Flinders University

MITRE eCTF 2025 Secure Video Decoder System

Technical Manual and User Guide

Team Flinders

Document Version: 2.0

Release Date: 2025/03/11

1. Introduction

1.1. Project Overview

The purpose of this document is to act as a design document and a user guide for owners of the Flinders Secure Video Encoder-Decoder. It describes the functionality of the device, the security features that the device possesses to enable secure decoding of video telecommunication links, and explanations of why said security features have been selected during the design process.

The Flinders Secure Video Decoder is a critical component of a secure satellite TV system, designed to securely decode live-streamed video content transmitted via unidirectional communication links. It operates as part of a broader system, which includes the Uplink, Encoder, Satellite, and Host Computer.

The Decoder runs on an Analog Devices MAX78000FTHR development board, which interfaces with a Host Computer through a USB-serial connection. Its primary responsibilities include maintaining active channel subscriptions, securely decoding TV frame data, and adhering to high-level security protocols designed to protect the integrity and confidentiality of transmitted content.

1.2. Objectives

Primary objectives are:

- Secure encoding/decoding of satellite TV streams.
- Protection against unauthorised channel access.
- Real-time broadcast capabilities.

1.3. Scope

The documentation details the design and implementation of a Secure Encoder/Decoder system for the MITRE Embedded Capture the Flag (eCTF) competition. This system is implemented on the MAX78000FTHR development board. The document covers system implementation, security features, and operational procedures, all within the constraints of hardware and MITRE rules. The deliverable consists of firmware implementation and technical documentation.

2. System Considerations

2.1. 2.1. MAX78000FTHR

The MAX78000FTHR microcontroller, manufactured by Analog Systems, features three main subsystems that define its operational capabilities for this project:

- Processor subsystem
- Memory subsystem
- Power subsystem

2.2. Processor Subsystem

At its core, the processor subsystem consists of a 100MHz ARM Cortex-M4 processor, and a 60MHz RISC-V co-processor. Given the processor's low computing power, the onboard AES hardware accelerator makes AES encryption an optimal method for cryptographic operations. The system also implements secure boot functionality to protect code integrity.

2.3. Memory Subsystem

The memory subsystem includes 512KB of flash memory, 128KB of SRAM, a 16KB cache and a Boot ROM. Memory management requires careful consideration as 128KB SRAM is shared across all applications, creating a constrained environment. The 512KB flash memory must accommodate program code and application data.

2.4. Power Subsystem

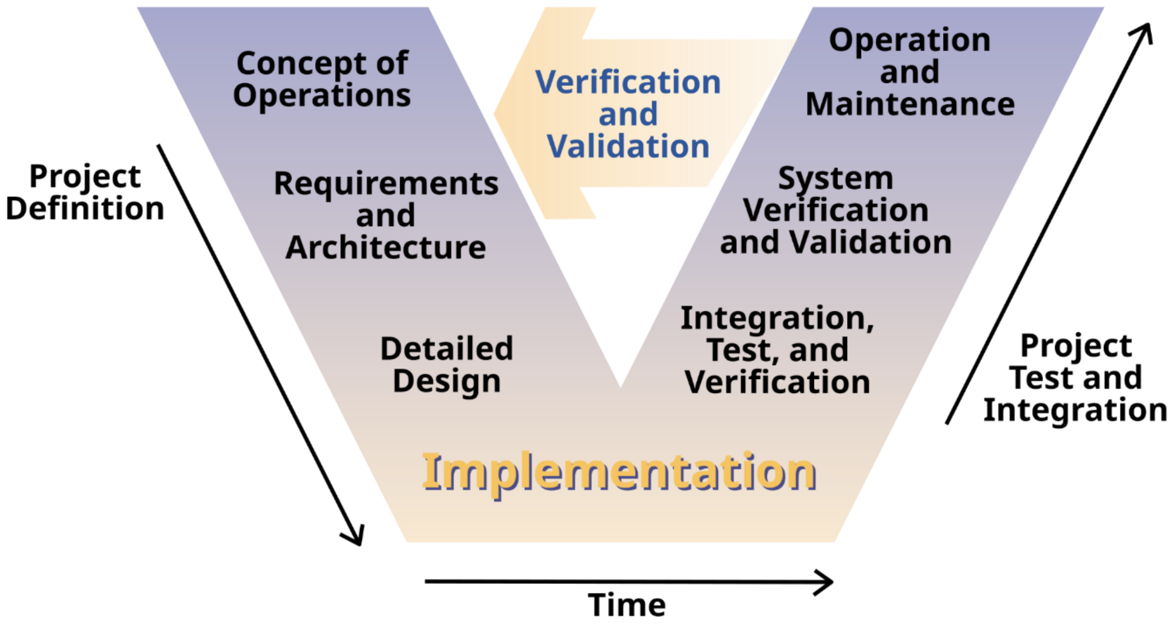
The power subsystem operates within a voltage range of 1.8V and 3.3V. It offers three power modes: active mode for high power consumption, sleep mode for limited functionality, and backup mode for minimal operations. While the microcontroller includes comprehensive power management features, the current implementation uses USB power, making energy-efficient considerations less critical. This power configuration eliminates concerns about battery management and power optimisation.

These specifications establish the constraints that guide the implementation of secure encoder and decoder systems.

3. Systems Engineering and Architecture

3.1. Introduction

The approach used for Flinders Secure Video Encoder-Decoder system is a standard *systems engineering*. This involves defining the "Concept of Operations" - an envisioning of how the system will work once developed, undertaking a needs analysis to develop qualitative objectives, and then development of requirements and a system functional architecture.


Concept of Operations
Verification and Validation
Operation and Maintenance
Project Definition
Requirements and Architecture
System Verification and Validation
Detailed Design
Integration, Test, and Verification
Project Test and Integration
Implementation
Time

"V-Model" Systems Engineering Process

This approach allows for planning from a high-level, as informed by the concept of operations, all the way down to extremely granular design implementations, and then later testing, verification and validation to ensure that the original objectives of the design have been met. Note that this approach does not lock the team into a "waterfall" design methodology, and instead provides a framework under which work can be planned.

The following sections will outline the envisioned concept of operations, a needs analysis (including threat modelling), and then verifiable requirements arising from this analysis and modelling. Finally, the section ends with a functional architecture of the system, outlining a high-level design for the encoder.

3.2. Concept of Operations

The video encoder-decoder system will allow the secure transmission and decoding of TV frame data using a unidirectional satellite link. The system includes a Uplink, an Encoder, a Host Computer, and a Decoder.

The Uplink sends raw TV frames to the Encoder, which encodes the data and transmits it to the Satellite for broadcast. The Satellite then broadcasts the encoded data to all listening Host

Computers, which then forwards the frames to a connected Decoder over a USB-serial interface.

The Decoder, implemented on an Analog Devices MAX78000FTHR development board, is responsible for securely decoding the frames. This includes decryption and integrity verification before reconstructing the original TV frames. The Decoder also manages active subscriptions to multiple data channels, ensuring that only the correct frames are processed.

Once decoding is complete, the frames are sent back to the TV application, which displays them on-screen.

3.3. Needs Analysis

The design of the Flinders Secure Video Encoder-Decoder System was shaped by a combination of mandatory requirements and additional requirements identified by the Flinders team to enhance functionality, security, and system robustness.

The mandatory functional and security requirements establish core capabilities such as channel management, secure subscription handling, and frame decoding. These requirements were imposed to meet cybersecurity standards recommended by the MITRE Corporation.

To determine relevant requirements beyond the mandatory standard, a threat analysis was also undertaken as detailed in the next section of the document. This analysis informed the design team’s understanding of the threat landscape, including likely vulnerabilities, common weaknesses, and attack vectors. Following the threat analysis, the Flinders team identified and implemented additional requirements to improve system resilience against likely attack vectors. These enhancements are intended to introduce an advanced level of security, beyond the baseline specifications.

Together, these two sets of requirements guided the development of the Encoder and Decoder, ensuring a secure, efficient, and reliable video transmission system.

3.4. Requirements

3.4.1. Mandatory Requirements

3.4.1.1. Functional

1) The Decoder shall maintain a list of channels with valid subscriptions.
2) The Decoder shall require a valid subscription to decode TV frame data for any non-emergency channel.
3) The Decoder shall process subscription updates through valid update packages.
4) The Decoder shall only use the most recent subscription update for each channel.
5) The Decoder shall always decode emergency broadcast channel (channel 0) frames without requiring a subscription.
6) The Decoder shall decode TV frame data from all channels with valid, active subscriptions.
7) The Decoder shall decode all emergency broadcast frames.

3.4.1.2. Security

8) The Decoder shall prevent frame decoding for channels without valid, active subscriptions.
9) The Decoder shall only decode TV frames from its provisioned Satellite System.
10) The Decoder shall only decode frames with strictly monotonically increasing timestamps.

3.4.2. Additional Requirements

3.4.2.1. Architectural

11) The Decoder shall implement FreeRTOS, a microkernel-based real-time operating system, to enforce both isolation and task scheduling.
12) The Decoder shall utilise the Cortex-M4 Memory Protection Unit (MPU) to enforce permission-based access to memory.
13) The Decoder shall run application-level processes with restricted privileges.
14) The Decoder shall compartmentalise processes to prevent faults from affecting the whole system.

