[redacted] Design Document

[redacted]

Overview

This document was created to detail the design of our system and detail how it meets the functional and security requirements of the 2025 eCTF competition. The document is organized into 3 primary sections. A breakdown of the individual components we are responsible for, a discussion of the security requirements and how our design meets these requirements, and a look at guarding against timing side channel attacks. Our code is modified from the original reference design provided by Miter to implement security features.

Components

Subscription Generator overview

Important formats
The subscription generate output contains the following information before encryption

Update Packet Format

- Unsigned int device_id: 4 bytes
  - Stores the decoder id from the subscription generator input
- Unsigned int start: 8 bytes
  - Stores the starting timestamp for the subscription from the input
- Unsigned int end: 8 bytes
  - Stores the ending timestamp for the subscription from the input
- Unsigned int channel of max size: 4 bytes
  - Stores the decoder id from the function input

device_id	start	end	channel

Process.

The subscription generator takes in the decoder id, secrets file, channel ID's, and times stamps (start and end). The subscription generator performs sanity checks on the provided input (including size of the variables, and that start <= end). After the sanity check we pack the data into the packet variable, which is in binary format, and add padding to make sure it is 16 byte aligned for our encryption algorithm.

We then use the AES-128-ECB mode encryption algorithm to encrypt the packet using the secret_key, which is generated in the global secrets file as part of the build process. The result of this is returned as the output of our subscription generator.

Required Secrets.

The encoder requires a master AES secret key, provided from the generated secrets file to function

Encoder Overview

Our encoder takes as input a secret file, containing an AES encryption key, a channel, a timestamp, and the contents of the frame and applies two layers of encryption to the frame. The first is to encode the contents of the message, and the second encodes the entire frame, including the channel, timestamp and contents.

Important formats

Frame format

- Channel ID (4 bytes) - Identifies the communication channel.
- Timestamp (8 bytes) - Marks the frame's valid time window.
- size (4 bytes) - tracks the size of the incoming frame data
- Data Section (64 bytes) - The encrypted payload. This will be padded to the nearest 16 bytes to meet our decoding requirements.

Channel ID	Timestamp	size	Data

The channel ID is a 4 byte channel ID provided to the encoder
The Timestamp is an 8 byte value provided to the encoder
The size is a 4 byte unsigned int that tracks the frame data size
The data is an array of FRAME_SIZE, containing the contents of the frame. Frame size is equal to the maximum size of the frame information, I.E 64 bytes. If the incoming frame information is smaller than this, it is padded to the nearest 16 bytes.

Process details

- The TV frame data is checked to see if it is =< 64 bytes, and is padded to the nearest 16 bytes.
- The TV frame is encrypted using AES-128 in ECB mode first, this is the first layer of encryption.
    - The key used is derived from the secure secrets file and the channel ID. *See the secret generator description.*
    - This means that the validity of the frame is checked in the decoder, before the full frame information is encrypted in memory, meaning that unless the frame

    - contents would be decrypted anyway, it will not be present in plaintext in the decoder's memory.
    - Channel zero uses the same encryption key for layer one and layer two, as it is required to be decrypted by the decoder.
- The header is then packed with the encrypted frame data to form the frame packet
    - More padding is added if required for 16 byte alignment.
- The frame packet is then encrypted with a master key that is shared with the decoder. This key is not derived from the channel ID.
- This encrypted frame packet is returned from the encode function.

Required Secrets.

The encoder requires a master AES secret key, provided from the generated secrets file to function. Additionally, it requires a table of 10007 keys, which are used with the modulo of the channel ID to pick the secret key for a given channel.

Decoder Overview

Our decoder implements the functional requirements by building off the reference designs functionality. With the exception of the security changes mentioned, our design follows the reference design outline for now, and so will satisfy the functional requirements. (Draft statement, the full documentation will detail all steps, even if they were from the reference design originally. Matching with the encoder, we use AES encryption to ensure the frames and subscription requests where generated from the same deployment.

Important Formats

Frame format

- From the Encoder section.

Subscription packet

- From the subscription generator section.

Important data structures/variables

Flash_entry_t decoder_status

This stores the current state of the decoder's subscribed channels, and a variable to indicate if this is the first boot of the machine or not. Channel subscription information is stored in a MAX_CHANNEL_COUNT array of channel_status_t called subscribed channels.

channel_status_t contains.

