eCTF 2025 UCI Design Documentation: Secured Satellite TV

The BugEaters

April 4, 2025

1 Introduction

The following is team UCI's design for the 2025 MITRE's eCTF competition, where the challenge is to implement a secure decoder and encoder for a satellite TV system.

This document will introduce the functional requirements of the encoder and decoder, the TAICHI security algorithm and its security analysis, subscription protection, key management, and hardware level protection on the MAX78000FTHR embedded board.

2 Functional Requirements

2.1 Encoder

The Encoder is entirely implemented in Python. It satisfies the required functionality with an additional layer of encryption.

2.1.1 Encode Frame

Frame encoding is described in Section 3.3.2. One frame will be maximum 64 bytes.

2.1.2 Generate Secrets

Secrets are generated in the JSON format and stored as global.secrets. It will contain an array of channel IDs, a flash key, and a list of nested JSONs.

Each sub-JSON will contain a:

- Corresponding channel ID
- Mask key
- Message key
- Data key
- Subscription key
- Checksum

Mask key, Message key, and Data key will be used in the TAICHI algorithm for data encryption and decryption; while Subscription key and Checksum will be used to ensure the safety of subscription update messages.

Algorithm 1 Generate Subscription Algorithm

function GEN_SUBSCRIPTION(secrets, device_id, start, end, channel)

    Input: secrets (16 bytes), device_id (4 bytes), start (8 bytes), end (8 bytes), channel (4 bytes)

    Output: Encrypted subscription data

    $secrets\_dict \leftarrow json.loads(secrets)$

    $sub\_key \leftarrow secrets\_subscription\_key$
    $check\_sum \leftarrow secrets\_checksum$

    $sub\_info \leftarrow struct.pack(\text{" } < IQQ\text{"}, device\_id, start, end)$

    $check\_sum\_channel \leftarrow ast.literal\_eval(check\_sum)$

    $interwoven\_data \leftarrow interweave(sub\_info, check\_sum\_channel)$

    $encrypted\_data \leftarrow encrypt\_subscription(interwoven\_data, sub\_key)$

    return $channel + encrypted\_data$

end function

Algorithm 2 Encrypt Subscription Algorithm

function ENCRYPT_SUBSCRIPTION(interwoven_bytestring, checksum)
    Input: interwoven_bytestring (48 bytes)
    Output: Encrypted subscription data

    $data \leftarrow pad(interwoven\_bytestring, 16)$

    $channel\_key \leftarrow get\_channel\_subscription\_key(channel)$
    $iv \leftarrow secret\_gen.token\_bytes(16)$

    $cipher \leftarrow encoder.sym\_encrypt(channel\_key, iv, data)$

    return *cipher*

end function

Algorithm 3 Get Subscription Key

function GET_CHANNEL_SUBSCRIPTION_KEY(channel, secrets)
    Input: channel (int)
    Output: Subscription key bytes (4 bytes)

    if channel not in secrets['channels'] **then**
        raise ValueError("Channel not found in secrets")
    end if

    $subscription\_key\_bytes \leftarrow str.convertToBytes(secrets[channel\_key])$

    return $subscription\_key\_bytes$

end function

2.1.3 Generate Subscription

To generate a subscription update, we concatenate the device ID, starting timestamp and ending timestamp into bytestring with length = 20. The channel ID will be attached to the front of the bytestring.

The corresponding length 24 byte checksum will be interwoven with the subscription info bytestring, then encrypted with the Subscription key.

Once the decoder receives the update packet, it will use the first 4 bytes to identify the channel ID and obtain the corresponding Subscription key from the secrets file. The decoder will then decrypt the update message, de-interweave the byte string, and update the subscribed_channels list correspondingly.

A subscription update packet will be 68 bytes long.

2.2 Decoder

The decoder is implemented in C based on the reference design with revised security measures.

2.2.1 Initialization

The initialization process occurs during the first boot. DMA interrupts are first disabled across DMA0-4. The custom implemented advanced flash interface is also activated.

The global timestamp counter is set to 0 and 8 programmable channels are all initialized to default values. The default values for each channel are described below. Channel 0 is reserved for emergency broadcasts.

Default channel values:

```
#define DEFAULT_CHANNEL_TIMESTAMP 0xFFFFFFFFFFFFFFFF
#define DEFAULT_CHANNEL_ID -1

channel.start_timestamp = DEFAULT_CHANNEL_TIMESTAMP;
channel.end_timestamp = DEFAULT_CHANNEL_TIMESTAMP;
channel.active = false;
channel.id = DEFAULT_CHANNEL_ID;
channel.fresh = false;]
```

Channel status structures with defaults are all written into the protected flash memory region. From this point onward, channel statuses can only be accessed with flash_privileged_read() and flash_privileged_write().

Finally, the initialization process is concluded with a call to init_secret() which loads all keys and secrets into the protected flash memory region. mpu_setup() uses the onboard Memory Protection Unit to protect the flash memory and set read/write privileges for different areas.
The whole initialization flow has been visually discribed in Figure 1.

2.2.2 List Channels

List Channels has not been changed. The subscribed channels array is looped over and active channels are returned.

2.2.3 Update Subscription

The update subscription is implemented as a function in C. It takes in the length of an incoming encoded packet and a pointer to an array of encoded channel updates, which is further explained in the general outline.

