Design of a Secure Vehicle Entry System (eCTF 2023)

BRIAN MAK, STEVEN MAK, JEFFREY ZHANG, VICTOR HO, CHIARA KNICKER, JACKSON KOHLS, NANCY LAU, IAKOV TARANENKO, STEPHEN LU, EYA BADAL ABDISHO, ALVARO A. CARDENAS, University of California Santa Cruz, USA

This document summarizes our design considerations for MITRE's Embedded Capture the Flag 2023 competition.

1 OVERVIEW OF DESIGN CONSIDERATIONS

Our design includes the analysis of cryptographic protocols and the use of memory-safe languages.

In particular, we consider the following general security principles:

(1) Data protection in transit: We use authenticated encryption to protect the integrity (and confidentiality) of every message exchanged. We looked at the winners of the CAESAR competition as well as the recommendations and best practices for this mode. We chose `XChaCha20-Poly1305` over AES in GCM mode because it is faster in software implementations.
(2) Data protection at rest: We need to keep all secret keys and the PIN secure. This is achieved by ensuring that all secrets are stored in the EEPROM, which is not directly accessible to attackers. In addition, we zeroize RAM used to temporarily store secrets following each usage of a secret.
(3) Trust: We use ECDSA digital signatures to verify our keys and features, creating a chain of trust.
(4) Key management: We have long-term symmetric keys shared between paired key fobs and cars, and long-term public keys to authenticate keys from the manufacturer. In pairing communications, we use an ephemeral Elliptic-Curve Diffie-Hellman key exchange. To authenticate a previously untrusted unpaired and paired fob for pairing, we exchange randomly generated public keys signed with a key generated by the manufacturer, which are also signed to guarantee authenticity.
(5) Secure cryptographic material generation: To generate secure cryptographic material, we use a ChaCha20-based CSPRNG seeded with entropy from various hardware sources. See section 7 for more details on how we gather entropy securely.
(6) Memory-safe code: To implement our designs, we choose Rust as our primary programming language, which is memory-safe. This reduces the probability of memory vulnerabilities, which account for a large portion of security issues in large codebases.

Our design uses the following symmetric keys:

k_{u1}: Key Fob Encryption Key. Known by a car and its paired key fobs. Unique to each car.
k_{u2}: Car Encryption Key. Known by a car and its paired key fobs. Unique to each car.

And the following asymmetric keys:

k_{fs}: Feature Signing Key. Known by the build system. Unique to each manufacturer.
k_{fv}: Feature Verifying Key. Known by all cars and fobs built by the same manufacturer.
k_{mps}: Pairing Manufacturer Paired Fob Signing Key. Known by the build system. Unique to each manufacturer.
k_{mpv}: Pairing Manufacturer Paired Fob Verifying Key. Known by all key fobs built by the same manufacturer.
k_{mus}: Pairing Manufacturer Unpaired Fob Signing Key. Known by the build system. Unique to each manufacturer.
k_{muv}: Pairing Manufacturer Unpaired Fob Verifying Key. Known by all key fobs built by the same manufacturer.
k_{ps}: Paired Fob Pairing Signing Key. Unique to each paired and unpaired key fob.
k_{pv}: Paired Fob Pairing Verifying Key. Unique to each paired and unpaired key fob.
k_{us}: Unpaired Fob Pairing Signing Key. Unique to each unpaired key fob.

Author's address: Brian Mak, Steven Mak, Jeffrey Zhang, Victor Ho, Chiara Knicker, Jackson Kohls, Nancy Lau, Iakov Taranenko, Stephen Lu, Eya Badal Abdisho, Alvaro A. Cardenas, University of California Santa Cruz, 1156 High St, Santa Cruz, CA, USA, 95064.

k_{uv}: Unpaired Fob Pairing Verifying Key. Unique to each unpaired key fob.

We now describe how we apply these principles to the different requirements.

2 BUILD ENVIRONMENT

Our project is written in Rust and C.

For Rust, our build environment includes the Rust toolchain with the `thumbv7em-none-eabihf` target and the host `gcc` toolchain for linking purposes. For C, our build environment includes the `arm-none-eabi-gcc` toolchain. Our build environment is contained within an OCI container image based on Arch Linux.

The only usage of C in the project is for our random number generation, to read from uninitialized memory. Rust's built-in safety prohibits us from reading from uninitialized memory without invoking undefined behavior, so we use C to work around that.

3 BUILD HOST TOOLS

The Build Host Tools step is responsible for building host tools to be used on the user's machine, and places them in the Host Tools volume, accessible by `ectf_tools`.

Host tools are used in all four use cases: Unlock, Pair, Enable Feature, and Package Feature. These tools are primarily used to facilitate host-to-firmware and firmware-to-host communication. The exception to this is the Package Feature host tool, which does not communicate with the car or key fob at all. Both the Pair and Enable Feature host tools initiate a connection to a key fob, while the Unlock host tool receives a message from the car when an unlock is successful. All host tools, except Package Feature, have been written to run in an insecure environment. The Package Feature host tool runs in a secure environment with access to secrets.

4 BUILD DEPLOYMENT

The Build Deployment step generates the secrets used by cars and key fobs in the deployment. These secrets are stored in a Host Secrets volume and are accessible to the build process for all cars and fobs in the deployment.

During the Build Deployment step, we generate three 256-bit keys: k_{mps}, k_{mus}, and k_{fs}. The Pairing Manufacturer Paired Fob Signing Key k_{mps} is used during the key fob build process to sign a key fob's Paired Fob Pairing Verifying Key k_{pv}. Similarly, the Pairing Manufacturer Unpaired Fob Signing Key k_{mus} is used during the key fob build process to sign an unpaired key fob's Unpaired Fob Pairing Verifying Key k_{uv}. The Feature Signing Key k_{fs} is used to sign packaged features during the feature packaging process. The Feature Verifying Key k_{fv} is derived from the Feature Signing Key k_{fs}, and is known by all key fobs and cars in a deployment. The Pairing Manufacturer Paired Fob Verifying Key k_{mpv} and Pairing Manufacturer Unpaired Fob Verifying Key k_{muv} are derived from their respective signing keys and known by all key fobs in a deployment.

5 BUILD CAR AND PAIRED FOB

The Build Car and Build Paired Fob steps build images for a car and a key fob paired to the car. Each image includes a firmware binary and EEPROM file. During this step, secrets are read from the Host Secrets volume and new secrets are generated that are unique to each device or shared between a car and paired fob. The new secrets are placed into the volume as well.

During the Build Car step, we generate two 256-bit keys, the Key Fob Encryption Key k_{u1} and the Car Encryption Key k_{u2}, and place them in the Host Secrets volume.

During the Build Paired Fob step, we generate another 256-bit key, the Paired Fob Pairing Signing Key k_{ps}. This key is also placed in the Host Secrets volume. The Paired Fob Pairing Verifying Key k_{pv} is derived from the Paired Fob Pairing Signing Key k_{ps} and signed with the Pairing Manufacturer Paired Fob Signing Key k_{mps} to produce a signature s_{pv}. This signature s_{pv} is placed in the Host Secrets volume as well.

