

2023 MITRE eCTF Design Document

As of: 2023-03-06



Team School

Plaid Parliament of Pwning
Carnegie Mellon University

Team Members

Eliana Cohen
Aditya Desai
Nandan Desai
Neha Gautam
Henry Howland
Ray Huang
Harrison Leinweber (Lead)

Ethan Oh
Palash Oswal
Anish Singhani
Carson Swoveland
Madeline Tasker-Fernandes
Suma Thota
Hanjie Wu

Advisors

Anthony Rowe
Patrick Tague

Maverick Woo

Table of Contents

1. Documentation Notes	3
1.1. Principals	3
1.2. Keys	3
1.3. Other Symbols	4
2. Build Steps	5
2.1. Build Environment	5
2.2. Build Host Tools	5
2.3. Build Deployment	5
2.4. Build Car and Paired Fob	5
2.4.1. Build Car	5
2.4.1.1. Security Requirements	6
2.4.2. Build Paired Fob	6
2.4.2.1. Security Requirements	6
2.5. Build Unpaired Fob	7
2.5.1. Security Requirements	7
2.6. Load Devices	9
2.7. Package Feature	9
2.7.1. Security Requirements	9
2.8. Enable Feature	10
2.8.1. Security Requirements	10
2.9. Pair Fob	12
2.9.1. Security Requirements	12
2.9.1.1. Notes	13
2.10. Unlock/Start Car	13
2.10.1. Security Requirements	13
3. Security Design	15
3.1. Cryptography	15
3.2. Cryptographic Storage (EEPROM)	15
3.3. EEPROM Encrypt	15
3.4. TRNG Design	16
4. Defenses	18
4.1. Pairing PIN Storage	18

4.2. Compiler Countermeasures	18
4.3. Fault Injection Defenses	18
4.4. Protections Against Bruteforce	19

1. Documentation Notes

This document informs the reader of the functional and cryptographic design implemented by the Plaid Parliament of Pwning in order to secure the Protected Automotive Remote Entry Device (PARED) in accordance with the design and security standards specified in the competition documentation.

Throughout this document, we aim to provide notation that is consistent and clear. Objects displayed in our protocol diagrams should be interpreted as bitstrings. The double pipe symbol, \parallel , represents concatenation, and braces, $\{\}$, represent encryption using the key located in subscript on the outer right side. For example:

$$\{A \parallel B\}_{K_x}$$

represents the bitstring A concatenated to bitstring B, then encrypted with K_x .

We begin by introducing the cryptographic symbols and abbreviations used.

1.1. Principals

C: Car representing the specific build during the attack phase (an integer from 0 to 5, inclusive).

PF_C : Paired Fob built for Car C in the current deployment.

UPF: Unpaired Fob built in the current deployment.

D: Deployment environment in which C, PF_C , and UPF are built prior to being released.

1.2. Keys

Ks_D : Deployment signature public key, so PF_C and C can verify features.

Ks_D^{-1} : Deployment signature private key, to allow for authentication of packaged features.

Ke_{UPF} : Unpaired fob encryption public key, so PF_C can encrypt messages to UPF.

Ke_{UPF}^{-1} : Unpaired fob encryption private key, allows UPF to decrypt messages sent to it.

Ke_C : Encryption public key for Car C so PF_C can encrypt the message to C during the unlocking process.

Ke_C^{-1} : Encryption private key for Car C so C can decrypt PF_C 's unlock message during the unlocking process.

Ks_C : Signature public key for Car C so PF_C can verify the nonce's authenticity during the unlocking process.

Ks_C^{-1} : Signature private key for Car C so C can sign the nonce during the unlocking process.

Ke_{PF_C} : Encryption public key for PF_C , so that C can encrypt messages to PF_C during unlock.

$Ke_{PF_C}^{-1}$: Encryption private key for PF_C for decrypting messages sent by C during unlock.

Ks_{PF_C} : Signature public key for PF_C C can verify messages from PF_C during unlock.

$Ks_{PF_C}^{-1}$: Signature private key for PF_C for signing messages sent to C during unlock.

