Raul Garcia

Priyanka Ravinder

Arvind Mohanraj

Professor Aly

ECE 4301.01

ECE4301 Midterm — Secure Video Streaming with Raspberry Pi 5 (Rust‑First) Report

Introduction:

For this midterm project, our goal was to build a secure video streaming system using two Raspberry Pi 5 boards and the Rust programming language. We wanted to send live video from one Pi to another, but do it in a way that keeps the stream private, encrypted, and verified from end to end. This meant using real cryptography, not just passwords or basic encryption, but actual public-key exchanges, hardware-accelerated AES-GCM encryption, and authenticated communication between the two devices. We also made sure the system worked in real time, so the receiver could play the live video with very little delay. Everything, from the encryption and key setup to the video transport, was coded directly in Rust. This helped make the project safer and more reliable since Rust prevents common memory bugs that can cause security issues.

Our system connects the two Raspberry Pi 5 devices over Ethernet, with one acting as the leader, or sender, and the other as the member, or receiver. The leader uses a camera to capture video frames through GStreamer, encrypts those frames using AES-128-GCM, and sends them over a TCP socket managed by Tokio, which handles asynchronous communication in Rust. The receiver then decrypts the frames, checks that nothing was tampered with, and displays the video using another GStreamer pipeline. We also measured important data such as CPU usage, memory, temperature, throughput, and latency, saving it all to CSV files for later analysis. What makes this project special is that it does not just show a video stream but demonstrates real-world secure media transmission on embedded hardware. By using both RSA and Elliptic Curve Diffie–Hellman for key establishment and by taking advantage of the ARMv8 crypto extensions built into the Pi 5 processor, we were able to prove that it is possible to combine strong security and good performance even on small, low-power systems.

System Overview:

The secure video streaming system was built using two Raspberry Pi 5 boards connected via Ethernet to ensure a stable and low-latency connection. One Raspberry Pi functions as the leader, or sender, while the other acts as the member, or receiver. The sender captures live video from an attached camera and transmits it securely to the receiver in real time. Using a wired connection helps maintain consistent throughput and reduces the risk of frame drops, which is especially important for encrypted streaming where data reliability directly impacts performance.

On the sender side, video frames are captured through the GStreamer framework, which provides an efficient way to handle camera input and multimedia processing. Each frame is encrypted using AES-GCM, a symmetric encryption standard that provides both confidentiality and authentication, ensuring that the transmitted video cannot be intercepted or modified without detection. The encrypted frames are then transmitted over a TCP socket using the Tokio library in Rust, which enables asynchronous I/O operations and prevents blocking during streaming.

The receiver listens for incoming encrypted frames, decrypts them using the same AES-GCM key, and verifies the authenticity of each frame before reconstructing the video. Once verified, the frames are passed through another GStreamer pipeline to display the live video feed.

This process allows for real-time playback of encrypted content while maintaining both security and performance throughout the transmission.

Both Raspberry Pi 5 boards run Raspberry Pi OS (64-bit) and use Rust version 1.79 for development. The Raspberry Pi 5's ARMv8 processor supports built-in AES and PMULL crypto extensions, which provide hardware acceleration for cryptographic operations. These extensions significantly improve encryption and decryption speeds while reducing CPU load. Additionally, the system logs important performance metrics such as CPU usage, memory consumption, temperature, latency, and throughput using a dedicated Rust module called metrics.rs. The collected data is written to CSV files for later analysis, allowing us to evaluate system performance under different conditions. In summary, the video streaming path can be represented as: Camera → GStreamer → AES-GCM Encryption → TCP → AES-GCM Decryption → GStreamer → Video. This design demonstrates that secure, real-time video transmission is achievable on small embedded systems using modern programming tools and hardware acceleration.

Design and Implementation:

The Rust workspace for this project is divided into three main components: the leader, the member, and the metrics logger. Each plays an essential role in the secure video streaming process. The leader handles encryption and key setup, the member manages decryption and playback, and the metrics logger collects performance data.

The leader, or sender, performs RSA and ECDH key exchanges to securely establish a shared AES session key with the receiver. It encrypts MJPEG video frames using AES-128-GCM, which ensures both data confidentiality and integrity. To maintain security, the leader rekeys approximately every ten minutes and logs handshake times, throughput, and CPU usage during operation.

The member, or receiver, completes the corresponding RSA and ECDH operations to derive the same AES session key. It then decrypts incoming frames, verifies their authenticity, and sends them to its GStreamer pipeline to reconstruct and display the video. The receiver also logs latency, temperature, and frame loss events to evaluate performance and stability.

