eCTF Design Document (UW)

Cryptographic Algorithms

The design uses various specific cryptographic functions that balance speed and security.

The symmetric cryptographic algorithm used for the design was ChaCha20-Poly1305. Testing of ChaCha and AES-GCM provided results that showed ChaCha being more than twice as fast as AES-GCM. ChaCha is more secure against power and timing attacks, so as a result of the added security and faster computation, ChaCha20-Poly1305 was chosen for this design.
The symmetric algorithm also computes a tag allowing for integrity check.

The hashing algorithm used for generating symmetric keys is SHA-256. Tests done on various hashing algorithms (MD5, SHA-1, SHA-256, and BLAKE2s) showed that BLAKE2s provided the best tradeoff between time of computation and increased security.

The (asymmetric) signature algorithm used in this design is Ed25519. This is an industry-standard algorithm used in various places that is reliable and has short signature sizes.

In addition to these algorithms, the HMAC Key Derivation Function was used due to its compatibility with SHA-256 and ease of use.

Situation:

1) The frames that are broadcast from the encoder are viewed by everybody.
2) The frames must be decrypted quickly (at least 10/second and 15/second on average).
3) The frames must only be viewable for those with a subscription.
4) The frames must only be viewable for those with a valid subscription to a specific range of frames.
5) The frames must be only decoded if they come from the non-attacker encoder.

- Symmetric Encryption
    - ChaCha20-Poly1305
        - Provides integrity for
            - Plaintext Metadata
            - Encrypted data
        - Tags can be checked to verify decodes
    - After performing speed tests and weighing possible power and timing attacks, we decided to use ChaCha-20-Poly1305 instead of AES-GCM.
        - AES-GCM
            - ~280µs per encryption
            - ~280µs per decryption
        - ChaCha-20-Poly1305
            - ~120µs per encryption

            - ~120µs per decryption
- Asymmetric Encryption
    - Ed25519 public/private keypairs
    - ECDSA over the secp256r1 elliptic curve for signatures
- Hashing
    - Tests:
        - ARM
            - MD5: 63µs per hash
            - SHA-1: 114µs per hash
            - SHA-256: 87µs per hash
            - BLAKE2s: 97 µs per hash
        - RISC-V
            - MD5: 341µs per hash
            - SHA-1: 532µs per hash
            - SHA-256: 508µs per hash
            - BLAKE2s: 418µs per hash
    - SHA-256
        - After performing hashing speed tests, SHA-256 was the second fastest algorithm when compared to MD5, SHA-1, and BLAKE2s.
        - SHA-256 provided a “best of both worlds” situation, allowing us to have a high security hashing algorithm, and a high speed algorithm.

Cryptographic Secrets

Encoder

- Root tree compression key for each channel (besides the emergency channel)
- Emergency channel key
- Private factory key (sign subscriptions)
- Subscription key derivation key (encrypt subscriptions)

Decoder

- Subscription
    - Varying levels of compression tree keys to decode frames in timestamp range
- Update subscription
    - Device key derived using subscription key derivation key
    - Public factory key (check signature)
- Emergency Channel
    - Factory key used to decrypt all frames

General Concept

- Each frame of each channel has its own symmetric key
- We build a binary tree for each channel from the root to each frame (fig 1)
- From any node in the tree it is possible to derive its children (and grandchildren, and so on), but not possible to derive its parents
    - Children can be derived by hashing the parent with a known salt

Encoding

- The encoder knows the root of the tree for each channel
- Maintains state in an in-order traversal of the key derivation tree
- Derives next key for every frame
- Amortized, requires <2 hashes per frame

Decoding

- The decoder has a library of subscription keys on the various channels
- Decoder gets the correct key from the library and uses it to derive the needed key
    - Decoder saves the stack of keys derived in the process, allowing sequential key derivations to be achieved much faster
    - Assuming sequential need of keys, <2 hashes per frame (amortized)
    - Maximum of 64 hashes, given random access from root level key
- Frames are checked against their Poly1305 tag to ensure fidelity of the data

Generate Subscription

- The subscription generator creates a minimal set of keys to include
    - The subscription will not include any keys that can be used to derive keys outside the subscription
    - Example is a subscription of frames 16-20, as shown in figure 1. We would include the keys *f(x)* and *f(f(g(k)))* (circled in blue) in the subscription
- The entire subscription packet is encrypted using a key derived using a key derivation function and the decoder id, meaning that only the correct decoder can read the subscription then signed
    - If another decoder tries to read the subscription, the keys contained within it will be incorrect, and the decoder won't recognize the frames decoded using this key as having come from the satellite
    - Signing ensures that the subscription has not been tampered with and that it came from the factory