- Decrypt subscription using subscription key

- Extract pre-encoded message
- Extract subscription information and checksum from inter-weaved message
- Update subscription

The information used to extract information from interwoven messages is 4

Algorithm 4 Extractor algorithm

```
1: function EXTRACT(*Intrwvn_Msg, *Subs_Info, *Chksm)
2:     Input: Interwoven message (48 bytes), Subscription Information, Checksum
3:     Output: 0 if successful, -1 if error
4:                                                          ▷ Validate input pointers
5:     if Intrwvn_Msg = NULL or Subs_Info = NULL or Chksm = NULL then
6:         return -1                                                          ▷ Error
7:     end if
8:                                                          ▷ Extract interwoven data
9:     uint8[20] Temp_Subs_Arr
10:    for i = 0 to 40 do
11:        if i  mod 2 = 0 then
12:            Temp_Subs_Arr[i/2] ← Intrwvn_Msg[i]
13:        else
14:            Chksm[i/2] ← Intrwvn_Msg[i]
15:        end if
16:    end for
17:                                                          ▷ Null-terminate arrays
18:    Temp_Subs_Arr[24] ← \0
19:    Chksm[24] ← \0
20:                                                  ▷ Parse Subscription Information
21:    Subs_Info.decoder ← Temp_Subs_Arr[0 : 4]                          ▷ 4 bytes
22:    Subs_Info.start_timestamp ← Temp_Subs_Arr[4 : 12]                 ▷ 8 bytes
23:    Subs_Info.end_timestamp ← Temp_Subs_Arr[12 : 20]                  ▷ 8 bytes
24:
25:    return 0                                                           ▷ Success
26: end function
```

2.2.4 Decode Frame

The decode frame is implemented as part of the the decoder. The decode frame takes uint_int16 of the length of the packet and a pointer of the packet itself. Here is the control flow of the decode frame:

Algorithm 5 Decode Frame

function DECODE_FRAME(pkt_len, new_frame)
    Input: Packet length (pkt_len), Encrypted frame packet (new_frame)
    Output: Decrypted frame data

    Initialize $output\_buf$ to zero
    Extract $channel$ and $timestamp$ from new_frame
    ▷ Decrypt $c1$ and validate timestamp
    Load $mask\_key$, $message\_key$, and $data\_key$
    Compute $c1\_key$ as XOR($timestamp, mask\_key$)
    Hash $c1\_key$
    XOR $c1\_key$ with $message\_key$ to obtain decryption key for $c1$
    Decrypt $c1$ using $c1\_key$ to get ts_prime
    Extract $nonce$ from ts_prime
    ▷ Decrypt $c2$ to obtain frame data
    Compute $c2\_key$ as XOR($nonce, data\_key$)
    Compute $c2\_length$ as pkt_len- header size
    Decrypt $c2$ using $c2\_key$ to obtain $frame\_data$
    return $frame\_data$
end function

2.2.5 Validate Timestamp

validate_timestamp() takes the following arguments:

- **Channel ID:** 32 bit integer indicating the channel number
- **Plaintext Timestamp:** The timestamp in plaintext. The extracted timestamp will be checked against this value to ensure the legitimacy of the claimed timestamp
- **Extracted Timestamp:** The extracted timestamp to be checked against plaintext timestamp to ensure the legitimacy of the claimed timestamp

This function will invoke the following three functions, and it will validate a timestamp if and only if all these functions are satisfied: *check_two_timestamp*(), *check_increasing*(), and *within_frame*(), respectively:

1. ***check_two_timestamp*()**: Calls *memcmp*() and passes the addresses of timestamp in plaintext and the extracted timestamp, along with the length of timestamp. It returns 1 if the extracted timestamp matches the plaintext timestamp
2. ***check_increasing*()**: This function is called once the two timestamps match. It takes the Channel ID and the Extracted Timestamp as arguments. It first reads the **channel status** structure from flash memory, and it then ensures that the extracted timestamp is greater than the current (last frame's) timestamp. If satisfied, the function returns 1
3. ***within_frame*()**: After it makes sure that the timestamp is strictly increasing, this function is called with arguments Channel ID and Extracted Timestamp. It ensures that the extracted timestamp is within the range (starting timestamp, ending timestamp). Function returns 1 if the extracted timestamp falls within said interval

The C structure **channel status** is defined as follows:

- **Activity Status:** Boolean value indicating whether there's an active subscription associated with the channel or not
- **Channel ID:** 32-bit integer representing the channel number
- **Start Timestamp:** 64-bit integer indicating the beginning of the timestamp
- **End Timestamp:** 64-bit integer indicating the end of the timestamp

- **Fresh:** Boolean value indicating whether the channel has received frames
- **Structure definition:**

```
typedef struct {
    bool active;
    channel_id_t id;
    timestamp_t start_timestamp;
    timestamp_t end_timestamp;
    bool fresh;
} channel_status_t;
```

The additional global variable **Current Timestamp** is a 64-bit integer indicating the latest valid timestamp received.

3 TAICHI: Symmetric-Key-based Proof of Knowledge

3.1 Algorithm Introduction

For this year's challenge, we designed **TAICHI**: *Totally Asynchronous Interval Cipher for Honesty Identification* as the foundation for encrypting, decrypting, and verifying the integrity of data-frame transmitted. Given the fact that the Encoder has static memory throughout the process, this algorithm overcomes the issue that nonce-based encryption is nearly impossible. The main goal of TAICHI is to add encryption randomness while maintaining ciphertext integrity (CI). The proofs in this section will be hand-wavy for the sake of development time.

