eCTF 2025 - Embedded Systems Design

26 Jan 2025

Chapman, Ethan; Lawrence, Jacob; Ly, Jason; Dalpias, Denver; Miller, Jake; Hubert, Ty; and Watson, Jon

Department of Electrical and Computer Engineering, Department of Computer Science, United States Air Force Academy, Colorado Springs, CO, USA

Advisor: Baek, Stanley

Department of Electrical and Computer Engineering, United States Air Force Academy, Colorado Springs, CO, USA

Contents

Abstract

This technical document presents the design and implementation of a secure embedded system for the 2025 MITRE Embedded Capture the Flag (eCTF) competition, a challenge that simulates real-world cybersecurity threats in embedded systems. Our project integrates robust cryptographic measures and access control mechanisms while maintaining functionality and performance efficiency. The system is designed to securely encode, transmit, and decode data in a controlled environment, preventing unauthorized access and ensuring data integrity. We used Rust for its memory safety and concurrency advantages. Our approach employs AES-256 encryption, RSA-based message authentication, and SHA-256 hashing to safeguard sensitive information. The document details our multi-phase development strategy, including design, validation, and adversarial testing, to reinforce the resilience of our solution. Through this competition, we aim to advance our understanding of secure system architecture, cryptographic protocols, and reverse engineering techniques, preparing for real-world cybersecurity challenges in embedded applications.

1 List of Symbols and Abbreviations

The following contains many of the protocols and symbols intended for use within our design, as explained throughout this initial design document.

- **AES256** - Stands for Advanced Encryption Standard 256 bits
- **RSA** - Stands for Rivest-Sharmir-Adleman
- **SHA256** - Stands for Secure Hash Algorithm 256 bits
- **UART** - Universal Asynchronous Receiver/Transmitter
- **S** - Subscription key
- **P** - Encrypted packets sent via UART
- K_d - Decoder-dependent key produced by SHA256
- **d** - Decoder ID number
- **M** - Array of bit-masks
- t_s - starting timestamp

2 Introduction

Learning Objectives:

1. Gain a comprehensive understanding of embedded system vulnerabilities and learn to integrate security measures to protect against different attacks.
2. Develop the ability to design secure systems that balance functionality and security.
3. Acquire proficiency in using hardware and software tools to develop embedded systems.
4. Enhance skills in reverse engineering, secure coding, and deepen technical expertise in cyber security practices/protocols.
5. Foster teamwork and communication skills by collaborating in a multiphase competitive environment involving design, validation, defense, and attack.

2.1 Project Overview

This document details the design and implementation of a secure embedded system for a Satellite TV broadcasting solution. The system must securely encode and decode satellite TV data streams while preventing unauthorized access to protected channels.

2.2 Problem Statement

How can embedded systems be designed to seamlessly integrate robust security measures while maintaining functionality and meeting customer requirements in real-world applications?

2.3 Description

This capstone explores the integration of security and functionality in embedded systems, focusing on addressing vulnerabilities while meeting performance and usability standards, particularly ensuring that the customer's personal information is kept confidential from the adversary's grasp. The team will use this respective first semester to prepare and learn to understand embedded systems and learn the attack vectors; meanwhile, when the second semester rolls out, the team will be using specialized hardware and software tools to design a system that is both secure and efficient with a clear focus on balancing technical rigor and user satisfaction. Our customer is MTIRE, the eCTF competition host.

2.4 Phases

Our customer has delegated time constraints in three different phases: Design Phase, Hand-Off Phase, and Attack Phase. Each respective phase has its own goals and purposes.

1. **Design Phase** - During this initial phase, our team will work in a defined time frame to design and implement a secure embedded system based on the security requirements. For the 2025 competition, this involves our team creating a satellite TV system that securely decodes encrypted data streams while protecting against unauthorized access. Furthermore, our team will focus on integrating robust security measures into our embedded system while ensuring it meets all the functional specifications. Therefore, this phase emphasizes secure system architecture, cryptographic implementation, and usability design.
2. **Hand-Off Phase** - Once our team decides that our design is sufficient, we will submit our system to MITRE for proper evaluation. In this phase, the device must pass rigorous functionality testing to demonstrate it performs as intended. Additionally, the system must comply with at least one security requirement. Therefore, there is no explicit requirement to meet all security requirements to pass the evaluation. So, this phase ensures that the system is practical and secure; otherwise, our team will have to redesign our system until it passes the respective test.
3. **Attack Phase** - During this phase, the competition focuses on attacking, where teams will analyze and attempt to exploit vulnerabilities in the systems designed by other contractors (teams). Our team needs to pass the Hand-Off Phase testing to proceed into this phase. Therefore, this phase provides a hands-on opportunity to understand adversarial preservatives, uncover weaknesses, and learn how to reverse engineer systems.