PIN_C : Pairing pin for Car C

K_H : Key used for hashing the pairing pin

1.3. Other Symbols

HT: Host Tools

MHC: Manufacturers Host Computer (runs in a secure environment, has access to Host Secrets)

HC: Host Computer (runs in an insecure environment)

Disk: location where Host Secrets and packaged features are stored

$H(msg, K)$: Hash of msg with key K

F: Feature Number (an integer from 1 to 3, inclusive)

A: Active Feature Bitvector (0 to 7, inclusive)

2. Build Steps

2.1. Build Environment

The build environment step initializes a Docker container, installs all updated packages, and performs environmental configurations that are required to run future build steps and generate secrets. We verified that the tools packaged within the container have no known vulnerabilities.

2.2. Build Host Tools

Host tools are written in Python3. Data size will be verified for every message transmitted and received. We ensure that the secrets used by the host tools will not be leaked within the binaries. Types and sizes of all the parameters provided to the host tools will be checked.

2.3. Build Deployment

Build Deployment will generate the following information, to be stored in Host Secrets:

- Public/Private encryption key pair for the unpaired fob

$$Ke_{UPF}, Ke_{UPF}^{-1}$$

- Public/Private signing key pair for deployment (to be used by Host Tools)

$$Ks_D, Ks_D^{-1}$$

2.4. Build Car and Paired Fob

In this step, the secure host computer running the build tool generates public/private key pairs for the car and paired fob. The keys are built into the car to allow for secure communication and authentication with the paired fob.

This step produces four pairs of public/private keys: one pair for encryption and authentication for each device:

$$Ke_C, Ke_C^{-1}, Ks_C, Ks_C^{-1}$$
$$Ke_{PF_C}, Ke_{PF_C}^{-1}, Ks_{PF_C}, Ks_{PF_C}^{-1}$$

Produced secrets are written to Host Secrets. Ke_C^{-1} , Ks_C^{-1} , Ke_{PF_C} , and Ks_{PF_C} are used to build the car, and Ke_C , Ks_C , $Ke_{PF_C}^{-1}$, and $Ks_{PF_C}^{-1}$ are built into the paired fob for C in the next step. The build tool also reads Ks_D from Host Secrets.

2.4.1. Build Car

The following secrets are used to build the car firmware and EEPROM:

$$Ke_C^{-1} || Ks_C^{-1} || Ke_{PFC} || Ks_{PFC} || Ks_D$$

Additionally, the car's EEPROM contains the car ID, the nonce seed, and random padding used for RNG. We randomly generate the nonce seed and random padding in this step.

2.4.1.1. Security Requirements

SR1: In the build step, the car's EEPROM is loaded with the secrets required to be able to securely communicate with a paired fob. These secrets are only stored in EEPROM and the manufacturer's secure environment.

SR2: Not applicable

SR3: The secrets loaded into the car's EEPROM in the build process allow for secure communication so an attacker cannot use their observations to unlock the car in the future.

SR4: Not applicable

SR5: In this step, the car is given the Deployment signature public key, which is used to verify that the enabled features were packaged by the manufacturer.

SR6: The deployment key is used to verify that the features were packaged for a particular car.

2.4.2. Build Paired Fob

In this step, the secure manufacturer computer running the build tool has access to the Host Secrets file and uses it to build the public and private keys into the paired fob.

To build the paired fob, the build tool reads the following from Host Secrets:

$$Ke_{PFC}^{-1} || Ks_{PFC}^{-1} || Ke_C || Ks_C || Ks_D || Ke_{UPF}$$

Using the secrets and the pairing pin input in this step, the tool builds the firmware and EEPROM containing the following information:

$$C || Ke_{PF}^{-1} || Ks_{PF}^{-1} || Ke_C || Ks_C || Ks_D || Ke_{UPF} || H(PIN_C, K_H) || K_H$$

Additionally, the EEPROM contains information about whether it's a paired fob or not, and any active features. A portion of the remainder of the EEPROM contains random padding used for RNG. We randomly generate the key used for hashing and the random padding in this step.

2.4.2.1. Security Requirements

SR1: Only a fob that is paired with a particular car will have been built with the keys needed to communicate during the unlock step.

