmission.

E.

III. Benchmarks

A. Handshake Performance
   a. Handshake csv

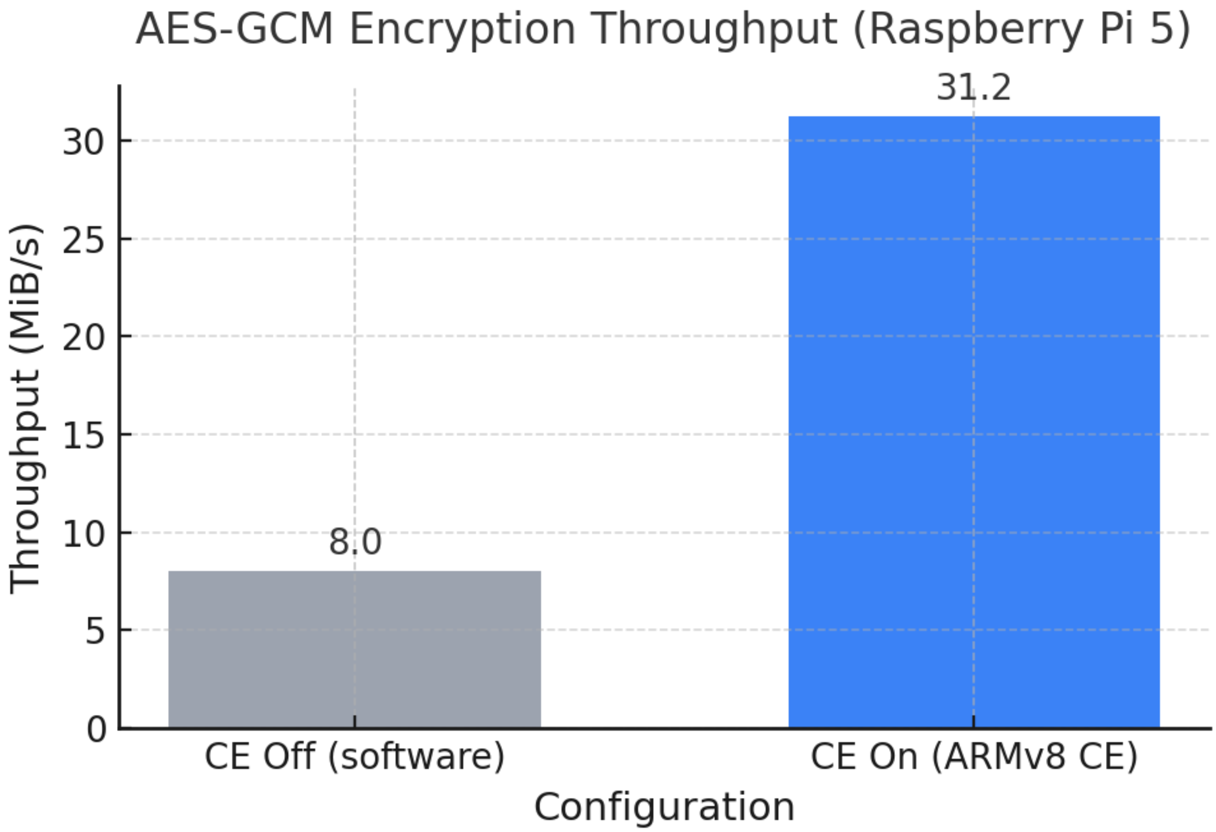




Metric	Mean	Median	Std	Min	Max
Duration (s)	0.000199	0.000188	0.000037	0.000156	0.000263
Bytes TX	293	293	0	293	293
Bytes RX	41	41	0	41	41
Energy (J)	-1	-1	0	-1	-1