- Bool active
    - A variable which is set to true when a channel is subscribed to the first time. Note that this variable is not a check to see if the board timestamps are valid.
- Channel_id_t id
    - Stores the 4 byte channel id

- Timestamp_t start_timestamp
    - Stores the 8 byte starting timestamp
- Timestamp_t end_timestamp
    - Stores the 8 byte ending timestamp.

These entries are stored in flash memory, so that they persist across board resists in accordance with the functional requirements.

latest_timestamp

- Keeps track of the last timestamp seen, checked in the last.

Global secrets

The global secrets are retrieved from the secure.h header file during the build process, and stored on the decoder board, this includes the complete table of 10007 possible channel keys.

Decode frame functionality

The decode frame function is called from the main process of the board, when a message is received with the D opcode, and follows the following algorithm

- Message received by the decoder contains the “D” opcode
- Decoder sends back ACK to host
- Host will send up to 256 byte message
    - We cap the size of the incoming UART communication to MAX_PACKET_SIZE to prevent buffer overflows. This means the requirement as no incoming subscription packet will be larger than this.
- We add a small random delay in the form of a random length while loop in front of the rest of the decode function.
- Use the master key to decrypt the frame packet's second level of encryption, allowing access to the channel ID, timestamp, and size.
- Call an is_subscribed function which.
    - Returns 1 if the timestamp is valid, and there is a valid subscription to the channel (timestamp between start and end, channel id active)
    - Fails if the timestamp is less then the last valid timestamp, or if there is not an active subscription.
- Calculate the size of the frame data
- If valid packet is on channel 0 (emergency channel)
    - Use the master key to decrypt again.
- If not channel 0
    - Decrypt with the channel key matched with the associated channel. This key is obtained using the same modulus operation the encoder uses on the table of 10007 keys.
- If is_subscribe returns -1, or any decode function fails, we return -1 the channel is not subscribed. Which outputs an error.

List channels functionality
The list channels functionality reads the contents of the subscription table, and outputs them on the UART according to the message protocol. It does this by reading from the channel_status_t and outputs any subscriptions with the active flag enabled.

Channel subscription functionality.

- Message received by the decoder contains the "S" opcode
    - Decoder sends back ACK to host
- Decoder receives the rest of the encoded packet
    - We cap the size of the incoming uart communication to MAX_PACKET_SIZE to prevent buffer overflows. This means the requirement as no incoming subscription packet will be larger than this.
- Decrypt the packet update using the shared key provisioned to the decoder board, using AES 128 ECB mode, though the wolfssl library.
    - If the packet is not the correct size (multiple of 16) or there is some other error the wolfssl will return an error code, and we will exit with an error message.
- Cast the raw packet to the package data structure.
- Trim off any extra padding added from the decrypted package.
- Compare decoder ID and reject if this packet is not for this decoder.
- Update the list of active channels with the new channel. If we cannot find a free channel slot, we return an error.

Secret Generator Overview

The secret generator generates a master_key that is used for the second layer encryption on the frames, and for the subscription generator. It also generates 10007 keys to be used to generate the per channel key, by taking 10007 % channel_id. This may lead to collisions, but it does fit on the board, and prevents the need from putting the channel keys in the subscription packet.

Running the secret generator creates a global secrets file in the form of a header, which is read in during build and which contains the master key and the generated key. Subscription_gen and encoder both read the keys from this file.

Security Requirements

- **Security Requirement 1**
    - An attacker should not be able to decode TV frames without a Decoder that has a valid, active subscription to that channel.
        - We require a shared key, and a shared decoder ID protected by that AES key for a subscription to be valid and the board to be updated. Without that, the check during decoding will fail due to the lack of valid subscription.

- **Security Requirement 2**
    - The Decoder should only decode valid TV frames generated by the Satellite System the Decoder was provisioned for. The definition of provision given in the functionality requirements is that encoder-decoder pairs generated from the same shared secrets are provisioned for each other.
        - By requiring an shared AES key, we prevent frames from a different encoder deployment from being decrypted on our board.
        - We check channel and timestamp viability before we decode the frame data in memory using a per channel key from the same shared secrets.
        - We store the channels in flash memory so they persist across power cycles.
- **Security Requirement 3**
    - The Decoder should only decode frames with strictly monotonically increasing timestamps.
        - We store the last frame timestamp encountered in memory, and reset it to zero in the init function.
        - Counter is across all channels.
        - We check the frame's timestamps using this counter.