3.4.2.2. Memory Management

15) The Decoder shall utilise memory-safe code within the FreeRTOS environment.
16) The Decoder shall utilise memory-safe code for peripheral drivers.
17) The Decoder shall utilise memory-safe code for system input handling.
18) The Decoder shall validate any data handled by non-memory-safe code with a memory-safe air gap.
19) The Decoder shall zeroise Flash Memory where it would not interfere with operation.

3.4.2.3. System Initialisation

20) The Decoder shall perform a whole-of-system health check during initialisation.
21) The Decoder shall validate the integrity of the process table during initialisation.
22) The Decoder shall initialise all cryptographic processes prior to allowing input.
23) The Decoder shall disable all unused peripherals.
24) The Decoder shall take 1 second to fully initialise from boot.

3.4.2.4. Timing and Clocks

25) The Decoder shall monitor system clock stability through a Clock Monitor.
26) The Clock Monitor shall monitor clock cycle ratios to assure expected behaviour.

3.4.2.5. Process Management

27) The Decoder shall use FreeRTOS scheduler to manage task properties and execution.
28) The Decoder shall use FreeRTOS task prioritisation to assign the highest priority level to critical processes: crypto manager, validation manager, serial interface manager, subscription manager, and channel manager.
29) The Decoder shall monitor system health through a Sentry Process.
30) The Decoder shall maintain a Failsafe Process for system recovery.
31) The Decoder shall monitor system performance through an Error Manager.
32) The Error Manager shall monitor other processes for unexpected behaviour, and log errors if they occur.
33) The Sentry Process shall monitor other processes for unexpected behaviour, and will activate a Failsafe Process if more than 1 error occurs.
34) The Failsafe Process shall enforce system integrity by rebooting the Decoder at the request of the Sentry Process.

3.4.2.6. Input Management

35) The Decoder shall process serial input at 115200 baud.
36) The Decoder shall use 8N1 UART configuration.
37) The Decoder shall discard transmissions not conforming to 115200 baud and 8N1.
38) The Decoder shall not enable hardware flow control.
39) The Decoder shall not enable software flow control.

3.4.2.7. Cryptography and Validation

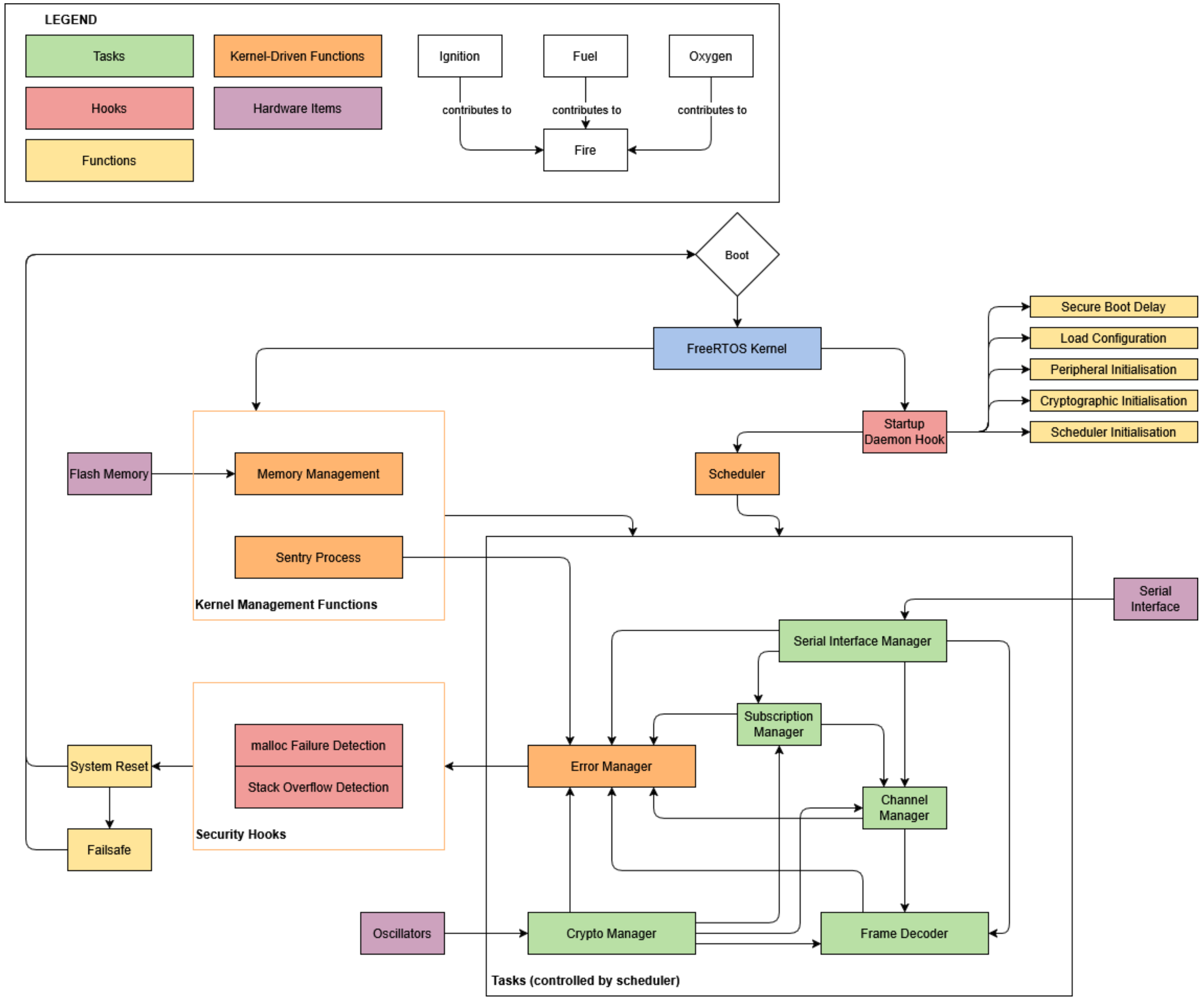
40) The Decoder shall make use of the AES encryption algorithm in a Crypto Manager process.
41) The Crypto Manager shall make use of the AES hardware acceleration engine.
42) The Decoder shall utilise message integrity checks (MIC).

3.5. Functional Architecture

3.5.1. Functional Architecture Description

With the needs analysis, threat modelling, and requirements developed, the functional architecture of the system begins to take shape. An initial functional architecture is provided below, noting that it may change as the design is implemented if it is discovered that some aspect of the design is not possible, or there is a better way to meet a requirement.

3.5.2. Functional Architecture Diagram



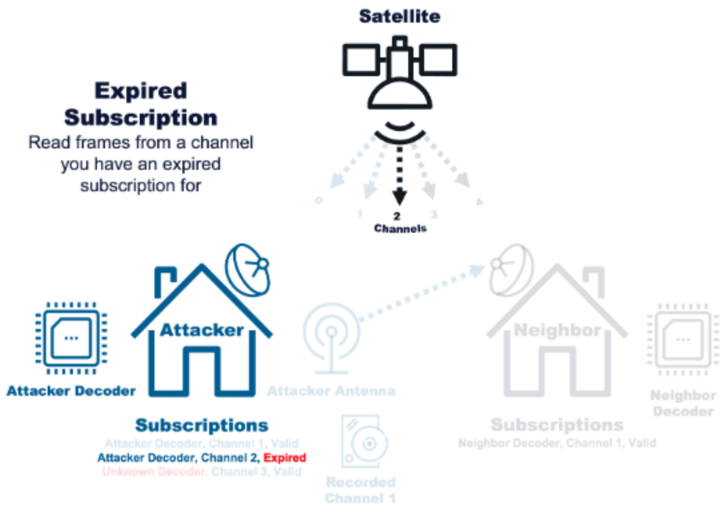
4. Threat Analysis

The Threat Analysis identifies potential vulnerabilities across software, hardware, and network layers. Each threat is classified by its type, attack vector, impact, and mitigation strategy. The analysis ensures system security, integrity, and reliability. Mitigating these threats is critical in safeguarding sensitive data, preventing unauthorised access, and ensuring reliable connectivity. While some attacks and their mitigation mechanisms are beyond the scope of this discussion, it is important to acknowledge their existence.

4.1. Expected Attack Scenarios

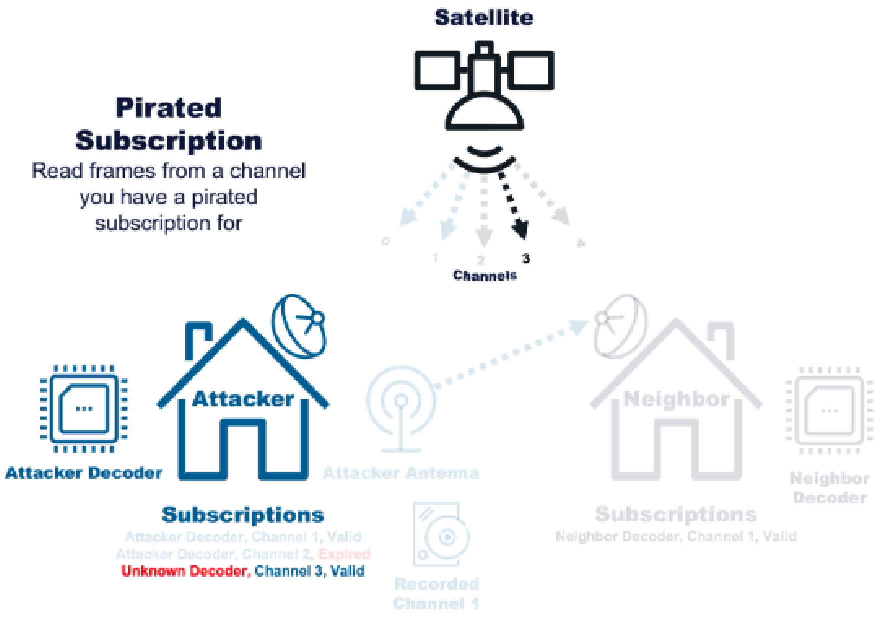
Specific attack scenarios expected against a decoder were modelled and defences against these scenarios have been integrated into the system's design.

