ECE 4301 Midterm Report: Encrypted Video Streaming

Design

Our encrypted video streaming system comprises two major components: the GStreamer video pipeline, and the encrypt/decrypt module.

On the sender side, the data flows from a Video4Linux2 source through a decodebin element which turns the picture series from the webcam into a video format. It continues through an h264 encoder which compresses the video stream to reduce the work required by the encryption module. GStreamer channel capabilities are added using the GDP protocol. Then the data arrives at an Appsink element where it is picked up by the encrypter module. The encrypter module polls the Appsink element for a sample, and when it is ready the module encrypts the sample and sends the ciphertext over the network to the receiver side.

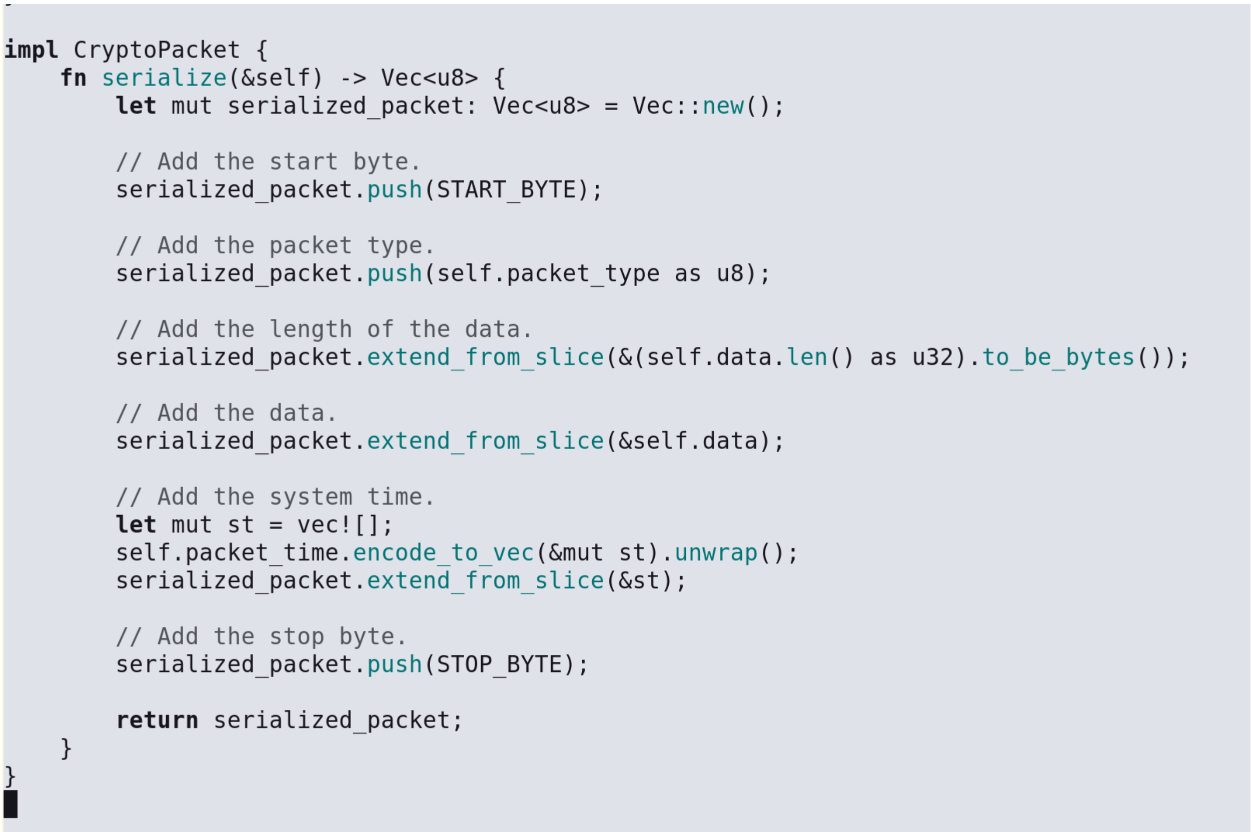
On the receiver side, the ciphertext is received from the sender over the network. The receiver decrypts the payload and then pushes the plaintext to an Appsrc element at which point the data re-enters the gstreamer pipeline. Channel capabilities are discovered by a gdpdepay element, and then the video is forwarded to an h264 decoder to reproduce the original video stream. This video stream goes into an fpsdisplaysink element which renders the video feed and the FPS of the video.