During both the car build step and paired key fob build step, we generate a secret seed. This secret seed is unique per device and is used as one of the sources of entropy for the pseudorandom number generator.

The EEPROM of the car and paired key fob each contain the Car ID, Secret Seed, Key Fob Encryption Key k_{u1}, Car Encryption Key k_{u2}, and Feature Verifying Key k_{fv}. The EEPROM of the car additionally contains the Unlock Message, Feature 1 Message, Feature 2 Message, and Feature 3 Message. The EEPROM of the paired key fob additionally contains the Pairing Byte that is set to `0x01010101` to indicate that the key fob is paired, the Pairing Manufacturer Paired Fob Verifying Key k_{mpv}, the Paired Fob Pairing Signing Key k_{ps}, the Paired Fob Pairing Verifying Key Signature s_{pv}, the Pairing Manufacturer Unpaired Fob Verifying Key k_{muv}, the Pairing PIN, and the Pairing Longer Cooldown Byte which is set to 0 to indicate that the fob has not received any invalid PINs yet. The general layout of the EEPROM, and the layout for the car and paired key fob after the build step can be seen in Figure 1.

Following the key generation process, the build process of the car and paired key fob produces three artifacts each: the firmware ELF, the code sections from the ELF, and the EEPROM file. The ELF executables are generated using the Rust toolchain, and the code sections are extracted from the ELF using `objcopy`. The EEPROM files are generated by Rust build scripts that are run before building the key fob and car firmware.

6 BUILD UNPAIRED FOB

The build process of the unpaired key fob is the same as the paired key fob, except for the EEPROM file generation.

During the Build Unpaired Fob step, we leave the Pairing Byte in the EEPROM zeroed to indicate that the fob is not paired. The unpaired key fob also lacks the Key Fob Encryption Key k_{u1}, the Car Encryption Key k_{u2}, the Car ID, and the Pairing PIN. These EEPROM fields are left zeroed in the unpaired key fob. Additionally, we generate the Unpaired Fob Pairing Signing Key k_{us}. The Unpaired Fob Pairing Verifying Key k_{uv} is derived from this, then signed with the Pairing Manufacturer Unpaired Fob Signing Key k_{mus} to produce a signature s_{uv}. The Unpaired Fob Pairing Signing Key k_{us} and Unpaired Fob Pairing Verifying Key Signature s_{uv} are then placed in the Host Secrets volume and stored in the unpaired key fob's EEPROM. The EEPROM layout of the unpaired key fob can be seen in Figure 1.

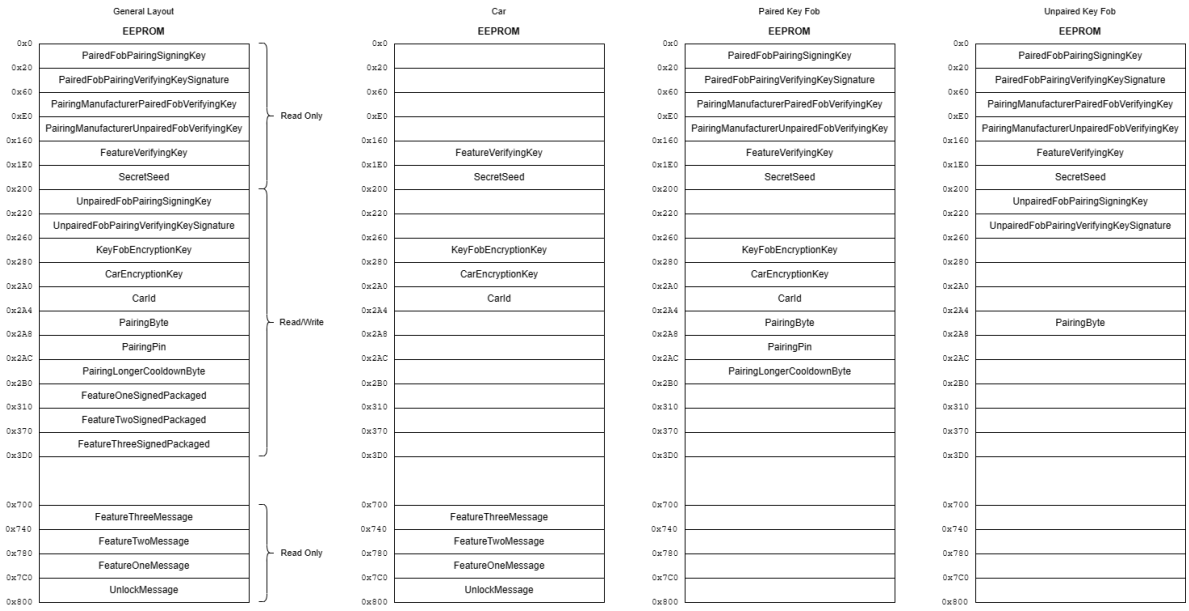

Fig. 1. EEPROM layouts after building

7 RANDOM NUMBER GENERATION

To create random numbers within cars and fobs, which are used for a variety of processes such as creating challenges, we need a source of entropy. Certain pseudorandom number generators, such as the Mersenne Twister are not cryptographically secure, so instead, we use a cryptographically-secure pseudorandom number generator based on the ChaCha20 algorithm. To seed this pseudorandom number generator, we take randomness from several sources on the board.

We take a SHA3-256 hash of data from the following sources: uninitialized memory, clock drift between the hibernation RTC and CPU clock, a secret seed generated per device, and ADC output from the internal temperature sensor. Using this hash, we have enough entropy to pass into the ChaCha20 pseudorandom number generator.

To change our uninitialized memory upon CPU reset, we replace it on start-up to guarantee a different seed from this source of entropy each time. We take a SHA3-256 hash of the memory and the secret seed and use this to seed a second ChaCha20 pseudorandom number generator. This pseudorandom number generator is used to cycle the memory. Thus, the next read to this block of memory will result in a unique block of data.

8 RUN UNLOCK

8.1 Functional Requirements

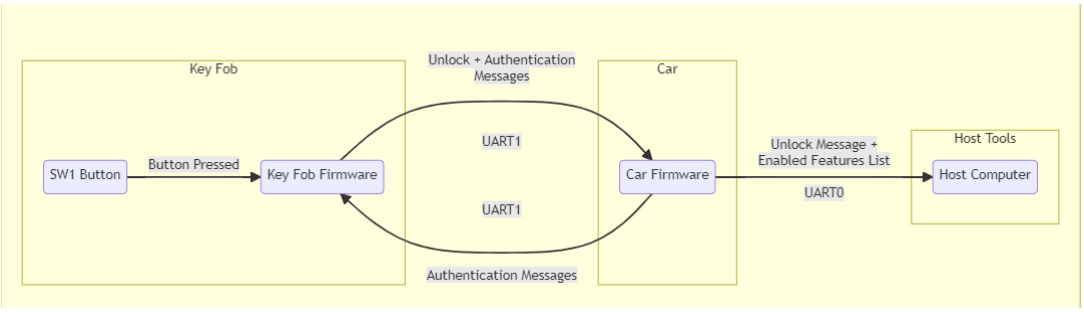

Fig. 2. Overview of Unlock Functionality

To unlock a car, the user must ensure their key fob is paired with the car they want to unlock. See section 9 for how pairing a key fob to a car works. Once the key fob is paired, the UART1 pins of the paired key fob and the car must be connected to facilitate a communication link.

