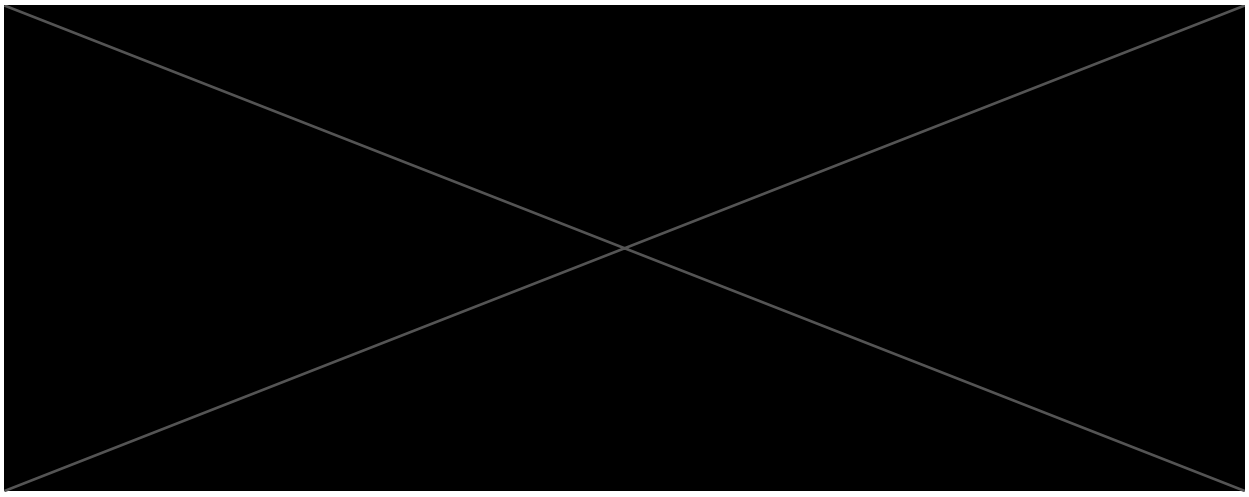
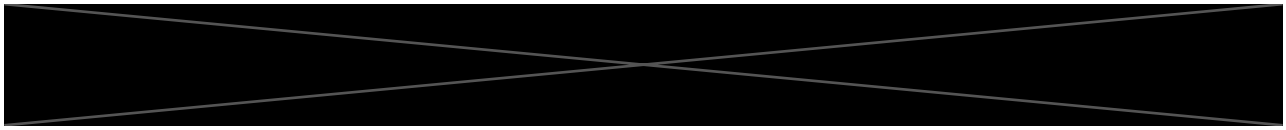
2025 MITRE eCTF
Design Document for [redacted]



Abstract

The 2025 MITRE Embedded Capture the Flag (eCTF) competition challenges teams to design and implement a secure Satellite TV System, ensuring the confidentiality, integrity, and authenticity of TV data streams. This document outlines the functionality and security requirements of our system, detailing our approach to implementing robust security mechanisms within the Decoder and Encoder firmware.

1. Introduction



1.1. Initial Considerations

Our design is built using C. We decided that switching to another language would pose the issue of having to maintain a new build toolchain and having to rely on third-party ports of the MSDK HAL, if there existed any. In addition, the C language is simple to understand, allowing all team members regardless of experience to contribute to the design. Languages such as Rust offer a robust type system and enhanced security features, including protection against buffer overflows. However, this would be an unnecessary risk given the steep learning curve of the language and additional infrastructure needed.

2. Build Environment and Setup

The build environment will host all of the required tools, packages, dependencies, and other developer tools necessary for the device to run and operate. Docker will be the main way these dependencies will be added and used in the design. Docker is a containment management system that will create a reproducible build environment (container) that compiles code exactly the same way, regardless of the host platform. The design environment is created using a "*docker build*" based on the requisite packages provided in the "*DockerFile*" by the competition organizers.

The dependencies included in the "*DockerFile*" are *gdb, gdb-multiarch, gcc-arm-none-eabi, binutils-arm-none-eabi, make, python3.10, git, wget, libtool, and texinfo.* The Docker deployment will need to be created after the secrets folder is generated. Once the Docker container has been successfully created, a subscription update will need to be generated. The Decoder firmware will have to be flashed onto the MAX78000FTHR boards, which is needed to build the design.

