MITRE Embedded Capture the Flag Competition

Design Document | Final Submission

Johns Hopkins University

Version: R002

February 26, 2025

Table of Contents

Introduction

Design Overview

This document outlines Johns Hopkins University's 2025 eCTF design, which aims to provide a secure satellite television system solution. Figure 1.1 outlines the design framework, which consists of the following seven components:

(1) A secret generator which creates keys for data encryption and authentication
(2) A subscription generator to generate subscriptions for individual decoders
(3) An encoder that encodes data frames to be viewed on the TV
(4) An uplink that is responsible for passing frames to a satellite
(5) A satellite that broadcasts the encoded data received from the uplink station
(6) A television that receives data from the satellite
(7) A decoder that decodes the received data and passes it back to the TV for viewing

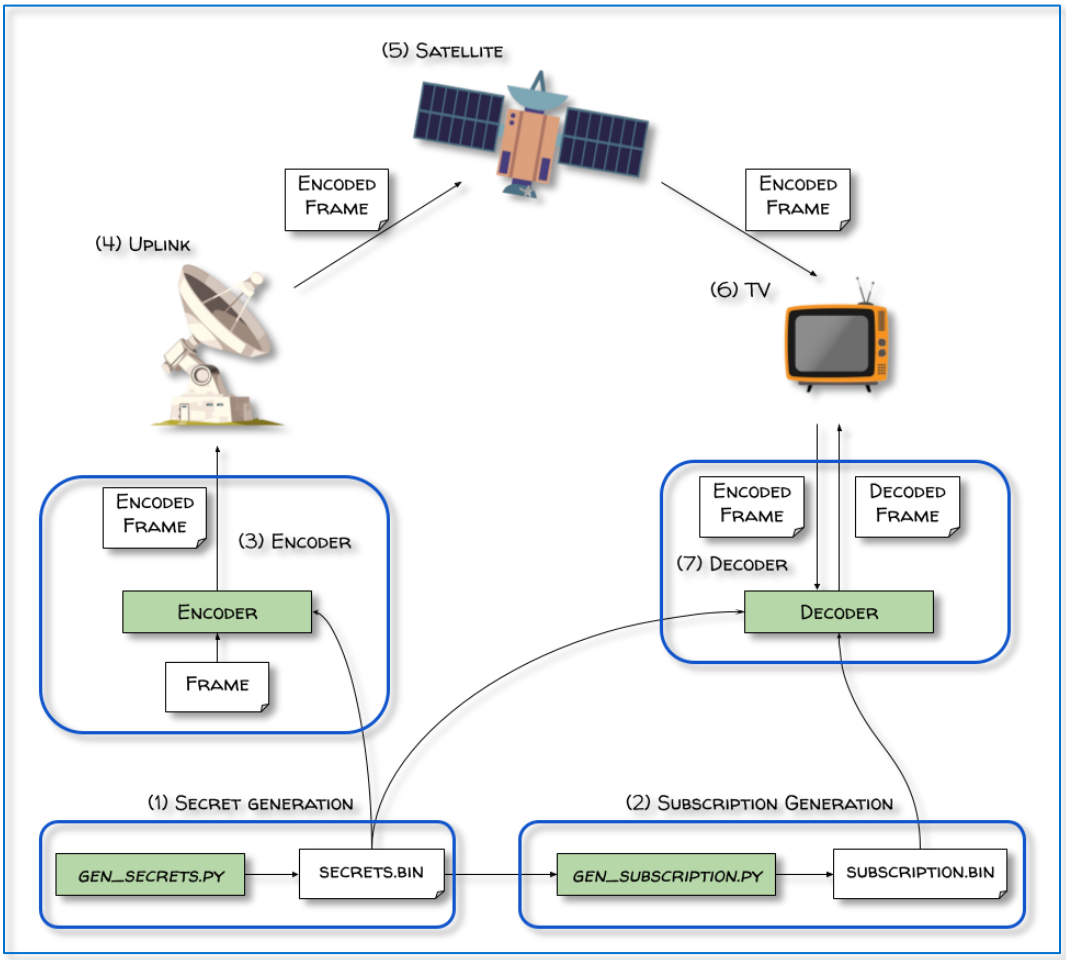

Figure 1: Overview of the JHU's design of the secure satellite system.