The car must then be connected to the host computer containing the `unlock_tool` to display the unlock message and the list of enabled features on successful unlock. The user must press the SW1 button on the paired key fob to unlock the car. Refer to Figure 2 for a visual representation of this description. See subsection 8.4 for the high-level design of this system.

8.2 Security Requirements

To ensure the security of our unlock protocol, we must meet the following security requirements.

(1) A user should only be able to unlock a car if they have a genuine key fob paired to the car.
   - This entails that there should be no way for a user to unlock a car using an unpaired key fob, a key fob paired with another car, or an unauthorized key fob. If their key fob is already paired to a particular car, they should not be able to use the traffic received and sent to it to unlock any other car.
   - Additionally, a user should not be able to clone a paired key fob. This requirement will be addressed in section 9.

(2) Even if a user previously had physical access to a paired key fob, they should not be able to unlock the car the fob was paired to after they no longer have access to the key fob.
   - Because the user previously had access to a paired key fob, we must assume that they had direct access to the communications sent to and from that key fob, meaning they could replay it at a later time, potentially jamming/delaying any particular communication and/or modifying it in the process.
   - Just like the previous security requirement, this also means that an attacker should not be able to clone a paired key fob.

(3) Eavesdropping on communications sent between a key fob and a car for the unlock protocol should not allow the user to unlock the same car in the future.
   - This requirement is similar to the previous one. However, this one is focused on passively listening in on traffic.

8.3 Potential Attack Vectors

The following attack vectors were considered and mitigated in the design of our unlock protocol.

(1) Man-in-the-middle attacks
- In this attack, an attacker could intercept the unlock transmission from a key fob paired with another car and modify the transmission to unlock another car. This attack type defeats Security Requirements #1 and #2 outlined above.

(2) Jamming/replay attacks
- In this attack vector, an attacker could replay an unlock transmission from a key fob to a car at a later time. This attack type defeats Security Requirement #1 and potentially #2 or #3 outlined above.

Assuming the correct implementation of our design, mitigating these potential attack vectors should be sufficient to uphold all security requirements. See subsection 8.5 for how the mentioned attack vectors are mitigated.

8.4 High Level Design

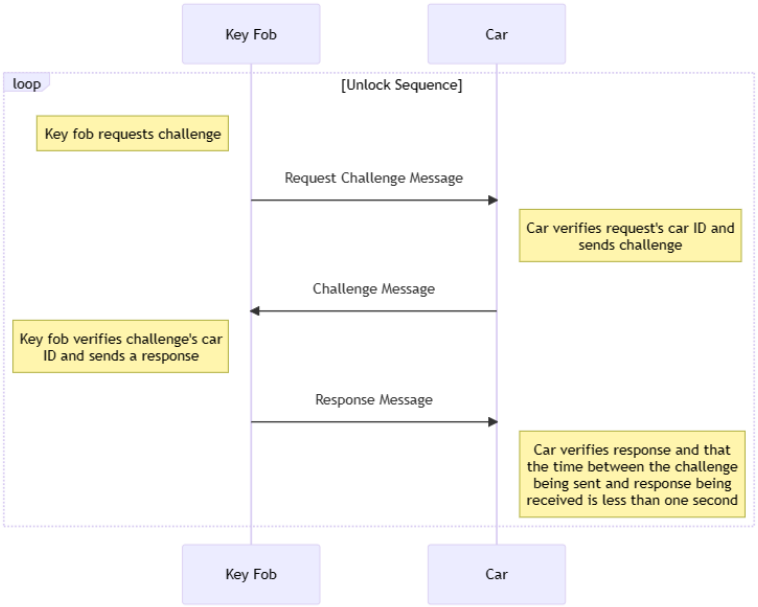

Fig. 3. Unlock sequence

Figure 3 shows our design for unlocking a car. This is a challenge-response protocol.

The key fob and car share two different secret keys: one for authenticating and encrypting messages from the key fob to the car, Key Fob Encryption Key k_{u1}, and one for authenticating and encrypting messages from the car to the key fob, Car Encryption Key k_{u2}. In both cases, we use the authenticated encryption algorithm `XChaCha20-Poly1305` (based on IETF RFC 8439) due to its popularity and efficiency that does not rely on hardware support, unlike AES-GCM. We use `XChaCha20-Poly1305` instead of `ChaCha20-Poly1305` because of its extended nonce, which is better for long-lived keys, like in our use case.

Upon pressing the button on the key fob, the key fob will send a request message to the car, initiating the challenge-response protocol. This request consists of just the Car ID of the car that the key is paired to, encrypted with Key Fob Encryption Key k_{u1}. The unencrypted request message structure is shown in Figure 4.

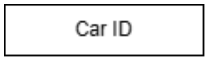

Fig. 4. Unlock request message structure

After receiving this request, the car decrypts the message and verifies that the Car ID is correct. If the Car ID is correct, the car will generate a challenge for the key fob. This challenge consists of the Car ID and a securely generated 128-bit nonce, encrypted with Car Encryption Key k_{u2}. The unencrypted challenge message structure is shown in Figure 5.

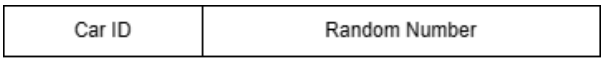

Fig. 5. Unlock challenge message structure

Once the key fob receives the challenge, it decrypts the challenge and verifies the Car ID included. If the Car ID is correct, the key fob will generate a response. This response begins the same way the challenge message in Figure 5 does but is encrypted with Key Fob Encryption Key k_{u1} instead, and additionally contains up to three pieces of optional features. The optional feature data will be used to allow for packaged feature(s) to be sent from the key fob to the car. See subsection 10.1 and subsection 10.4 for more details on this. See Figure 6 for the unencrypted challenge response structure.

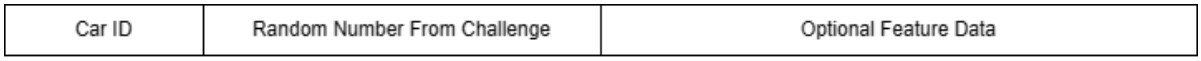

Fig. 6. Unlock challenge response message structure

The car verifies the response by checking if it was received within one second of the challenge being sent. Then, it decrypts the response and checks whether both the Car ID and the nonce from the response match the values sent out in the challenge. If it took longer than one second for the key fob to respond, the car will invalidate the response and stay locked. If the car receives a valid response within one second, it will unlock. If there is optional feature data, the car will also verify the packaged feature(s) using the Feature Verifying Key k_{fv}. The car then notifies the host computer that it has been unlocked by sending the host the unlock message and feature messages for each verified enabled feature. See Figure 7 and Figure 8 for state diagrams of the car and key fob for the unlock sequence.