Figure 1: 2025 MITRE eCTF Competition Timeline

2.5 Objectives

1. Ensure secure encoding of TV frames for transmission over the satellite.
2. Implement secure decoding to prevent unauthorized access to protected channels.
3. Design robust subscription management to control access rights.
4. Secure all communication between the encoder and decoder to prevent interception or tampering.

2.6 Hardware and Software Used

Microcontroller: MAX FTHR 78000eval kit Crypto Modules Used: AES-256, SHA-128, PKCS, Rust memory safety against reverse engineering

3 System Architecture

3.1 High-Level System Diagram

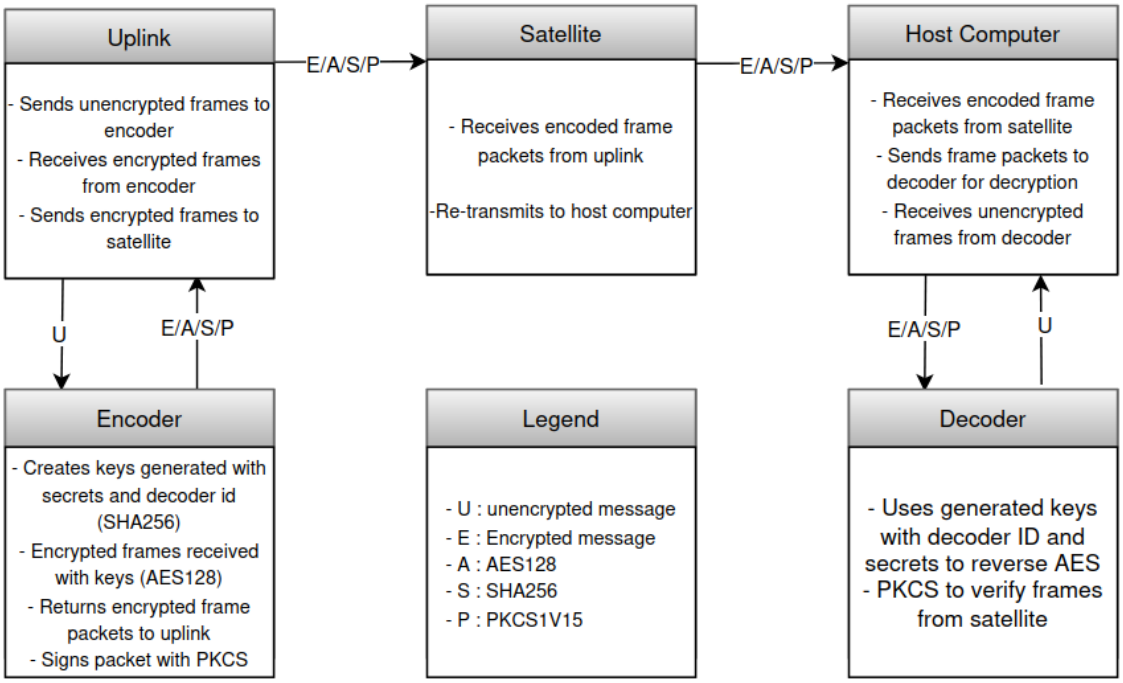

Figure 2: eCTF 2025 System Design

3.2 System Components

- Uplink: The Uplink component of the Satellite TV system is responsible for sending frames to the Satellite to be broadcast to all Decoders. The Uplink will first send raw frames to the Encoder and will then forward the encoded frames to the satellite. The Uplink is implemented as a Python script from the ectf25 module, networked with the Encoder and Satellite.
- Encoder: Implemented an Encoder design (to be discussed in design section) utilized by the Satellite System. Generated TV data frames are passed through the Encoder before being broadcast to to a listening Decoder. The Encoder is implemented as a Python script from the ectf25 design module, networked with the Uplink.
- Satellite Communication: The Satellite component of the Satellite TV System will run in a cloud environment managed by the eCTF Organizers. The Satellite takes encoded frames sent to it by the Uplink and forwards them to all listening TVs running on Host Computers. The Satellite is implemented as a Python script from the ectf25 module, networked with the Uplink.
- Host Computer: The host computer is a general-purpose computer (i.e., your laptop or desktop) used to communicate with the Decoder over a serial interface through a number of Host Tools. These tools will be used to initiate the various functionalities of the Decoder device and to read back status messages and data.
- TV: The TV is the primary functionality running on the Host Computer. It connects to the Satellite to

receive encoded packets, forwards them to the Decoder to be decoded, and prints the decoded frames to the screen. The TV is implemented as a Python script from the ectf25 module.

- Decoder: The main focus of your our work is on the Decoder device. Our Decoder is implemented on an Analog Device MAX78000FTHR development board connected to a Host Computer over a USB-serial interface. The Decoder will be used to decode data being live streamed over a uni-directional communications link across multiple channels. The Decoder is responsible for maintaining active subscriptions and correctly decoding data frames on subscribed channels.

3.3 System Requirements