3.2 Threat Model and Assumptions

3.2.1 Threat Model

We assume that there are software vulnerabilities that could allow a malicious party to perform the following attacks against the embedded device:

- The attackers have polynomial bounded computational power. To weaken our assumption for practical scenarios, we assume that the attacker can break AES 128 in 30 days
- The attackers can capture, modify, and concatenate any cipher text from the communication channel.
- The attackers can jam and eavesdrop on any message.

3.2.2 Security Assumption

- As given by MITRE, we assume that the global secret is secure and not available to any attacker for the sake of this security proof. We will discuss how we protect the flash on the hardware level in Section 6.
- We model the hash function H as a random oracle(RO), meaning that H behaves as a truly random function that provides a unique, random output for every distinct input. In the real-world implementation, we will replace such H with $SHA-256$.
- We assume the encryption scheme we use satisfies semantic security under the chosen plaintext attack (IND-CPA). This means that no polynomial-time adversary can distinguish between the encryptions of two chosen plaintexts with non-negligible advantage. In this year's competition, we will replace such encryption function Enc with $AES128$ with CBC mode.

3.3 Algorithm Description

The algorithm consists of three main functions, $Key\_Gen(\lambda)$, $Enc(key\_set)$, $Dec(key\_set)$, as is standard in any private key cryptosystem. We show only the algorithm for a particular channel. For multiple channels, the algorithm simply duplicates itself and runs independently of each other.

- $Key\_Gen(\lambda)$
- $Enc(key\_set, data\_frames)$
- $Dec(key\_set, ciphertext)$

3.3.1 Key Generation Algorithm

Key Genλ:

The Key generation algorithm takes in security parameter λ, outputs three AES keys for each channel. These keys are called a Key Set, defined as:

$$KeySet = \{\text{Mask Key}, \text{Message Key}, \text{Data Key}\}$$

Each of these keys is a 16-byte character string, uniformly chosen at random to ensure security and uniqueness.

Algorithm 6 Key Generation Function

```
function KEY_GEN(λ)
    char[16] Mask_key ← Random(16 bytes)
    char[16] Msg_Key ← Random(16 bytes)
    char[16] Data_Key ← Random(16 bytes)
    Key_Set ← {Mask_key, Msg_Key, Data_Key}
    return Key_Set
end function
```

3.3.2 Encoder Algorithm

The satellite dish encodes and sends the following components to the decoder, described in Algorithm 7:

- Computation of T_s':
$$T_s' = \text{Nonce}_l \,||\, T_s \,||\, \text{Nonce}_r$$
where Nonce_l and Nonce_r are a p-byte random string Nonce_p split into half, T_s is just the given timestamp.
- Computation of c_1:
$$c_1 = \text{Enc}\Big(\text{Hash}(\text{Mask} \oplus T_s) \oplus \text{MsgKey}, T_s'\Big)$$
- Computation of c_2:
$$c_2 = \text{Enc}(\text{Nonce}_p \;\oplus\; DataKey, \text{Data})$$
- Output: The encoder sends the following to the decoder:
$$\{T_s, c_1, c_2\}$$

Algorithm 7 Encoder Algorithm for Satellite Dish Communication

function ENCODER(T_s, Data, Mask, MsgKey, DataKey, Nonce_p)
    Input: T_s (Timestamp), Data, Mask, MsgKey, Nonce_p (17 bytes)
    Output: T'_s, c_1, c_2

    $T'_s \leftarrow \text{Concat}(Nonce\_p, T_s)$ ▷ Append nonce to timestamp
    $H \leftarrow \text{Hash}(\text{Mask} \oplus T_s)$ ▷ Compute hash of Mask XOR Timestamp
    $c_1 \leftarrow \text{Enc}(H \oplus \text{MsgKey}, T'_s)$ ▷ Encrypt with MsgKey and T'_s
    $c_2 \leftarrow \text{Enc}(Nonce\_p \oplus \text{Data Key}, \text{Data})$ ▷ Encrypt nonce and data

    return $\{T'_s, c_1, c_2\}$ ▷ Return encoded components
end function

3.3.3 Decoder Algorithm

The decryption algorithm we used to retrieve the timestamp works as follows. Pseudo code was shown in Algorithm 8

$$\text{Dec}\big(\text{Hash}(\text{Mask} \oplus T_s) \oplus \text{Msg Key}, C1\big) = TS' = \text{Nonce} \,\|\, TS$$

$$Data = \text{Dec}\big(Nonce \oplus DataKey, C1\big)$$

Algorithm 8 Decoder Algorithm for Satellite Dish Communication

function DECODER($T'_s, c_1, c_2, \text{Mask Key}, \text{Msg Key}, \text{Data Key}$)
    Input: T'_s (Nonce $\|$ T_s), c_1, c_2, Mask Key, Msg Key, Data Key
    Output: T_s (Timestamp), $Data$ (Decoded Data)

    $\text{Nonce}_p \leftarrow \text{ExtractNonce}(T'_s)$ ▷ Extract the first 17 bytes Nonce
    $T_s \leftarrow \text{ExtractTimestamp}(T'_s)$ ▷ Extract the remaining bytes (Timestamp)