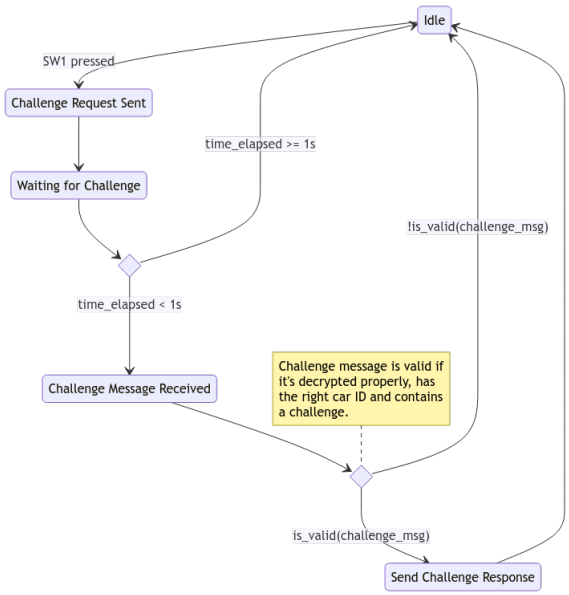

Fig. 7. State diagram of key fob communications

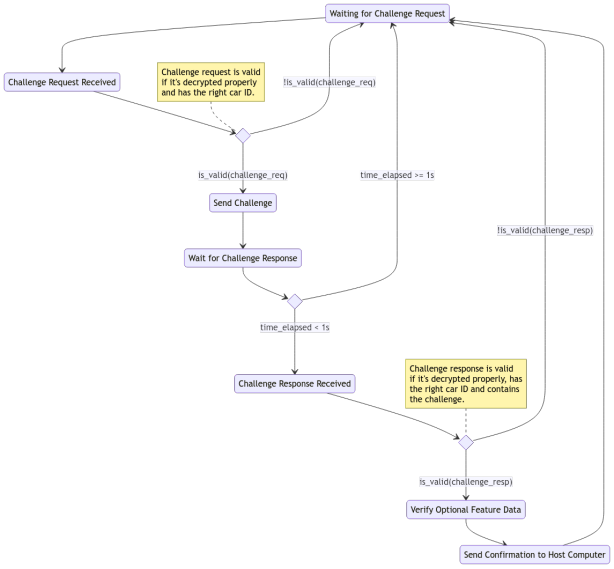

Fig. 8. State diagram of car communications

8.5 Security Analysis

In order to uphold the security requirements pertinent to our unlock sequence listed in subsection 8.2, our design incorporates security measures to protect against the potential attack vectors listed in subsection 8.3. These security measures are listed below.

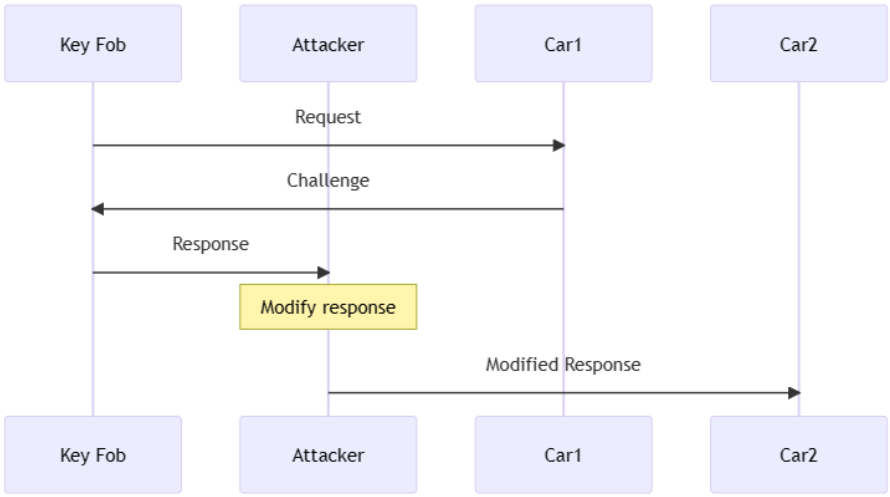

Fig. 9. Man-in-the-middle attack

Man-in-the-Middle:

We protect against man-in-the-middle attacks as shown in Figure 9 above because we use `XChaCha20-Poly1305` to encrypt all communications bi-directionally. This guarantees message confidentiality meaning that an attacker that listens in will not be able to read the contents of the message. This also ensures message authenticity meaning that an attacker will not be able to forge or modify any message. Therefore, a man-in-the-middle attack is not possible.

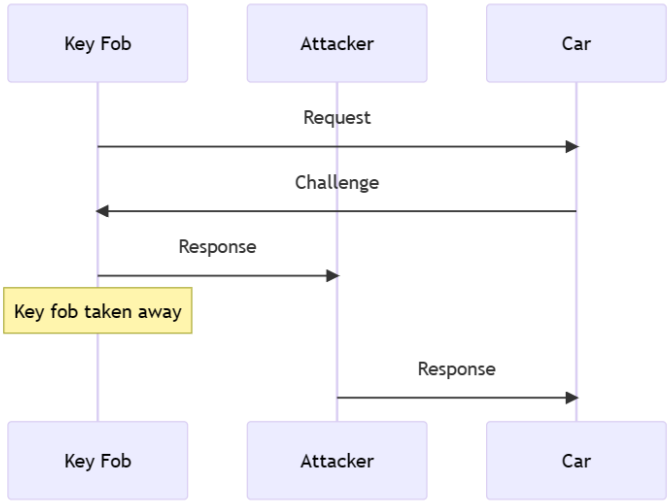

Fig. 10. Jamming and replay attack

Jamming/Replay:

We are protecting against attacks that replay the challenge by ensuring that every challenge sent by the car is unique and invalidated after getting a valid response. We do this by securely generating a 128-bit random number for every challenge. Even though the challenge response is the same as the challenge itself, the challenge is encrypted with Car Encryption Key k_{u2} when sent to the key fob and the challenge response is encrypted with Key Fob Encryption Key k_{u1} when sent to the car, ensuring that the challenge cannot be replayed back to unlock the car.

Our design is protected against replay attacks involving jamming as shown in Figure 10 above because we also invalidate challenges after not receiving a response within one second. This means that an attacker would have to use the obtained response within one second.

8.6 Design Considerations

Rolling Codes vs Challenge-Response:

Our design uses a challenge-response type of transaction to unlock the car instead of the traditional rolling code method. This is due to the fact that rolling codes are susceptible to attacks that a challenge-response is not. Rolling codes rely on a counter to introduce freshness into each message to prevent the replaying of messages.

A common attack that can be executed on rolling code systems is the RollJam attack. This is executed by jamming two consecutive unlock signals and replaying the first one to unlock the car for the user. The second message can then be used by the attacker at a later time. To protect against this, we would have to include a

timestamp in the rolling code. This timestamp is used to expire the message after a short period of time. However, this would be difficult to implement and synchronize correctly between a car and key fob.

In addition, rolling codes are challenging to implement correctly when the attacker has full physical access to the car and key fob. An attacker could listen to transmissions, and then reset the rolling code counters by re-flashing the car and key fob's firmware. This would allow the attacker to replay transmissions captured prior to the re-flashing, resulting in a successful unlock.