As a team, we decided to standardize our development environment by using Linux across all systems to ensure seamless collaboration and effective troubleshooting. All devices were wiped, and a fresh install of Linux was performed. By working within a standardized development environment, we eliminated inconsistencies that could arise from varying operating systems, tool configurations, or dependencies. This approach allows us to replicate issues precisely, streamline debugging processes, and ensure comparability with the tools required for embedded system development and security testing. The shared environment fostered better teamwork, reduced setup discrepancy, and additionally allowed us to focus on solving technical challenges collaboratively!

1. **Git** - An open-source version control system. This allows our team to collaborate on a single code base while actively managing a history of all the distinctive changes one or many people have made. Git is required for the eCTF to submit our team's design for testing and progression to the Attack Phrase.
2. **Open On-Chip Debugger (OpenOCD)** - An open-source program that provides control and access to the DAP (Debug Access Point) on a respective microcontroller. Therefore, in this competition, the program allows for debugging and flashing firmware onto the embedded system, enabling direct interaction with the hardware during development and testing.
3. **arm-none-eabi-gcc** - A version of the GNU Compiler Collection (GCC) specifically designed for compiling code for ARM-based microcontrollers and processors, typically used in embedded systems. It is a cross-compiler used to compile C or C++ code for ARM-based embedded systems that do not rely on an operating system.
    - **arm** - Indicates that this toolchain is intended for ARM architecture.
    - **none** - Refers to the absence of an operating system intended for bare-metal programming.
    - **eabi** - Stands for "Embedded Application Binary Interface," which is a standard that defines how programs interact with the respective system's hardware and the ABI (Application Binary Interface).
    - **gcc** - Refers to the GNU compilers Collection, an open-source set of compilers for various programming languages, including C and C++.

    It is a cross-compiler used to compile C or C++ code for ARM-based embedded systems that do not rely on an operating system. Therefore, it is used to build and compile the source code into machine code suitable for the ARM board in the competition.
4. **make** - A build automation tool usually used in software development that manages the compilation and building of projects, particularly in environments with multiple files or dependencies.
5. **Python3.11+** - A version of the Python programming language, an interpreted, high-level, and general-purpose language.

6. **Docker** - An open-source platform that automates applications' deployment, scaling, and management inside lightweight, portable containers.
    - **Containers** - A container is an isolated environment that includes everything needed to run an application, such as code, libraries, and dependencies.
    - **Docker Images** - A Docker Image is a snapshot of a container, essentially a template used to create containers.
    - **Docker Engine** - The Docker Engine is the runtime that manages Docker containers. It allows users to build, run, and manage containers on their machine or in the cloud.