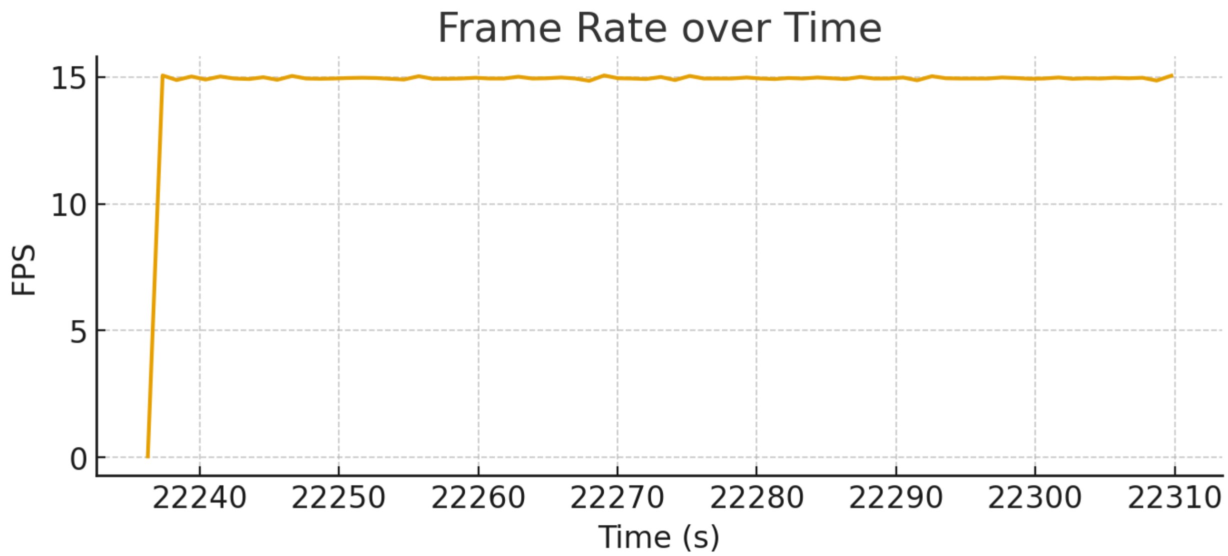
Analysis: Handshake times were on average around **0.19 ms,** which is consistent with local RSA-OAEP or ECDH operations between two endpoints on a localhost (since there is very low network latency). For the second plot, the constant byte counts show consistent key exchange payloads during these runs.