MACs vs `XChaCha20-Poly1305`:

Our design uses `XChaCha20-Poly1305` to provide message authenticity and confidentiality. We use this instead of a MAC because MACs solely provide message authenticity without confidentiality. Message confidentiality prevents an attacker from being able to view the data within our messages, so that a potential attack cannot be launched using knowledge of specific message fields, such as the nonce sent in the challenge.

9 RUN PAIR

9.1 Functional Requirements

To pair an unpaired key fob, there are three items the user must have: a paired key fob, an unpaired key fob, and a Pairing PIN. The host computer will establish a link to both key fobs over UART0. The user enters the Pairing PIN into the pairing host tool running on the host computer, which is sent to the paired key fob for validation. If the Pairing PIN is valid, the two key fobs will then begin communicating with each other over UART1. After pairing, both key fobs should be able to unlock the same car and pair other unpaired key fobs.

9.2 Security Requirements

Our pairing protocol should be designed to satisfy the following requirements.

(1) A user should not be able to pair an unpaired key fob to a car without both a key fob paired to that car as well as its Pairing PIN.
   - We need to ensure that pairing requires both a paired key fob and the corresponding Pairing PIN. It should not be possible to extract the PIN from a paired key fob as this would make knowing the PIN unnecessary.

(2) A user should no longer be able to unlock a car once they no longer have physical access to a paired key fob.
   - The pairing host tool requires the user to physically connect a paired key fob. Thus, if a user no longer has physical access to a paired key fob, they will not be able to use the pairing host tool to pair a new unpaired key fob, even when in possession of the Pairing PIN.

9.3 Potential Attack Vectors

(1) Man-In-The-Middle
   - When the key fobs are communicating with each other, we need to prevent data extraction via eavesdropping. This includes the extraction of the Pairing PIN or secrets being shared by the paired key fob. In addition, we need to prevent the replaying of a pairing exchange.

(2) Brute Force attack
   - The Pairing PIN is a six-digit hex string; there are only 16,777,216 possible combinations, meaning an automated brute force attack of the PIN is possible.

(3) Malicious Inputs

- When the paired key fob and the unpaired key fob are pairing, the unpaired key fob or host may be able to send hostile data or garbage input to the paired key fob. To prevent erroneous behavior, there should be verification done on incoming data to prevent security issues from untrusted inputs.

9.4 High Level Design

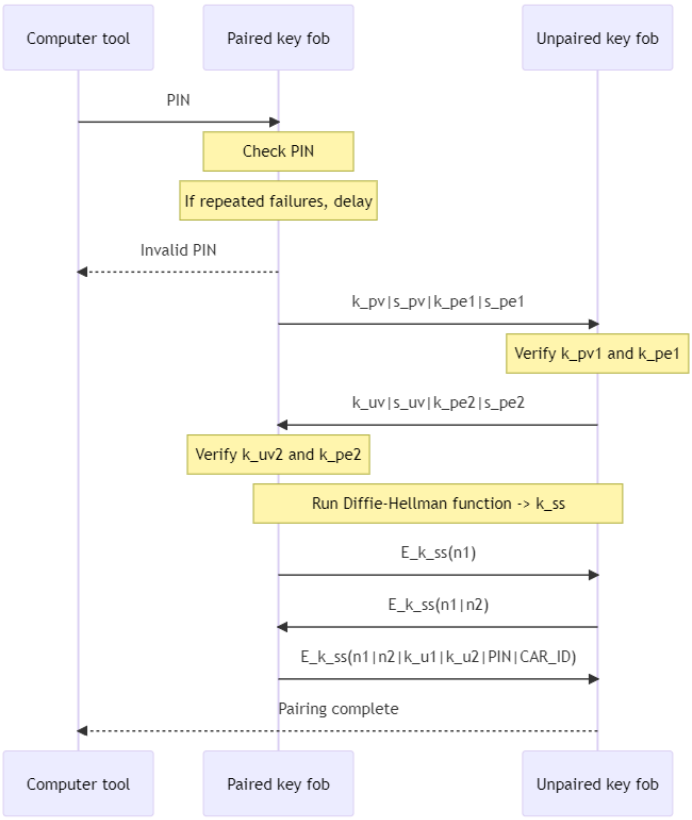

Fig. 11. Pairing Design. We assume that the first two messages between the paired key fob and the unpaired key fob are done in an authenticated channel, given the threat model and assumptions of the competition.

Our design for the pairing process can be seen in Figure 11. The pairing host tool first sends the PIN to the paired key fob so that it can check if the PIN is correct. If the PIN provided is incorrect, the paired key fob ends the current pairing attempt. Under normal circumstances, the paired key fob will ensure that just under one second passes before acknowledging that a PIN is correct. Additionally, to mitigate an attacker's ability to perform brute force attacks, after one incorrect attempt, the paired key fob will determine that it is under attack and increase this delay to just under five seconds.

We use various keys to establish a chain of trust that connects the manufacturer, individual key fobs, and pairing sessions. Figure 12 shows the relationships between these keys.

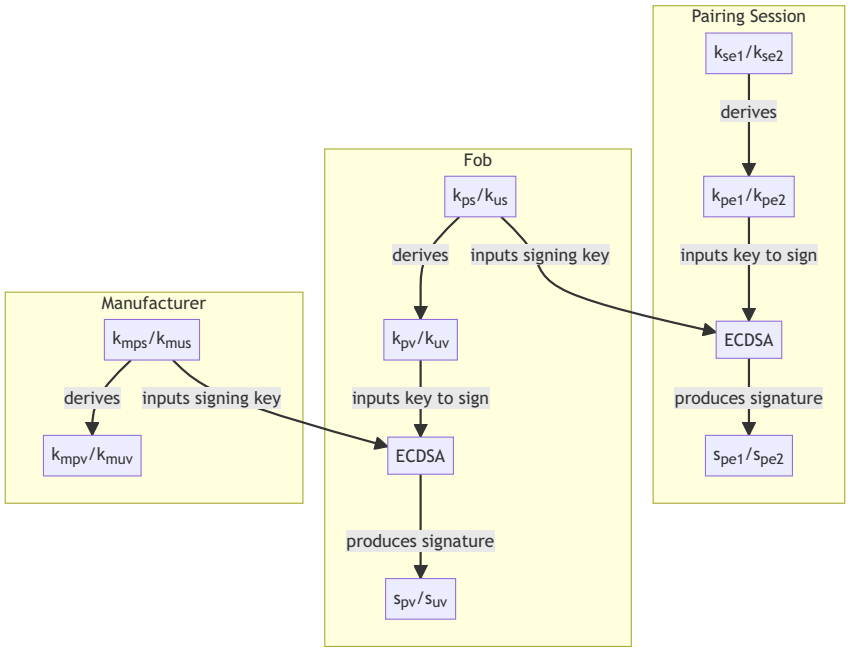

Fig. 12. Chain of Trust Diagram

The paired key fob and unpaired key fob verify each other's authenticity, and exchange keys to establish an ephemeral shared secret using the Elliptic-Curve Diffie-Hellman algorithm. The paired key fob generates an Ephemeral Secret Key k_{se1} and derives an Ephemeral Public Key k_{pe1} from it. The paired key fob then sends the unpaired key fob a message containing the Paired Fob Pairing Verifying Key k_{pv}, the Paired Fob Pairing Verifying Key Signature s_{pv} from the Pairing Manufacturer Paired Fob Signing Key k_{mps}, the Ephemeral Public Key k_{pe1}, and the Ephemeral Public Key Signature s_{pe1} from the Paired Fob Pairing Signing Key k_{ps}. The message structure is shown below in Figure 13.