4 General Requirements

4.1 Security Requirements

1. **Security Requirement 1** - An attacker should not be able to decode TV frames without a Decoder with a valid, active subscription to that channel.
    - Knowledge of the plain text contents of one encoded frame should not allow an attacker to violate this requirement for other frames
    - A valid subscription refers to a subscription installed through a subscription update, generated at the same Secure Facility matching this device's ID and selected channel.
    - An active subscription refers to a subscription for a channel where the current time falls within the active subscription window.
2. **Security Requirement 2** - The Decoder should only decode valid TV frames generated by the Satellite System the Decoder was provisioned for.
    - During the Attack Phase, the organizers will only create one Satellite System per team (i.e., all systems built using the same generated secrets).
3. **Security Requirement 3** - The Decoder should only decode frames with strictly monotonically increasing timestamps.
    - The timestamps do not need to be sequential, and our system should reject disordered frames if this would violate the security requirement
    - This requirement does not need to apply across power cycles

4.2 Functional Requirements

1. Build: the system, environment, and deployment, and decoder. Build is implemented with Docker.
2. Encoder: It is implemented as a function that accepts the raw frame to encode, the channel it will be sent on, and the timestamp of the frame.
3. Decoder: respond to update subscriptions, decode frames, and list channels commands successfully. Frames on channel 0 must always be decoded and subscriptions should be stored so that they persist if the decoder loses power and should always use the latest subscription update.

5 Security Considerations

5.1 Threat Models

Expired Subscription - A user with a previously valid subscription continues to attempt accessing encrypted TV content after their subscription has expired. This can occur if the decoder does not enforce expiration checks properly or if an attacker manipulates the system clock to extend access. **Pirated Subscription** - An attacker uses stolen or shared credentials to gain unauthorized access to encrypted TV content. **No Subscription** - An attacker without a valid subscription attempts to gain access to the encrypted TV content without purchasing a legitimate subscription. **Recording Play Back** - A user records encrypted TV frames while they have a valid subscription and later attempts to decrypt and watch them after their subscription has expired. This allows users to extend access to content beyond their paid period, bypassing subscription enforcement. **Pesky Neighbor** - An attacker spoofs the satellite signal to trick a nearby decoder into decoding frames meant for another user's subscription. This allows the attacker to watch protected channels without a valid subscription by injecting manipulated signals into the target decoder.

5.2 Threat Mitigation Techniques Used

Expired Subscription Mitigation Techniques: Implementing time based subscription expiration. Using key generation techniques to prevent the reuse of expired keys

Pirated Subscription: Using device-bound authentication with the decoder ID for decryption. Using Signature Authentication Techniques like PKCS1v15.

No Subscription: Using encryption techniques including secrets.json generation for SHA256 key generation and AES128 frame encryption. Using PKCS1v15 for message authentication.

Recording Play Back: Using session-based encryption with dynamic key creation based on time bit masks. Time stamp-based decryption to prevent replay of old content.

Pesky Neighbor Attack: Using key exchange for secure communication. Implementation of RSA through PKCS1v15 for mutual authentication between encoder and decoder.

Table 1: Encoder-Decoder Authentication Mapping

Authentication Step	Encoder Actions	Decoder Actions
Dynamic Key Encryption	characterize any range of timestamps with a set of different keys	Uses set of keys to decrypt frames using bitrange logic
Message Integrity Check	Signs authentication message using PKCS1v15	Verifies signature using the encoder's public key
Frame Key Establishment	Derives frame key using randomly generated secrets file each time with SHA256	Derives the same frame key for secure decryption with decoder ID
Frame Decryption	Encrypts video frames with AES128 and sends to decoder	Uses key to decrypt and display frames

6 Design

6.1 Rust

We opted for Rust over languages like Python or C for this competition due to its strong emphasis on memory safety without reliance on a garbage collector, which is crucial in resource-limited environments. Unlike C, where manual memory management can lead to errors like buffer overflows, Rust's ownership system ensures safe memory handling and prevents common bugs. Furthermore, Rust's concurrency model allows efficient parallel processing, such as decoding frames concurrently with other tasks, without introducing data races. While Python's higher-level nature leads to performance overhead, Rust provides low-level control with performance comparable to C, making it ideal for real-time data processing.

6.2 Security

Encoder:
S = Subscription
K_d = SHA256(secrets + d)
P = AES(S, K_d)

Decoder:
S = AES(P, K_d)
Store S for later decryption

We will store the global secrets on the host machine; they will not be uploaded or flashed to the board at any time.