4.1.1. Expired Subscription Attack


Satellite
Expired Subscription
Read frames from a channel you have an expired subscription for
0
1
2
Channels
3
4
Attacker Decoder
Attacker
Attacker Antenna
Neighbor
Neighbor Decoder
Subscriptions
Attacker Decoder, Channel 1, Valid
Attacker Decoder, Channel 2, Expired
Unknown Decoder, Channel 3, Valid
Recorded Channel 1
Subscriptions
Neighbor Decoder, Channel 1, Valid

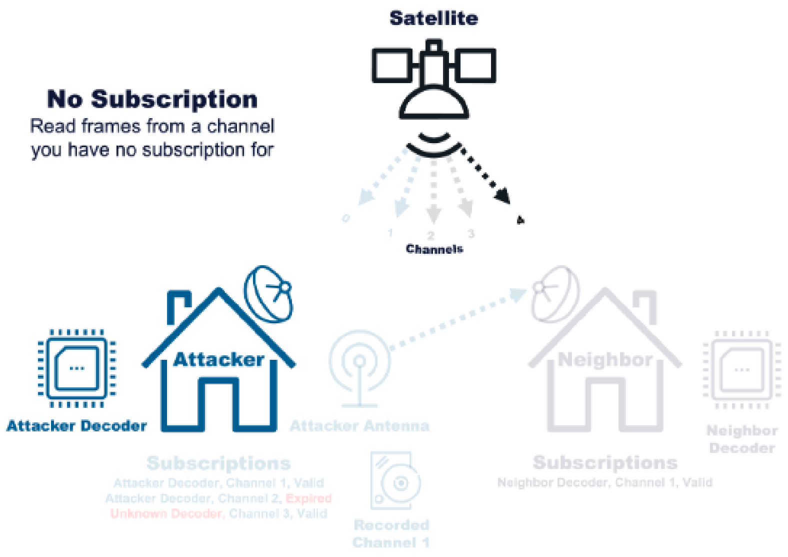
The expired subscription attack assumes the attacker has a previously valid subscription, essentially the contents of a subscription update message.

4.1.2. Pirated Subscription Attack


Pirated Subscription
Read frames from a channel you have a pirated subscription for
Satellite
0
1
2
3
Channels
Attacker Decoder
Attacker
Attacker Antenna
Neighbor
Neighbor Decoder
Subscriptions
Attacker Decoder, Channel 1, Valid
Attacker Decoder, Channel 2, Expired
Unknown Decoder, Channel 3, Valid
Recorded Channel 1
Subscriptions
Neighbor Decoder, Channel 1, Valid

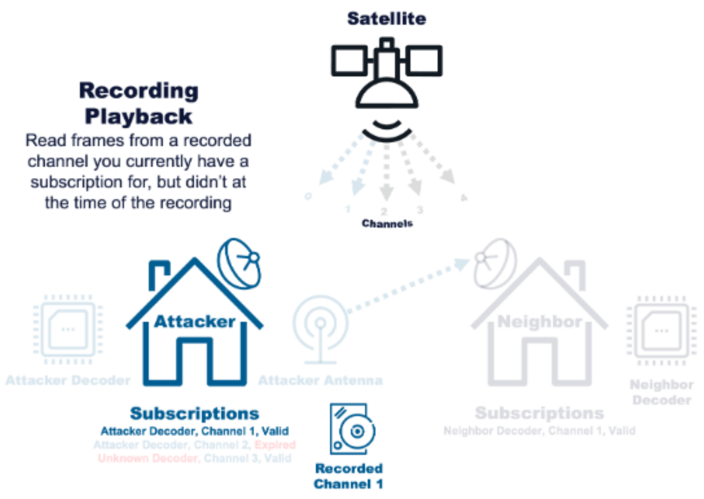
The pirated subscription attack relies on some decoder having a pirated subscription enabling it to decode frame data.

4.1.3. No Subscription Attack


No Subscription
Read frames from a channel you have no subscription for
Satellite
0
1
2
3
4
Channels
Attacker Decoder
Attacker
Attacker Antenna
Neighbor
Neighbor Decoder
Subscriptions
Attacker Decoder, Channel 1, Valid
Attacker Decoder, Channel 2, Expired
Unknown Decoder, Channel 3, Valid
Recorded Channel 1
Subscriptions
Neighbor Decoder, Channel 1, Valid

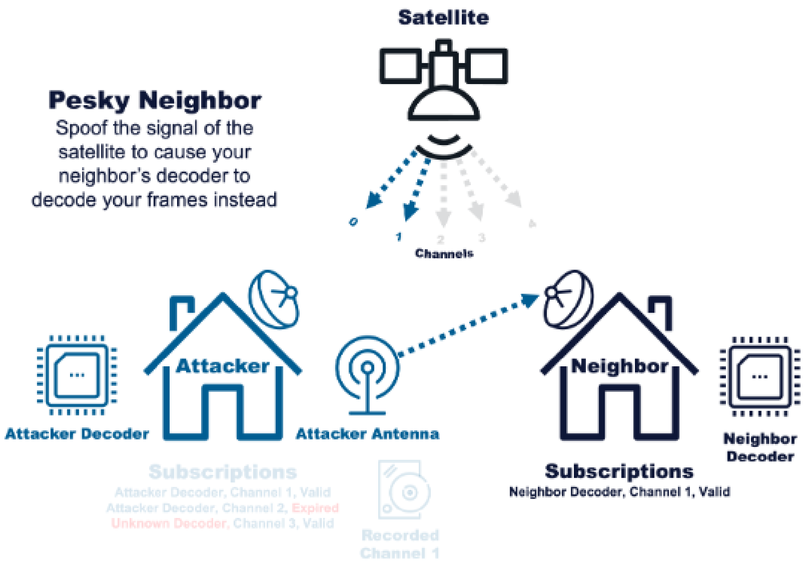
In this attack, an attacker aims to decode frame data without a subscription.

4.1.4. Recording Playback (Replay) Attack


Satellite
Recording Playback
Read frames from a recorded channel you currently have a subscription for, but didn't at the time of the recording
Channels
Attacker
Attacker Decoder
Attacker Antenna
Neighbor
Neighbor Decoder
Subscriptions
Attacker Decoder, Channel 1, Valid
Attacker Decoder, Channel 2, Expired
Unknown Decoder, Channel 3, Valid
Recorded Channel 1
Subscriptions
Neighbor Decoder, Channel 1, Valid

In this attack, the attacker is given a valid subscription and tries to decode previous frames.

4.1.5. Pesky Neighbour (Spoofing) Attack


Satellite
Pesky Neighbor
Spoof the signal of the satellite to cause your neighbor's decoder to decode your frames instead
0
1
2
3
Channels
Attacker
Attacker Decoder
Attacker Antenna
Neighbor
Neighbor Decoder
Subscriptions
Attacker Decoder, Channel 1, Valid
Attacker Decoder, Channel 2, Expired
Unknown Decoder, Channel 3, Valid
Recorded Channel 1
Subscriptions
Neighbor Decoder, Channel 1, Valid

In this attack, an attacker creates fake packets and tries to get a friendly decoder to decode the frames.

5. Decoder Implementation

5.1. Introduction

This section outlines the implementation of the functional architecture and the realisation of the final design. To meet the team's requirements for simplicity and security, FreeRTOS was selected as the real-time operating system (RTOS).

The most important distinction between a standard superloop implementation and an RTOS-based approach is task execution. In a traditional superloop, all tasks run sequentially, whereas in Team Flinders' solution, tasks execute in parallel. This is enabled by FreeRTOS, which provides a scheduler that distributes CPU time among tasks according to their assigned priority.

5.2. Differences from standard C superloop

FreeRTOS introduces two key concepts beyond standard C functions: hooks and tasks.