    $H \leftarrow \text{Hash}(\text{Mask Key} \oplus T_s)$ ▷ Compute hash of Mask Key XOR Timestamp
    $\text{Recovered\_}T'_s \leftarrow \text{Dec}(H \oplus \text{Msg Key}, c_1)$ ▷ Decrypt c_1 using the combination of Mask and Msg Key

    if $\text{Recovered\_}T'_s = T'_s \& Recovered\_T'_s > Previous\_T'_s$ **then** ▷ Verify the integrity of T'_s
        Update Subscription
    end if

    $Data \leftarrow \text{Dec}(\text{Nonce}_p \oplus DataKey, c_2)$ ▷ Decrypt c_2 using Nonce and Data Key
        ▷ The XOR operation ensures data recovery by reversing the encryption stream applied during encoding.

    return T_s, $Data$ ▷ Return the decoded timestamp and data
end function

By such an algorithm, we achieve the following.

1. **Restoration of T'_s:**
    - During decryption of c_1, the hash H (Hash of Mask Key XOR Timestamp) is recombined with the Msg Key using XOR.
    - The XOR operation reverses the earlier combination of H and Msg Key in the encoder, restoring the original T'_s.
2. **Validate Timestamp:**
    - The extracted timestamp should be the same as the timestamp before encryption.

- We extract the subscription information, validate if the timestamp is greater than it to validate the monotone increasing order.
- If the extracted timestamp is valid, we update the current timestamp for the subscription channel.

3. **Data Recovery:**

   - When decrypting c_2, the nonce is used as part of the encryption stream.
   - The XOR operation reverses the encryption applied during encoding, effectively recovering the original *Data*.
   - Mathematically:

     $$\text{Ciphertext} = \text{Data} \oplus \text{Encryption\_Stream}(\text{Nonce})$$

     During decryption:

     $$\text{Data} = \text{Ciphertext} \oplus \text{Encryption\_Stream}(\text{Nonce})$$

REMARK: In this algorithm description, we do not check explicitly how the timestamp is strictly, monotonically increasing. This is the job for the decoder firmware (by software protection), which will be discussed in later section(s).

3.4 Security Proofs

In this section, we proceed to prove (informally) the following:

- Semantic Security for the cipher text
- $IND - CPA$ Security
- $IND - CCA^1$ Security

3.4.1 Semantic Security

We prove the most fundamental passive attacks (eavesdropping attack) that is trying to decipher the encrypted text without proper key.

Lemma 1. *Any proper subset of the key set is **NOT** sufficient for decryption*

Proof. Given we have three keys for a particular channel

- $Mask_{key}$
- $Message_{Key}$
- $Data_{key}$

Recall the encryption scheme cipher-text

$$C1 = Enc(Hash(Mask_{key} \ \oplus \ TimeStamp) \oplus \ Message_{Key}, \ TS')$$

$$C2 = Enc(Nonce \oplus Data_{Key}, data)$$

A total of 3 subset (for choose 2) and 3 subset (for choose 1), prove by exhaustion we can obtain this lemma. □

Theorem 1. *Suppose the encryption box $Enc(Key, Message)$ is semantically secure, TAICHI is semantically secure*

Proof. Since each of the encryption process in TAICHI uses a fresh key. Our algorithm is semantically secure as if no correlation between $c1, c2$ □

3.4.2 CCA Security

Suppose the adversary plays a security game with the encoder. The adversary constantly has access to the decryption oracle, as the standard CCA1 game was constructed. The adversary can send polynomial many data tuple pairs $t_0 = (channel, data_0, TS_0)$ and $t_1 = (channel, data_1, TS_1)$ to the encryption oracle. The oracle chooses secret bit $b_i \in \{0, 1\}$ uniformly at random at the beginning of the game. Suppose the decryption oracle output $\perp$ when $c1, c2$ in a cipher text was manipulated, as checked by nonce. Change of any $c1, c2$ wont let the decryption oracle decrypt.

Suppose that at the challenge round the attacker modifies the data and resends to the encoder. The freshness of the nonce would yield the whole cipher text indistinguishable. Thus we achieve the CCA security.

Alternatively, we can show that each $c1$, $c2$ has one-to-one correspondence. That is, for any particular $c1$, WLOG, the probability of having two $c2_1$, $c2_2$ is negligible. This can be obtained by checking the nonce. Suppose semantically secured $c1$, forging two cipher text under the same encryption key is as hard as guessing the encryption key, given the PRF nature.

3.4.3 CPA Security

Since CCA security implies CPA security, we can follow this as a corollary from the section above.

4 Subscription Protection

The subscription protection is rather easy compared to the TAICHI algorithm. To protect the subscription, we only consider the possible forgery attack. but not the replay attack. We assume that security for subscription is mostly protected by the hardware. Also we assume no flag will be contained in the subscription package. Hence, our algorithm only ensures the following properties.

- The subscription package is generated by a genuine subscription generating script. This is ensured by the encryption of a unique key.
- The subscription is not being tempered by the adversary. This is ensured by a secret checksum.
- To update a subscription, the decoder follows a strict verification process that ensures the validity and security of the subscription package received.

In Algorithm 9, the package generated by the subscription consisted of two keys that are uniformly chosen at random. In Algorithm 10, the subscription is generated through string manipulation and encryption. In Algorithm 11, the subscription is updated after the extracted information is validated.