The metrics logger records data to CSV files such as handshake.csv, throughput.csv, and steady_stream.csv. It samples CPU, memory, and temperature every few seconds and verifies that AES and PMULL hardware acceleration are active. Together, these components create a modular system that combines strong security, real-time performance, and detailed measurement capabilities.

Key Establishment:

For this project, we implemented two different methods for setting up shared encryption keys: RSA key transport and Elliptic Curve Diffie–Hellman (ECDH) with HKDF. The main testing and measurements were done using the ECDH method. In this approach, both Raspberry Pi 5 boards generated temporary (ephemeral) P-256 key pairs and exchanged their public keys. From there, each side used its private key and the other's public key to derive the same shared secret, which was expanded through HKDF-SHA-256 into a 128-bit AES session key and a nonce base. To make sure our setup followed proper security practices, we also rekeyed on a timer and pulled randomness from the Pi's hardware-based random number generator.

When we ran the program, both devices logged that the ARMv8 crypto extensions were active (aes=true and pmull=true), meaning the hardware acceleration for AES and polynomial multiplication was successfully enabled. The handshake between the two devices was lightweight: on average, the leader sent about 95 bytes and received around 70 bytes per handshake, while the member sent roughly 105 bytes and received about 95 bytes. Our timestamps for the start and end of the handshake were identical, which made the recorded duration show up as 0.0 seconds. This is most likely due to how our time logging was implemented, since the actual handshake clearly completed correctly and quickly. Once the key exchange finished, both devices established the same AES-128-GCM session key and immediately began encrypted video streaming without any issues.

Results:

Once the system was running, both Raspberry Pi 5 devices streamed encrypted video steadily and reliably. In steady-state operation, the leader averaged about 14.9 frames per second (fps) with an encrypted data rate of roughly 9.6 Mb/s. The member showed similar performance, averaging around 15.0 fps and 9.3 Mb/s. Both CPUs stayed under heavy load, averaging only about 4.3% usage on the leader and 4.8% on the member. Temperatures remained within normal ranges during streaming, around 48°C on the leader and 57°C on the member.

Latency results were also solid. The receiver recorded an average latency of about 14 milliseconds across 5,420 frames, with a 100% success rate in decrypting and authenticating all frames. The median and 95th percentile both showed up as 0 ms in the CSV logs, likely because our timestamp precision was too coarse to capture sub-millisecond differences. A few frames (around 76 of them, or roughly 1.4%) logged a latency of 1,000 ms, but these are almost certainly timing or clock sync artifacts rather than actual video delay since playback appeared smooth. There were no AES-GCM tag failures and no dropped frames during steady streaming. The leader only reported a single send failure event late in the run, which didn’t affect the video stream.

Both devices again confirmed aes=true and pmull=true in the runtime logs, showing that hardware acceleration was being used throughout the session. Given the low CPU percentages and stable frame rates, it’s clear that the Pi 5’s ARMv8 crypto engine handled most of the encryption and decryption work efficiently.

Data Analysis:

The collected data points to three main conclusions. First, ECDH with HKDF worked extremely well for this system. The handshake data size was very small, only a few dozen to a hundred bytes in each direction, and it transitioned almost instantly into secure video streaming. Even though the handshake duration was recorded as 0.0 seconds, that was just a logging limitation rather than a performance issue. Second, the encryption overhead on the Raspberry Pi 5 was very low. The combination of hardware AES and PMULL acceleration allowed both devices to maintain around 15 fps and close to 9.5 Mb/s of encrypted throughput while keeping CPU usage under 5%. This shows that strong, authenticated encryption can be used on embedded systems without major performance loss. The temperature readings also stayed within safe limits, which indicates good efficiency and thermal stability. Finally, the latency data suggests that the system’s timing was more limited by logging precision than by actual network performance. The average latency of 14 ms aligns with smooth real-time playback, and the occasional 1,000 ms readings likely came from timestamp errors or temporary clock drift. Using a higher resolution or monotonic clock and syncing both Pis more precisely would probably eliminate these outliers. Overall, the encrypted stream was consistent, responsive, and secure, demonstrating that real-time AES-GCM video streaming is completely achievable on Raspberry Pi 5 hardware using Rust.

<u>Conclusion:</u>

Overall, this project successfully demonstrated secure video streaming on the Raspberry Pi 5 using Rust. Implementing the system in Rust allowed us to take advantage of strong memory safety features while integrating a reliable key management system and hardware crypto extensions. These elements were essential to achieving both the security and performance goals of the project, proving that real-time encrypted video streaming is practical on low-power embedded devices.