3. Functional Requirements

This section outlines the functional requirements of the design. The system consists of an Encoder that is responsible for encoding and broadcasting TV frames via Uplink, and a Decoder responsible for decoding and displaying TV frames for authorized users. The final design must adhere to all functional requirements before progressing to the attack phase.

3.1. Architecture

The architecture of the system is built around the Analog Devices MAX78000FTHR development platform, which serves as the hardware foundation for the Decoder. The MAX78000FTHR provides the necessary processing power and connectivity to support the system requirements.

3.2. Building Process

Global Secrets are generated by the host tool. There is a Docker based environment to install dependencies, compilers, packages, and build tools. The Decoder firmware is built from the design and flashed onto the MAX78000FTHR. Subscription updates are generated including channel numbers, Decoder IDs, timestamps, and information from Global Secrets.

3.3. Encoder Requirements

Once the system is built, the Encoder can be started. The Encoder is implemented as a function with raw frames, channel number, and secrets as inputs. The Encoder prepares the frame for Uplink by encoding it with the necessary metadata.

3.4. Decoder Requirements

With a functioning Decoder, the system is ready to accept signals from the satellite TV system. The Decoder must be able to respond to the list channels, update subscriptions, and decode frame commands. Using the Decoder from the reference design, and adding security measures on top of that would satisfy the functional requirements.

4. Cryptographic Primitives

In this section, we outline the cryptographic primitives used to fulfill the security requirements of our design. We selected well-established, standardized algorithms that have undergone rigorous research and cryptanalysis. This ensures our strong confidence in their algorithmic security.

4.1. HMAC

Hash-based message authentication codes [1] are used to verify message integrity. It achieves this by using a secret key and a hash function to generate a tag unique to the message. The HMAC is included with the outgoing message to allow the receiver to verify message integrity. The receiver does this by re-computing the HMAC and comparing tags.

Algorithm 1 HMAC Verification

Require: Message M, Key K, Received MAC T_{received}
Ensure: Valid or Invalid
1: computed_mac $\leftarrow$ **HMAC**(M, K)
2: **if** computed_mac $= T_{\text{received}}$ **then**
3: **return** Valid
4: **else**
5: **return** Invalid
6: **end if**

4.2. AES-CBC

The Advanced Encryption Standard is a block cipher utilizing a symmetric key to both encrypt and decrypt data. Since the cipher purely operates on a block of 16 bytes, it requires a mode of operation in order to operate on larger data. AES-ECB is the simplest yet most insecure mode of operation, which uses the same key to operate on all blocks leading to cases such as patterns in the plaintext mapping to patterns in the ciphertext, revealing information that should otherwise be hidden. A famous example is the "ECB Penguin [2]" illustrating this effect.

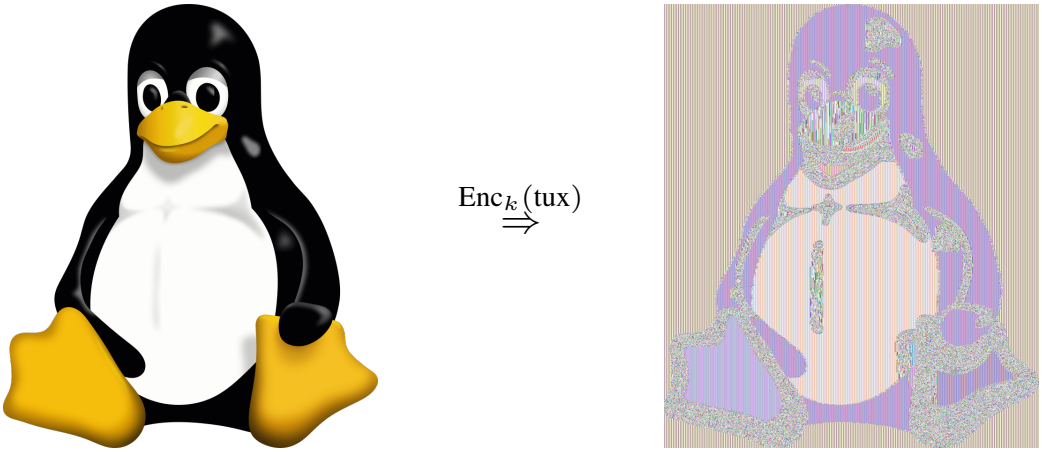


Figure 1: Image encryption using AES-ECB resulting in visually similar images [2].