Algorithm 9 Key Generation Function for Subscription

```
function SUBSCRIPTION_KEY_GEN(λ)
    char[20] Check_Sum ← Random(20 bytes)
    char[16] Subscription_Key ← Random(16 bytes)
    Subscription_Key_Set ← { Check_Sum, Subscription_Key }
    return Subscription_Key_Set
end function
```

Algorithm 10 Create Subscription

```
function CREATE_SUBSCRIPTION(Subscription_Key_Set, Subscription_Info)
    char[48] pre_encoded_msg ← Interweaving(Check_Sum, Subscription_Info)
    char[48] subscription_package ← Encrypt(Subscription_Key, pre_encoded_msg)
    return subscription_package
end function
```

Algorithm 11 Update Subscription

```
function UPDATE_SUBSCRIPTION(pkt_len, *packet)
    char[4] Channel_ID ← Extract first 4 bytes of Packet
    char[48] interwoven_encrypted ← Extract next 40 bytes of Packet + 8 bytes of padding
    char[16] IV ← Extract last 16 bytes of Packet
    Retrieve stored secrets from flash using Channel_ID
    char[48] interwoven_decrypted ← Decrypt(interwoven_encrypted, Subscription_Key, IV)
    { char[20] Subscription_Info, char[20] Checksum } ← De-interleave(interwoven_decrypted)
    if Validate(Checksum, Stored_Checksum) is False then
        return Failure
    end if
    if Found_Duplicate_Channel_ID() is True then
        return Failure
    end if
    if Channel_ID == Emergency_Channel then
        return Failure
    end if
    Store updated Subscription_Info in flash memory
    return Success
end function
```

5 Secret Key Management

In our design, a total of 4 types of keys are generated:

- Initial Vector for AES CBC mode
- Frame Encryption Keys
- Subscription Keys
- Hardware Flash Protection Key

5.1 Key Storage

All keys are generated randomly using Python scripts provided by the MITRE example. Key generation follows the implementation details of `gen_secrets.py`, ensuring randomness and security for each key type. To accommodate use for both C compilation and Python scripts, the storage formats are slightly different.

5.1.1 Host Machine File Format

The keys generated on the host are stored in a JSON file format for easy interoperability between Python scripts and external tools. These keys are stored as string representations of bytes and must be converted back to byte form for cryptographic use.

The secret_json file includes:

- **Channel IDs:** Represented as integer values.
- **Keys:** Represented as binary data encoded as strings.
  - **Flash Key:** A system-wide security key stored as a string-encoded binary value(16 bytes).
  - **Mask key, message key, data key, and subscription key:** Randomly generated for each channel (16 bytes).
  - **Checksum:** Random value stored along with the channel cryptographic keys. This is only 20 bytes since the first four bytes of channel id will not be encrypted. (20 bytes).
- **Example structure:**

```
{
    "channels": [0, 1, 2, 3],
    "flash_key": "f7d893159c1ae19da583f4941810c21d",
    "channel_0": {
        "channel_ID": "0",
        "mask_key": "865d04c724aa62db33e5a8bb037b8dee",
        "msg_key": "d3914b2ce81f63a79d8eaa4fb523a077",
        "data_key": "d79612833cc5dccd36aec7d276f2ddde",
        "subscription_key": "6b75d9c0b992da7033cb1d7f49f34c8a",
        "check_sum": "dfdc7f9c4cbcd816b501fe7000763c872fb5cb52"
    },
    ......
}
```

- **String-Formatted Keys and Conversion:**
  - The generated secrets are stored as **hexadecimal encoded** strings.
  - This ensures compatibility between JSON-based storage and Python-based cryptographic operations.
  - These values can be converted back to byte form in Python, for instance:

    ```
    "{hex_to_c_array(channel_data.get('mask_key', ''))}"
    ```

- **Command line format:**
  - To place the *secrets.json* file in the *secrets* directory when generating the JSON file, prepend '../../secrets' to the name of your JSON file. Example usage:

    ```
    python gen_secrets.py../../secrets/secrets.json 1 2 3
    ```

- **C Header File Generation:**
  - The *generate_secret_h.py* script processes the generated *secrets.json* file and creates a C header file *secret.h*, which contains an integer array of all the **Channel ID** and the integer **Array Size** indicating the size of the Channel ID array, as well as the C source file *secret.c*, which contains the secret structures for managing encryption keys and checksum validation. Each key is converted into an array in this format:

    ```
    char mask_key[16] = {0x9f, 0xc4, 0x3d, ... , 0xff, 0xa6}
    ```

5.1.2 MAX-78000FTHR Storage

The generate_secret_h.py script is responsible for generating the C header file, secret.h, based on the input JSON file. It takes the path to the JSON file as input. If the given file does not exist, it raises *FileNotFoundError* exception.