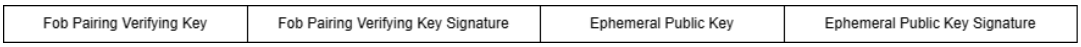

Fig. 13. Diffie-Hellman message

The unpaired key fob then verifies the paired key fob's Paired Fob Pairing Verifying Key k_{pv} using the Pairing Manufacturer Paired Fob Verifying Key k_{mpv}. It also verifies the Ephemeral Public Key k_{pe1} using the received Paired Fob Pairing Verifying Key k_{pv}. If successfully verified, the unpaired key fob generates an Ephemeral Secret Key k_{se2} and derives an Ephemeral Public Key k_{pe2} from it. The unpaired key fob sends back a response containing the Unpaired Fob Pairing Verifying Key k_{uv}, the Unpaired Fob Pairing Verifying Key Signature s_{uv} from the Pairing Manufacturer Unpaired Fob Signing Key k_{mus}, the Ephemeral Public Key k_{pe2}, and the Ephemeral Public Key Signature s_{pe2} using the Unpaired Fob Pairing Signing Key k_{us}. The message structure is the same as the message structure of the paired fob's message as seen in Figure 13. The unpaired key fob then runs the Elliptic-Curve Diffie-Hellman algorithm using its Ephemeral Private Key k_{se2} and the paired key fob's Ephemeral Public Key k_{pe1}.

After the paired key fob receives the response, the paired key fob verifies the signatures in the same way as the unpaired key fob but instead using the Pairing Manufacturer Unpaired Fob Verifying Key k_{muv} and the Unpaired Fob Pairing Verifying Key k_{uv}. If successfully verified, the paired key fob runs the Elliptic-Curve Diffie-Hellman algorithm using its Ephemeral Private Key k_{se1} and the unpaired key fob's Ephemeral Public Key k_{pe2}. This exchange leaves both the paired key fob and the unpaired key fob with an Ephemeral Shared Secret k_{ss}. This key is used to encrypt communications between the paired key fob and unpaired key fob for the rest of the pairing process.

For the next step of the pairing process, a pairing request is sent from the paired key fob to the unpaired key fob. The pairing request contains a randomly generated nonce n_1 used for freshness. The message structure is shown below in Figure 14.

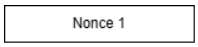

Fig. 14. Pairing request message

The unpaired key fob then sends the paired key fob a challenge. The challenge consists of the received nonce n_1 and a second randomly generated nonce n_2 as seen in Figure 15.

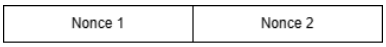

Fig. 15. Pairing challenge message

Upon receipt of the challenge, the paired key fob checks that the first nonce n_1 is the same as the one it sent in the pairing request. If it contains the correct nonce, the paired key fob responds to the challenge with a message containing both nonces and the key fob's pairing data. Both nonces are sent to ensure the freshness of the message. The sent pairing data contains the Key Fob Encryption Key k_{u1}, Car Encryption Key k_{u2}, the Pairing PIN, and the Car ID of the car the key fob is paired with. The message structure can be seen in Figure 16.

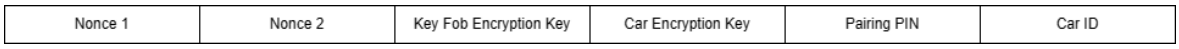

Fig. 16. Pairing challenge response message

The unpaired key fob then verifies the two nonces, and if successful, the key fob stores the unlock keys, Pairing PIN, and Car ID in the appropriate EEPROM fields. The newly paired key fob also erases the Unpaired Fob Pairing Signing Key k_{us} and Unpaired Fob Pairing Verifying Key Signature s_{uv} from its EEPROM. In addition, the newly paired key fob sets its pairing byte to `0x01010101` to indicate that it is now a paired key fob. The newly paired key fob then sends a confirmation to the host computer to notify the user that the pairing was successful. This completes the pairing process. See Figure 17 for a state diagram of the paired key fob while pairing and Figure 18 for a state diagram of the unpaired key fob while pairing.