6.3 Subscription Generation

To ensure that each subscription is only valid during its time range, we have devised a way to send a set of subscription keys S that each decrypts a certain range of frames. We use a bit-masking method to achieve this. We can define the following set of bit-masks M.

$$M = \{0, 1, 2, 3\}$$

Each number corresponds to the "width" of the bit-mask or how many of the least significant bits of the timestamp it covers. We can now define range values using a start timestamp and a mask index: (t_s, i). Here is an example of how that works:

$$(01010100_2, 2) \rightarrow 010101\text{XX}_2 \text{ or } [01010100_2, 01010111_2]$$

We will call this (t_s, i) a bit range. We can generalize this to represent any range of times by combining adjacent bit ranges. Here is the complete set of possible bit ranges for a 4-bit timestamp:

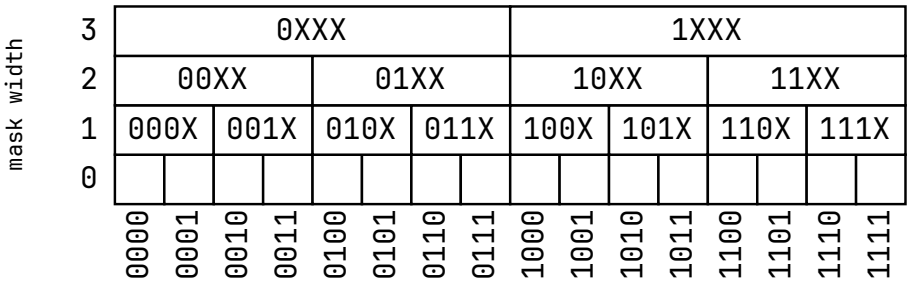

Figure 3: Possible 4-bit bit ranges

Here is an example range of timestamps encoded using bit ranges:

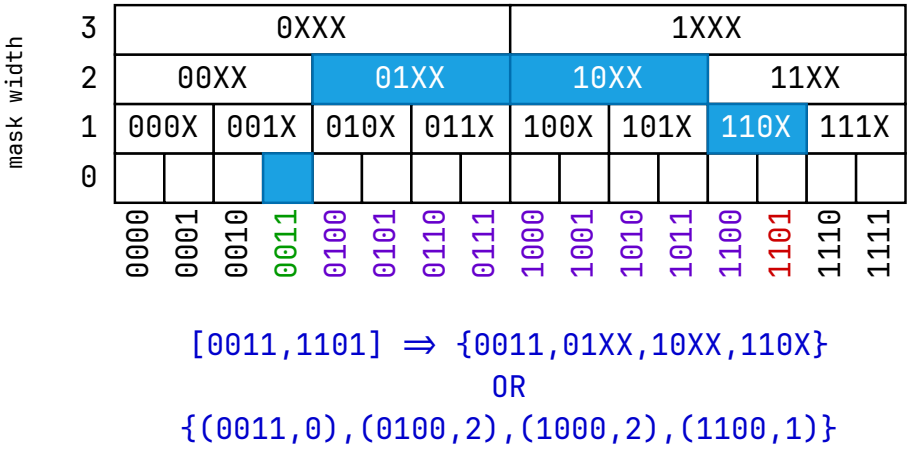

Figure 4: Time range encoded using bit ranges

We can give each bit range its channel-specific key by defining the following subscription key generation function that takes a start timestamp (t_s), a mask index (i), and a channel number (c) and hashes those data along with the global secrets:

$$K_s(\text{bitrange}, c) = \text{SHA256}(\text{secrets} + t_s + i + c)$$

When we generate a subscription, we will generate the keys of all the bit ranges required to span its time range and pair that with the bit range itself so the decoder knows when the key is valid. For the example in fig. 4, we would generate the following subscription S:

$$S = \{(0011, K_s(0011)), (01\text{XX}, K_s(01\text{XX})), (10\text{XX}, K_s(10\text{XX})), (110\text{X}, K_s(110\text{X}))\}$$

To turn this subscription into a packet we can send over the wire, we must ensure that only the decoder we want to use can understand the message. To accomplish this, each decoder will get its own 256-bit key K_d

that will be flashed to the board and is generated using the following function of the decoder id (d):

$$K_d = \text{SHA256}(\text{secrets} + d)$$

We can then encrypt our subscription with this key to obtain our packet (P):

$$P = \text{AES}(S, K_d)$$

The decoder can then use its K_d to obtain S and store this subscription for decoding.