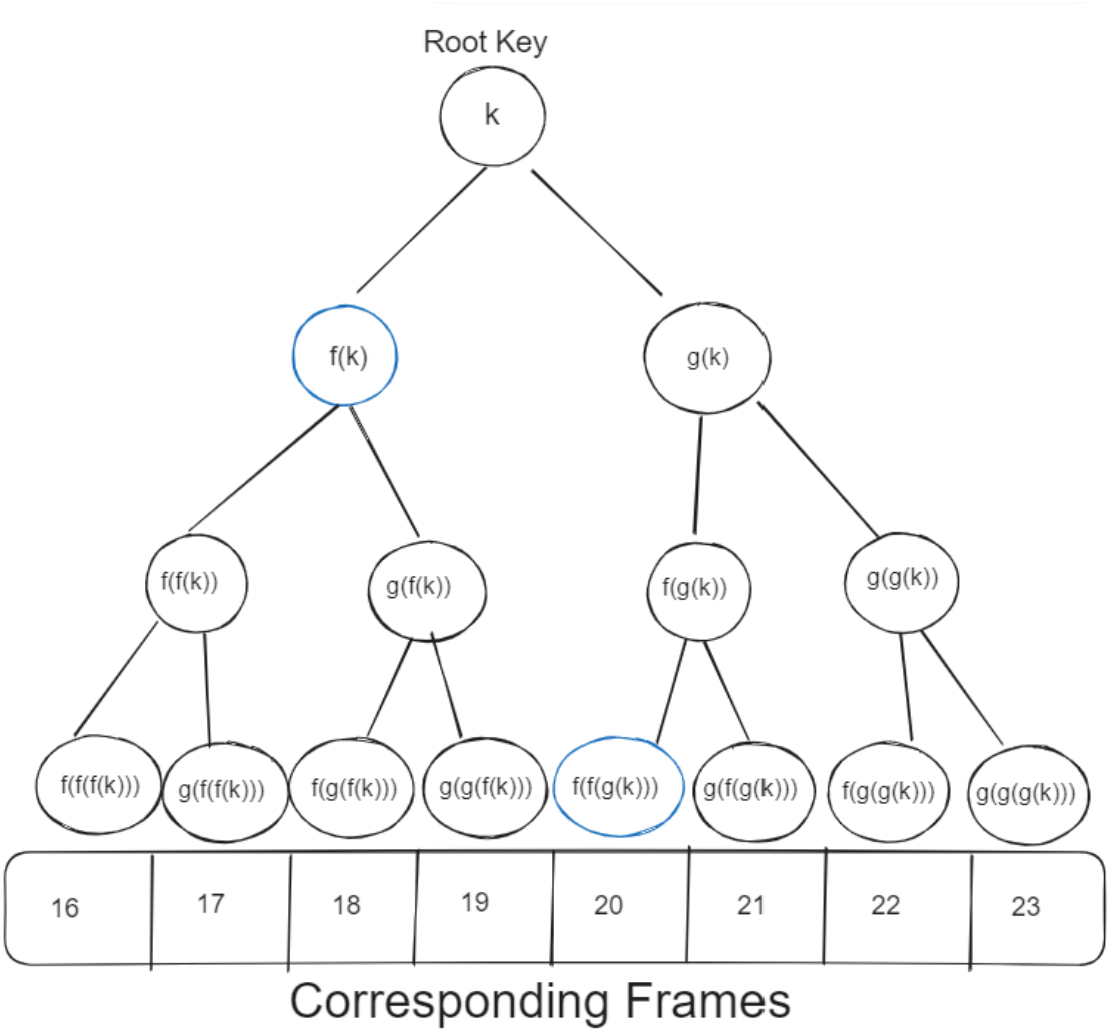