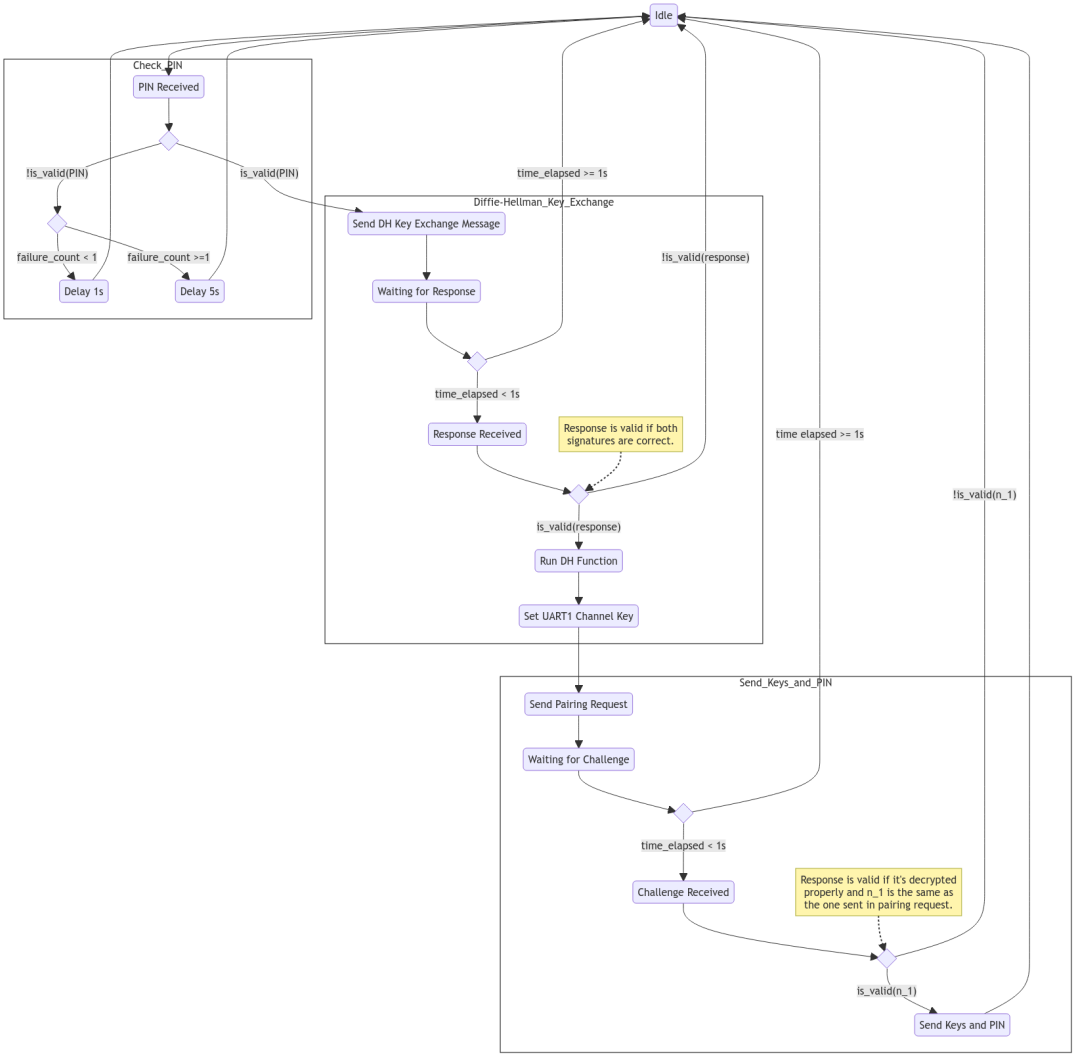

Fig. 17. State diagram of paired key fob communications

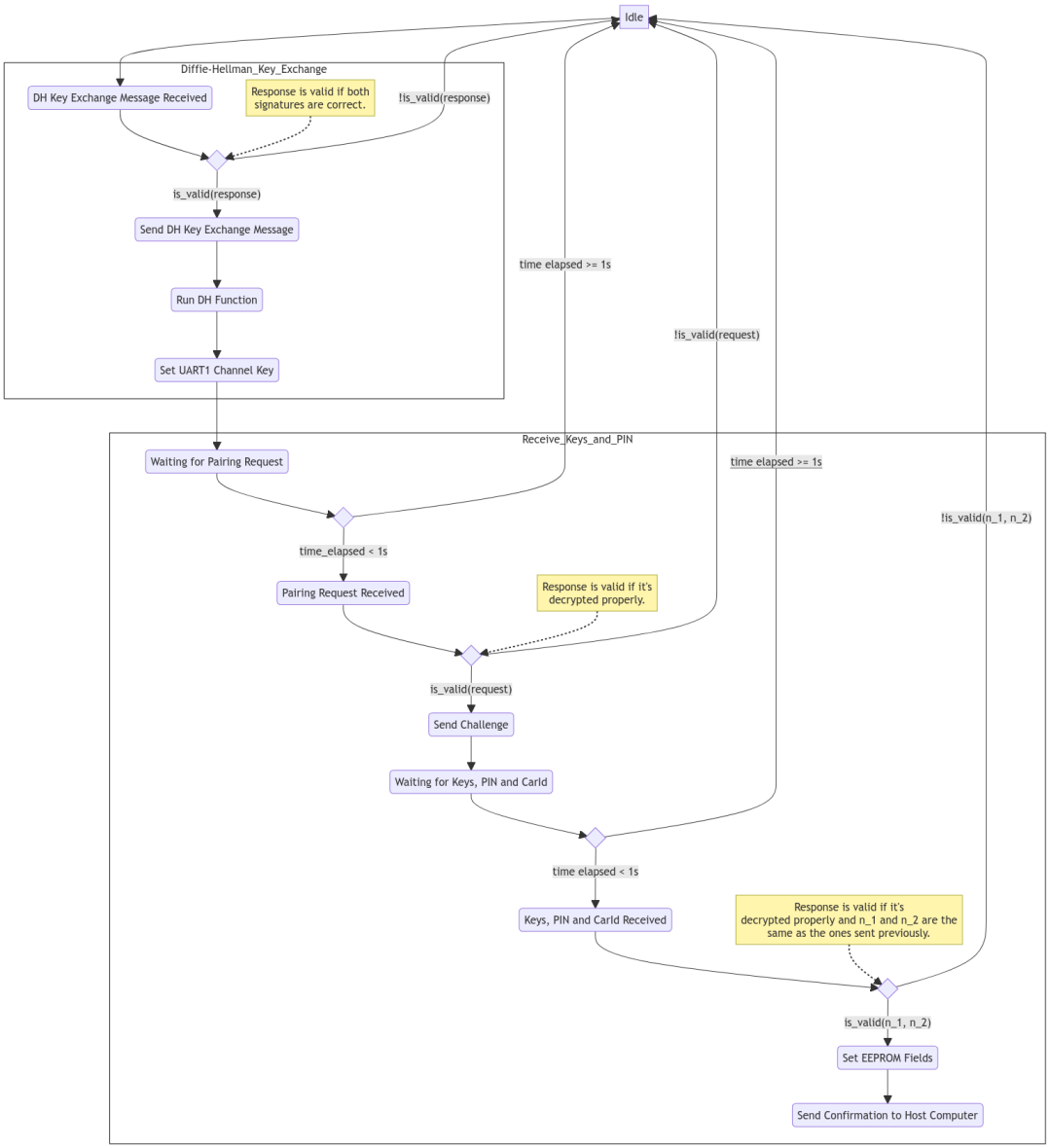

Fig. 18. State diagram of unpaired key fob communications

9.5 Security Analysis

Man-in-the-Middle:

An attacker would have access to communications sent over UART0 and UART1. We use an ephemeral Elliptic-Curve Diffie-Hellman key exchange to establish a shared secret between the unpaired and paired key fob. This shared secret is used as a symmetric key for `XChaCha20-Poly1305` authenticated encryption between the

unpaired and paired key fob to ensure that an attacker cannot view or modify messages being sent between them. We also prevent replay attacks by including nonces n_1 and n_2 to introduce freshness into each pairing exchange.

Brute Force:

The technical specifications require that this pairing process may take up to one second under normal conditions, or five seconds if we determine the system is under attack. To slow a potential brute force attack, we wait until one second has passed before reporting whether a PIN is valid or not. After one incorrect PIN has been entered, we consider the system to be under attack and increase this delay to five seconds. We also prevent an attacker from being able to have more than one PIN checked at a time. Assuming that one attempt takes five seconds, it would be possible to brute force a PIN in approximately 485 days on average ($\frac{1}{2} \cdot 16^6$ possible PINs $\cdot$ 5 seconds).

Malicious Inputs:

We perform input validation when parsing inputs. To ensure that an attacker cannot cause erroneous behavior by sending malicious input to the paired key fob, invalid inputs are ignored.

10 RUN PACKAGE FEATURE/ENABLE FEATURE

10.1 Functional Requirements

Only the manufacturer may configure a feature that will be sent to the user for use. This will be done by generating a Packaged Feature, which is a signed package containing the user's Car ID and the feature number. To enable a feature on a car, the user must have received a Packaged Feature from the manufacturer and have that Packaged Feature installed on their key fob. Enabling a feature on a key fob involves connecting the host computer to a key fob and uploading the package to it. When a user unlocks their car, the list of enabled features is transmitted to the car.

10.2 Security Requirements

To protect our revenue, we must meet the following security requirements:

(1) A user should not be able to enable a feature given a Packaged Feature from another user.
(2) A user should only be able to enable a feature given they have a corresponding Packaged Feature.

10.3 Unauthorized Use Cases

Our attack vectors include the following unauthorized use cases:

(1) User A purchases Packaged Feature #2 and gives it to others to enable Feature #2 on their cars.
(2) User A purchases Packaged Feature #1 and installs it onto their key fob(s). User A then modifies Packaged Feature #1 and installs the modified Packaged Feature to enable Feature #2.