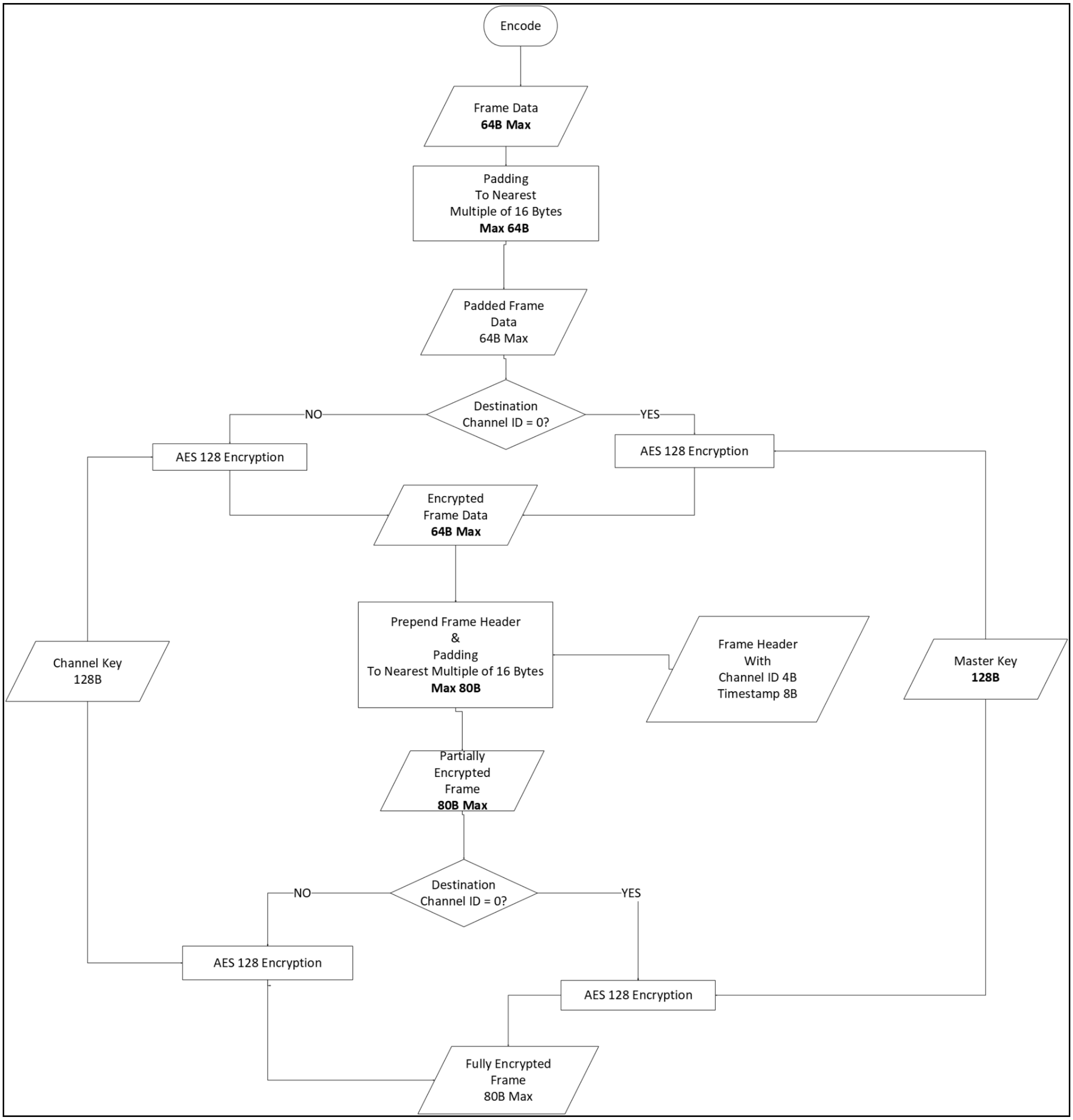
Security vulnerabilities and Mitigations.