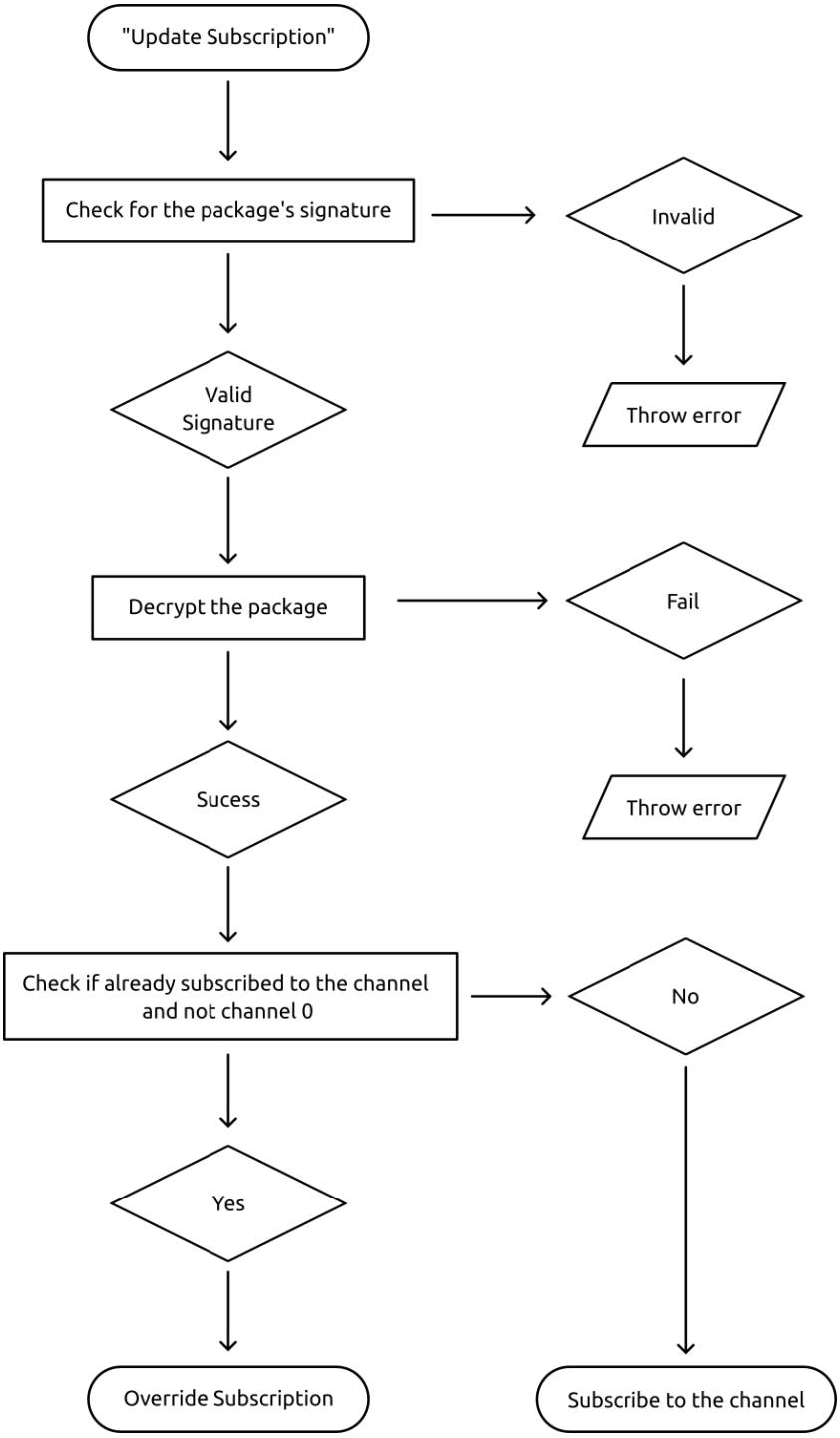