6.4 Frame Encoding

When encoding our frames, we must do so in a way that ensures any K_s for a bit range that spans the frame's timestamp can decode the frame. For example, let us say we are trying to encode a frame F at timestamp 0110_2. This would result in the following bit ranges needing to be able to decode the frame:

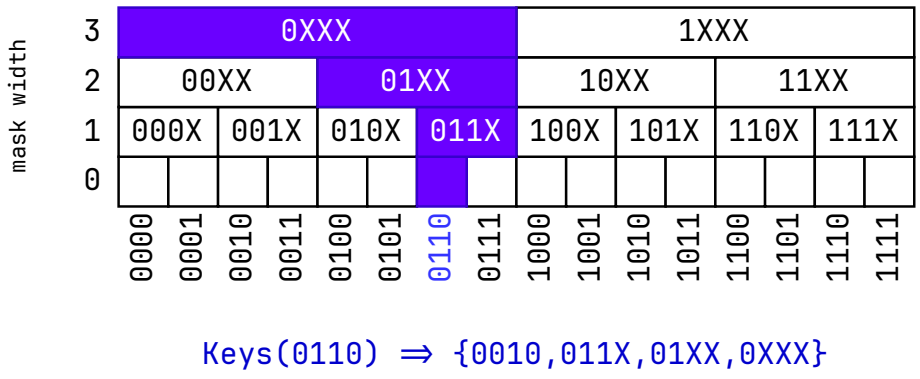

Figure 5: Bitranges corresponding to a timestamp

We will define a frame key that we will use to encrypt this frame:

$$K_f = \text{SHA256}(\text{secrets} + t + c)$$

Then we can encrypt this key with each of the subscription keys that should be able to decrypt the frame, resulting in the following encoded frame key data (E):

$$E = \{\text{AES}(K_f, K_s(0110)), \text{AES}(K_f, K_s(011\text{X})), \text{AES}(K_f, K_s(01\text{XX})), \text{AES}(K_f, K_s(0\text{XXX}))\}$$

This data would then be broadcast along with the frame encrypted with K_f for decoders to decode.

6.5 Frame Decoding

The decoder received the above-encoded frame E at timestamp 0110_2. This decoder also has the above subscription S. Since the decoder has S, it has $K_s(01\text{XX})$, which can be used to decode this frame key. Key

01XX has an i (mask width index in M) of 2, which means it is at index 2 in E (the third element due to zero-indexing). This allows the decoder to decrypt the frame key using AES:

$$K_f = \text{AES}(E_2, K_s(01\text{XX}))$$

We can substitute the value of E_2 into this equation to prove that this will work since the symmetric AES operations cancel out:

$$F = \text{AES}(\text{AES}(K_f, K_s(01\text{XX})), K_s(01\text{XX})) = K_f$$

Now that we have recovered K_f, we can use it to decrypt the frame.

6.6 Message Authentication

We will use PKCS1v15 to verify encoded frames, and a SHA256 hash of the unencrypted subscription data to verify subscriptions. We can use simple SHA256 to verify subscriptions because only an entity holding K_d can control unencrypted subscription hashes. This works since the subscription keys are encrypted with K_d, and the decoder decrypts them before hashing.

6.7 Optimizations

In all of the above examples, we used all available bit ranges. However, we can skip some by defining a set of masks like this:

$$M = \{0, 2, 4, 6, \ldots\}$$

As long as we have a mask width of zero, we can characterize any range using these bit ranges. Here is an example of this:

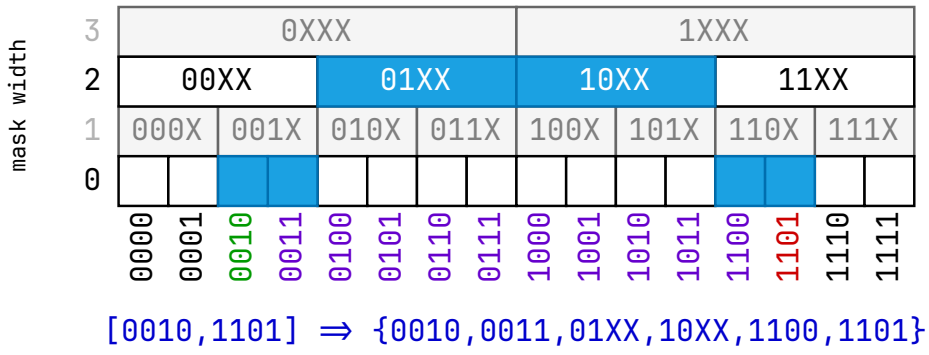

Figure 6: Time range encoded while skipping bit ranges

Skipping bit ranges allows E to be smaller since there are fewer possible bit ranges for a given timestamp, but S has to be larger since more bit ranges are needed to cover most time ranges.