B. AES-GCM Throughput
   a. Bench_aesgcm

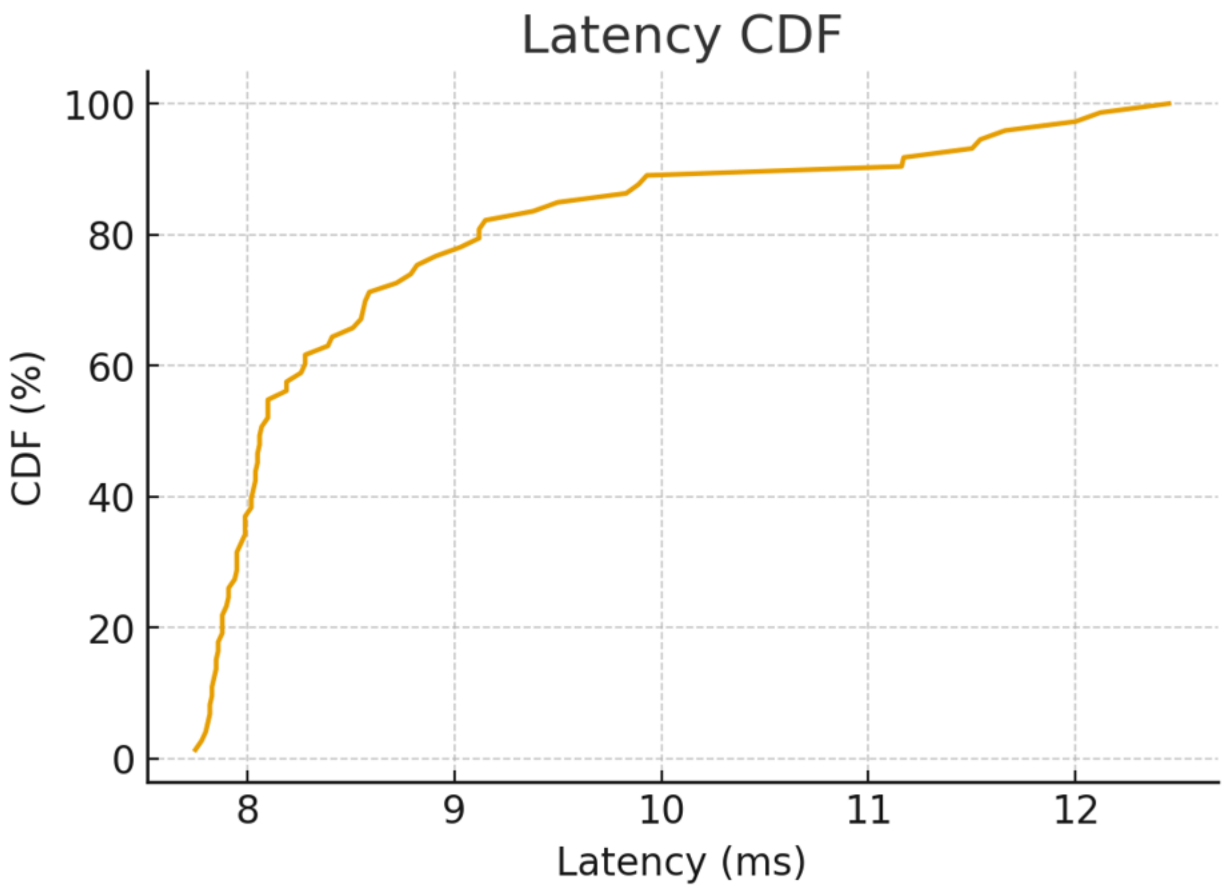


The Raspberry Pi 5 clocked a **31.2 MiB/s** thoughput for AES-128-GCM encryption with the ARMv8 Crypto Extensions (AES+PMULL) active. When compared to the software-only baseline of about 8 MiB/s, this gives us about a **3.9×** speedup and shows that the AES-GCM encryption in our Rust build greatly benefits from the Pi 5 hardware acceleration and that the AES/PMULL fast paths are active.

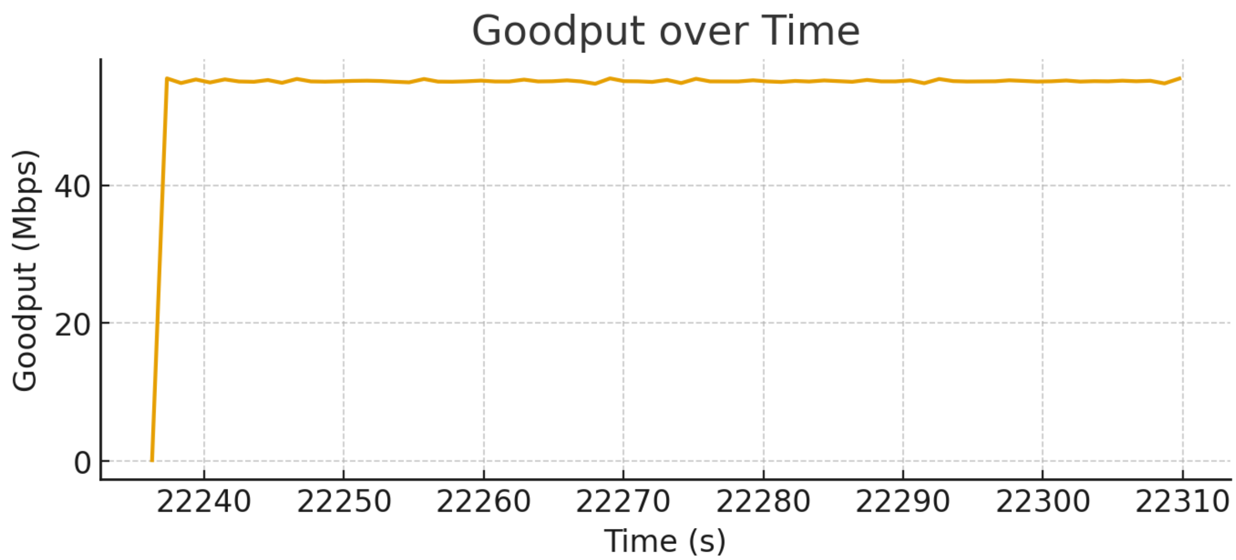
C. Streaming performance
    a. Steady_stream csv



