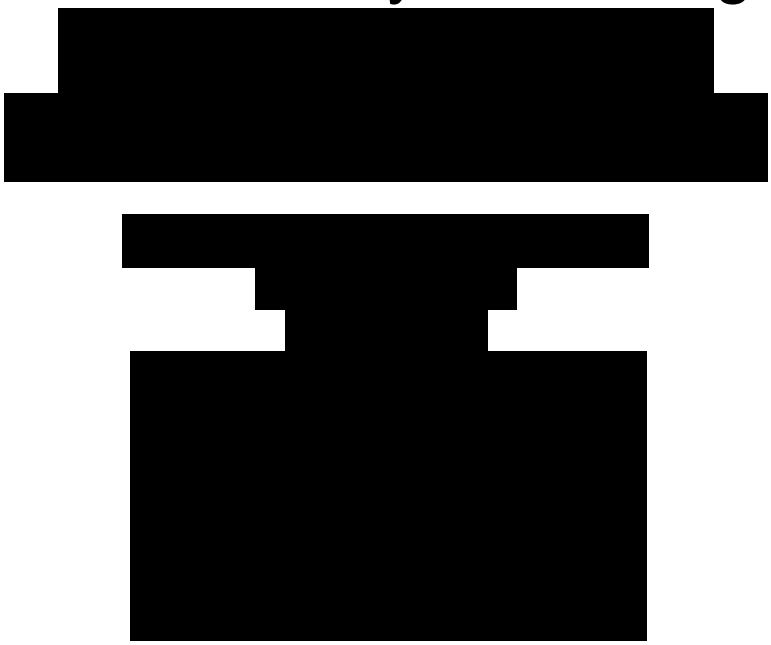
# Satellite TV System Design



# 1 Background

This document describes a system that meets the functional and security requirements of the MITRE 2025 eCTF challenge. The system is a simulated and simplified version of a satellite TV signal encoder/decoder. The decoder must run on the Analog Devices MAX78000FTHR board. Other parts including the encoder run on a computer (host).

## API

The host-based components must be Python callable modules, with functions implementing these signatures:

```
gen_secrets (channels: list[int]) -> bytes
gen_subscription (secrets: bytes, device_id: int,
   start: int, end: int, channel: int) -> bytes
encoder (secrets: bytes) -> encoder instance
encode (channel: int, frame: bytes, timestamp: int) -> bytes
```

The MAX78000-based decoder must receive commands via UART (baud 115200) on the USB port via a protocol described in Detailed Specifications. Messages being sent or received contain a header (1 byte magic, 1 byte opcode, 2 byte body length) followed by the body (data bytes). The header and each 256-byte chunk of the body must be ACKed by the other side before more data is sent. The receiver waits for the host to send a command, then sends a response before waiting for more commands.

The decoder must implement 3 functions, which must be written in C, C++, or Rust:

1. List Channels (opcode L)
   The request body is empty.
   The response body contains a 32-bit number of channels, followed by an array of [32-bit channel ID, 64-bit start time, 64-bit end time] for all valid subscriptions.
2. Update Subscription (opcode S)
   The request body contains bytes generated by gen_subscription().
   The response body is empty.
3. Decode (opcode D)
   The request body contains bytes generated by encode().
   The response body contains the bytes that were passed as input to encode().

## Functional Requirements

- The maximum number of subscribed channels per device is 8 (not including channel 0, the emergency broadcast channel).
- Channel IDs are 4 bytes, timestamps are 8 bytes.
- The maximum frame size is 64 bytes (prior to encoding).
- The device must take no longer than 1s to become responsive after powering on.
- The List Channels and Update Subscription functions must take no longer than 500ms.
- The decoder must be able to handle 10 64-byte frames per second (including data transfer time over UART).
- The encoder must be able to handle 1000 64-byte frames per second.