1. **Hooks** in FreeRTOS are special functions that execute at predefined system events, such as idle time or system ticks. They allow developers to insert custom behaviour without modifying the RTOS kernel. The Team Flinders decoder uses the **daemon startup hook**, **stack overflow hook**, and **malloc failure hook**:
   - The *daemon startup hook* runs when the scheduler starts, allowing initialisation of background tasks. It initialises the true random number generator, implements a randomised security delay at boot time, and prints a welcome message.
   - The *stack overflow hook* and *malloc failure hook* both call `system_reset()`, which first attempts to gracefully reboot the device, and then invokes `failsafe()`. This critical failure handling function disables all interrupts, attempts a device restart, and, if unsuccessful, places the device into an infinite loop to prevent unpredictable behaviour. It prints output to the console to keep the user aware of what is going on internally.
   - These hooks can be found in **src/FreeRTOS_hooks.c**.
2. **Tasks** are independent execution units managed by the FreeRTOS scheduler. Each task consists of a function running in an infinite loop and is scheduled based on its priority. Tasks can be created dynamically or statically and may communicate via queues, semaphores, or other synchronisation mechanisms. Team Flinders' decoder has distributed functionality among tasks as follows:
   - crypto_manager
   - subscription_manager
   - channel_manager
   - frame_manager
   - serial_interface_manager

These tasks are called by the scheduler as needed, and together provide the decoder's full functionality. Each task is given a fixed amount of memory, and if it exceeds that memory, then either the stack overflow or memory allocation hooks will be called to prevent attacks based on undefined behaviour arising from memory problems.

5.3. Configuration

The FreeRTOS configuration file can be found in **decoder/FreeRTOSConfig.h**. This file enables and disables features, as was deemed necessary for the correct operation of the secure decoder. Detailed information on the configuration items can be found here: https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/02-Customization.

In general, the Team Flinders secure video decoder is configured as follows:

5.3.1. Modes

- Preemptive (priority-based) scheduling is enabled, rather than cooperative scheduling.
- Tickless idle mode is disabled, due to no need for power saving modes.

5.3.2. Clocks

- The clocks are set to match the CPU clock and the board's real-time clock oscillator.
- The tick rate is set to 1kHz, so the RTOS ticks every 1ms.
- A 32-bit tick counter is in use.

5.3.3. Informational

- The trace facilities are set to provide some information about the tasks at boot time.

5.3.4. Scheduler

- There are five distinct priority levels for tasks.
- The minimum stack size is 128 words (32 bits = 4 bytes = 1 word).
- The idler, though disabled, is set to never surrender the CPU if it is somehow enabled.
- The device is using standard mutexes (mutual exclusions).
- The device is not using semaphores or queues at all.
- The device is utilising time slicing, which allows tasks of equal importance to share CPU time.

5.3.5. Memory

- Sections of the heap reserved by a task are cleared when freed by that task.
- Both static and dynamic memory allocation are enabled.
- The total heap size is 64KB.

5.3.6. Security

- Stack overflow detection is enabled via hook, using method 3.
    - Method 3 details available here: https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/09-Memory-management/02-Stack-usage-and-stack-overflow-checking)
- Memory allocation (malloc) failure detection is enabled via hook.

The configuration is designed to support system architecture with compartmentalised processes, secure memory management, and prioritised execution of security-critical components such as crypto manager and validation manager.

6. Cryptography

6.1. Introduction

The scenarios listed in section 4 were used to inform the development of the system architecture and cryptographic scheme. This provides basic system security; however, if the secret keys are leaked to an attacker, the attacker will be able to perform all of the aforementioned attacks.

6.2. Cryptographic Principles

This section describes the cryptographic principles and the reasoning used to influence our design decisions.

1. **Use AES wherever possible**

The design aims to use Advanced Encryption Standard (AES) and its derivatives wherever possible as the microcontroller used on the decoder has an AES accelerator keeping cryptographic performance acceptable.

2. **Use AES CTR for block chaining**

AES CTR ensures that the output for each block in the stream cipher is always different, even if the inputs are the same. This prevents pattern leakage and ensures that there are no recognisable patterns.

3. **Never reuse the same AES CTR none**

AES CTR uses a nonce which is incremented for each block. This block incrementing value is block encrypted and block XORed with plain text. If the same nonce is ever used pattern leakage can still happen so the nonce must be randomly generated. Consequently, some portion of the nonce must be randomly generated. However, the same nonce can be used twice if the input key for the cipher is different as the encrypted block counter output will be different.

4. **Never use the same cryptographic key twice on cryptographic material visible to attackers**

Even if attackers extract partial information from one encrypted packet, switching to a new key for every packet ensures that the leaked data cannot be combined with future transmissions to break the encryption. A key derivation function is used to generate different keys for each packet. However some

5. **Use AES CTR for KDF**

AES CTR with a random nonce ensures that the same key will not be derived twice.

6. **Do not use global secret cryptographic material raw in cryptographic material exposed to attacks**

This reduces the scope for accidental leakage of global secrets which must be kept secret at all costs. If the global secrets get leaked the attacker can easily get through the cryptographic implementation and decrypt all data. Consequently, global secrets are used via a KDF to encrypt and sign data.

7. **Base KDF input on as much context as possible**

Basing the input to the KDF on as much context as possible makes it harder for attackers to derive the same key as the attacker also needs to know the same context which changes for each packet.

8. **AES CMAC packet signature output should never be the same for multiple packets**

In AES CMAC, each packet's signature is generated based on its unique content (often combined with a nonce or sequence number). If the same signature appears on multiple packets, attackers could figure out that those packets share similar properties. This could result in replay attacks or even help the attacker forge valid signatures, ultimately compromising the integrity of the communication. Using derived keys that are never repeated prevents the signature from being the same even if the packet data is the same.

9. **Security through depth**

Where possible, implement multiple, independent layers of cryptographic protection. If one layer is broken, attackers must still overcome additional layers to get in.

6.3. Global Secrets

The global secrets are known only by the encoder and friendly decoders. Consequently, the information in these secrets is used to secure the communication between the broadcaster and friendly decoders attached to clients' TVs.

The global secrets contain:

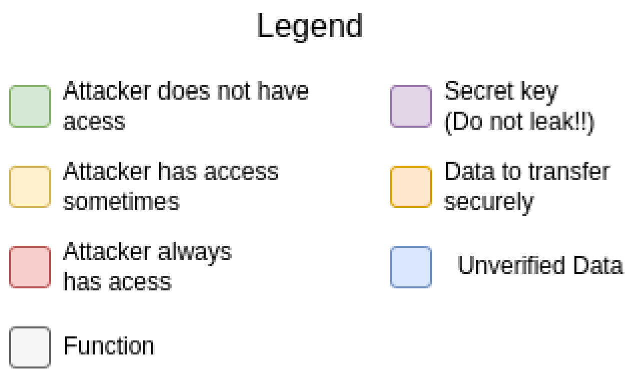
- Subscription key derivation function key
- Subscription cipher auth tag
- Frame key derivation function key
- Flash key derivation function key
- Flash key derivation function input magic bytes
- Channels active in the deployment
- Channel key derivation function input magic bytes for each active channel

The following sections detail how these global secrets are used to provide secure communication.

6.4. Subscription Update

Subscription updates are used by the broadcaster to provide subscription information to friendly decoders. Channel 0 is the emergency broadcast channel, so the decoder does not require a subscription update to decode the frames on this channel. Subscription encode is performed in “design/ectf25_design/gen_subscription.py” and subscription decode in “decoder/core/src/subscription.c”

The figure below shows the legend used in the block diagrams.

