Proposed System Design Document (v2)

Security Requirements (For Reference)

These security requirements **must** be met:

- SR1: A car should only unlock and start when the user has an authentic fob that is paired with the car
- SR2: Revoking an attacker's physical access to a fob should also revoke their ability to unlock the associated car
- SR3: Observing the communications between a fob and a car while unlocking should not allow an attacker to unlock the car in the future
- SR4: Having an unpaired fob should not allow an attacker to unlock a car without a corresponding paired fob and pairing PIN
- SR5: A car owner should not be able to add new features to a fob that did not get packaged by the manufacturer
- SR6: Access to a feature packaged for one car should not allow an attacker to enable the same feature on another car

Design Constraints

There are a few requirements within the system design that need to be met in order to achieve all Security Requirements:

- The car **must** be able to verify the paired fob's integrity (SR1)
  - Fobs will encrypt unlock messages using the shared key (AES)
- Replay attacks **must** be unachievable (SR2)
  - Two randomly generated nonces will be used in communications, preventing capturing one conversation and replaying it to another device.
- Communication is meaningless to those without private keys (SR3)
  - Communication will be signed (and/or encrypted?) using ECC
- Unpaired fobs should not be able to unlock the car, even with the pairing pin.
  - There will not be any correlation between verifying a pairing pin and the car's unlocking mechanism.
- Features are not forgeable (SR5)
  - Features are **signed** by the manufacturer
- Features packaged for one car should not be enableable on another (SR6)
  - Features will be futher signed with the car's public key.

Devices

There are two types of devices - cars and fobs. They all share common variables used for integrity verification in handshakes:

- `priv_key`: device unique private key
- `pub_key`: device unique pub key
- `signature`: factory signature of public key

These are all generated in factory and are unique to each device (except for factory signature) - they *never* change.

In addition, paired fobs and cars will both contain a `symmetric secret` used in the unlock process.

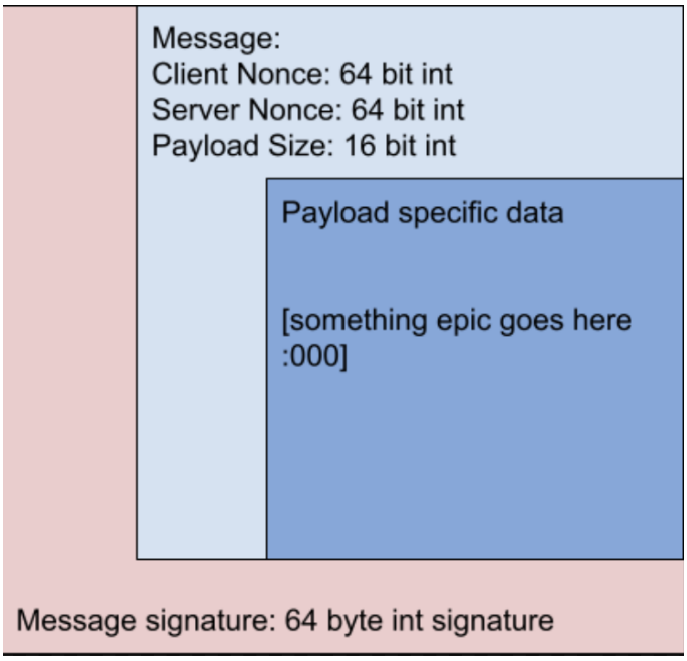
Messages

Messages are packets sent over UART1 from one embedded device to another. While all devices can send messages, only fobs can initiate a conversation between the two devices. They have the following structure:

```
struct Message {
    uint8_t magic; //packet type
    uint64_t nonce_c; //Randomly generated nonce by the sender (client)
(check if 8 is fine for nonce)
    uint64_t nonce_s; //Randomly generated nonce by the recipient (server)
- 2 nonces prevent "replay conversation" attacks
    uint16_t size;
    uint8_t[size] payload; //type-specific payload, signed by the
recipient's public key - ensures intented recipient is correct.
}

struct MessageSignature{
    uint8_t[64] signature; //Message signed by the factory public? key.
Ensures only factory devices in conversation
}
```

The following is an amazing diagram of the message structure:



The goal of the `Message` data structure is to prevent reply attacks and ensure the integrity of each message.