Security Mechanism Overview

Our design consists of 3 categories of security elements:

(1) Encryption: Encrypting broadcasted frames and subscription updates

(2) Authentication: Signing broadcasted data and subscription updates

(3) Subscription Checks: Based on information from valid subscription updates, reject incoming broadcasts that do not match the decoder ID, subscribed channel, or are outside the validity period of the subscription

The following keys are used for encryption and authentication, and described in detail in the subsequent Cryptographic Algorithms section:

(1) K_{CHx} = Symmetric channel keys, where x is the channel number (Encryption)
(2) K_S = Symmetric subscription key (Encryption)
(3) K_{Priv} = Private key for broadcast data and subscriptions (Authentication)
(4) K_{Pub} = Public key for broadcast data and subscriptions (Authentication)

Threat Model

Security Requirements

(SR1) An attacker should not be able to decode TV frames without a decoder that has a valid, active subscription to that channel

(SR2) The decoder should only decode valid TV frames generated by the Satellite System the Decoder was provisioned for.

(SR3) The decoder should only decode frames with strictly monotonically increasing timestamps.

Attack Scenarios

(AS1) Expired Subscription: Attackers previously held a valid subscription for one channel but attempt to read frames after subscription expiration.

(AS2) Pirated Subscription: Attackers have access to a valid subscription, but it belongs to a different decoder ID.

(AS3) No Subscription: Attackers attempt to read frames from a channel they do not have a subscription for.

(AS4) Recording Playback: Attackers have a valid subscription for a channel, but attempt to decode frames recorded at a time prior to their subscription validity period.

(AS5) Pesky Neighbor: Attackers attempt to spoof the satellite broadcast signal, replacing the legitimate broadcast with their own.

Overview of Security Controls

The following security controls are used to meet the 3 key security requirements.

Security Requirement	Control
(SR1)	Sign subscription updates
	Check subscription validity period
	Check target channel
	Check target decoder ID
(SR2)	Sign TV frames
(SR3)	Check incoming frame timestamps

Figure 2: Table of security controls to meet security requirements.

Additionally, the table below describes specific threats associated with each attack and the corresponding security control implemented as a defense.

Defends Against	Threat Model	Control
(AS1) Expired Subscription	Attacker tries to decode frame	Frame encryption
	Attacker modifies timestamp in the subscription	Sign subscriptions
	Attacker reuses expired subscription	Subscription time frame check in decoder
	Attacker extracts channel key from subscription	Channel key encryption
(AS2) Pirated Subscription	Attacker tries to decode frame	Frame encryption
	Attacker modifies decoder ID in the subscription	Sign subscriptions
	Attacker uses pirated subscription	Invalid subscription rejection
	Attacker extracts channel key from subscription	Channel key encryption
(AS3) No Subscription	Attacker tries to decode frame	Frame encryption
(AS4) Recording Playback	Attacker tries to decode frame	Frame encryption
	Attacker uses current subscription to decode recorded frame	Subscription timeframe check in decoder
(AS5) Pesky Neighbor	Attacker spoofs satellite arbitrary data as channel 0 to neighbor decoder	Sign encoded frames

Figure 3: Table of security controls implemented against each attack scenario.

The following sections describe the security controls outlined above in detail.

Security & Design

Cryptographic Algorithms

To ensure broadcast data and subscription data cannot be tampered with or utilized by unauthorized parties, we encrypt and sign the frames that are broadcasted as well as the subscriptions that are generated for receiving decoders. The following sub-sections discuss the encryption and signing algorithms used, as well as our motivations behind each algorithm.

AES-256-GCM Encryption

AES-GCM (Advanced Encryption Standard – Galois/Counter Mode) is an authenticated encryption mode that combines AES-CTR for encryption and GHASH for authentication. In our design, this encryption scheme is used for encrypting both the broadcast frames and subscription updates, utilizing the following keys:

(5) K_{CHx} = Symmetric channel keys, where x is the channel number
(6) K_S = Symmetric subscription key

The AES-GCM scheme takes the following inputs:

(1) 256-bit key – Randomly generated using the PyCryptodome library
(2) Initialization Vector (IV) – Randomly generated using PyCryptodome and used to both initialize the counter for encryption and the Galois Hash function for authentication
(3) Plaintext – Either the broadcast frames to be transmitted or the subscription update file

This encryption scheme is ideal for frame data because it eliminates the need for padding and incorporates the use of a counter, both of which improve processing speed. It allows for a unique IV per frame, which helps defend against replay attacks. Additionally, frames are encrypted independently, which prevents errors in one frame from propagating to the next. Finally, the authentication component helps ensure that individual frames cannot be modified without detection.