## 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 (except for the broadcast channel, which does not require subscription).*

***SR2.*** *The Decoder should only decode valid TV frames generated by the Satellite System the Decoder was provisioned for (as opposed to fabricated frames from an attacker or frames from another system).*

***SR3.*** *The Decoder should only decode frames with strictly monotonically increasing timestamps (this does not apply across power cycles).*

## Flag Attack Scenarios

The system must defend against the following attack scenarios in order to prevent other teams from capturing flags:

1. **Expired Subscription**
   Read frames from a channel you have an expired subscription for
2. **Pirated Subscription**
   Read frames from a channel you have a pirated subscription for
3. **No Subscription**
   Read frames from a channel you have no subscription for
4. **Recording Playback**
   Read frames from a recorded channel you currently have a subscription for, but didn't at the time of the recording
5. **Pesky Neighbor**
   Spoof the signal of the satellite to cause your neighbor's Decoder to decode your frames instead

# 2 Security Design

## Guiding Principles

- All data received by the decoder should be encrypted and signed.
- All data stored in flash should be encrypted.
- Assume that the attacker may be able to gain any secret key known to the device, and try to minimize the damage they can do with it.
- Assume that the attacker may be able to gain read/write access to the flash memory or the RAM. Wipe secrets from memory when they are not needed.
- Use public-key cryptography for signing and verification, so that the compromise of a public key on the device will not allow the attacker to impersonate the host system (since they still lack the private key).
- The decoder should not have any channel keys to start with (except channel 0); keys will be received via subscription updates.
- The decoder must store subscription data for each channel in flash, in order to survive a power cycle.
- The decoder must keep track of the last-seen timestamp in order to check Security Requirement 3 [increasing timestamps] but this does not need to be saved to flash.

## Overview

The system implements a secure satellite TV decoder using encryption and authentication. In particular, it relies on public key cryptography for signing and verifying messages, which are encrypted and decrypted using a fast symmetric cipher. All private keys reside only server-side, while public and symmetric keys are shared with individual decoders on an as-needed basis.

Each channel has a symmetric key and a public/private key pair, with the encoder holding all private and symmetric channel keys and individual decoders receiving public and symmetric channel keys through signed and encrypted subscription updates. The system uses a subscription seed and decoder ID to derive device-specific symmetric and public/private subscription keys. This allows subscription messages to be signed and encrypted specifically for each decoder.

During manufacturing, each decoder gets a device-specific firmware image containing an encrypted payload with its essential secrets (including the public and symmetric subscription keys and the public and symmetric emergency broadcast channel keys). At runtime, the decoder maintains a data structure in RAM that holds active subscription information and the necessary channel keys.

Frame decoding follows a strict verification process: the decoder first validates that a purported (unencrypted) channel ID matches an active subscription, then decrypts the frame using the corresponding channel's symmetric key, verifies the embedded signature using the channel's

public key, checks that the intended (decrypted) channel ID matches, and checks that the timestamp is within the subscription boundaries and increasing. This process protects against various attacks including subscription sharing, replay attacks, and encoded frame forgery.

The design also employs various techniques to defend against potential attacks (see Section 4, Threat Analysis and Mitigations).

## Secrets and Cryptographic Primitives

| **Secret** | **Algorithm** | **Inputs** | **Purpose** |
|---|---|---|---|
| Private channel key(s) | Ed25519 | PRNG | Sign frame for one specific channel |
| Public channel key(s) | | | Verify frame signature for one specific channel |
| Symmetric channel key(s) | ChaCha20-Poly1305 | PRNG | Encrypt / decrypt video frame for one specific channel |
| Subscription seed | *N/A* | PRNG | Used with Decoder ID to generate subscription keys |
| Decoder ID | *N/A* | | Unique identifier for decoder device |
| Private subscription key | Ed25519 | Decoder ID, Subscription seed | Sign subscription for one specific device |
| Public subscription key | | | Verify subscription signature for one device |
| Symmetric subscription key | ChaCha20-Poly1305 | | Encrypt / decrypt subscription update for one device |
| Symmetric flash key | ChaCha20-Poly1305 | PRNG | Encrypt / decrypt data stored in flash |

The Python-based PyCryptodome library is used to generate keys and/or encrypt and sign messages on the host side, and the C-based wolfCrypt library is used to decrypt messages and verify their authenticity on the decoder.