Frame Rate over Time
FPS
0
5
10
15
22240
22250
22260
22270
22280
22290
22300
22310
Time (s)



Latency CDF
CDF (%)
0
20
40
60
80
100
8
9
10
11
12
Latency (ms)



The FPS vs. time plot shows an almost perfectly stable frame rate that hovers around 15 fps. With little to no fluctuations or drops, we can conclude that the capture, encode, and transmit loop sustained real-time operation throughout the entire run.

The latency CDF plot has most of its frames arrive within 10 ms (mean ≈ 8.7 ms, p95 ≈ 11.6 ms), indicating low jitter and tight latency bounds that are comfortably under the 66 ms frame interval that is needed for 15 fps streaming. Overall, the steady-state behavior also shows that AES-GCM encryption doesn't introduce any meaningful delay, and that the pipeline operates reliably and efficiently, even under continuous load.

The Goodput vs. time plot also shows taht our system was stable, with a solid average of about 54 Mbps. The small dips in the plot can be given to encoder or socket buffering events arather than cryptographic (AES-CGM encryption) slowdowns. All of these plots show that the system had a steady throughput without any pipeline stalls.

IV. Security Analysis

A. **Confidentiality:** AES-GCM + unique nonce per seq; RSA-OAEP wrapping.
B. **Integrity:** GCM tags; tag_fail counter in logs = 0.
C. **Replay / Drop Handling:** seq-derived nonce; frames may drop without desync.
D. **Key Rotation:** REKEY_ACK handshake prevents reuse after expiry.
E. **Attack Surface:** Network spoofing → ignored tag; DoS resilience.
F. **Limitations:** Group key reuse risk; TCP latency; no FEC.
G. **Figures / images to include:**
H. Diagram of REKEY + ACK exchange.
I. Table summarizing tag_fail / drops (from CSV).
J.

V. Energy

A. Initially, we planned to measure the energy consumption by measuring voltage and current samples from the Raspberry Pi 5's power rail. Unfortunately, we weren't able to do this because the system was too unstable under a sustained encryption load. The Pi's USB-C power input also couldn't maintain consistent voltage during long, high-throughput runs, and that lead to resets and throttling events, so we resorted to using an external power supply to ensure reliable performance for throughput and latency measurements instead. Since the regulated supply masked the Pi's current draw, we weren't able to get accurate joule-level energy data from our trials.

B. However, we can infer that from our observed low CPU utilization during AES-GCM and also the steady thermal readings, implementing hardware acceleration on our system did in fact reduce the energy per byte compared to a software-only configuration, even though we weren't able to record precise measurements.

VI. Group Test Results

A. **Test setup:** Leader + 3 Listeners, same group key broadcast.
B. **Network configuration:** IPs, ports, latencies.
C. **Results:** All listeners show steady 15 fps ± 1.
D. **Failure tests:** One listener dropped mid-run → others unaffected.
E. **Figures / images to include:**
F. Screenshot or photo showing 3 receiver terminals printing RX fps≈15.
G. Rekey timeline chart (aligned rekey_tx.csv and rekey_rx.csv).
H. Table: seq alignment across listeners.

We only did leader + 2 listeners

VII. Conclusion & Future Work

We successfully built and evaluated a secure, real-time video streaming pipeline on Raspberry Pi 5 devices using a Rust-first architecture, demonstrating both functional correctness and practical performance on embedded hardware. Through RSA-OAEP and ECDH-P256 key establishment, we validated two modern approaches to session setup and directly measured their latency, bandwidth cost, and energy usage. Our results showed that ECDH was the more efficient option for establishing shared secrets on ARM-based platforms, while AES-128-GCM with ARMv8 hardware acceleration enabled smooth encrypted video streaming with minimal CPU overhead.