SR2: Not applicable

SR3: Not applicable

SR4: The pairing pin is hashed and securely stored in EEPROM so it cannot be read.

SR5: The paired fob is given the Deployment signature public key, which is used to verify that the enabled features were packaged by the manufacturer.

SR6: The Deployment key is used to verify that the features were packaged for a particular car.

2.5. Build Unpaired Fob

The unpaired fob is built in a secure environment, so encryption is not needed during the build step. The keys are used for secure communication between the unpaired fob and a paired fob while the Pair Fob tool is being run.

To build the unpaired fob, the build tool will read the following from Host Secrets:

$$Ke_{UPF}^{-1}$$

This will be used in the build process, which will produce the firmware and EEPROM for the unpaired fob. Additionally, the EEPROM contains information about whether it is a paired fob or not, and random padding used for RNG. We randomly generate the padding in this step.

2.5.1. Security Requirements

SR1: An unpaired fob is not loaded with the keys required to communicate with the car, so will not be able to successfully unlock any car.

SR2: Not applicable

SR3: Not applicable

SR4: Without going through the proper pairing process, an unpaired fob does not have the keys required to unlock a car.

SR5: Not applicable

SR6: Not applicable

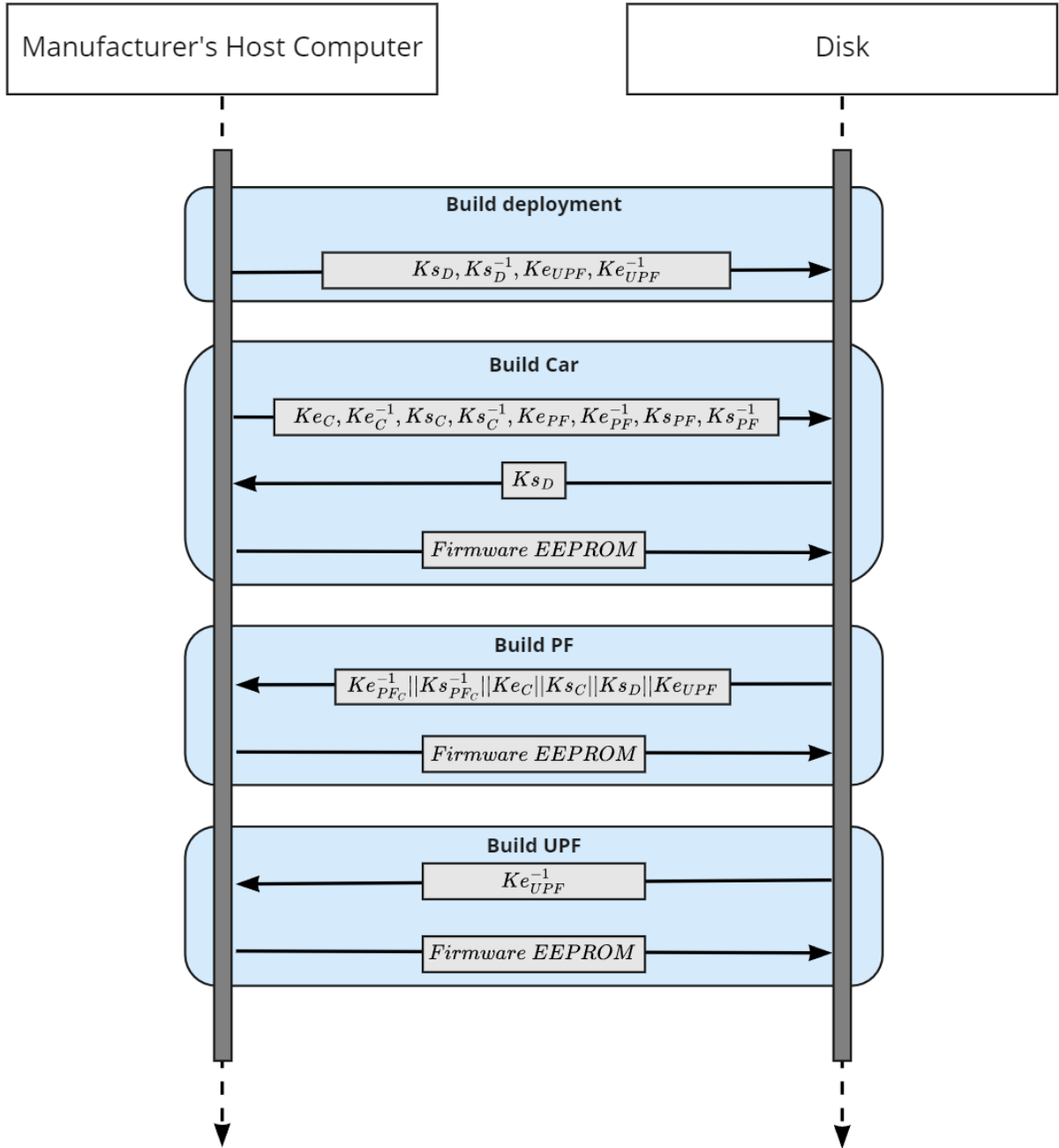


Figure 1: Build phase

2.6. Load Devices

Tools for this step are provided by the organizers and are not a permitted attack vector.

2.7. Package Feature

This function is designed to be run in a secure environment by only the manufacturer, so encrypted communication is not needed. When the Package Feature tool is run, the tool will take C and F as inputs. The tool will read K_D^{-1} from the Host Secrets file, and use K_D^{-1} to sign the package. This packaged feature will then be stored until it is used by the Enable Feature tool. The packaged feature is the following:

$$C||F||\{C||F\}_{K_D^{-1}}$$

2.7.1. Security Requirements

SR1: Not applicable. The package feature functionality does not influence the pairing status between fobs and cars.

SR2: Not applicable. The package feature functionality does not influence the pairing status between fobs and cars.