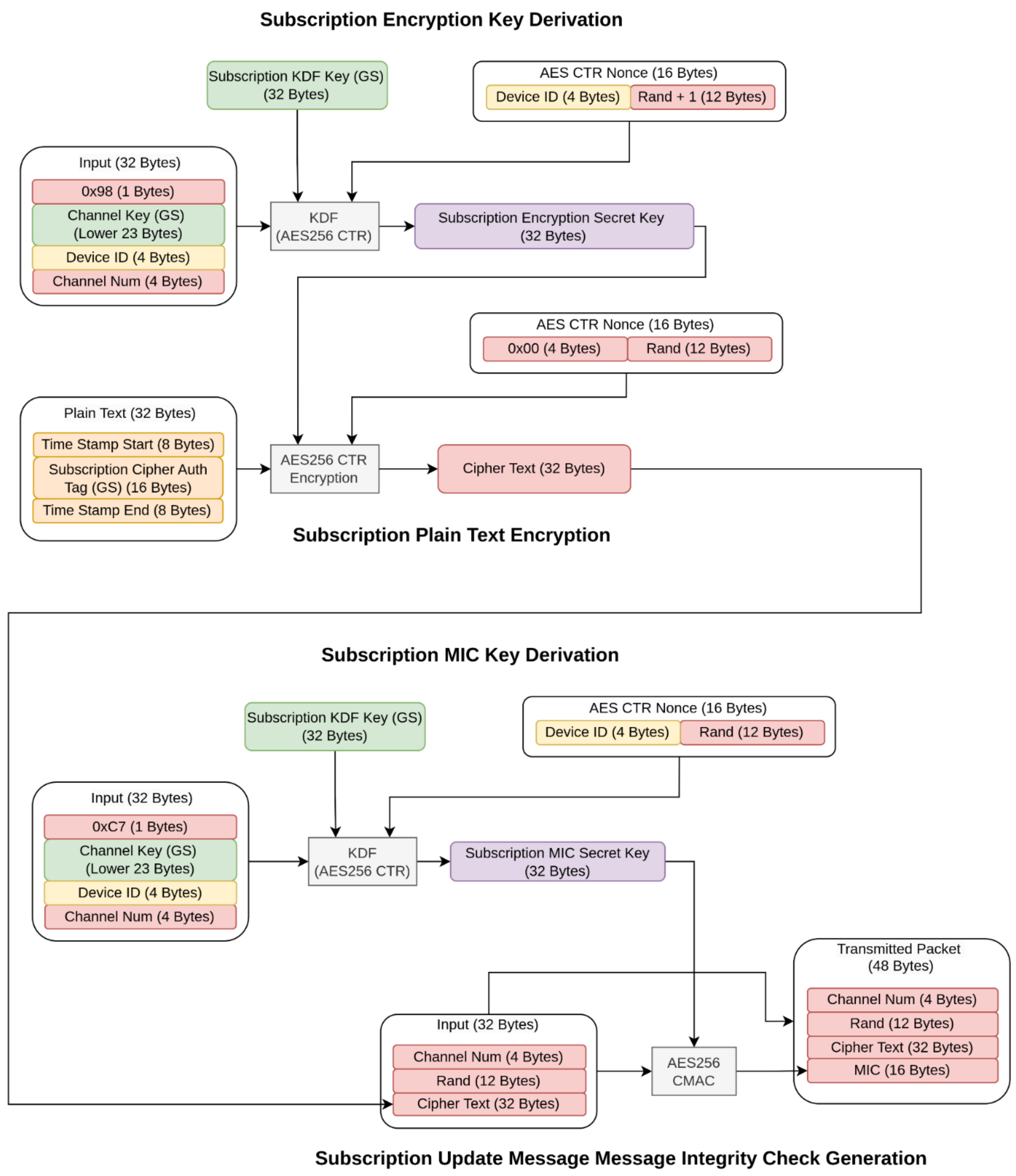


Legend
Attacker does not have acess
Secret key (Do not leak!!)
Attacker has access sometimes
Data to transfer securely
Attacker always has acess
Unverified Data
Function

Block Diagram Legend

6.4.1. Subscription Update Encode

This section covers the subscription update encode process run with python at the broadcaster. This process is illustrated in the figure below.

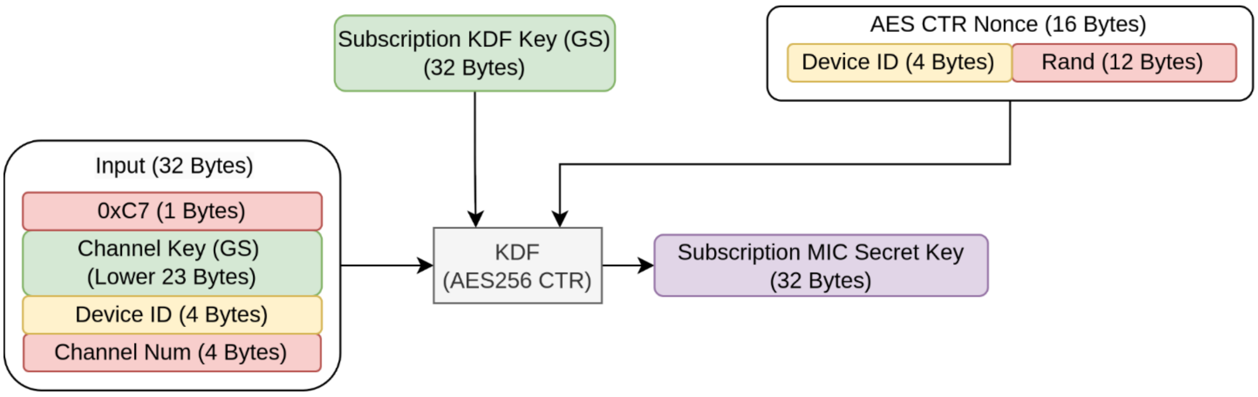


Subscription Update Encode Overview

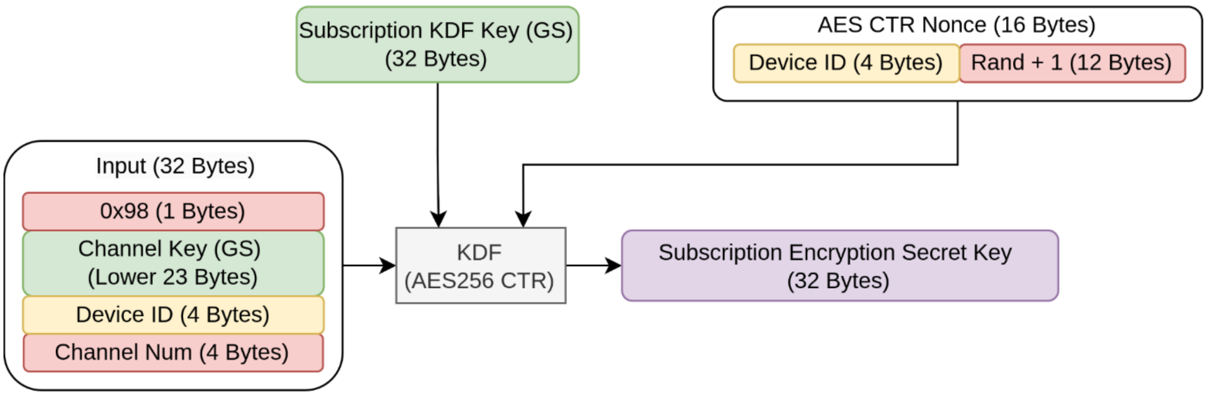
The subscription update command sends the following information to the decoder: the channel number, start and end timestamp to determine when the subscription for a particular channel is active. The design requires no cryptographic information to be sent in the subscription update packet, as the frame encoding and decoding rely on the global secrets being known only to friendly decoders. Encryption is used to hide information and a message integrity check (MIC) is used to sign the packet to prevent tampering.

The first part of the cryptographic process for the subscription update is deriving the subscription encryption and MIC key. The encryption key is used to encrypt the plain text. This

MIC key is used to sign the subscription update message sent to the decoder. In line with Principle 3, the same key can never be used twice so two keys must be derived for the two cryptographic processes. If an attacker modifies the subscription update message, the decoder could decode frame data even though it does not have a valid subscription. This MIC signature prevents this. Consequently, these two keys must be able to be derived at both the subscription update service and the decoder. The derivation of these keys is shown in the two figures below.



Subscription Update MIC Key Derivation Overview



Subscription Update Encryption Key Derivation Overview

In this design, the KDF used is AES256 CTR. The key for the KDF is the subscription KDF key stored in global secrets. The input for the KDF is a magic byte, part of the channel key, stored in global secrets, device ID and channel number. The decoder can derive the same key as the subscription KDF key, and the channel key is stored in the global secrets. The device ID is stored in the firmware, and the channel number is in the transmitted frame. The rand bytes for the nonce are also in the transmitted frame.

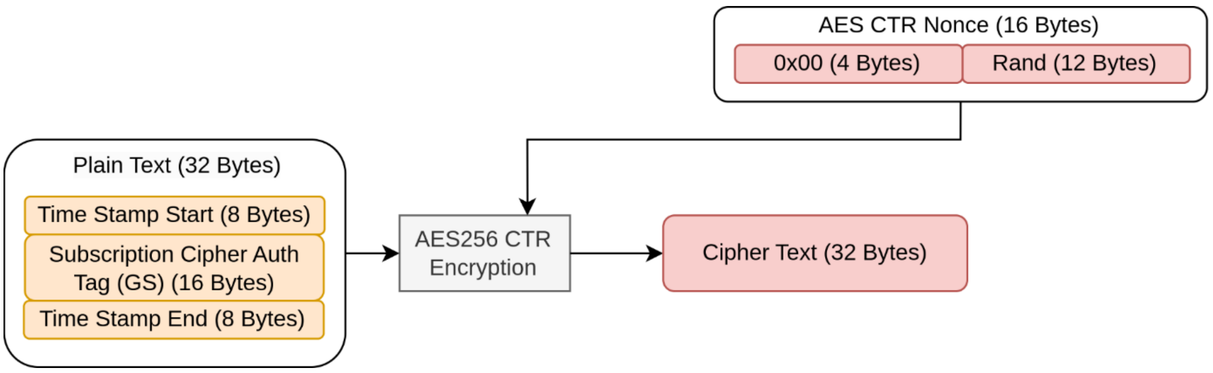
The KDF input structure byte offsets are as follows:

- [0]: Magic byte (0xC7 for MIC KDF, 0x98 for Encryption KDF)
- [1:23]: Lower 23 bytes of channel key for channel “Channel Num”
- [24:27]: Device ID (Little Endian)
- [28:31]: Channel number (Little Endian)

As the KDF is based on global secrets, this prevents an attacker from being able to derive the same keys to calculate a new valid MIC, if it modifies the packet, or decrypts the packet. The input magic byte is the only difference between the two keys however due to the nature of AES CTR, the output stream is vastly different.