10.4 High Level Design

Digital Rights Management systems running on untrusted hardware or software are almost always broken, so our design ensures that only genuine car firmware can use the Packaged Feature. During the Package Feature step, we concatenate the Car ID and Feature Number and sign the result with ECDSA using the Feature Signing Key k_{fs}. The Packaged Feature has the structure shown in Figure 19.

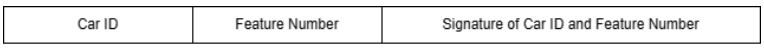

Fig. 19. Packaged Feature

During the Enable Feature step, the Packaged Feature is sent to the paired key fob. When the paired key fob receives the Packaged Feature, it verifies the signature using the Feature Verifying Key k_{fv}. The Car ID of the Packaged Feature is also verified. If the verification fails, a negative acknowledgment is sent to the Host Computer. Otherwise, a positive acknowledgment is sent after saving the feature to the appropriate field in the EEPROM. Every time the user unlocks their car, the key fob sends the stored Packaged Feature(s) to the car as part of the optional feature data in the unlock sequence's challenge response. See Figure 6 for how the optional feature data is incorporated into the challenge response. The car then verifies the signature of the Packaged Feature(s) using the Feature Verifying Key k_{fv} stored in its EEPROM and compares the Car ID of the Packaged Feature(s) with its own. If the verification pass, the car retrieves the appropriate feature message(s) and sends it to the host computer with the unlock message. A figure displaying the feature-enabling process can be seen in Figure 20.

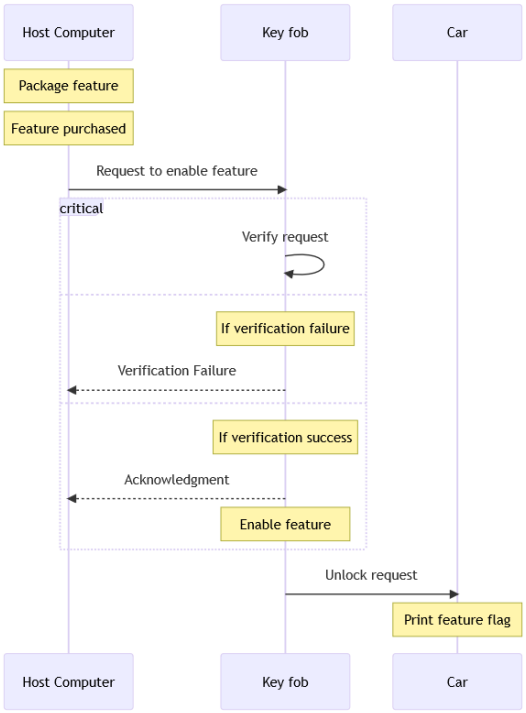

Fig. 20. Packaging and Enabling Features

10.5 Security Analysis

Unauthorized Use Case #1:

Our firmware runs a check against the Car ID in the Packaged Feature, and its own Car ID so that the Packaged Feature will only work with the car it was originally packaged for. This Car ID cannot be tampered with due to our usage of ECDSA which guarantees message authenticity, preventing users from sharing Packaged Features with others, which upholds Security Requirement #1. See Figure 21 below for a diagram of the unauthorized use case.

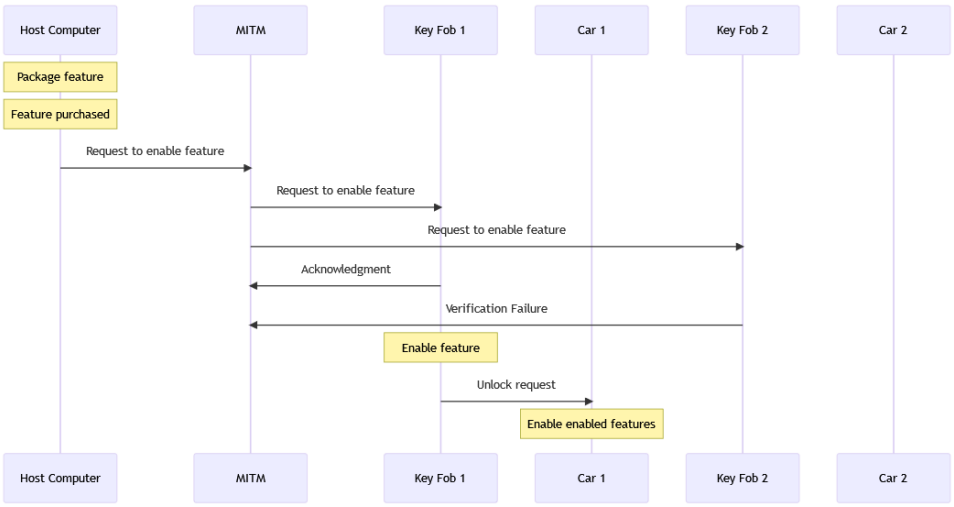

Fig. 21. Unauthorized Use Case #1

Unauthorized Use Case #2:

Our firmware verifies the Packaged Feature using the Feature Verifying Key k_{fv}, which makes changing the feature number impossible because as stated in Unauthorized Use Case #1, ECDSA guarantees message authenticity, preventing a user from modifying the payload, upholding Security Requirement #2 as well. See Figure 22 below for a diagram of the unauthorized use case.

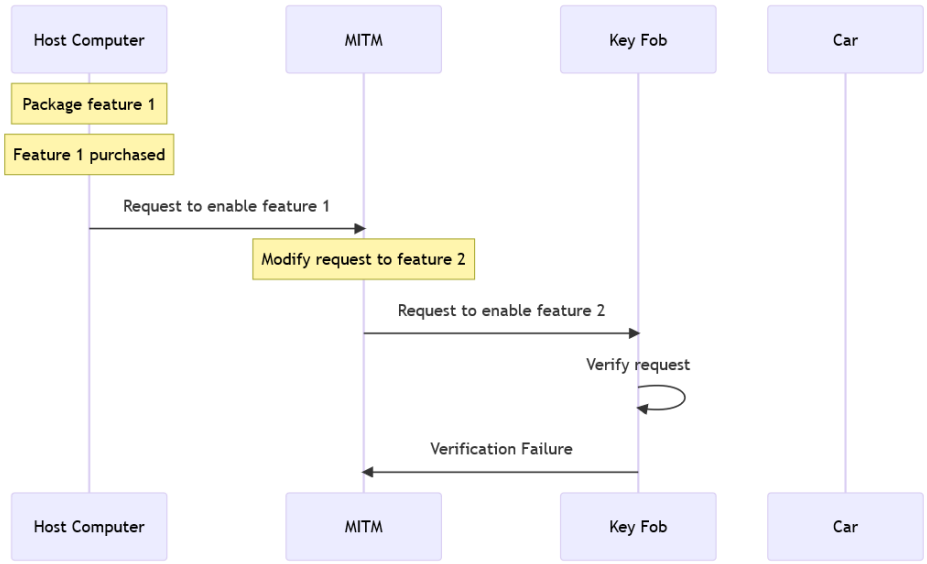

Fig. 22. Unauthorized Use Case #2