Ed25519 was chosen for signing and verification because: it is one of the NIST recommended ciphers for public key cryptography; it has smaller keys and signatures compared to alternatives and predecessors (such as ECDSA and RSA); and wolfCrypt provides a timing-resistant implementation of this cipher. ChaCha20-Poly1305 was chosen due to its security level and efficiency compared with alternative symmetric ciphers (such as AES-GCM) in the absence of hardware acceleration, and because it includes message authentication.

Subscription keys are generated deterministically by hashing a combination of the subscription seed and decoder ID, since both the subscription generator and the decoder build process need to derive the same set of keys independently. We use separate hashing algorithms (Blake2b and SHA3-256) to derive the symmetric and private keys.

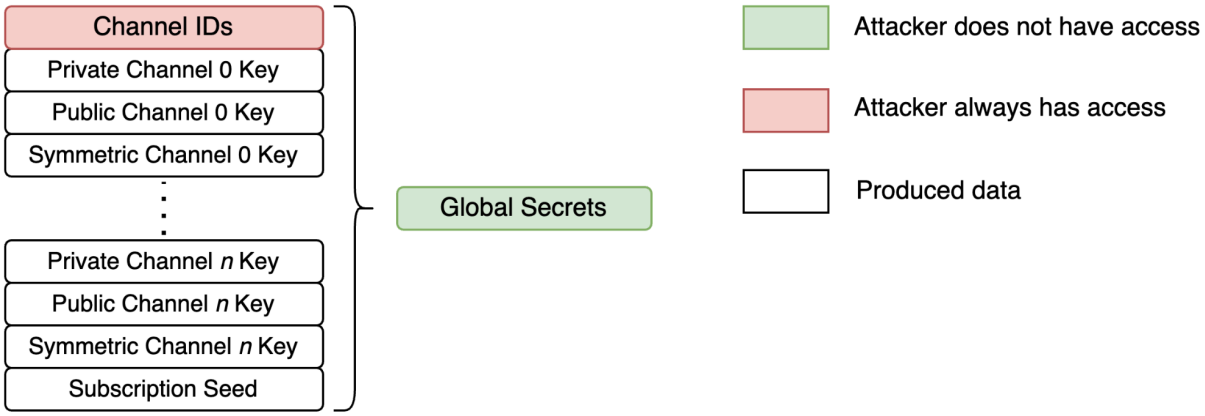
ChaCha20-Poly1305 uses an initialization vector (IV) that can be openly shared but should never be reused. The IV for all encrypted messages to the decoder (Subscribe or Decode) will be randomly generated nonces, preprended in plain to the messages. The flash key is only used to decrypt flash on the device, so a single randomly generated nonce will be stored in the firmware at build time, together with the key.

# Component Design

## Secrets Generation

The gen_secrets function:

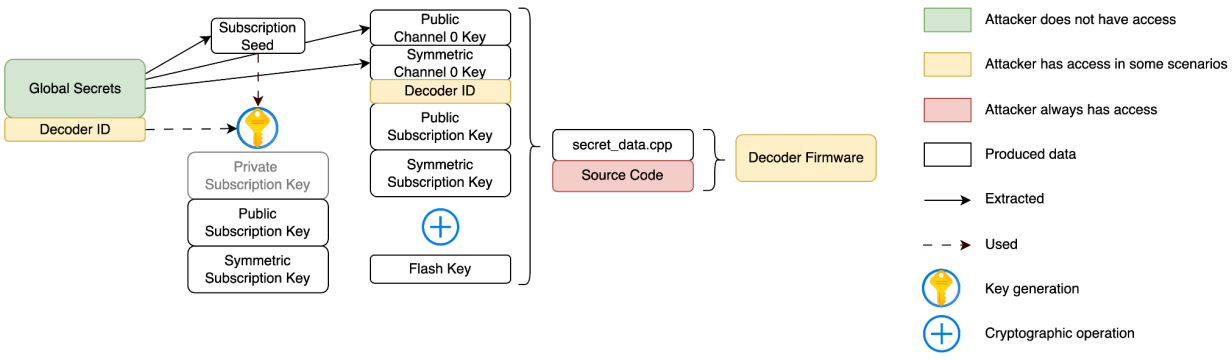
- Takes as input the channel IDs.
- Generates:
    - a symmetric key and a public/private key pair for each channel, including the emergency broadcast channel 0, and
    - a seed for deriving subscription keys (32 random bytes).
- Outputs a secrets blob containing the channel IDs and above generated items.