7 Hardware Implementation

This project is written in Rust and utilizes cross-compilation to target the ARM Cortex-M4F architecture. Various tools and dependencies have been configured to build and flash firmware onto the MAXFTHR board. The target platform is `thumbv7em-none-eabihf`, corresponding to the ARM Cortex-M4F architecture with hardware floating-point support.

7.1 Dependencies

Key dependencies (specified in `Cargo.toml`):

- `max7800x-hal`: Hardware Abstraction Layer for low-level peripherals.
- `cortex-m` and `cortex-m-rt`: Libraries for ARM Cortex-M runtime handling.
- `embedded-hal-nb`: Trait-based abstraction for embedded peripherals.
- `panic-halt`: Halts the program on panics, suitable for bare-metal systems.
- `rkyv`
- `sha2`
- `rsa`
- `aes`

7.2 Flashing and Debugging the Firmware

Firmware development, flashing, and debugging are facilitated by OpenOCD and GDB.

7.3 OpenOCD Configuration

Below is the configuration file for OpenOCD:

```
source [find interface/cmsis-dap.cfg]
source [find target/max78000.cfg]

gdb_port 50000
tcl_port 50001
telnet_port 50002
```

- `cmsis-dap.cfg` is used for the CMSIS-DAP debugging interface.
- The `max78000.cfg` file specifies the MAXFTHR board configuration.

7.4 GDB Commands

Here is an example GDB script to debug the firmware:

```
target extended-remote :50000
break DefaultHandler
break HardFault
break main
monitor arm semihosting enable
load
stepi
```

This script:

- Sets up breakpoints to debug crashes (e.g., hard faults).
- Loads the firmware onto the MAXFTHR.
- Facilitates stepping through instructions for debug analysis.

7.5 Cryptographic Hardware and Secure Communication

This project leverages the cryptographic functionality of the MAXFTHR board to secure communication between devices.

7.6 Key Generation

Keys for secure communication are generated using cryptographic hash functions:

```
def gen_key(secrets: bytes, start_timestamp: int,
            mask_width: int, channel: int, decoder_id: int) -> bytes:
    return hashlib.sha256(secrets + struct.pack("<QBBI",
           start_timestamp, mask_width, channel, decoder_id)).digest()
```

7.7 Deployment Pipeline

The deployment pipeline involves three primary stages:

1. **Build the Firmware**: The firmware is compiled for `thumbv7em-none-eabihf`.
2. **Binary Conversion**: The binary is converted to a flashable file using `arm-none-eabi-objcopy`.
3. **Flashing and Debugging**: OpenOCD and GDB are used to flash and debug the firmware.

7.8 Python Bindings for High-Level Usage

The project provides Python bindings to simplify high-level interaction with the encoder API. Here is an example:

```
from ectf25_design_rs import Encoder
encoder = Encoder(secrets_file.read())
encoded_frame = encoder.encode(channel, frame.encode(), timestamp)
```

