LED Signaling
We use a slightly modified color scheme for the RGB LED:

- Car: yellow before first unlock -> green after first unlock
- Unpaired fob: cyan
- Paired fob: white
- Unrecoverable error: red
- Operation failed: blinking purple for up to 5 seconds after a failed interaction

Device Configuration Modifications

- We increased the stack size to accommodate the larger stack sizes of the functions we are calling.
- We enabled the board's PLL to increase the internal clock frequency from the 16 MHz of the on-board oscillator to the maximum of 80 MHz.
- We enabled the MPU and configured it to prevent any code execution from SRAM.

Randomness Generation
At build time, we generate a randomness seed and store it in EEPROM. We then use the ChaCha12 RNG to generate randomness, immediately overwriting the EEPROM seed with the RNG output to ensure that the random output is different for each boot. We also start a periodic timer on boot and reseed the RNG, mixing in the current timer value, after a user interaction with the device. This ensures that devices with the same random seed will have divergent randomness. We do not use the ADC as an entropy source because an attacker could easily tie the analog input pins to a known value.

Data Storage
We store the following data in the car EEPROM:

- `0x00-0x1f: HMAC-SHA256("Server Key", argon2id(car_start_auth, car_start_salt))`
- `0x20-0x3f: SHA256(HMAC-SHA256("Client Key", argon2id(car_start_auth, car_start_salt))`
- `0x40-0x4f: car_start_salt`
- `0x50-0x7f: 0xFF padding`
- `0x80-0x9f: RNG seed`

The two values using HMAC-SHA256 are used in SCRAM-SHA256 and are precomputed as part of the build process.

We store the following data in the fob EEPROM:

- `0x00-0x03: car ID (0xFFFFFFFF if unpaired)`
- `0x04-0x23: unpaired_auth_base (if unpaired) | HMAC-SHA256("Server Key", argon2id(paring_auth, pairing_auth_salt)) (if paired)`
- `0x24-0x43: SHA256(HMAC-SHA256("Client Key", argon2id(pairing_auth, pairing_auth_salt)) (if paired)`

- `0x44-0x53: pairing_auth_salt (if paired)`
- `0x54-0x73: car_start_base (if unpaired) | car_start_auth (if paired)`
- `0x74-0x80: 0xFF padding`
- `0x80-0x9f: RNG seed`

Protocol changes
Both pairing new fobs and starting the car involve mutual authentication between the devices involved, so we implement this by having both sides independently compute a shared secret as the input password to a modified version of the SCRAM-SHA256 authentication protocol. After both sides prove to each other that they possess the same secret, they take their respective actions, authenticating the subsequent messages in the session with HMAC and a key derived from part of the authentication state. In addition, we ensure that all of our messages have an expected fixed size, ensuring that there is no lack of a terminator symbol that might cause buffer overflow errors otherwise.

SCRAM-SHA256 modifications
We refer readers to RFC 5802 for the original definition of SCRAM-SHA1. Besides of using SHA256 as the hash function, we make the following additional changes to the protocol:
- We use argon2id for deriving a salted hashed password instead of PBKDF2, tuning memory consumption and rounds independently. We also use hardcoded parameters for the password hashing instead of exchanging them during authentication. Our goal with the parameter tuning is to spend most of the computation time here and to use the majority of the device's memory.
- We omit negotiation of channel binding support, instead including the car ID in the header when performing fob pairing.
- Because all the fields we are transmitting are fixed-size, we do not need field delineation or base64 encoding for message transmission.
- Channel binding for subsequent communication is accomplished by extracting a variant of the standard protocol_key of HMAC-SHA256(SHA256(client_key), "PARED session key" || client_key || auth) (where the difference is in the constant string) and using it as an HMAC key for future messages in that session. An attacker would need the client_key to compute this value, which would not be available to attackers.
- The client confirms authentication success by sending an empty MAC-protected success message to the server, instead of signaling authentication failure closing the connection as would be done with TCP in a non-embedded system. (The server would have confirmed authentication success previously by sending the ServerSignature message, so the other direction is unnecessary.)

Car starting:
The car_start_auth secret authenticated between the car and the paired fob is computed as the HMAC-SHA256 of the car ID using a global secret car_start_base. (While we use argon2id as a password hash function, we still want to apply a one-way function to car_start_base to ensure

that it cannot be recovered from the paired fob.) car_start_base is only present on unpaired fobs, as both the car and its paired fobs can just store (derivatives of) car_start_auth directly. Having car_start_auth depend on the car ID ensures that the authenticated secret is different from each car and that extraction of the secret from one car would not assist attacks on other cars.

In the SCRAM framing, the fob is the client that initiates the authentication, and the car is the server that can perform the car start action after authenticating the fob and receiving the list of features stored on the fob, which is protected by an HMAC using the protocol key as the key. By sending the feature list, the client (fob) affirms to the server (car) that it has authenticated the car.

Fob pairing:
The pairing_auth secret authenticated between the paired fob and the unpaired fob is computed as the HMAC-SHA256 of the car ID and the pairing PIN using a global secret unpaired_auth_base. unpaired_auth_base is only present on unpaired fobs, as both the car and its paired fobs can just store (derivatives of) pairing_auth directly. Depending on the car ID ensures that the authenticated secret is different from each car and that extraction of the secret from one car would not assist attacks on other cars, while depending on the pairing PIN ensures that both fobs receive the same correct pairing pin.

In the SCRAM framing, the unpaired fob is the client that initiates the authentication, and the paired fob is the server, so that the unpaired fob initiates communication. The paired fob can send its car ID first to the unpaired fob, with the unpaired fob transmitting it back as part of the auth variable. (If an attacker corrupts the car ID that is sent, the fobs will have different pairing_auth variables and SCRAM will fail, so we do not need to add integrity protections to this first message.) After authentication is confirmed, the newly paired fob commits its new paired state to the EEPROM.

Feature enabling:
We secure our feature package with a p256 ECDSA digital signature, computing the signature over the concatenation of the car ID and the feature ID. We hardcode the public key as part of the binary and store signed features in the fob's flash. The fob receives the signed feature and verifies both the car ID and the signature before storing it. Then, the fob transmits the unmodified signed package to the car during a car start sequence, and the car verifies the signatures and discards any features with invalid signatures. (The actual feature transmission during car start omits the car ID, since both sides would have confirmed possession of the same car ID by that point, but the signature's dependence on the car ID prevents features from being transposed from one car to another.)

Changes to host tools
For fob pairing and car unlocking, we only need to include a success/failure code so that the host tools can report on a failed pairing or car start. Unlike in the reference design, the fob pairing tool transmits the pin to the unpaired fob for computing the pairing_auth variable.

For feature enabling, the feature packaging tool will sign the feature with its private key and send the feature along with its signature to the fob. The feature enabling tool only needs to transmit the packaged feature to the fob.

As noted above, we modified the communication protocols to use fixed-length messages.

Threat Model
We assume that all communications can be monitored and tampered with by an external adversary. However, we assume that an adversary is not going to have access to the unprotected binary or EEPROM image (except for the unprotected binaries of car 0).