The script utilizes a simple for loop, iterating over the available channels in the *channels* key of the JSON file. It extracts the values of each key and stores them in the C structure defined as follows:

- **Channel ID:** An integer value representing the channel number
- **Mask key:** Randomly generated 16 byte number
- **Message key:** Randomly generated 16 byte number
- **Data key:** Randomly generated 16 byte number
- **Subscription key:** Randomly generated 16 byte number
- **Checksum:** A 24 byte random number

- **Structure definition:**

```
typedef struct {
    uint32_t channel_id;
    char mask_key[16];
    char msg_key[16];
    char data_key[16];
    char subscription_key[16];
    char check_sum[20];
} secret_t;
```

The generated C header file contains the function *init_secret*() in which these structures are instantiated and populated with the data from JSON file, then written to flash memory by invoking *write_secret*() function. The *memset*() function is called after each write to flash in order to erase the SRAM. The following is an example of a struct, populated with data extracted from a JSON file:

```
secret_t channel_1 = {
    1,
    { 0x8c, 0xab, 0x4a, ... , 0x8d, 0x35 },
    { 0x40, 0xe0, 0x66, ... , 0x94, 0x2e },
    { 0x0f, 0x62, 0xab, ... , 0xa4, 0x10 },
    { 0x11, 0xe8, 0x8a, ... , 0xf8, 0x92 },
    { 0x29, 0x23, 0x76, ... , 0x9c, 0xc8 }
};
```

5.1.3 MAX-78000FTHR Retrieval

Retrieving the key for any channel for frame or subscription decoding was done via the function read_key(*channel_id*, *secret**) and write_key(*secret**) Additional hardware protection was done on the flash to protect the key's integrity.

6 Hardware Protection

6.1 Memory Protection Unit

The ARM Cortex-M4 processor is in one of two states at any given time: privileged mode or user mode. These states are typically used by embedded operating systems to create barriers of control between user applications and the kernel; we instead use them to add an additional layer of protection over our most critical data.

We employ the **Memory Protection Unit or MPU** to add hardware level protection over our memory space. The MPU allows us to create distinct regions of memory and define access permissions for those regions. These permissions include disabling instruction execution, restricting read/write access depending on the current privilege level, and configuring cache and buffer properties to secure peripheral data transfers.

We have defined three regions of memory, each with a distinct set of permissions.

Background Region

- Region Space: All memory
- Access Permission: Read/write for all users
- Execute Permission: Disabled

This region is the default region that will be used if no higher priority region is set.

Code Space

- Region Space: All memory
- Access Permission: Read/write for all users
- Execute Permission: Enabled

Permanent Secrets Region

- Region Space: 0x1007.C000 - 0x1007.DFFF
- Access Permission: Read-only for privileged users only
- Execute Permission: Enabled

After we write our secrets to flash on our first boot, we disable writes to this region completely and only allow reads from a privileged state. This region is never changed after our first boot.

Secrets Overlay Region

- Region Space: 0x1007.8000 - 0x1007.FFFF
- Access Permission: No access for any user
- Execute Permission: Disabled

This is a region that overlaps with the Permanent Secrets Region, but has lower privilege than it. If an adversary is reading flash memory byte-by-byte, this no access region acts as a guard, causing an error before the adversary reads the secrets region.

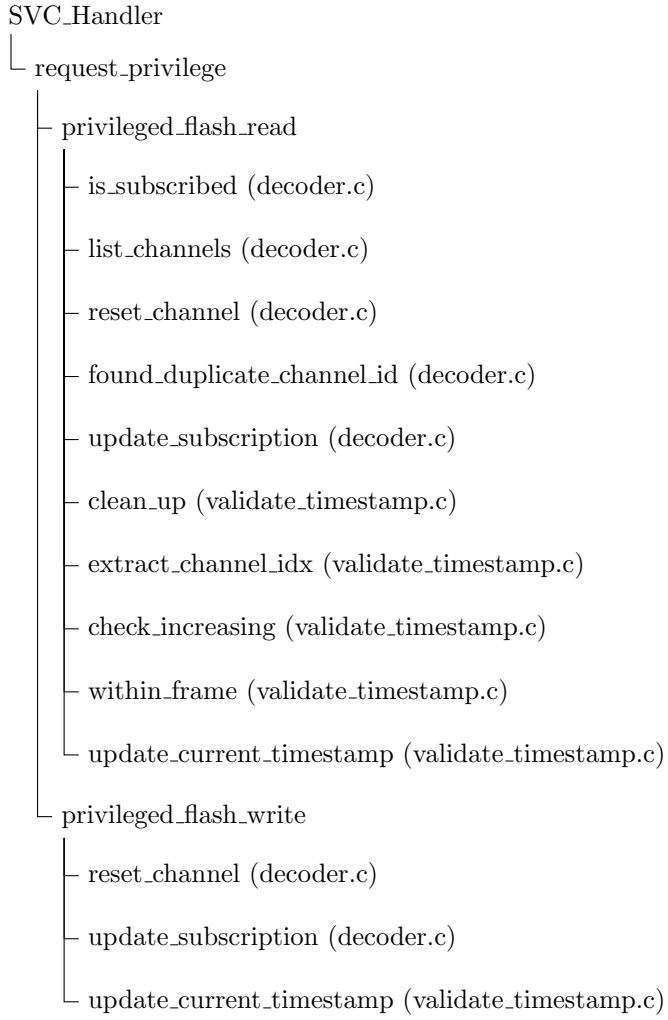
We implemented address validation.

6.2 Conditional Privilege Escalation

While the MPU is great for protecting the system from accesses from unprivileged users, that protection is meaningless if the process of escalating privilege is trivial. We employ **Conditional Privilege Escalation** to protect our critical data in the case of malicious code injection.