Ways the message can get discarded (fail validation):

- Invalid magic (see below)
- Invalid nonces
- Payload cannot be decoded by the recipient's private key (wrong recipient)
- MessageSignature does not match with the Message when decoded (not factory device)
- Invalid packet magic? (tentative)
- Payload size does not match packet type size
- Key_ex packet not sent during a conversation initialization (Attacker sends a random challenge packet to try to get output)

The following are valid packet types (magics):

```
key_exchange (0x59) | any <--> any
unlock       (0x6A) | paired fob <--> car
pair         (0x7B) | unpaired fob <--> paired fob
feature      (0x8C) | factory <--> paired fob
```

Messages are signed with either device public keys or factory public keys. If the signature does not match the contents, it will be discarded.

```
The `key_exchange` message is special in that it is the only message that
does not require a valid device signature. Instead, the payload contents
(a public key) **must** be signed using the factory's private key. This is
to ensure only legitimate devices can communicate with one another.
```

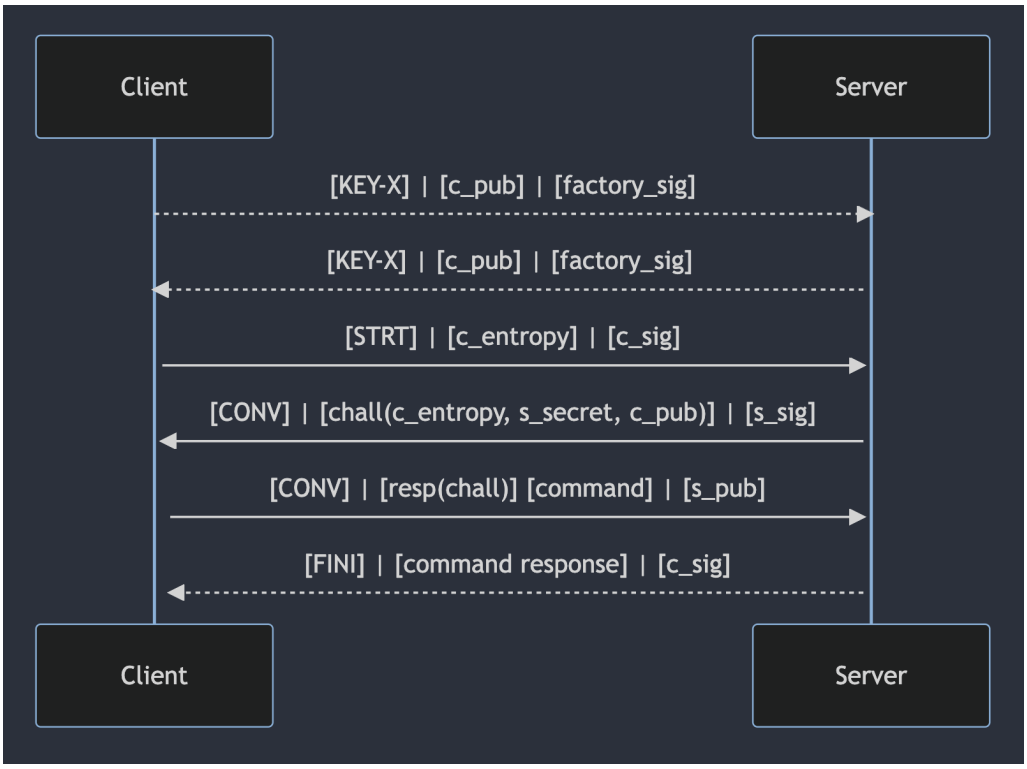
Nonce generation:

We plan on having two nonces in our messages. This will prevent an attacker from capturing one conversation and replaying it later (as per SR3). Only `key_ex` packets can reset a nonce. When a client begins a conversation, they will only send one nonce, their own randomly generated number. The server will take that, save it, and send another with the nonce of the client and their nonce (see Message data structure). After that, the client saves the server Nonce and continues the conversation as usual. Devices will discard their nonce when either the messages get discarded or when the conversation ends.

Conversations

Conversations are a **two way** handshake between embedded devices comprised of *messages*. The implementation goal is to ensure communication secrecy and disallow replay attacks (As per SR3).

Before any conversation, it is necessary to exchange certificates to ensure messages have not been tampered with using the `key_exchange` message. If a **factory-signed** public key is presented, it is stored in the recieving device and used to verify succeeding messages in the sequence.




```
coloring is hard - the first two messages are part of the key exchange;
the following four are part of the conversation.
```

Each conversation consists of a **client** and a **server**. The **client** always initiates the conversation with a `key_ex` message and the following `payload`:

```
struct key_ex {
    uint8_t[64] device_pub;
}
```

Only fobs can initialize a conversation (send a key_ex message).