Timing attacks

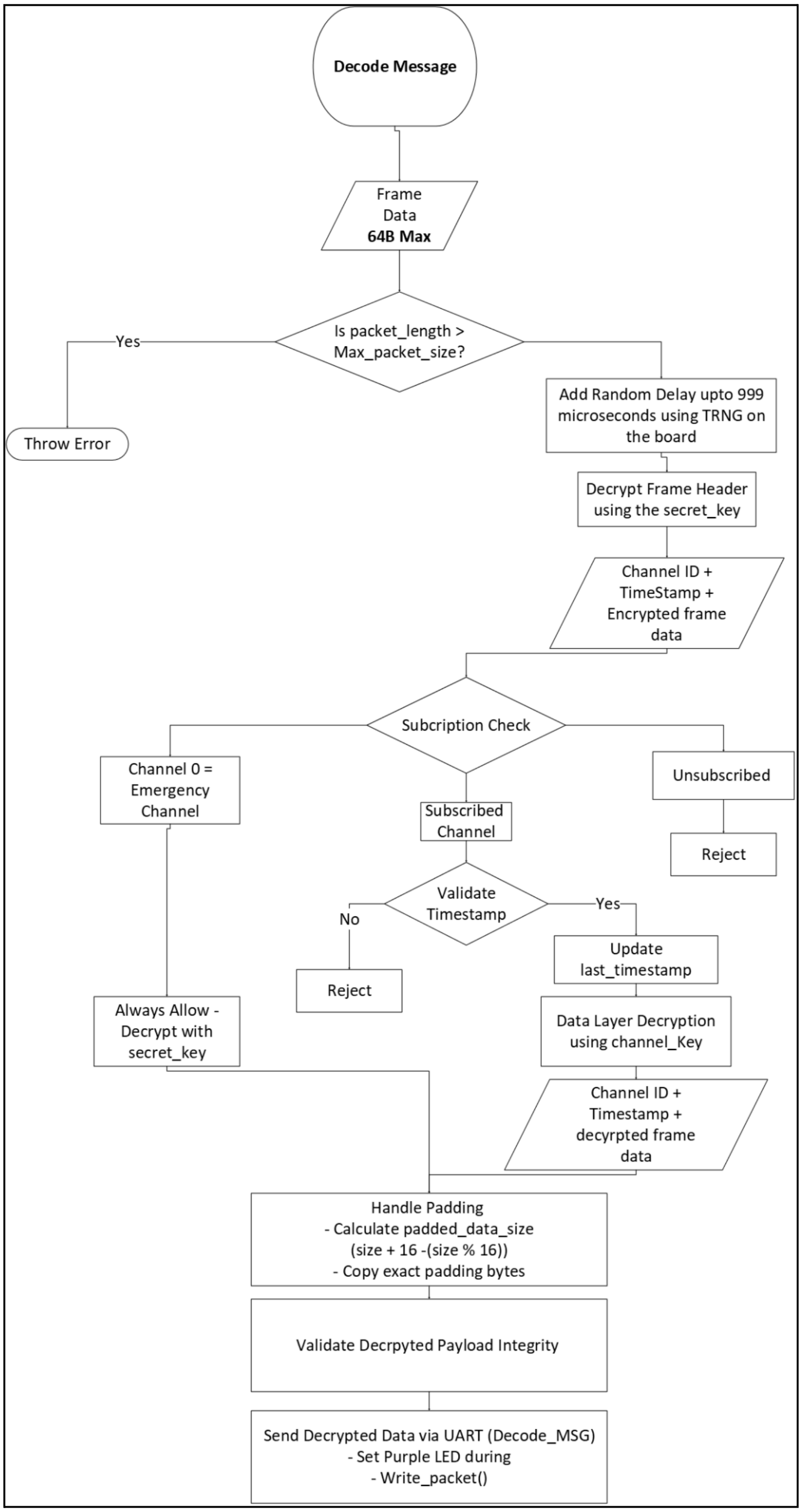
To make timing attacks more difficult, we add a randomly generated delay using the board's internal random number generator to the decoding process. We cause this delay using a simple for loop that lasts a random amount of time, but will always keep the time for decoding under the speed requirement.