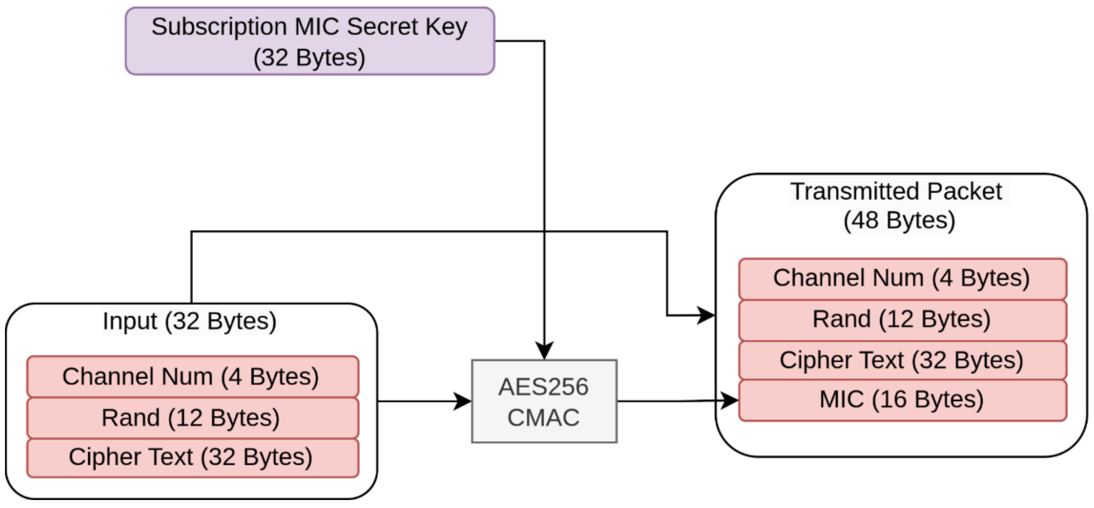
Using a KDF and not the subscription KDF key raw to sign and encrypt the data provides several benefits as listed in Principle 6. First, the message data is checked twice. If any part of the subscription update message has been changed the derived key will be different resulting in the calculated MIC being wrong. The MIC will also be wrong due to the wrong data and key. Additionally, if due to a poor implementation, the system leaks the subscription secret key, it is only valid for one transmission. Every subscription update message will have a different key due to different inputs to the KDF and random nonce.

The encryption flow is shown in the figure below. While encryption is not necessary it adds another layer of security. The encryption of the timestamp start and end adds extra randomness and the subscription cipher auth token, stored in global secrets, adds an extra security check to the subscription update process.



Subscription Update Encryption Overview

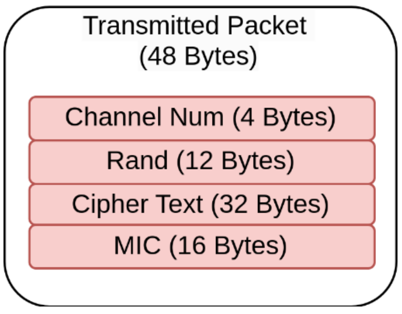
The MIC is calculated using AES256 CMAC (Chiper-Based Message Authentication Code) using the aforementioned derived subscription MIC key and the output 16 bytes are attached to the end of the subscription update frame. This message is sent to the TV and subsequently to the decoder. The contents of the subscription update message is shown below.



Subscription Update MIC Calculation Overview

The subscription update packet structure is illustrated in the figure below and bytes offsets are additionally listed.

- [0:3] Channel Number (Little Endian)
- [4:15] Rand Nonce (Big Endian)
- [16:47] Cipher Text
- [48:63] MIC

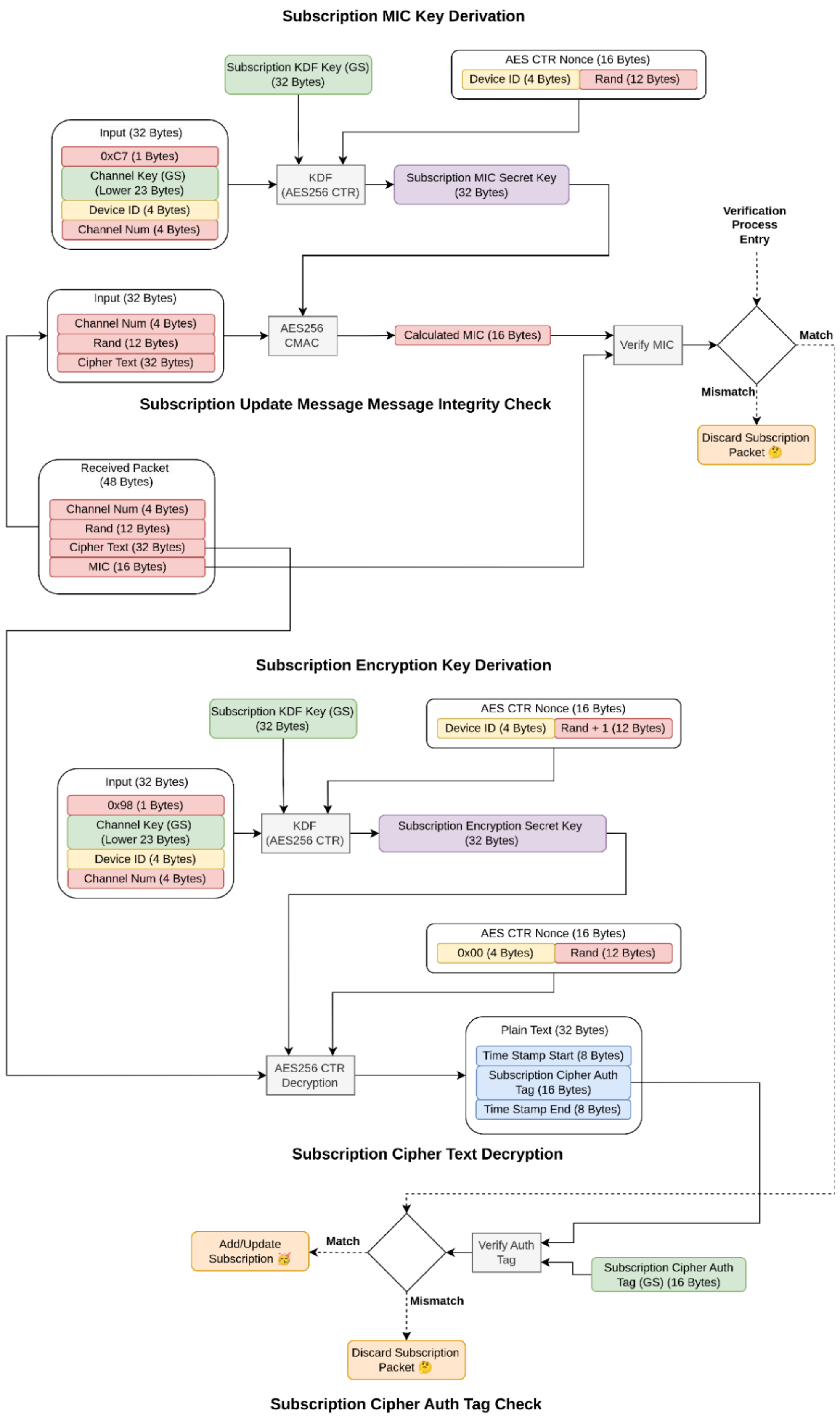


Subscription Update Packet Structure

Multiple subscription update packets can be appended together and the decoder will process them all independently. The next section covers the subscription update decode process which runs on the decoder.

6.4.2. Subscription Update Decode

This section covers the subscription update decode process which is fed packets from the TV to decode and update the list of valid subscriptions if valid. The subscription update decode process involves running the same KDFs and validating the MIC and subscription cypher auth tag. This process is illustrated in the below figure.


Subscription MIC Key Derivation
Subscription KDF Key (GS) (32 Bytes)
AES CTR Nonce (16 Bytes)
Device ID (4 Bytes)
Rand (12 Bytes)
Input (32 Bytes)
0xC7 (1 Bytes)
Channel Key (GS) (Lower 23 Bytes)
Device ID (4 Bytes)
Channel Num (4 Bytes)
KDF (AES256 CTR)
Subscription MIC Secret Key (32 Bytes)
Verification Process Entry
Input (32 Bytes)
Channel Num (4 Bytes)
Rand (12 Bytes)
Cipher Text (32 Bytes)
AES256 CMAC
Calculated MIC (16 Bytes)
Verify MIC
Match
Mismatch
Discard Subscription Packet
Subscription Update Message Message Integrity Check
Received Packet (48 Bytes)
Channel Num (4 Bytes)
Rand (12 Bytes)
Cipher Text (32 Bytes)
MIC (16 Bytes)
Subscription Encryption Key Derivation
Subscription KDF Key (GS) (32 Bytes)
AES CTR Nonce (16 Bytes)
Device ID (4 Bytes)
Rand + 1 (12 Bytes)
Input (32 Bytes)
0x98 (1 Bytes)
Channel Key (GS) (Lower 23 Bytes)
Device ID (4 Bytes)
Channel Num (4 Bytes)
KDF (AES256 CTR)
Subscription Encryption Secret Key (32 Bytes)
AES CTR Nonce (16 Bytes)
0x00 (4 Bytes)
Rand (12 Bytes)
Plain Text (32 Bytes)
Time Stamp Start (8 Bytes)
Subscription Cipher Auth Tag (16 Bytes)
Time Stamp End (8 Bytes)
AES256 CTR Decryption
Subscription Cipher Text Decryption
Add/Update Subscription
Match
Verify Auth Tag
Subscription Cipher Auth Tag (GS) (16 Bytes)
Mismatch
Discard Subscription Packet
Subscription Cipher Auth Tag Check

Subscription Update Decode Overview

The subscription decode process is similar to the encode process with a few exceptions. The KDFs are based on the packet channel number and rand bytes. The calculated MIC is compared with the packet MIC and if there is a mismatch the packet is discarded. Subsequently, the cipher text is decrypted and the cipher auth tag is compared with the one in global secrets as another security check that the packet has not been modified.