## Build Decoder Process

The Build Decoder process:

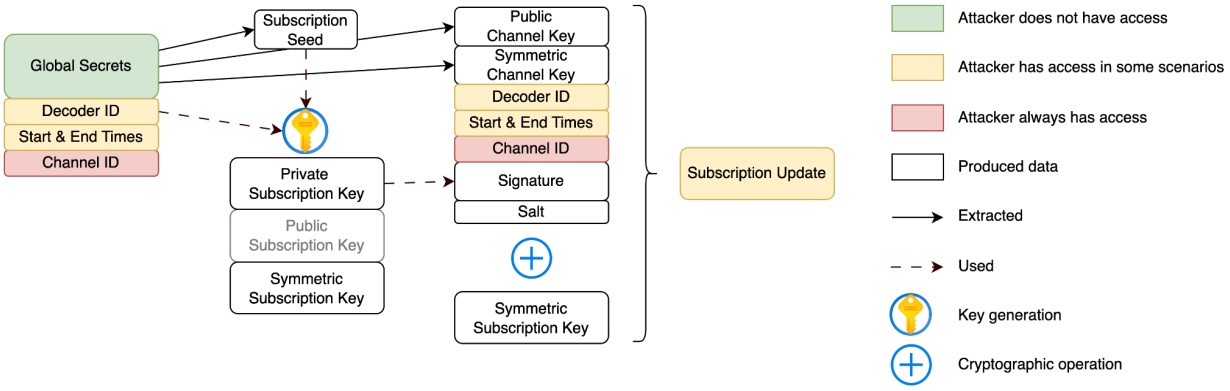
1. Takes as input the:
    a. Source code
    b. Secrets
    c. Decoder ID
2. Extracts from the secrets the:
    a. Subscription seed
    b. Public channel 0 key
    c. Symmetric channel 0 key
3. Derives public, private, and symmetric subscription keys from the subscription seed and decoder ID.
4. Generates a random symmetric key for flash (persistent) storage.
5. Bundles and encrypts with the flash key:
    a. Decoder ID
    b. Public channel 0 key
    c. Symmetric channel 0 key
    d. Public subscription key
    e. Symmetric subscription key
6. Stores the flash key and the above encrypted data in a secret_data.cpp file.
7. Builds the decoder image from the source code, including the secret_data.cpp file.



## Subscription Generation

The gen_subscription() function:

1. Takes as input the:
    a. Secrets
    b. Decoder ID
    c. Channel ID
    d. Start and end times
2. Extracts the subscription seed and public and symmetric channel keys from the secrets.
3. Derives device-specific public, private, and symmetric subscription keys from the subscription seed and decoder ID.
4. Bundles and signs with the private subscription key:
    a. Decoder ID
    b. Channel ID
    c. Start and end times
    d. Public channel key
    e. Symmetric channel key
5. Bundles the above signed payload with random salt (1 byte containing a random number *n* followed by *n* random bytes), and encrypts it using the symmetric subscription key.