The encrypt/decrypt module additionally frames data into packets, allowing data packets to be freely intermixed with control packets to support live rekeying.

The program is configured at execution time with command line flags to set the program mode---sender or receiver, the asymmetric encryption mode---rsa or ecdh, and the ip address and port of the connection.

It was written entirely in Rust, using the aes-gcm crate for steady-state streaming encrypt/decrypt, which can be compiled with or without hardware acceleration, allowing us to easily examine the effects of hardware acceleration.

Protocol Diagram

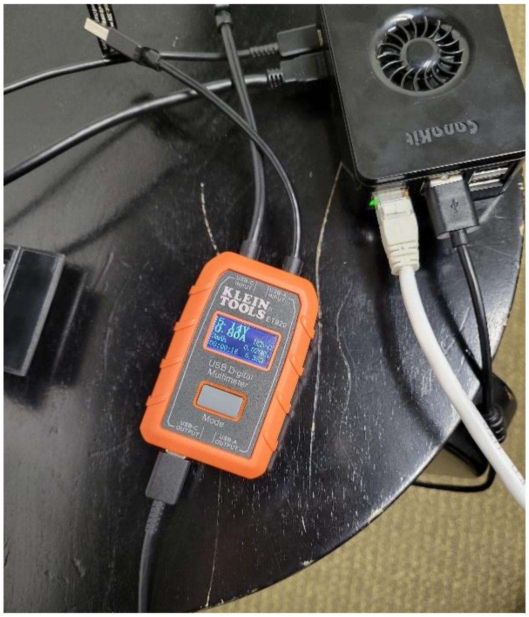


```
impl CryptoPacket {
    fn serialize(&self) -> Vec<u8> {
        let mut serialized_packet: Vec<u8> = Vec::new();

        // Add the start byte.
        serialized_packet.push(START_BYTE);

        // Add the packet type.
        serialized_packet.push(self.packet_type as u8);

        // Add the length of the data.
        serialized_packet.extend_from_slice(&(self.data.len() as u32).to_be_bytes());

        // Add the data.
        serialized_packet.extend_from_slice(&self.data);

        // Add the system time.
        let mut st = vec![];
        self.packet_time.encode_to_vec(&mut st).unwrap();
        serialized_packet.extend_from_slice(&st);

        // Add the stop byte.
        serialized_packet.push(STOP_BYTE);

        return serialized_packet;
    }
}
```

Security Choices

We use RSA 3072 to send a session key; the transmitter sends the session key to the receiver via the receiver’s public key. On each rekey request, the transmitter generates a new session key, and the receiver generates a new public key. In the case of ECDH, both generate new public keys.