SR3: Not applicable. The package feature functionality does not influence the pairing status between fobs and cars.

SR4: Not applicable. The package feature functionality does not influence the pairing status between fobs and cars.

SR5: The private deployment key (K_D^{-1}) is known only by the trusted manufacturer computer which can access Host Secrets. The features are packaged with the car ID and signed with the deployment private key. Since the attacker does not have access to this key, this signature prevents an attacker from tampering with the packaged features and escalating privileges on a feature-limited fob.

SR6: Not applicable. The package feature functionality does not influence this security requirement

2.8. Enable Feature

The Enable Feature function instructs the host tool of the current deployment to authorize fobs with specific features if packaged through the Package Feature process.

- 1) HT -> HC: request features
- 2) HC -> HT: $C||F||\{C||F\}_{Ks_D^{-1}}$
- 3) HT -> PF: "e" (the letter e in ASCII)
- 4) HT -> PF: $C||F||\{C||F\}_{Ks_D^{-1}}$

2.8.1. Security Requirements

SR1: Not applicable. The enable feature functionality does not influence the pairing status between fobs and cars.

SR2: Not applicable. The enable feature functionality does not influence the pairing status between fobs and cars.

SR3: Not applicable. The enable feature functionality does not influence the pairing status between fobs and cars.

SR4: Not applicable. The enable feature functionality does not influence the pairing status between fobs and cars.

SR5: The signature on the packaged features prevents tampering since K_D^{-1} is only available on the manufacturer's secure computer. The paired fob will use K_D to verify the signature, and will not accept a packaged feature if it cannot verify that it was packaged by the manufacturer in the particular deployment.

SR6: The paired fob will have access to the deployment public key, so will be able to verify the signature of the packaged features. The paired fob will verify that C matches the C stored on the paired fob. If C does not match, the paired fob will not accept the packaged feature. This prevents someone from enabling a feature that was packaged for a different car.