Ed25519 Signing

Ed25519 is an elliptic curve digital signature algorithm based on Curve25519 and is used for both broadcast and subscription data authentication in our design. This algorithm is ideal for broadcast applications as it is fast, provides collision resistance, has a high security target of 2^{128}, and has a small signature size of 64 bytes – it is both fast and secure. This algorithm utilizes a private key for signing, and a public key for verification. The following keys are included in our design:

(1) K_{Priv} = Private key used by encoder to sign broadcast data, and subscription generator to sign subscription updates

(2) K_{Pub} = Public key used by decoder to verify incoming broadcast data and subscription updates

For the broadcast data, this signature is applied over the entire broadcast packet. This is critical as only the frame data is encrypted and metadata (channel number, timestamp, decoder ID) are left in plaintext. This prevents attackers from tampering with metadata critical to decoder operation.

Secrets

The "*gen_secrets.py*" script is responsible for generating and packaging all encryption and authentication keys for distribution to the encoder, decoder, and subscription generator. The output of this script is a binary file and takes on the following format:

```
:format specification for secrets.bin:
    32 bytes: subscription_key (Ks)
    32 bytes: ecc_public_key   (KPub)
    32 bytes: channel_key_0    (KCH0)
    32 bytes: channel_key_1    (KCH1)
    32 bytes: channel_key_2    (KCH2)
    32 bytes: channel_key_3    (KCH3)
    32 bytes: channel_key_4    (KCH4)
    ...
    32 bytes: channel_key_x    (KCHx)
    32 bytes: ecc_private_key  (KPriv)
```

Figure 4: Format specification for *secrets.bin*, which holds keys for signing and encrypting data.

This binary file is generated once per deployment and is included in the build process.

Subscriptions

The "*gen_subscription.py*" script is responsible for generating subscription updates provided to individual decoders. These updates are provided at runtime and are not included in the initial build process, meaning that subscriptions can be issued on a rolling basis. Additionally, multiple subscriptions can be issued to a single decoder and will remain persistent across boots. To generate subscriptions, the following steps are performed:

(1) From secrets.bin, the script accepts channel private keys (K_{CH0-8}), subscription encryption key (K_s), and the Ed25519 private key for signing (K_{Priv}).
(2) Based on user input, the channel, validity period, and decoder ID is defined
(3) The entire subscription packet is encrypted
(4) The entire subscription packet is signed
(5) The subscription update file is saved as a binary file, and can then be sent to the decoder using the provided update tool

The output of the subscription generation script is a binary file containing the intended channel, validity period, intended decoder ID, the channel key (K_{CHx}) for the corresponding channel, and the ed25519 signature. This binary takes on the following format:

```
:format specification for subscriptions:
  4 bytes: decoder_id
  8 bytes: start_timestamp
  8 bytes: end_timestamp
  4 bytes: channel number
 60 bytes: KCHx (12 bytes nonce + 32 bytes KCHx + 16 bytes tag)
 64 bytes: ed25519_signature
```

Figure 5: Format specification for the payload encoded by the encoder and broadcast over the air via the uplink station.

Encoder

Our encoder is responsible for two primary tasks: (1) encrypting the frame, and (2) signing the payload broadcast over the air. Both tasks are achieved using the secret keys generated by *gen_secrets.py*. A summary of how the encoder encrypts and signs the payload can be seen in Figure 6.