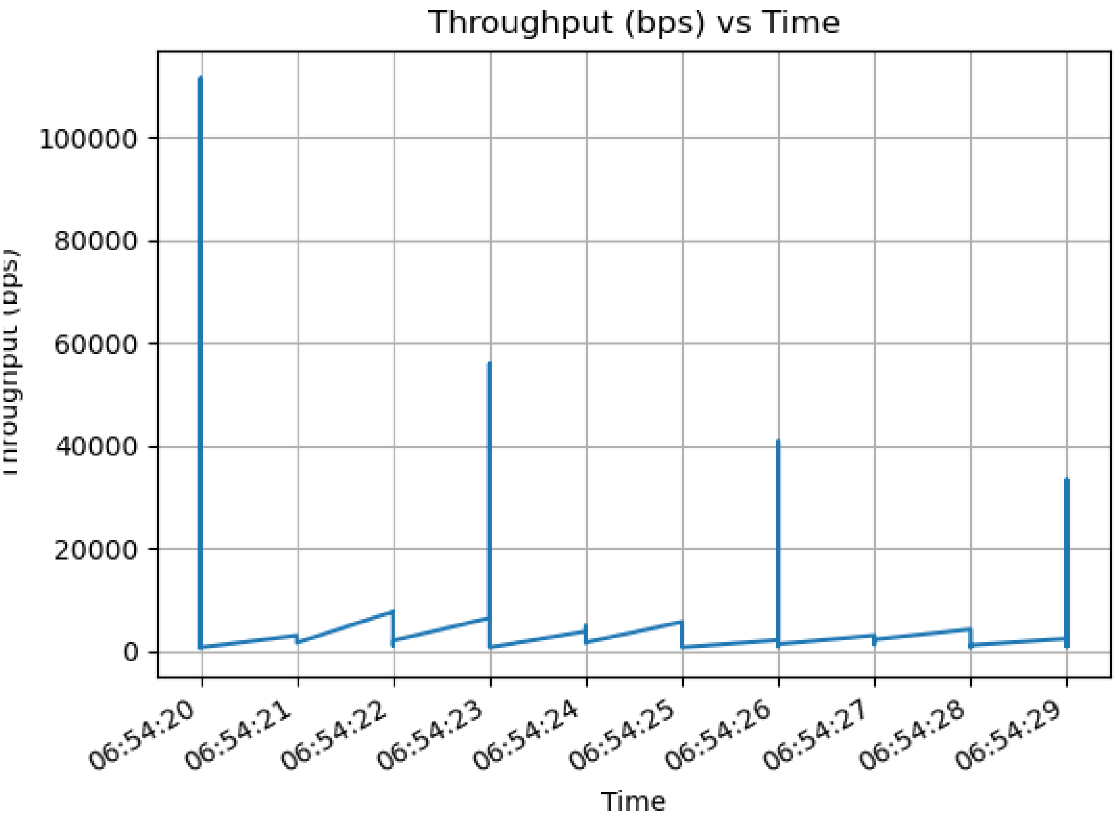
Measurement Setup



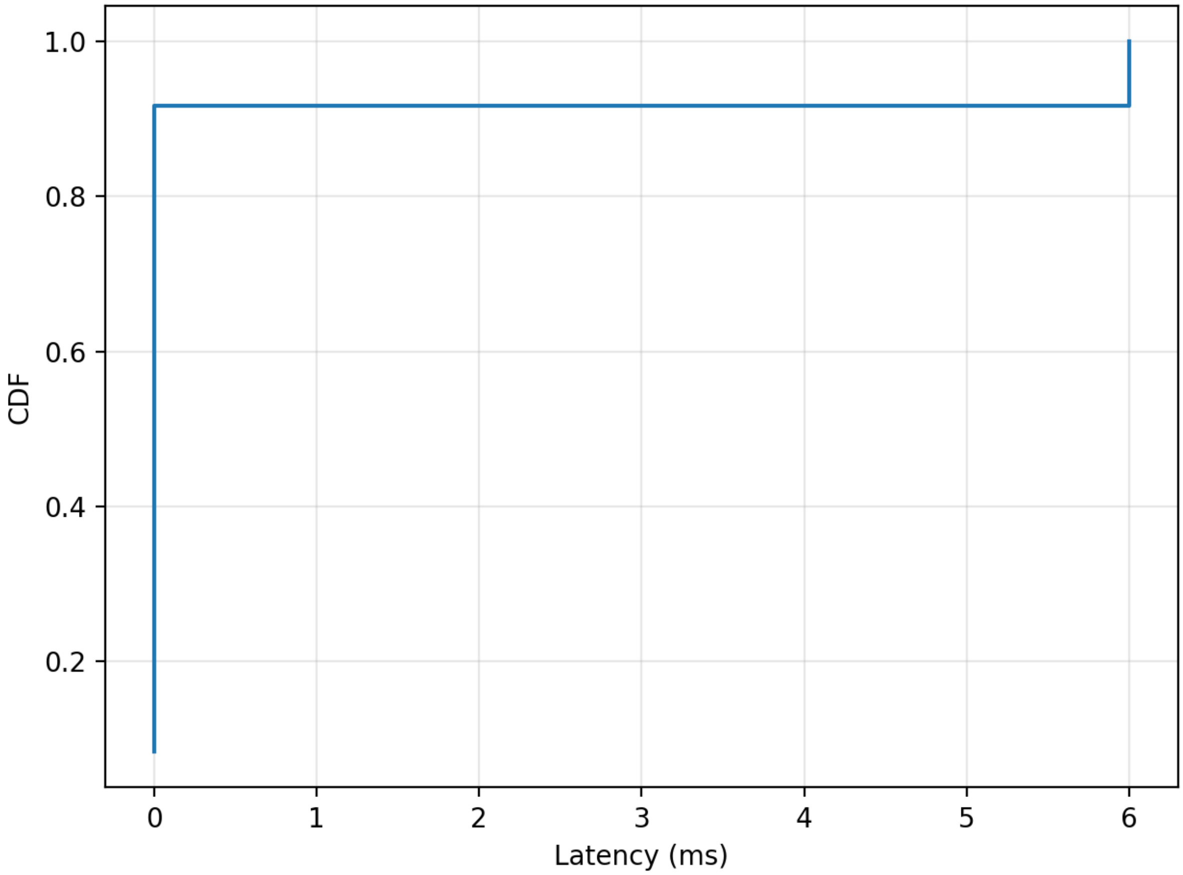
Results

We were able to acheive a framerate of 30fps with a 640p camera feed without any significant jitter. We observed that ECDH key exchange was trivial, finishing in usually under a millisecond, while RSA key exchange could take up to a half a second. We did not observe a significant change in power usage depending on whether hardware acceleration was turned on or off. Below are graphs of our data.