The way this is achieved is simple: check the return address stored in the link register `lr` in order to see from where the function is being called and compare it against our own pre-determined addresses. If the comparison fails, privilege is not escalated. If there are more functions needed in order to escalate privilege, we validate the caller at every function call. This means that these "privileged" functions can only be called from addresses that we have approved. If code injection is achieved, our most sensitive data would still be protected.

A tree diagram of these caller validations is given below.



6.3 Stack Protection

One of the most common vulnerabilities in embedded systems is the **Buffer Overflow** vulnerability. While we make sure to create safe software, there is always a chance for a zero day to exist that can exploit the embedded system. In order to protect against a buffer overflow vulnerability, we employ the use of the stack canary.

The **stack canary** protects the stack by inserting a random canary value at the end of the stack, so if an attacker uses a buffer overflow attack, they will have to write over the canary to change the return address stored on the stack. Since they write over the canary, the program will detect that it was tampered with and immediately terminate before returning to prevent execution of malicious code.

6.4 Encrypted Flash

We use the AES accelerator peripheral on the FTHR to encrypt data with a 128-bit key before writing it to flash; then, if the flash is dumped (e.g. during a buffer overflow attack), attackers will need to decrypt the data in order to use it for further attacks. The AES peripheral stores the key it uses for AES operations in a separate memory-mapped physical flash chip, so it is inaccessible if the flash is dumped.

Note: the Maxim SDK describes the interface for the AES peripheral inconsistently. The code sample below contains the enum for selecting the mode in which the peripheral is in:

```
/**
* @brief Enumeration type to select AES key source and encryption type
```

```
  *
  */
typedef enum {
    MXC_AES_ENCRYPT_EXT_KEY = 0, ///< Encryption using external key
    MXC_AES_DECRYPT_EXT_KEY = 1, ///< Encryption using internal key
    MXC_AES_DECRYPT_INT_KEY = 2 ///< Decryption using internal key
} mxc_aes_enc_type_t;
```

6.5 Receiving Packets

We also implemented additional checks within the host messaging tools to improve security. Based on the type of message, we check that the packet is valid in length. In our Encoder implementation, we send encrypted frames of no more than 128 bytes and subscription requests are no more than 68 bytes. Larger packets will be ignored.

After reading, we flush the UART receive buffer to prevent extraneous bytes from getting read in future `read_packet` calls.