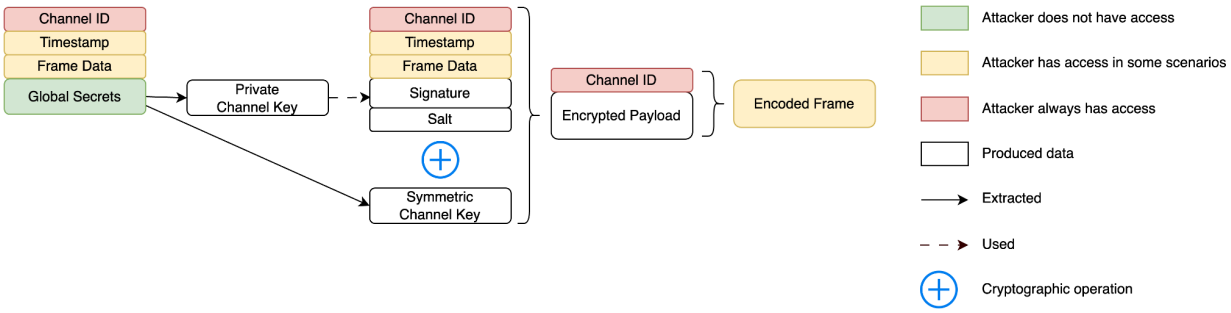


## Encoder

The Encoder class constructor extracts from the secrets the items necessary to encode frames: the list of channel IDs, and the corresponding private and symmetric keys for each channel.

The encode() method:

1. Takes as input the:
    a. Channel ID
    b. Frame timestamp
    c. Data frame
2. Bundles and signs the above with the private channel key.
3. Bundles the above signed payload with random salt, and encrypts it with the symmetric channel key.
4. Outputs the unencrypted channel ID followed by the encrypted data.




## Decoder Startup

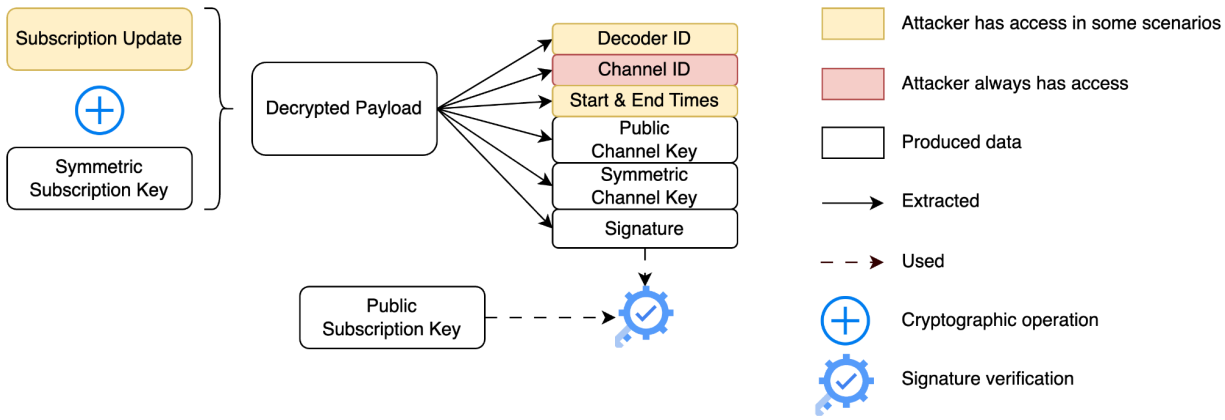
The decoder boot sequence:

1. Uses the flash key to decrypt stored secrets:
    a. Decoder ID
    b. Public channel 0 key
    c. Symmetric channel 0 key
    d. Public subscription key
    e. Symmetric subscription key
2. Initializes a “Channel State” data structure that maintains subscription information. This includes the public and symmetric keys for channel 0 as well as any subscribed channel.
3. Attempts to load subscriptions from flash if they exist, decrypts each using the symmetric subscription key, verifies authenticity using the public subscription key, and updates the Channel State (following the process described below in “Update Subscription”, with the exception of saving to flash). If the device detects memory tampering during this step, it will reboot.
4. Initializes a global variable that stores the last seen frame timestamp to 0.