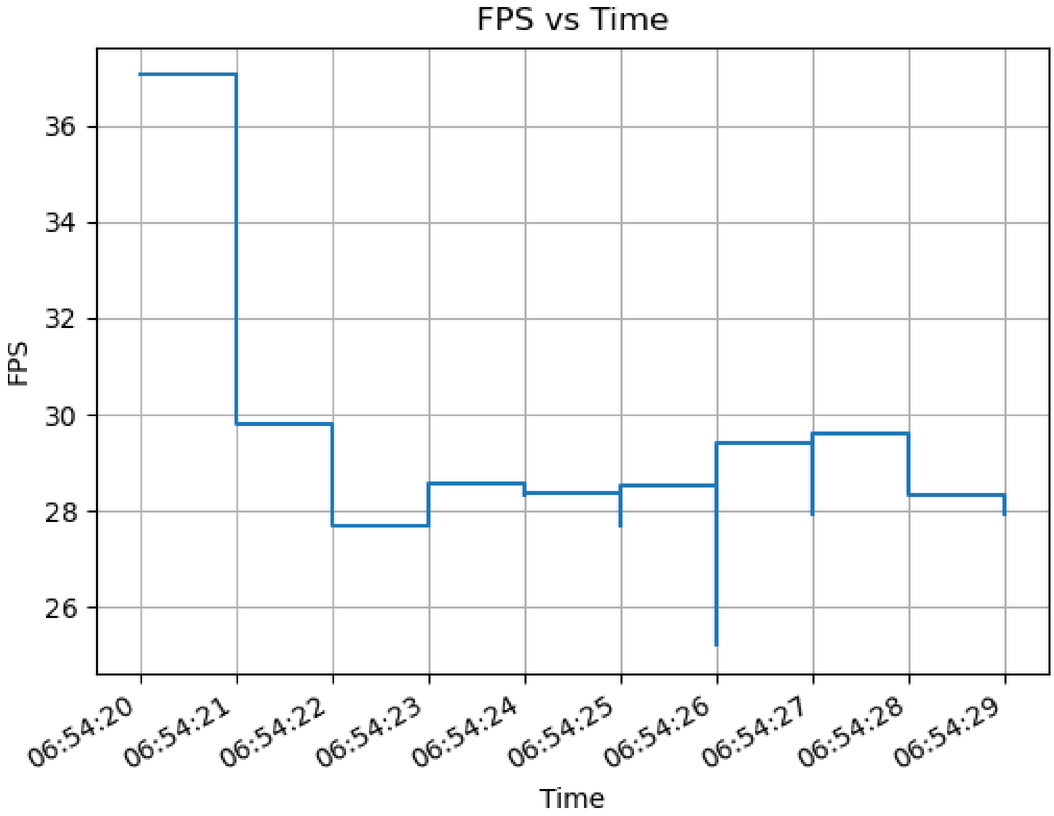
Steady State Throughput



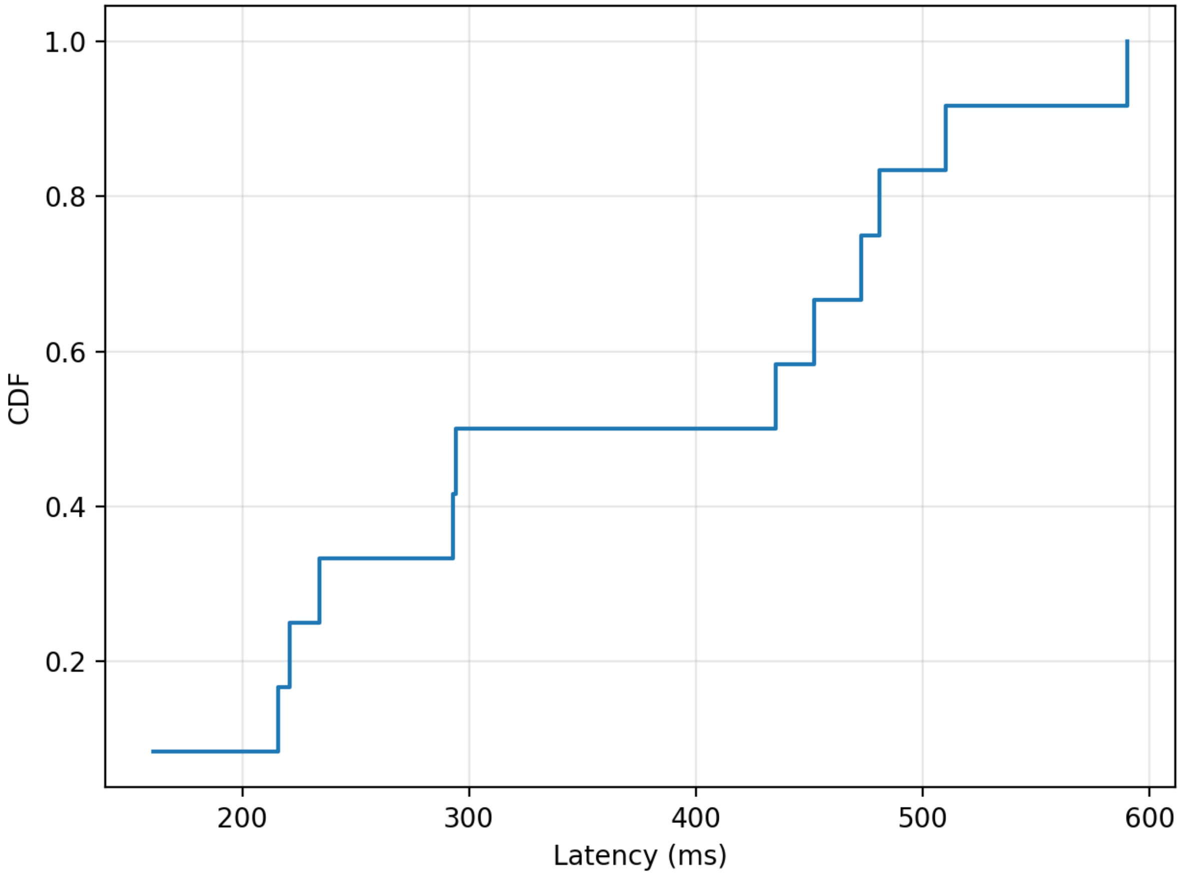
ECDH Latency



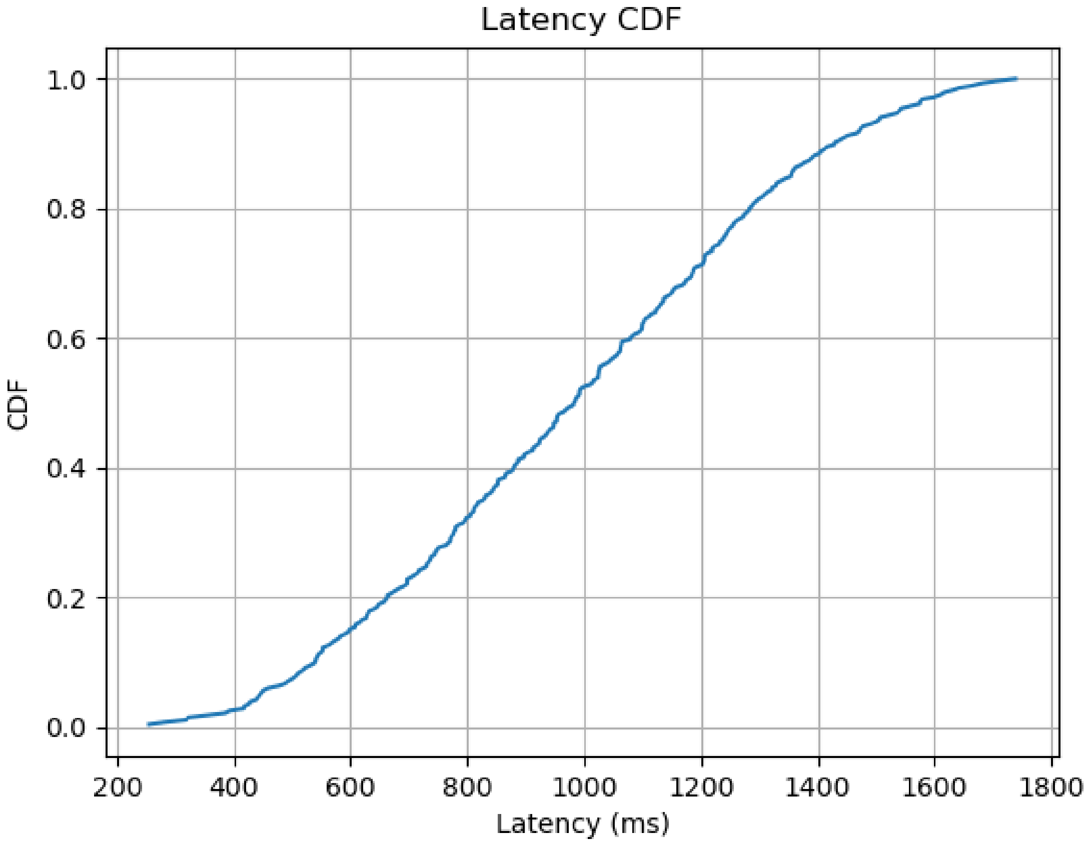
Steady State FPS



RSA Latency



Steady State Latency



Discussion

Because we did not observe any significant changes in throughput or execution time with respect to enabling or disabling hardware acceleration, we believe that the Pi crypto extension does not provide significant value to this type of application. This is likely because for this sort of application, the video processing compute dominates the encryption compute. So, the crypto accelerator on the raspberry pi is likely of *limited* utility.