References

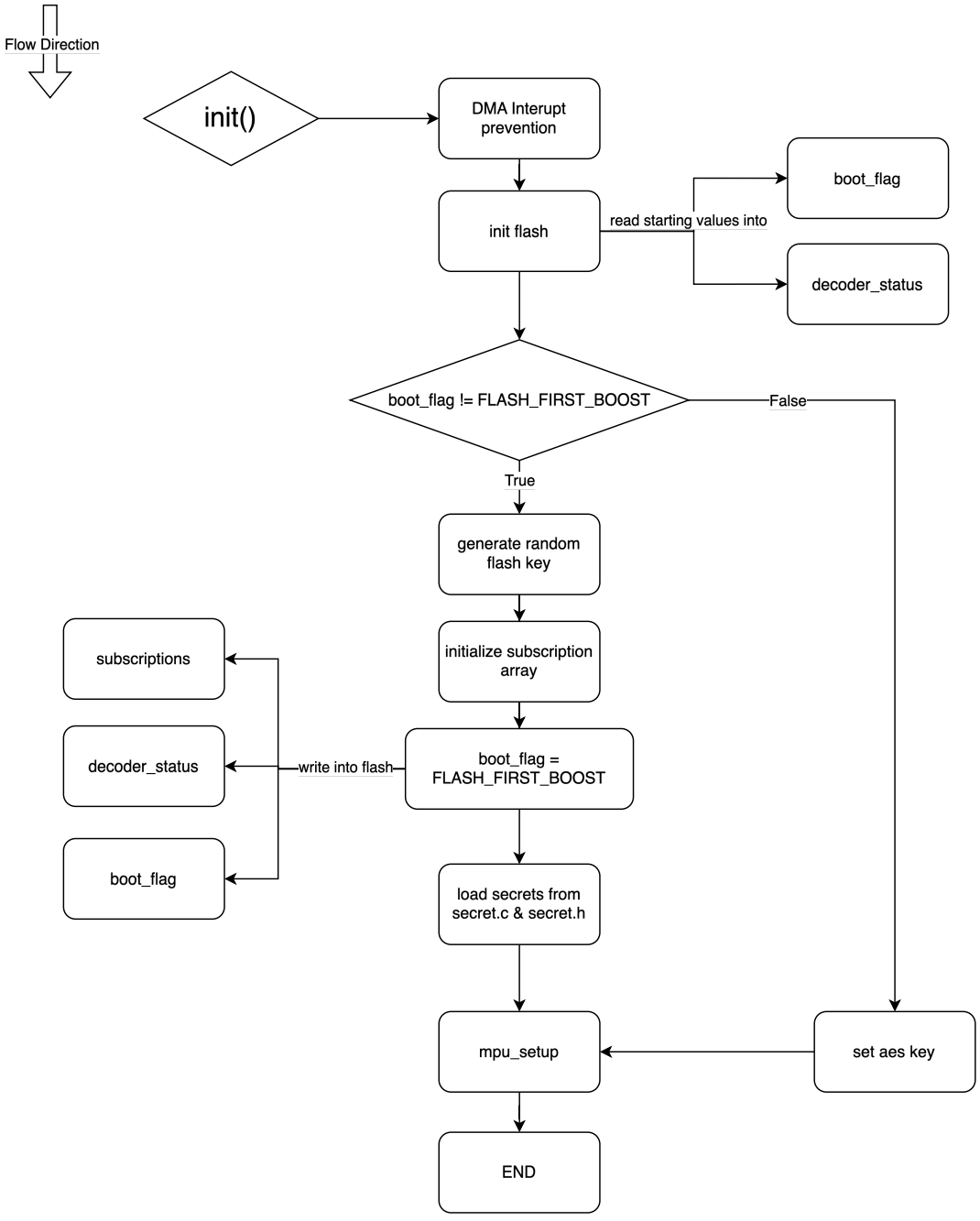


Figure 1: init flow

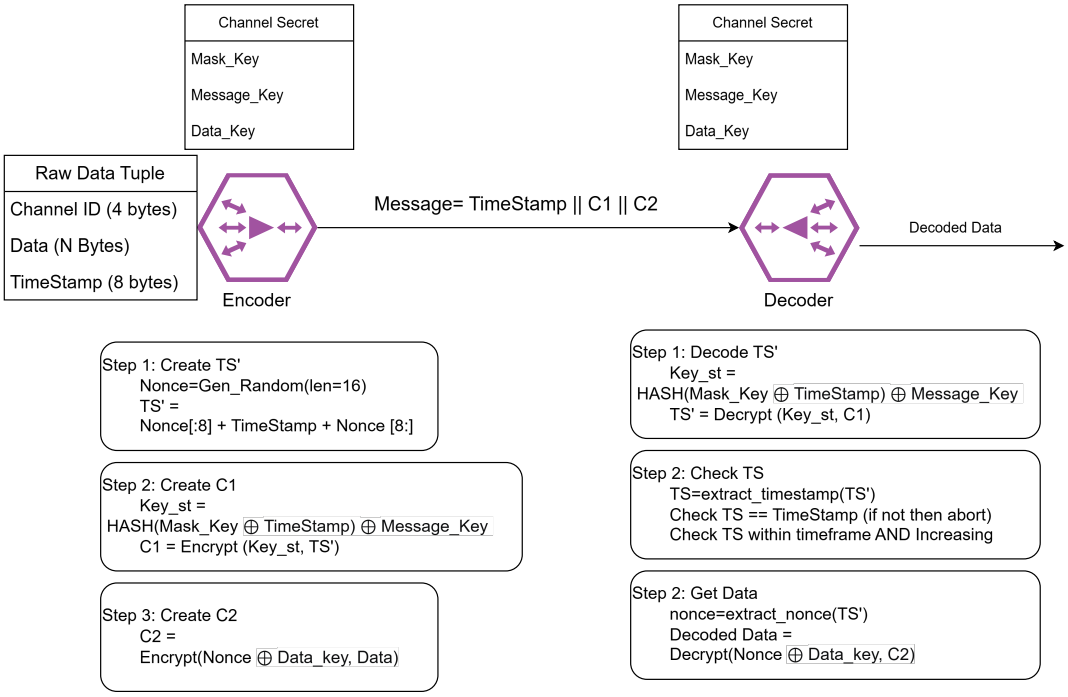


Figure 2: TAICHI Protocol

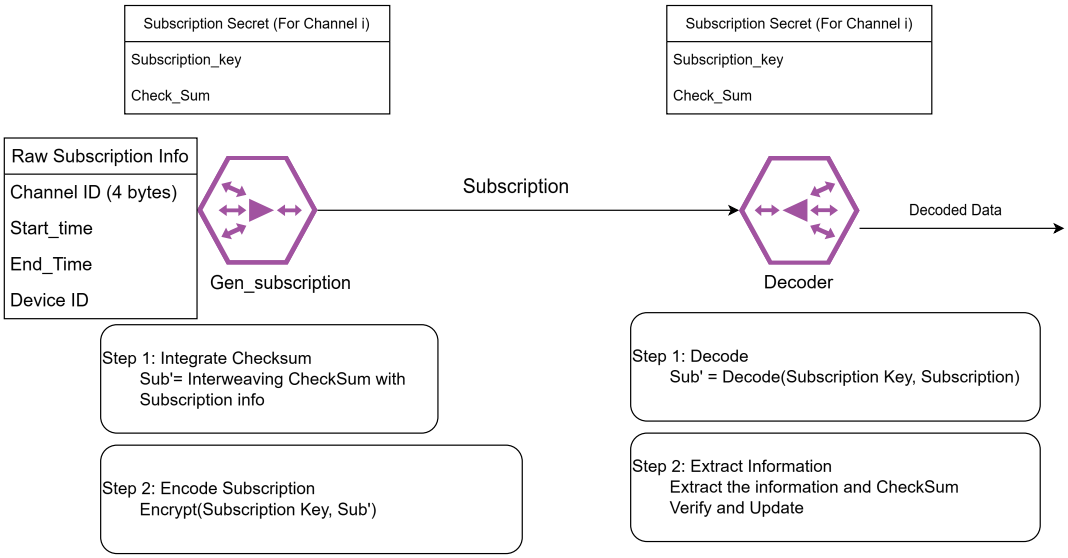


Figure 3: Subscription Protocol