Diagrams

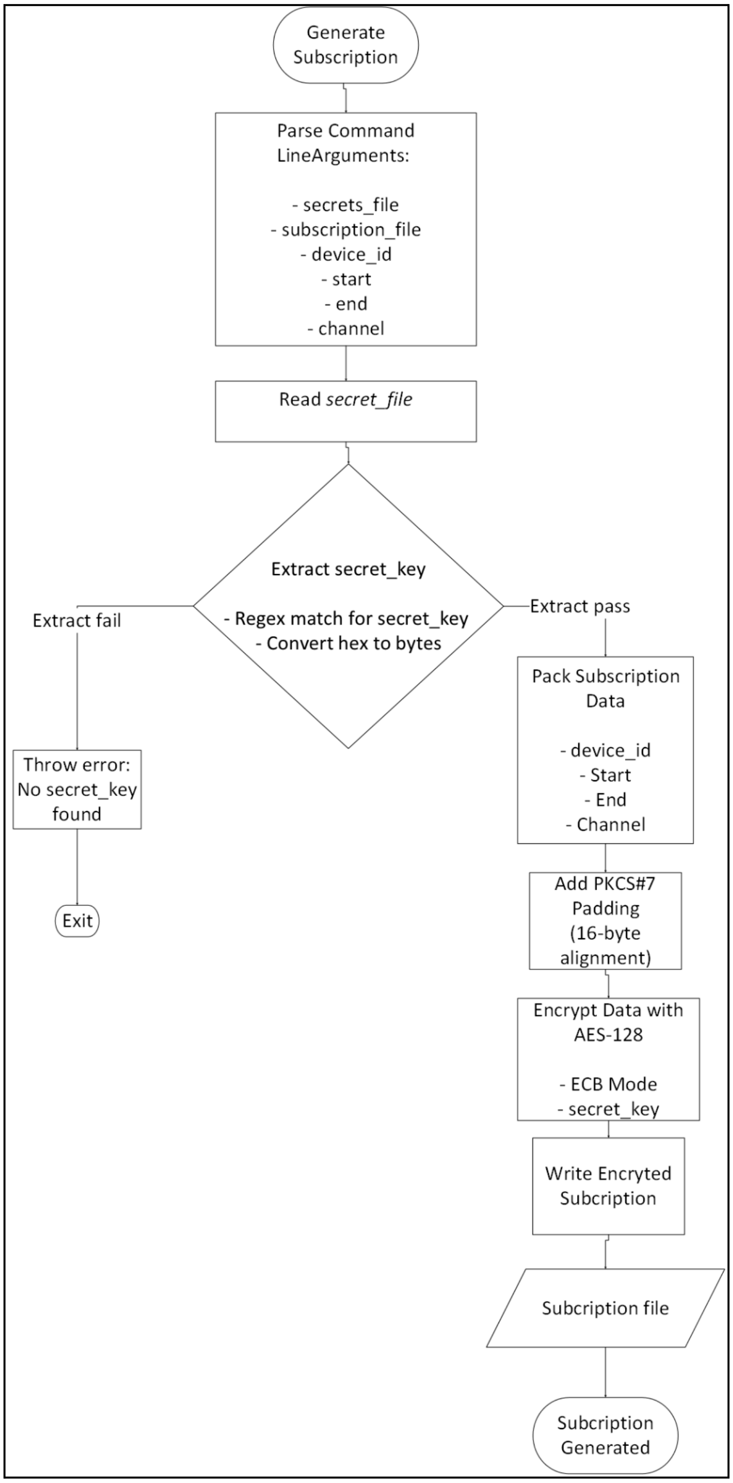
Below are flow diagrams to better illustrate the intended process flow of the encoder, the decoder, and the subscription generator.



Encoder flow diagram


Decode Message
Frame Data **64B Max**
Is packet_length > Max_packet_size?
Yes
Throw Error
Add Random Delay upto 999 microseconds using TRNG on the board
Decrypt Frame Header using the secret_key
Channel ID + TimeStamp + Encrypted frame data
Subcription Check
Channel 0 = Emergency Channel
Subscribed Channel
Unsubscribed
Reject
Validate Timestamp
No
Yes
Reject
Update last_timestamp
Always Allow - Decrypt with secret_key
Data Layer Decryption using channel_Key
Channel ID + Timestamp + decyrpted frame data
Handle Padding
- Calculate padded_data_size
(size + 16 -(size % 16))
- Copy exact padding bytes
Validate Decrpyted Payload Integrity
Send Decrypted Data via UART (Decode_MSG)
- Set Purple LED during
- Write_packet()

Decoder flow diagram



Subscription generator diagram