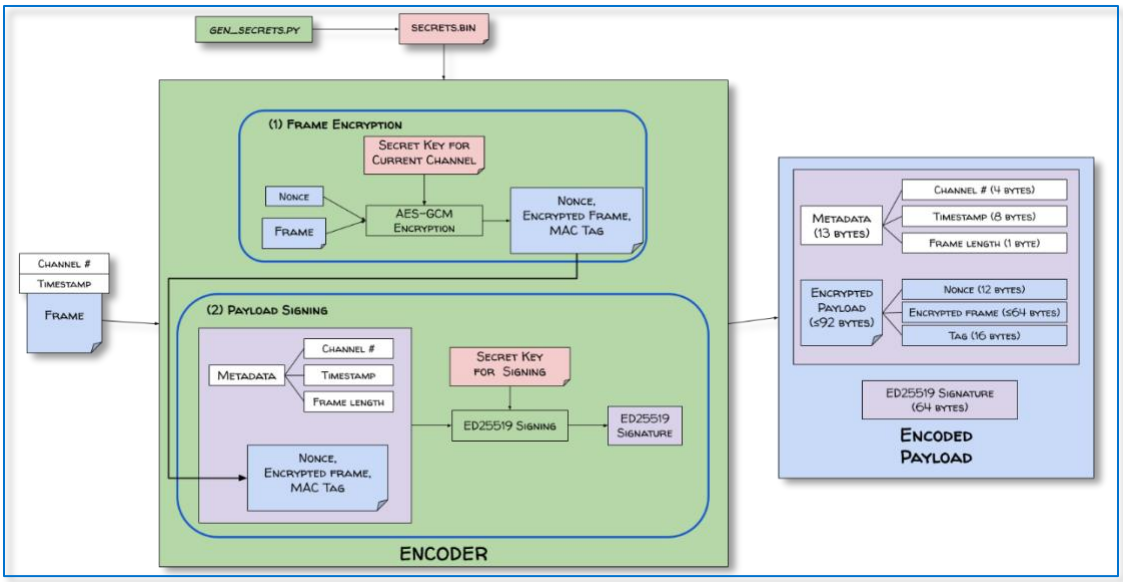

Figure 6: Diagram showing the encoder encrypting the frame and signing the payload.

First, the encoder retrieves the channel number to which the incoming frame will be broadcast. This channel number is used to identify the corresponding secret key (K_{CHx}) stored in "secrets.bin" to encrypt the frame, as shown in Figure 6, and will be used in conjunction with a randomly generated 12-byte nonce to perform AES-GCM encryption of the frame. The resulting output consists of the 12-byte nonce, the encrypted frame, and the Message Authentication Code (MAC) tag that the decoder can use to verify the source of the frame.

Note that the frame may have variable length. Rather than padding to the maximum 64 bytes as defined by the specifications, our encoder's output expands the pre-existing metadata to include not only the channel number and timestamp, but the length of the frame as well.

Secondly, because a mechanism to verify the authenticity of the payload is required, the overall payload, consisting of the unencrypted metadata and the combined nonce, encrypted frame, and MAC tag, is signed using the *ecc_private_key* (K_{Priv}), stored in "secrets.bin", through the ED25519 signing algorithm. Although the metadata is unencrypted when transmitted, attackers cannot manipulate the channel number, timestamp, or frame lengths due to the generated ED25519 signature.

The final encoded broadcast data is passed to the uplink in a binary format taking on the following structure:

```
:format specification for broadcast data:
--- header
  4 bytes: channel_number
  8 bytes: timestamp
  1 byte ciphertext_length
--- body
 92 bytes: encrypted_frame (12-byte nonce;64-byte ciphertext;16-byte tag)
 64 bytes: ed25519_signature
```

Figure 7: Format specification for the encoded payloads broadcast over the air to the decoder.

Decoder

The decoder receives the encoded frames from the TV and decodes them into readable frames only if it has a valid subscription to the frame's channel upon the time of receiving the frame. As a result, the functions of the decoder can be separated into two, as depicted in Figure 8: (1) checking the decoder's subscription status with the incoming frame's channel, then (2) decrypting the frame and transmitting it back to the TV.