## Update Subscription

When the decoder receives a subscription update command (opcode S: Subscribe) over UART, the command handler:

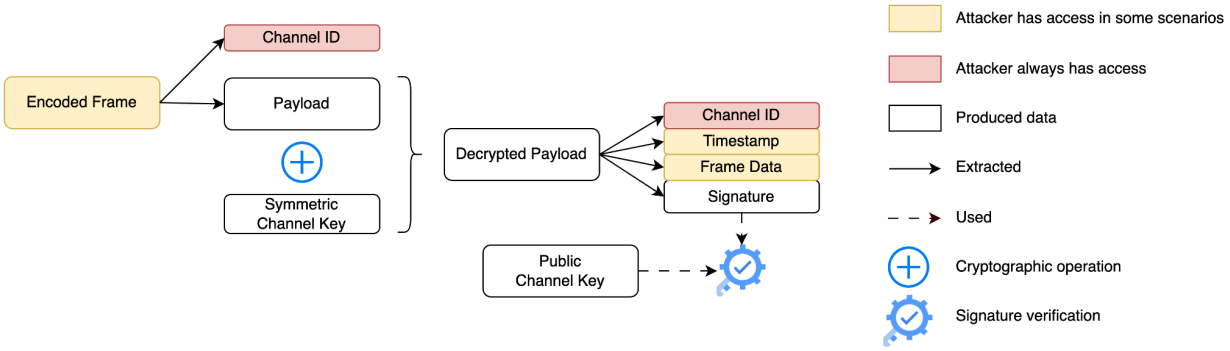
1. Decrypts the command payload using the symmetric subscription key and verifies authenticity using the public subscription key.
2. Extracts the subscription data:
    a. Decoder ID
    b. Channel ID
    c. Subscription start and end times
    d. Public channel key
    e. Symmetric channel key
3. Performs checks and updates the Channel State. It returns an error if the decoder ID does not match or the channel ID is new and the maximum number of channels has been reached. Otherwise, it replaces any information for the given channel with the newly provided information.
4. Stores the new subscription data in flash if it is valid, potentially overwriting older subscription data for the same channel. Subscriptions for individual channels are stored on separate pages in flash.
5. Removes the channel keys from the Channel State if the subscription has expired, detected by comparing the end time with the last seen valid frame timestamp.



### Decode

When the decoder receives a decode frame command (opcode D: Decode) over UART, the command handler:

1. Extracts the purported channel ID (first 4 bytes) from the command payload, and checks that it is either channel 0 or a channel with an active subscription.
2. Decrypts the remainder of the command payload using the symmetric channel key for the given channel, and verifies authenticity using the public channel key.
3. Extracts all elements from the decrypted data:
    a. Channel ID
    b. Frame timestamp
    c. Data frame
4. Performs validity checks:
    a. The purported and intended channel IDs must match.
    b. The timestamp must fall within the subscription range for the given channel.
    c. The timestamp must be strictly greater than the last seen frame timestamp.
5. Updates the Channel State (by erasing the channel keys) if the subscription has expired.
6. If all validity checks have passed, updates the last seen frame timestamp and returns the decrypted data frame. Otherwise it returns an error.




# 3 Meeting 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 (except for the broadcast channel, which does not require subscription).*

The encoder encrypts every frame using a symmetric channel key specific to the frame's channel. The encryption is strong enough to prevent brute-force key guessing, meaning that attackers must use a decoder to perform decryption.

Decoders initially start without any channel keys other than the ones for the broadcast channel. The only way to acquire channel keys is via subscription messages, which are also encrypted in order to keep the channel keys secret. It is impossible for a decoder to decode frames on channels for which it does not have a subscription.

When a decoder has a valid but inactive subscription (either expired or with a future start time), it will refuse to decode frames for that channel after comparing the decrypted frame timestamp against the subscription start/end time.

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

This requirement can be broken down into two parts:

*2a. Frames from a different Satellite System should not be valid in the current system.*

Channel keys are randomly and independently generated for each system. One system's channel keys will not be able to decrypt another system's encoded frames.