Upon recieving a valid key_ex message, a conversation is initiated between the sender and recipient of the messages, comprised of a two way *Challenge* followed by an accept/reject message by the server.

The next step is an `acknowledge` message:

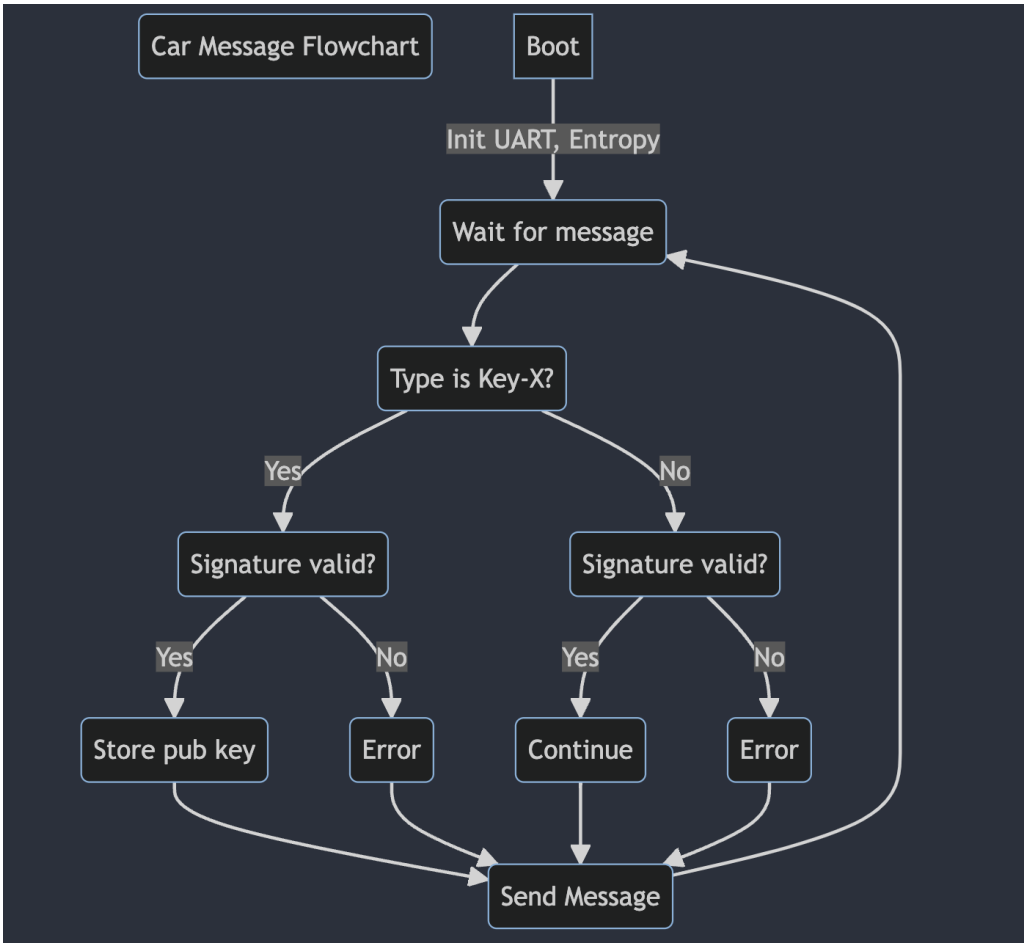
```
struct ack {
    uint8_t[64] chal; //f(c_entropy, c_pub, s_entropy)
}
```

Upon verification that the response is legit, the client will send a request packet to actually execute a command:

```
struct exe {
    uint8_t[64] resp; //f(chal)
    uint8_t[size - 64] command; //command specific data
}
```

Finally, an accept/reject message is sent to the client from the server:

```
struct status {
    uint8_t[64] msg; //char[] message
}
```