We chose to use AES-CBC [3] (Cipher Block Chaining) using an IV (Initialization Vector) to introduce randomness and prevent the aforementioned ECB problem. We also use a PKCS#7 [4] padding scheme to ensure that any sized data can be decrypted. Upon decryption failure, a generic error message is returned to reduce the risk of padding oracle attacks. The CBC mode operation for encryption and decryption is defined as follows:

Encryption with PKCS#7 Padding

Given plaintext blocks $P_1, P_2, \ldots, P_n$, initialization vector IV, and block cipher E_k, the plaintext is first padded using PKCS#7 to ensure its length is a multiple of the block size. Let $P = P_1 \| P_2 \| \cdots \| P_n$ be the original message, and $\widetilde{P} = \textbf{PKCS7}(P)$ the padded plaintext, split into blocks $\widetilde{P}_1, \ldots, \widetilde{P}_m$.

$$
\begin{aligned}
C_0 &= IV \\
C_i &= E_k(\widetilde{P}_i \oplus C_{i-1}) \quad \text{for } i = 1, 2, \ldots, m
\end{aligned}
$$

Decryption with PKCS#7 Padding

Given ciphertext blocks $C_0, C_1, \ldots, C_m$ and block cipher D_k:

$$
\widetilde{P}_i = D_k(C_i) \oplus C_{i-1} \quad \text{for } i = 1, 2, \ldots, m
$$

After decryption, the original plaintext P is recovered by removing the PKCS#7 padding:

$$
P = \textbf{RemovePKCS7}(\widetilde{P}_1 \| \cdots \| \widetilde{P}_m)
$$

In addition, CBC by itself does not have an authentication mechanism like GCM, so we employ HMAC in an encrypt-then-HMAC fashion. This approach requires separate keys for encryption and authentication unlike GCM which uses a single key for both. Even if the HMAC key is compromised attackers cannot decrypt messages due to this key separation.

4.3. PBKDF2

Password-based key derivation function [5] generates a key from a password and a salt by appending the salt to the password and repeatedly hashing the value. The number of hashes performed is called the iteration count and derives security from its key stretching process making brute-force attacks computationally expensive.

5. Security Requirements

This section details the necessary security requirements the design must follow and how we implement each of them. The following requirements are put in place to ensure the protection of data and functionality critical to the secure system operating as intended. To prevent unauthorized access an attacker should not be able to decode TV frames without a Decoder that has a valid, active subscription to that channel. Additionally, to maintain system integrity the Decoder should only decode valid TV frames generated by the Satellite System the Decoder was provisioned for. This requirement ensures that our system cannot be used to decode frames in any context other than what our team has designed it for. Lastly, the Decoder should only decode frames with strictly monotonically increasing timestamps. This will ensure that frames are only processed and displayed in the order intended by the TV station.

5.1. Security Requirement 1: Channel Subscription Verification

For this requirement, the Decoder should only be able to decode TV frames from a channel it has an active, valid subscription for. This requirement is crucial for protecting our system from attackers using pirated and out of date subscriptions to access frames from channels they are not permitted to use.

The question arises of how to ensure that only the intended decoder receives the subscription data. To begin, outgoing subscription update packets need to be encrypted with a symmetric key only the Decoder and Encoder know. This key is called the subscription update key K_s and is derived from a salt S_s from Global Secrets and the Decoder ID. The salt S_s is generated using the Host's CSPRNG (**urandom**).

$$K_s := \textbf{Hash}(S_s \parallel \text{DecoderID})$$
$$P_s := \textbf{Dec}_{K_s}(C_s)$$

This guarantees that only valid and intended Decoders can decrypt subscription update messages. The subscription update payload can now be decrypted and written to the Decoder's flash. Upon every frame decode, the provisioned Decoder checks to see if the current frame is within the valid time window specified by the subscription, if not it rejects the frame.