Figure 1. Binary Key Derivation Tree

Protocols

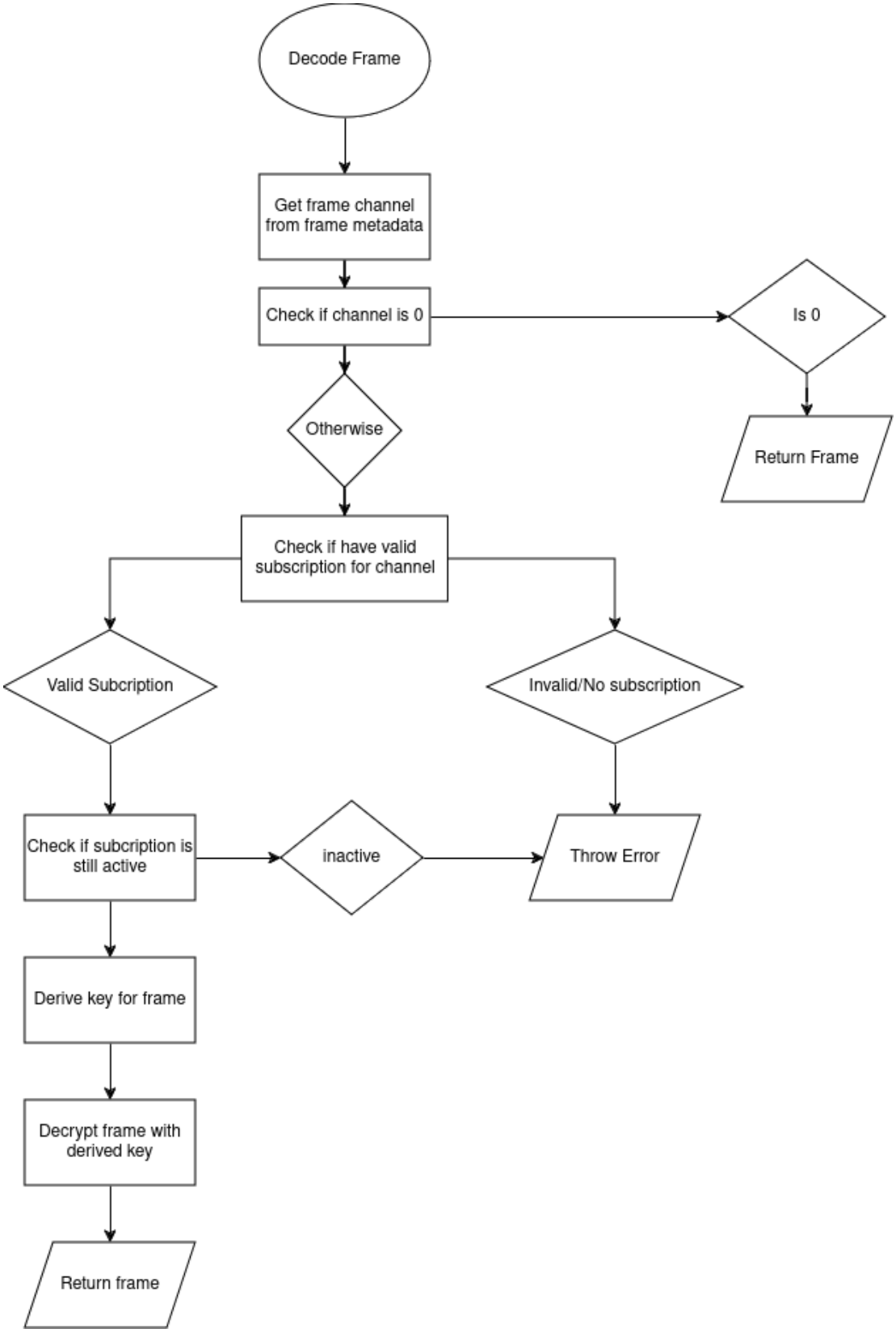
Commands

- List Channels
- Update Subscription



Control flow chart for update subscription command

- Decode Frame



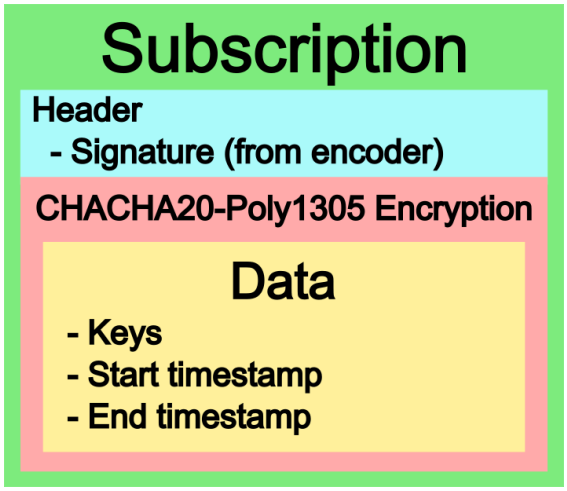
Serialization Formats/Other Important Data Structures

Design Package

Generate Secrets

- 16 bytes of a unique symmetric key (a per-decoder key)
- 32 bytes of encoder's public key

Generate Subscription



The format of the subscription blob is as follows:

- Signature (64 bytes)
- Encrypted subscription data (4632 bytes)
    - Start (uint64_t)
    - End (uint64_t)
    - Keys (This is an array of size MAX_SUBSCRIPTION_KEYS * sizeof(struct key) = 128 * 36 bytes = 4608 bytes
        - Each key object is 36 bytes, consisting of start and end (uint64_t), key (16 bytes), and level (uint32_t)
    - Decoder ID (uint64_t)
- Note: since this is larger than 256 bytes, it will need to be split into chunks (with acknowledgements) before being sent to the decoder.