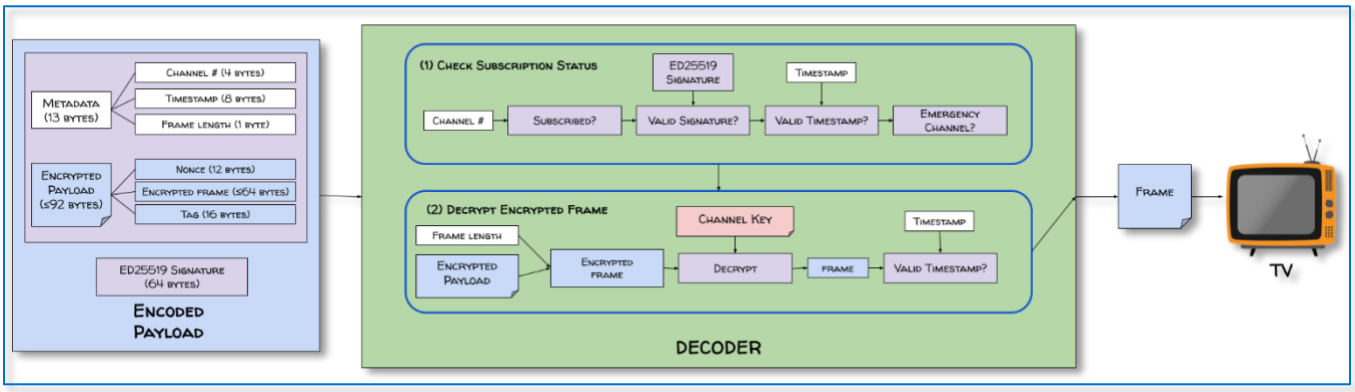

Figure 8: Diagram showing the decoder verifying the subscription status and frame before the decrypted frame is sent back to the TV.

Because the process of verifying a signature is time-consuming, the decoder first checks the channel number to ensure that the decoder has a valid subscription to this channel. Once the decoder verifies that it is subscribed to this channel, it checks the authenticity of the message to ensure the frame was sent from the authorized encoder/uplink/satellite system. The fact that the channel number is checked prior to the signature verification does not jeopardize the security of the system, since even if an attacker tampers with the channel number, the payload would have been rejected because it would have resulted in an invalid signature.

Once the authenticity of the message is verified, the timestamp of the message is checked to meet the functional requirements (i.e. monotonically increasing timestamps) *and* to ensure that the decoder has a valid subscription to this channel for the given timestamp. If this has been cleared, the decoder performs a final check to see if it is the emergency channel, since frames sent via the emergency channel should always be processed and sent to the TV. If all of these checks pass, the decoder finally decrypts the frame and performs another timestamp check before passing it over to the TV.

Tools

The host tools allow communication between the host computer and the Satellite TV system, and generate key components used in the design. Below outlines the provided tools and sample usage in our design:

Tool	Description	Example Usage
gen_secrets	Generates secrets.bin (keys)	python -m ectf25.tv.gen_secrets global.secrets/secrets.bin <CH Numbers>
build decoder	Build the decoder image	docker build -t decoder . docker run -v $(pwd)/decoder/:/decoder -v $(pwd)/global.secrets:/global.secrets:ro -v $(pwd)/<build_name>:/out -e DECODER_ID=<decoder_id> decoder_new
flash	Flash design to the MAX78000	python -m ectf25.utils.flash ./<decoder_name>/max78000.bin /dev/tty.usbmodem<port>
gen_subscription	Generates new subscription.bin	python -m ectf25.tv.gen_subscription global.secrets/secrets.bin <CH Numbers>
subscribe	Apply subscription to a decoder	python -m ectf25.tv.subscribe <name>.sub /dev/tty.usbmodem<port>
list	List active subscriptions on a decoder	python -m ectf25.tv.list /dev/tty.usbmodem<port>

uplink	Start the uplink	python -m ectf25.uplink global.secrets/secrets.bin localhost 2000 0:10:frames/<name>.json 1:10:frames/<name>.json 3:10:frames/<name>.json
satellite	Start the satellite	python -m ectf25.satellite localhost 2000 localhost 0:2003 1:2001 3:2002
tv	Start the TV	python -m ectf25.tv.run localhost 2002 /dev/tty.usbmodem<port>

Figure 9: Table of tools used to implement the secure satellite system.

Libraries

The following libraries are used our satellite TV system design:

(1) PyCryptodome (Python)
(2) PyNaCl (Python)
(3) WolfSSL (C)
(4) MSDK Libraries (C)