Subscription keys are unique to each device, having been generated from a system-specific random seed and the device ID. Subscription messages intended for a decoder on one system cannot be decrypted by any other decoder (regardless of system), meaning that channel keys from one system cannot be passed to a decoder in another system.

*2b. An attacker should not be able to spoof an Encoder.*

The encoder signs the frame data (channel ID, timestamp, frame content) with the private key of the corresponding channel before encrypting it with the symmetric channel key. Decoders are able to verify the signature using the channel's public key, obtained via subscription update. Even if an attacker could steal the encryption key for a channel, or otherwise modify encrypted frames, they would not be able to change the frame data and update the signature to match, since that requires knowing the private channel key.

***SR3.*** *The Decoder should only decode frames with strictly monotonically increasing timestamps.*

The decoder remembers the timestamp of the last-seen decoded frame and can compare the current frame timestamp against it. If the frame timestamp is not larger, the frame is rejected.

# 4 Threat Analysis and Mitigations

## Remote Code Execution

**Threat**: Attackers may gain the ability to execute arbitrary code on the device by exploiting a vulnerability such as buffer overflow. This could be used to inject code to extract keys, which would allow attackers to decrypt frames for all channels that the device is subscribed to, regardless of timestamp.

**Mitigations**: We minimize the attack surface by reducing the decoder's dependency on functionality from the Maxim Software Development Kit library (MSDK), avoiding interrupts and exceptions, and using a safer programming language (C++ rather than C). The code has been carefully audited for buffer overflow vulnerabilities. The device also handles message payloads that exceed the maximum expected size, by reading them without saving to memory and returning an error (these payloads are either malicious or malformed).

## Power Analysis

**Threat**: Attackers can derive information about the decoder's internal operations by monitoring the power consumption of the device using a probe that can sample voltage or current at high frequency. Certain operations use slightly more power depending on the data being processed, which can be used to deduce keys, given a sufficiently high number of observations.

**Mitigations**: The decoder firmware incorporates random delays throughout its execution paths in order to make it harder to determine when the secrets are being used. Additionally, all decryption operations include decoy decryptions performed using random keys, to make the power profile more difficult to analyze.

## Timing Analysis

**Threat**: Attackers can gain information about computations being performed based on the timing of the decoder's response. By making small adjustments to the input and observing changes in timing, they may be able to deduce keys known to the device.

**Mitigation**: We deny attackers the opportunity to gain information from response times by implementing constant time responses for operations that perform decryptions. Device boot-up always takes around 900ms, Subscribe commands take 450ms, and Decode commands take 90ms (90% of the maximum allowed values). We achieve this by setting a timer that waits until the desired deadline before returning a response.

## Voltage Glitching

**Threat**: Attackers could manipulate the CPU supply voltage, causing it to malfunction in a way that can be exploited, for example by skipping CPU instructions. Specifically, they could precisely time these voltage disturbances in order to bypass security checks or corrupt memory in a way that is beneficial to them. This might allow them to decode any frame on a subscribed channel, regardless of timestamp, without needing to obtain keys.

**Mitigations**: We perform all critical checks twice, with a random delay in between, which makes it very unlikely that both checks could be bypassed. We also duplicate critical memory writes (start/end times for subscriptions) with a random delay in between for the same reason.

## Cryptanalysis

**Scenario**: Attackers may try to infer decryption keys by analyzing the ciphertext-plaintext pairs of encoded and decoded frames. They may also exploit weaknesses in encryption algorithms or implementation mistakes, in order to construct forged messages from valid ones without knowing the key. Brute force key guessing is also a possibility, though very unlikely when using modern ciphers appropriately.

**Mitigations**: Our main defense is the use of ciphers and key lengths that meet current security best practices. Random nonces are used to ensure that all encryption operations use different initialization vectors. Plaintext is padded (before and after) with random salt in order to eliminate any common prefixes or suffixes that could create exploitable patterns in the ciphertext.