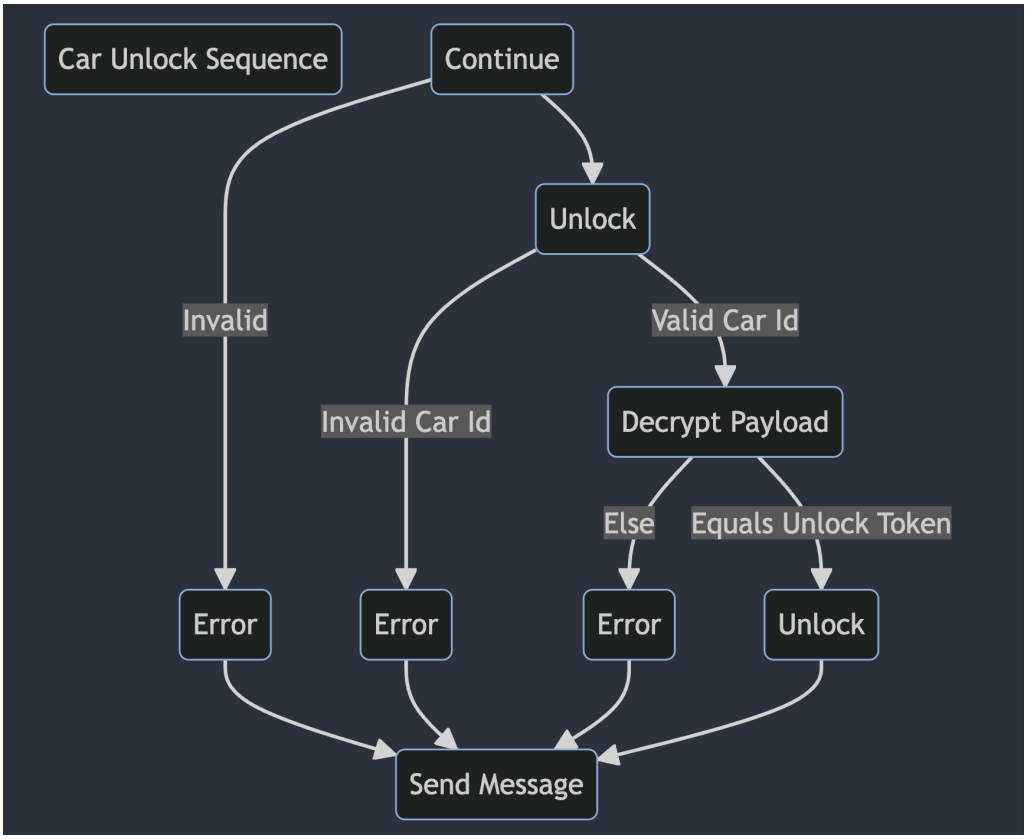


Commands

Unlock

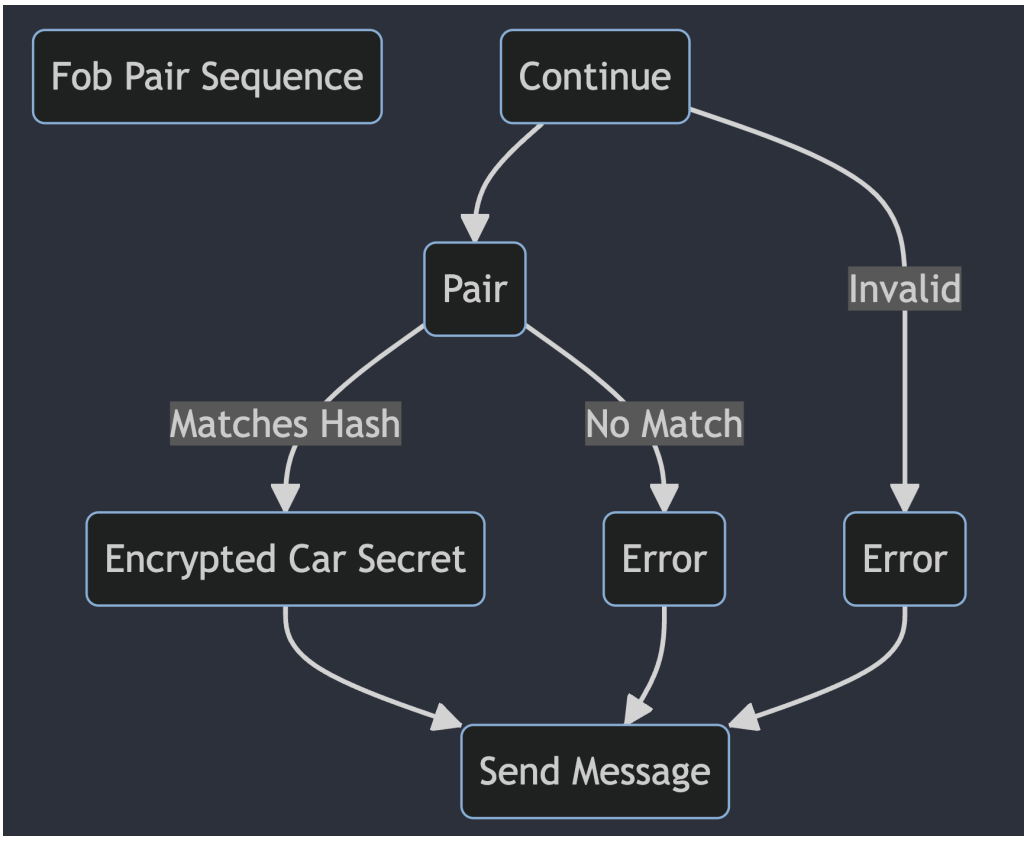
```
struct Unlock {
    uint8_t unlock_magic;
    uint8_t car_id; // Intended car
    uint8_t[64] encrypted_unlock_msg; //if the car can decrypt this with
its shared key, it will unlock
```

```
}
```