8 Testing and Benchmarks

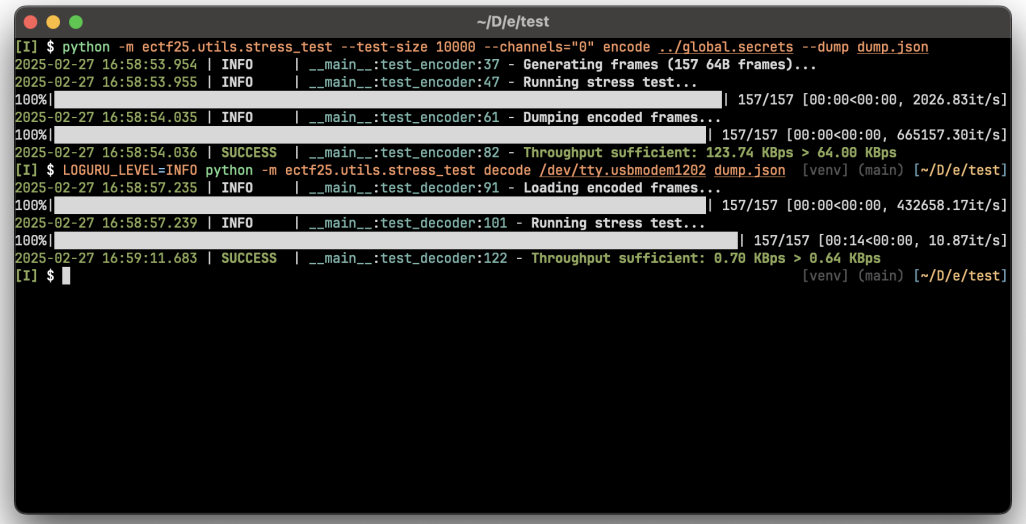

Figure 7: Throughput test results showing encoder and decoder performance.

8.1 Functional and Security Requirements Compliance

To ensure the secure Satellite TV system meets all functional and security requirements, we conducted extensive testing on timing, throughput, component size, and channel constraints. The throughput test results, shown in Figure 7, confirm that our implementation meets the required performance benchmarks.

8.2 Timing Requirements

The system adheres to strict timing constraints to guarantee real-time performance. The following operational timing requirements were met:

Table 2: Timing Requirements Compliance

Operation	**Maximum Time for Completion**
Device Wake	1 second
List Channels	500 milliseconds
Update Subscription	500 milliseconds
Decode Frame	150 milliseconds

The decoder successfully maintained a maximum frame decode time of 150 ms, ensuring seamless playback without noticeable delay.

8.3 Throughput Requirements

As demonstrated in Figure 7, the system meets the required throughput constraints for both encoding and decoding:

Table 3: Throughput Requirements Compliance

Operation	Minimum Throughput
Encoder	1,000 64B frames per second
Decoder	10 64B frames per second (5s rolling avg)

8.4 Component Size Requirements

The system was designed to stay within memory constraints, with all components adhering to the specified size limits:

Table 4: Component Size Compliance

Component	Size
Channel ID	32b unsigned int
Decoder ID	32b unsigned int
Timestamp	64b unsigned int
Decoded Frame	Max 64 bytes

8.5 Channel Availability Requirements

The decoder was verified to support the required number of standard and emergency channels, ensuring full functionality.

Table 5: Channel Availability Compliance

Channel Type	Required
Standard Channels	8 channels
Emergency Channel	1 channel (always Channel 0)

8.6 Security Compliance

Beyond performance constraints, the system integrates robust security mechanisms to prevent unauthorized access and protect transmitted content:

- **Expired Subscription Prevention:** Implemented time-based subscription expiration and key rotation to prevent reuse of expired keys.
- **Pirated Subscription Prevention:** Uses device-bound authentication with decoder ID validation and PKCS1v15 signature authentication.

- **No Subscription Prevention:** Enforces SHA-256 key generation from ‘secrets.json‘ for AES-128 encrypted frames , with PKCS1v15 authentication for message integrity.
- **Recording Playback Prevention:** Implements session-based encryption with dynamic key creation based on time bit masks and timestamp-based decryption to prevent replay attacks.
- **Pesky Neighbor Attack Prevention:** Uses secure key exchange and RSA-based mutual authentication (PKCS1v15) to protect against signal spoofing.

8.7 Conclusion

Through rigorous testing, we have validated that the secure Satellite TV system meets all functional and security requirements. The throughput test results in Figure 7 confirm that both the encoder and decoder operate within the required constraints, ensuring real-time, secure, and uninterrupted performance.

9 Future Work

While our current system meets functional and security requirements, future improvements can focus on mitigating hardware-based exploits that could compromise the integrity of the decoder. Embedded systems are susceptible to attacks such as side-channel analysis, fault injection, and hardware tampering.

By integrating these hardware-level security measures, future iterations of this system can further protect against advanced embedded system exploits, ensuring robust, tamper-resistant security.