Subscription Update Decrypted Payload

Name	Offset	Size (bytes)
Device ID	0x00	4
Start Timestamp	0x04	8
End Timestamp	0x0C	8
Channel ID	0x14	4
Channel Key	0x18	16

5.2. Security Requirement 2: TV Frame Validation

For this requirement, the Decoder should validate TV frames before decoding them, making sure that the TV frames were generated by the satellite system for which the Decoder was provisioned for. With this, TV frames injected by a malicious party who do not have control of the satellite system will not be decoded, preventing malicious operations from being run on the Decoder.

We utilize HMAC tags embedded in every message to verify integrity. The Global Secrets contain an HMAC key generated using **urandom**. This guarantees that only valid decoders can verify message authenticity using the shared key. We utilize HMAC with the SHA256 hash algorithm. HMAC computation is as follows:

Let M be the message payload, τ the HMAC tag, and K_h the HMAC key.

$$\tau = \textbf{HMAC}(M, K_h)$$

Generic Message Structure

Name	Offset	Size (bytes)	
Channel ID (?)	0x00	4	(Frames only)
HMAC Tag	0x00	32	
IV	0x20	16	
Payload	0x30	??	

Incoming messages to the Decoder are then checked for authenticity with **HMACVerify**(M_p, τ, K_h) with M_p being the message payload.

5.3. Security Requirement 3: Frame Order Integrity

This requirement ensures that the Decoder will only decode frames with strictly monotonically increasing timestamps, preserving the correct order of frames as intended by the TV station.

To ensure this, the Decoder validates that each incoming frame's timestamp is strictly greater than the timestamp of the last successfully processed frame. Frames with older or equal timestamps are discarded. Additionally, to prevent replay attacks, the integrity of the timestamp is secured by HMACs, ensuring that the timestamp cannot be altered or replayed. As a result, frames with invalid timestamps are automatically discarded, ensuring that the frames are decoded and displayed in the order intended.

However, there is still the problem of the "first timestamp". Since we maintain a previous timestamp value, if we assume the value is 0 upon initialization, then an incoming timestamp of 0 will be rejected. We solve this by saving whether we have received a frame for a specific channel, if not we accept any incoming frame. This sets the reference timestamp for future frames.

Let $B \in \{0,1\}^8$ denote a byte representing the frame reception status for 8 channels. Each bit B_i (with $i = 0$ being the least significant bit) corresponds to channel i:

$$B_i = \begin{cases} 1 & \text{if a frame has been received for channel } i \\ 0 & \text{otherwise} \end{cases} \qquad \text{for } i = 0, 1, \ldots, 7$$

Thus, a bitwise representation like $B = \texttt{0b01001001}$ indicates that frames have been received for channels 0, 3, and 6.

Let $E \in \{0,1\}^8$ be a second byte representing the emergency frame reception status. We define:

$$E = 0xFF \iff \text{An emergency frame has been received}$$

Otherwise, if $E \neq 0xFF$, no valid emergency frame has been received.

6. Additional Mitigations

Since sensitive data is stored on the stack when processing messages, we take the precaution and zero out those values to ensure that in the case of a stack-based or RCE attack, attackers will not be able to extract leftover keys.

In addition, we reject all messages longer than 256 bytes to prevent attackers from overflowing the UART buffer. Since the maximum longest message for our protocol is less than 256 bytes, we can afford this.