Pair

```
struct Pair {
    uint8_t pair_magic;
    uint8_t[6] pin; //no need for this to be hidden from an attacker
}
```



TODO: How is response going to be encrypted?

Feature

```
struct Feature {
    uint8_t feature_magic;
    uint8_t[64] feature_proof; //signature
    uint32_t[3] feature; //The feature to be added
}
```

If the payload structs we have listed above do not meet our size requirements, they will be subject to change.

Concerns

Entropy

Entropy is hard. By using entropy from two completely separate devices, we can hopefully make it much harder to create predictable / repeatable patterns.

We have 2 plans for introducing entropy on our machines, based on how much RAM and memory our devices have. Both are based on a 128 bit seed, which will be generated during our build process:

- Scenario 1: Each seed will be randomly generated by the factory and sent to all the devices.

- Scenario 2: When scenario 1 does not work. Our seed is generated by a combination of the starting temperature and the RAM of the device on startup (which appears to be semi-random according to this study).

Our rand() function will be a combination of the seed and the analog system of the two devices during a conversation. Our hope is that even if attackers can reliably determine the randomness of one device, the second device can add a layer of additional randomness.

Pin Concerns

A six digit pin is easily brute-forcible. Normally, this could be prevented using a brute-force detection system, but by completely reflashing the board, this can be bypassed. In the attack scenarios where the attacker gets a pairing pin, they do not get access to a paired fob, and vice versa. This makes brute-force pairing/unlocking much more difficult, even if the board is reflashed to circumvent brute force detection.

Our brute force detection system will automatically delay any conversation messages after 5 (an arbitrary number) attempts, increasing in time exponentially with more failed attempts.

Build Environment

We see no reason to modify this from a security standpoint; all secrets will be inacessible to attackers (see *Build Deployment*).

Build Tools

Because attackers have full access to build tools and can modify them at will, we shall assume they are compromised and therefore will not contribute to cryptography whatsoever.

Build Deployment

Because our design relies on factory secrets for signature verification (and encryption), these values **cannot** be leaked to attackers in the build environment. Therefore, cryptographic secrets shall be dynamically generated in the secrets volume upon each build. Therefore, each device will have separate cryptographic keys that cannot be found in the build process. Entropy will also be added to devices at this stage of the build process, according to one of our scenarios listed above in the entropy section.

Build Car/Fob Pair

Because our car/fob pair's unlocking design relies on the fob knowing the car's secret id, we will add the car's id into paired fob designs. Because we will be building these at the same time, we have decided that the Car/Fob pairs built at this stage will have the same public/private key pair. This should theoretically be inaccessible to potential attackers.

Build Unpaired Fob

This will generate a fob without the car's secret key, to be added later during pairing.

Run Unlock, Run Pair, Run Package Feature, Run Enable Feature

Because attackers will have full access to each device, we are going to assume these are compromised and therefore will not contribute to security. See above for how we will try to secure these requirements. No keys/entropy/secrets will be added to the device.

Other

For our cryptography library, we have decided to use BearSSL, a lightweight, modular library intended for embedded applications.

We have chosen this for its:

- small package size
- side-channel fortification
- compatability with arm-v7 architecture