If all checks match the subscription is updated if a subscription already exists for the channel else it is added to the list of valid subscriptions. Subscription information is stored in flash to ensure that subscriptions persist across power cycles.

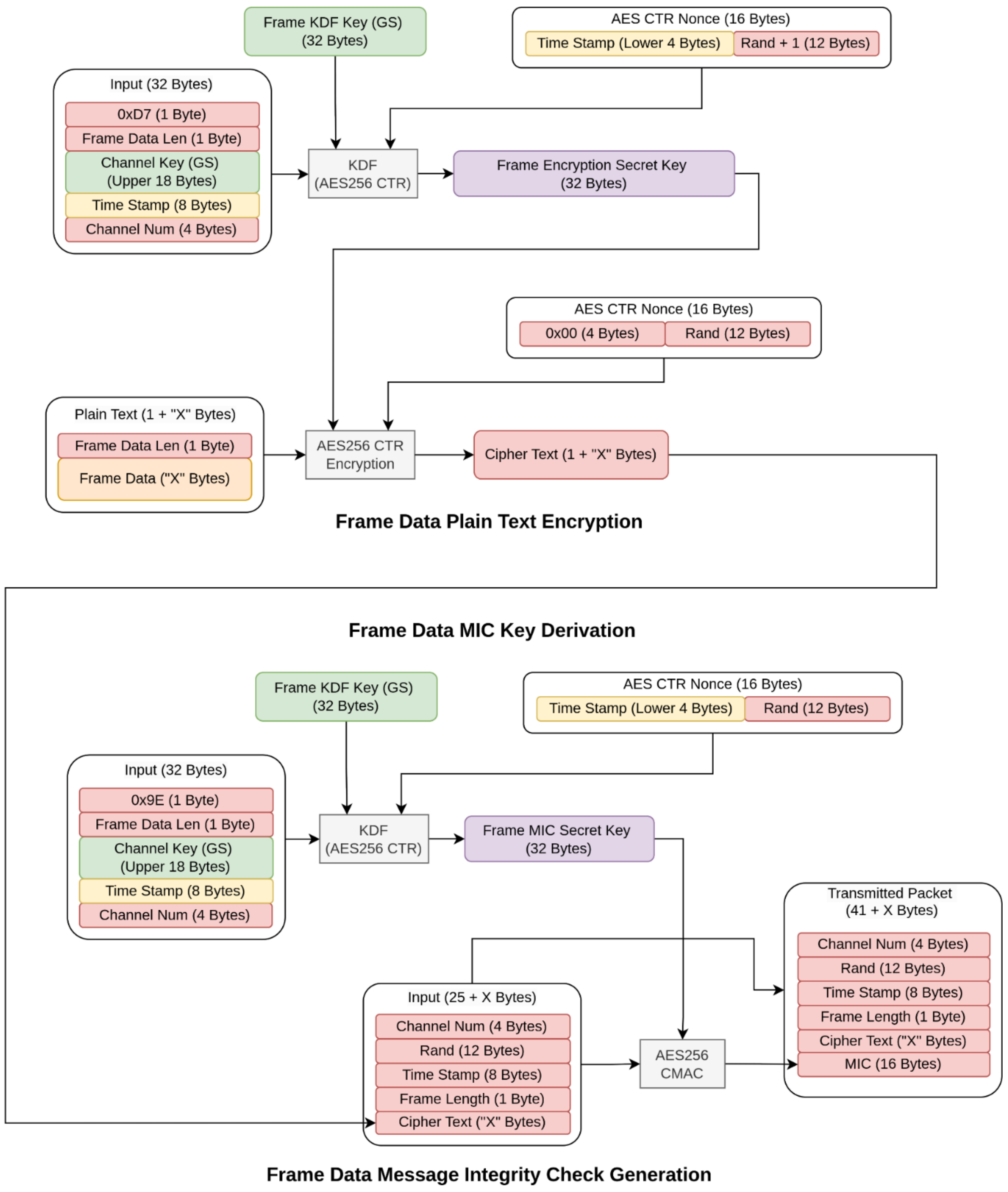
Subscriptions can now be securely transmitted to each friendly device. The next section discusses the secure frame transmission design.

6.5. Frame Transmission

This section discusses the design used to encode frames at the broadcaster and ensure secure transmission to friendly decoders which securely decode and send the frames back to the TV. Frame transmission encode is performed in “design/ectf25_design/encoder.py” and frame transmission decode in “decoder/core/src/frame.c”

6.5.1. Frame Transmission Encode

This section covers the frame transmission encoded process run with python at the broadcaster to securely broadcast their frames to clients with valid subscriptions. The frame transmission encoding process follows a similar approach as the subscription update process except with the addition of variable length frame data and no cipher auth tag in the plain text. This process is illustrated in the figure below.



Frame Transmission Encoding

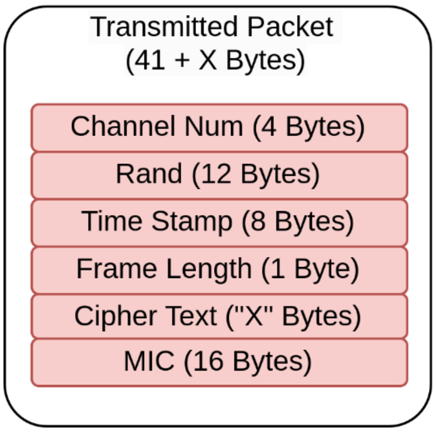
The KDF input structure byte offsets are as follows:

- [0]: Magic byte (0x9E for MIC KDF, 0xD7 for Encryption KDF)
- [1]: Frame data length
- [2:19]: Upper 18 bytes of channel key for channel “Channel Num”
- [20:27]: Time Stamp (Little Endian)
- [28:31]: Channel number (Little Endian)

The KDF, based on AES256 CTR, is used to derive the frame MIC and encryption keys. The key will change each transmission due to the timestamp always monotonically increasing. All data in the transmitted frame is signed with AES256 CMAC to prevent an attacker from modifying the data.

The frame transmission packet structure is illustrated in the figure below and bytes offsets are additionally listed.

- [0:3] Channel Number (Little Endian)
- [4:15] Rand Nonce (Big Endian)
- [16:23] Time Stamp
- [24] Frame Length
- [25:25+X-1] Cipher Text
- [25+X:25+X+16-1] MIC