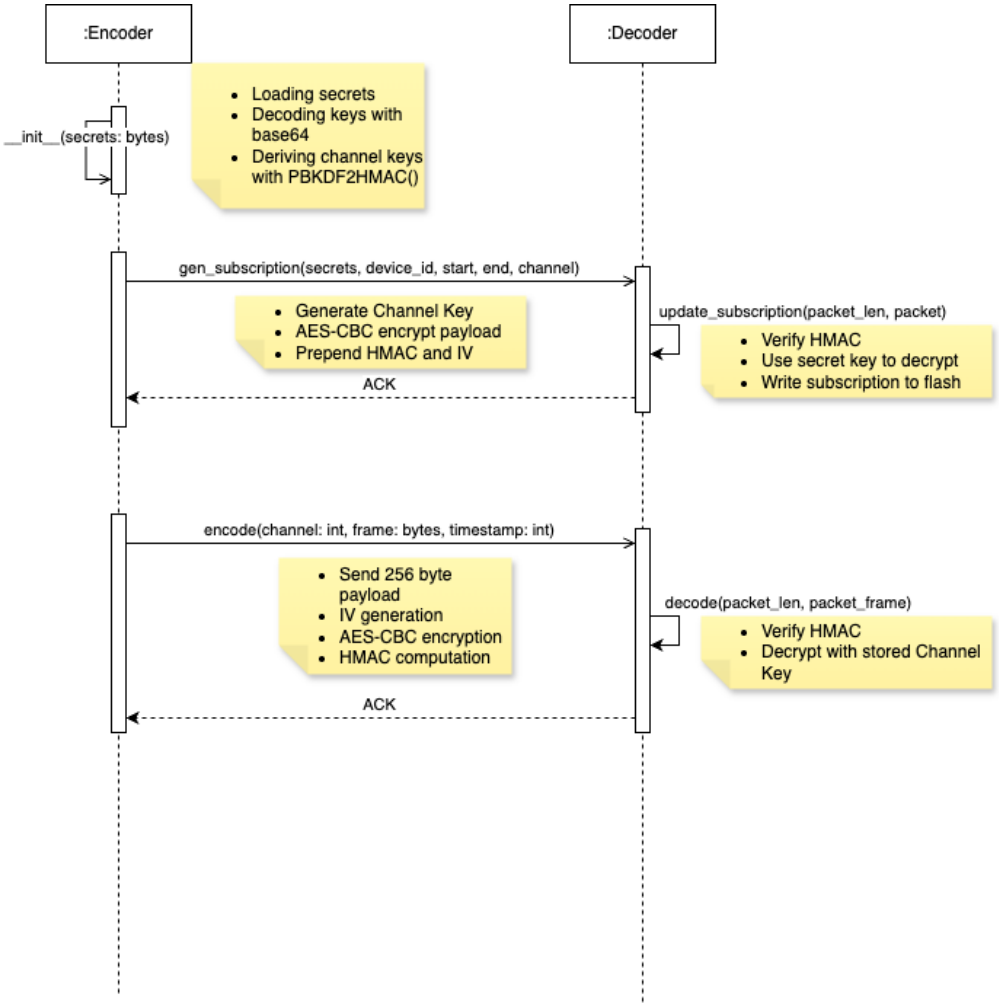


Figure 2: System Design diagram

Appendix A — Definition of Terms

Term	Definition
Frame	Actual TV image data. Guaranteed to be a maximum of 64 bytes.
Encoder	Device that generates and broadcasts encoded frames to be received by Decoders.
Decoder	Device that is responsible for receiving and decoding the frames accordingly.
Host	Trusted system designed to contain Encoder infrastructure and broadcast messages to the Decoder via UART.
Global Secrets	Shared information between the Encoder and Decoder. This is meant to contain cryptographic secrets and other values to facilitate secure communication. Valid decoders maintain Global Secrets upon provisioning.
AES-CBC	AES (Advanced Encryption Standard) using the CBC mode of operation
SHA	Secure Hashing Algorithm
HMAC-SHA256	HMAC (Hash-based Message Authentication Code) using SHA256 as the hash function
K_c	Channel specific symmetric key
K_s	Subscription update symmetric key
K_h	HMAC key
S_s	Subscription update salt
S_c	Channel key PBKDF2 salt

References

[1] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, Feb. 1997. [Online]. Available: https://www.rfc-editor.org/info/rfc2104

[2] Wikipedia contributors, "Block cipher mode of operation — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Block_cipher_mode_of_operation&oldid=1279034580, 2025, [Online; accessed 7-April-2025].

[3] S. Frankel, K. R. Glenn, and S. G. Kelly, "The AES-CBC Cipher Algorithm and Its Use with IPsec," RFC 3602, Sep. 2003. [Online]. Available: https://www.rfc-editor.org/info/rfc3602

[4] B. Kaliski, "PKCS #7: Cryptographic Message Syntax Version 1.5," RFC 2315, Mar. 1998. [Online]. Available: https://www.rfc-editor.org/info/rfc2315

[5] K. Moriarty, B. Kaliski, and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1," RFC 8018, Jan. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8018