Below are the tentative in-memory data structures we will use to store the subscription data. Note that we believe that MAX_SUBSCRIPTION_KEYS is at most 128.

```
struct subscription {
        uint8_t sig[64];
        uint8_t enc_sub[sizeof(struct subscription_data)];
}

struct subscription_data {
        uint64_t start;
        uint64_t end;
        struct key[MAX_SUBSCRIPTION_KEYS];
        uint64_t decoder_id;
}

struct key {
        uint64_t start;
        uint64_t end;
        uint8_t key[16];
        uint32_t level;
}
```

Encode

The format of each encoded frame is as follows:

- Channel Number (uint16_t)
- Timestamp (uint64_t)
- Encrypted Frame Data (64 bytes)

Decoder Firmware

List Channels

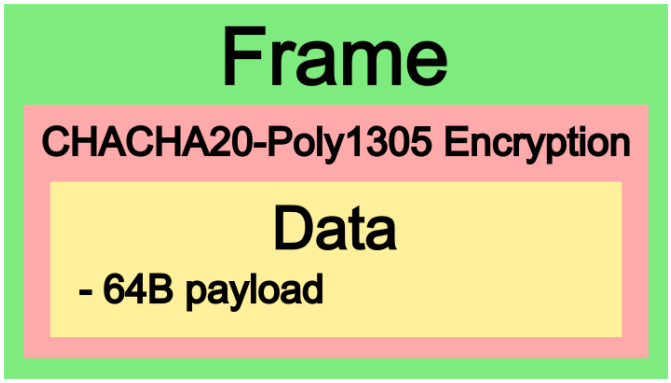
- Host command: empty
- Decoder response:
    - Num Channels (uint32_t)
    - Subscription (total of “Num Channels” entries)
        - Channel Number (uint16_t)
        - Start (uint64_t)
        - End (uint64_t)

Update Subscription

- Host Command:
    - Signature (65 bytes)
    - Encrypted subscription data (4632 bytes)
        - Start (uint64_t)
        - End (uint64_t)

        - Keys (This is an array of size MAX_SUBSCRIPTION_KEYS * sizeof(struct key) = 128 * 36 bytes = 4608 bytes
            - Each key object is 36 bytes, consisting of start and end (uint64_t), key (16 bytes), and level (uint32_t)
        - Decoder ID (uint64_t)
- Decoder Response: empty

Decode Frame



- Host Command:
    - Channel Number (uint16_t)
    - Timestamp (uint64_t)
    - Encrypted Frame Data (64 bytes)
- Decoder Response:
    - Channel Number (uint16_t)
    - Timestamp (uint64_t)
    - Decrypted Frame Data (64 bytes)

Countermeasures

The design requires several specifications to be met in various attack scenarios.

SR_1: Firstly, a decoder must have an active and valid subscription in order to decode our frames. This will be relevant in all attack scenarios except the Pesky Neighbor scenario, under the specifications of all those scenarios valid subscription which consists of a key, start, and end must contain a mechanism which prevents the decoding of frames that are not within the start and end interval.

We affirm the subscription is valid though a symmetric key assigned on a per-device basis. We hold the integrity of our subscription by using a custom binary cryptographic hash that derives a unique key for each frame from a master key. By including only the keys for the frames within the valid time frame in the subscription we prevent frames outside of the subscription from being encrypted as they do not possess the key.

SR_2: Secondly, our decoder should only decrypt frames from the correct satellite. If an attacker decoder has a subscription with access to some frame keys, we need to prevent them from modifying the data and sending modified frames with valid keys to our decoder. This is important in the Pesky Neighbor attack scenario. So to stop this we plan on signing each frame with the encoder so that each frame can be verified to be sent from the encoder

SR_3: Finally, a decoder should only decrypt our frames in a monotonically increasing order. An attacker should not be able to access frames that come before or after their assigned timeframe, nor should they be able to decrypt frames in reverse order. This is especially relevant to the Recording Playback scenario. We will address this requirement by tracking a “current” frame at all times, and checking if any frame we try to decrypt is strictly increasing in timestamp. Our implementation of frame key encryption addresses the other attack scenarios relevant to this security requirement.