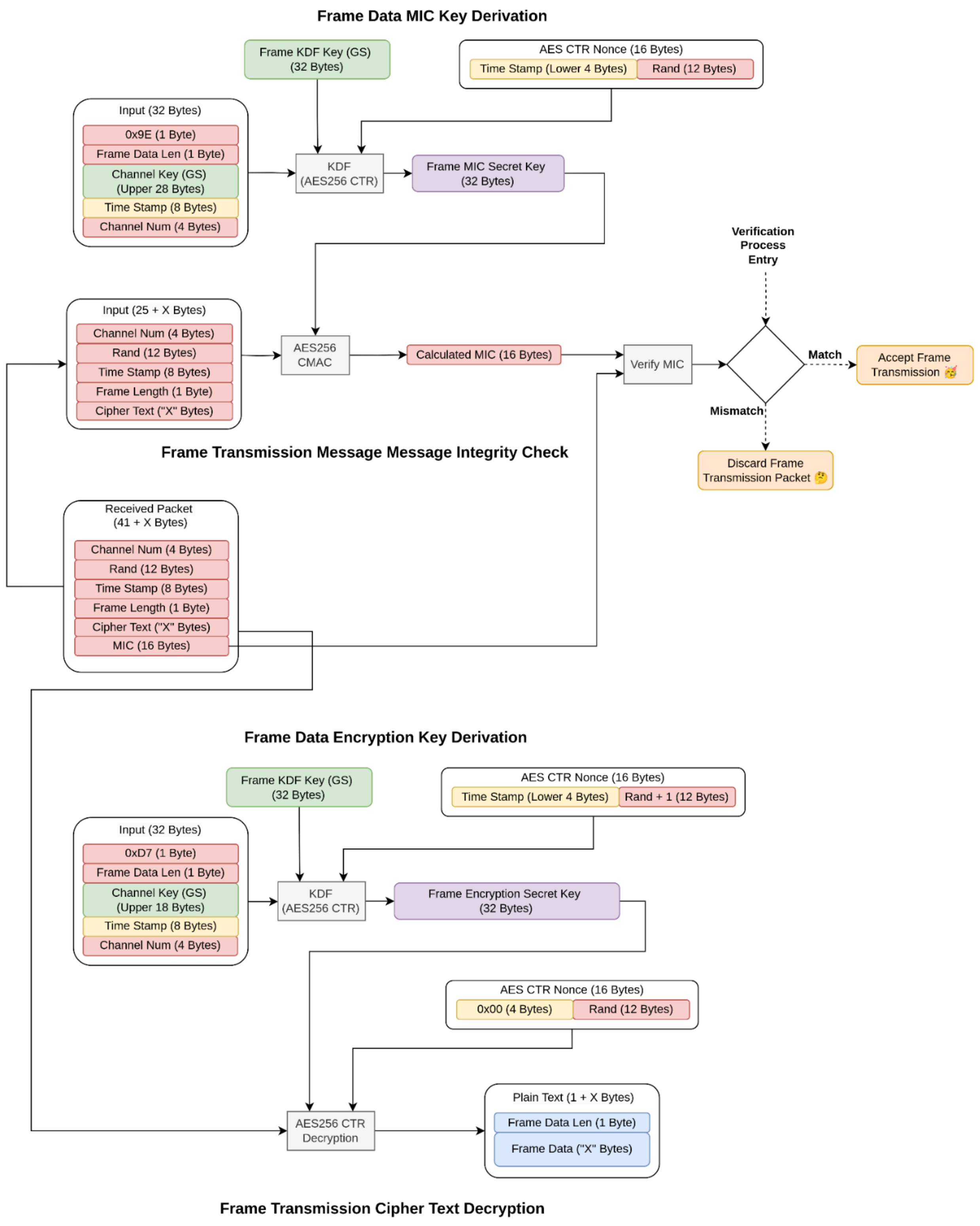


Frame Transmission Packet Structure

6.5.2. Frame Transmission Decode

This section covers the frame transmission process which is fed packets from the TV to decode and return decrypted frames to the TV to display. Frame decoding follows a similar approach to encoding. First, the frame secret key is derived and used to calculate the MIC which is compared with the MIC in the received frame packet, ensuring the packet has not been modified.

Subsequently, the timestamp is checked to ensure the time stamp has monotonically increased, preventing replay attacks. If both the signature is valid and the time stamp has increased the encrypted frame data is decrypted and sent back to the TV as it has been verified. This is illustrated in the following figure.

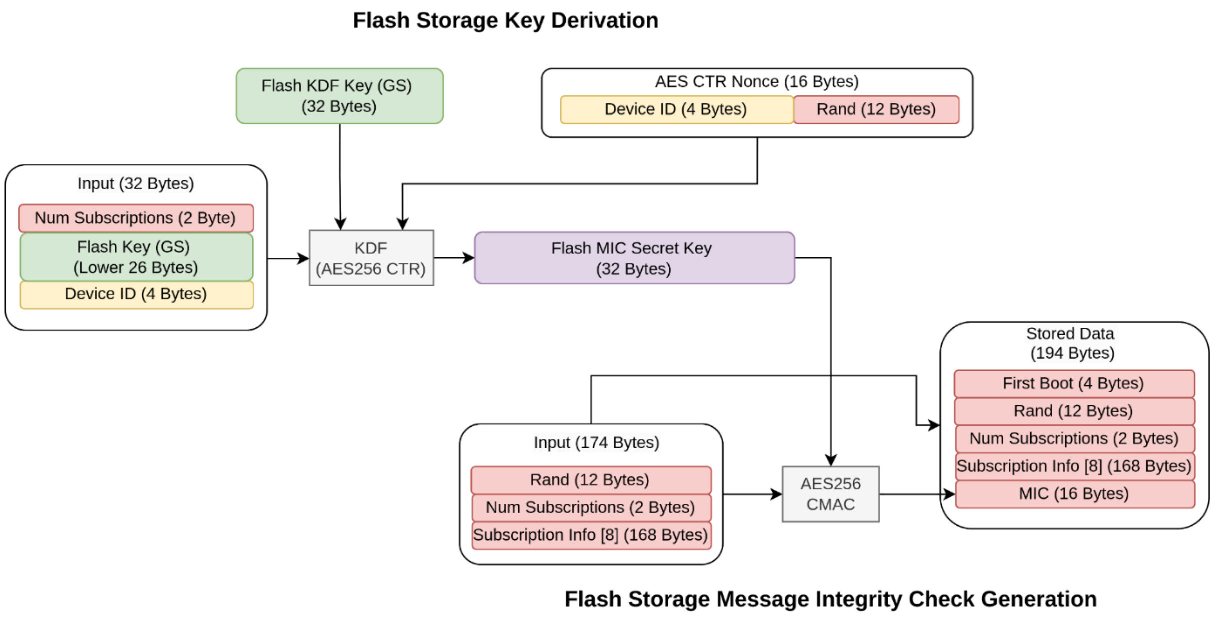


Frame Transmission Decoding

The use of global secrets, accessible only to friendly decoders, in the key derivation function safeguards against all attack scenarios. These include “Expired Subscription”, “Pirated Subscription”, “No Subscription”, “Recording Playback Attack“ and “Pesky Neighbour”. Additionally, the validation that the timestamp is always monotonically increasing prevents replay attacks. Signature verification ensures that only packets from trusted devices are accepted, discarding those from attackers.

6.6. Flash Active Channels Storage

Subscription updates tell the decoder when a particular channel is active. This information is stored in flash on the decoder and protected with a message integrity check to prevent attacks from directly writing to flash to add new susbcriptions. The flash MIC generation is shown in the below figure.


Flash Storage Key Derivation
Flash KDF Key (GS) (32 Bytes)
AES CTR Nonce (16 Bytes)
Device ID (4 Bytes)
Rand (12 Bytes)
Input (32 Bytes)
Num Subscriptions (2 Byte)
Flash Key (GS) (Lower 26 Bytes)
Device ID (4 Bytes)
KDF (AES256 CTR)
Flash MIC Secret Key (32 Bytes)
Stored Data (194 Bytes)
First Boot (4 Bytes)
Rand (12 Bytes)
Num Subscriptions (2 Bytes)
Subscription Info [8] (168 Bytes)
MIC (16 Bytes)
Input (174 Bytes)
Rand (12 Bytes)
Num Subscriptions (2 Bytes)
Subscription Info [8] (168 Bytes)
AES256 CMAC
Flash Storage Message Integrity Check Generation

Flash MIC Generation

The same process is used for the flash MIC check, except the calculated MIC is compared against the MIC in the stored data. If the message integrity check is incorrect, the subscription information is discarded, and the flash data is regenerated.

7. Design Implementation

The decoder’s architecture is based upon a segmented real-time operating system (RTOS) capable of managing multiple processes simultaneously. FreeRTOS - a specific implementation of RTOS designed for microcontrollers and small embedded systems - is used to implement robust task scheduling, memory management, and segmented processes.

Each component runs as a FreeRTOS task with defined priority levels and memory partitions. Task isolation is achieved through FreeRTOS’s memory protection features

There are eleven components, each with its protected memory region. This architecture employs memory isolation mechanisms, whereby each process operates within its defined boundaries, mitigating buffer overflow vulnerabilities and preventing inter-process memory access.

The architecture integrates processes that manage subscription, channel control, cryptographic operations, initialisation processes, serial interfaces, memory allocation, and error handling. These operate alongside the monitoring systems, including sentry, failsafe, and clock supervision processes.

The following subsections describe how each process functions in further detail.

7.1. Initialisation Manager

The initialisation manager is the “startup daemon hook”. It starts on boot, before the scheduler, and initialises the system.

7.2. Serial Interface Manager

The serial interface manager decodes serial packets at a baud rate of 112500 and in 8N1 configuration. It ensures input is valid, of the expected size, and not malformed in any way. As with all tasks, it is monitored by the Sentry function and a failsafe is triggered in the event of a stack overflow or memory allocation failure due to bad input.

7.3. Crypto Manager

The cryptographic manager performs all cryptographic operations as requested using the hardware AES peripheral on the microcontroller. It is the only process with access to the global secrets. The crypto manager performs all key derivation functionality to minimise the number of processes with access to key material.

7.4. Subscription Manager

The subscription manager takes in subscription update packets from serial and checks the signature through the cryptographic manager. The subscription manager only has access to the check signatures functionality of the cryptographic manager which will return a simple true or false. This reduces the potential attack surface to the global secrets.

If the subscription is valid, it stores the subscription information in a section of flash memory allocated to it by the Memory Manager. The sentry process is notified if subscription update messages with a bad format or invalid signature are received.

7.5. Channel Manager

The channel manager keeps a list of all channels for which the decoder has a valid subscription and responds to the list channels command. Additionally, on startup the subscription manager retrieves all valid subscriptions from the flash memory, ensuring subscription persistence through power cycles. It does not have access to any cryptographic information to reduce the attack surface of the global secrets.

7.6. Frame Decoder

The frame decoder receives encrypted frame data from the serial interface manager. It communicates with the crypto manager to verify signatures and decrypt encrypted frame data. It gets subscription information from the channel manager to know when a subscription for a particular channel is valid.

If frame messages with a bad format or signature are received it communicates this to the sentry process which can act accordingly.

7.7. Sentry Functionality

Two hooks are implemented within FreeRTOS to ensure that both stack overflows and memory allocation failures are detected. These will both trigger a failsafe.

7.8. Error Manager

The error manager is implemented utilising FreeRTOS functionality, and is configured specifically to detect and catch Memory Allocation failures, as well as Stack Overflows utilising **Method 3**. If a soft system reset does not succeed as a result of an error, the failsafe process is invoked.

See more:
https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/09-Memory-management/02-Stack-usage-and-stack-overflow-checking

7.9. Failsafe Process

The failsafe process locks the system in the event that a soft system reset does not succeed. It disables interrupts, which effectively also disables FreeRTOS functionality, prints a message to inform the user of what is happening, and enters an infinite loop.