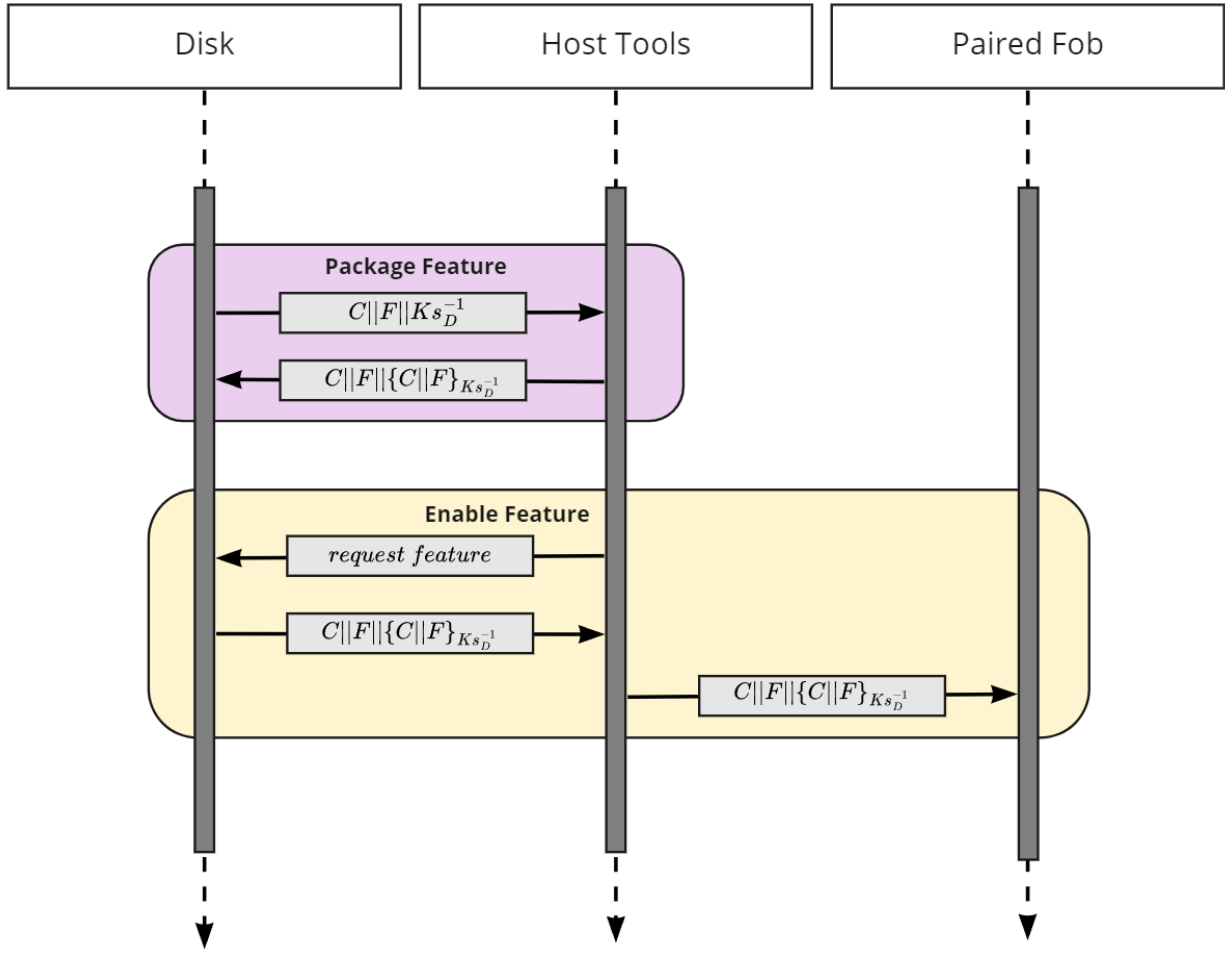


Figure 2: Package and enable feature tools

2.9. Pair Fob

- 1) HT -> PF: "p" (the letter p in ASCII)
- 2) HT -> UPF: "p"
- 3) HT -> PF: PIN_C
- 4) PF -> UPF:

$$\{C||Ke_C||Ks_C||Ks_D||Ke_{UPF}||Ke_{PF}^{-1}||Ks_{PF}^{-1}||H(PIN_C, K_H)||K_H\}_{Ke_{UPF}}$$
- 5) UPF -> HT: "Paired"

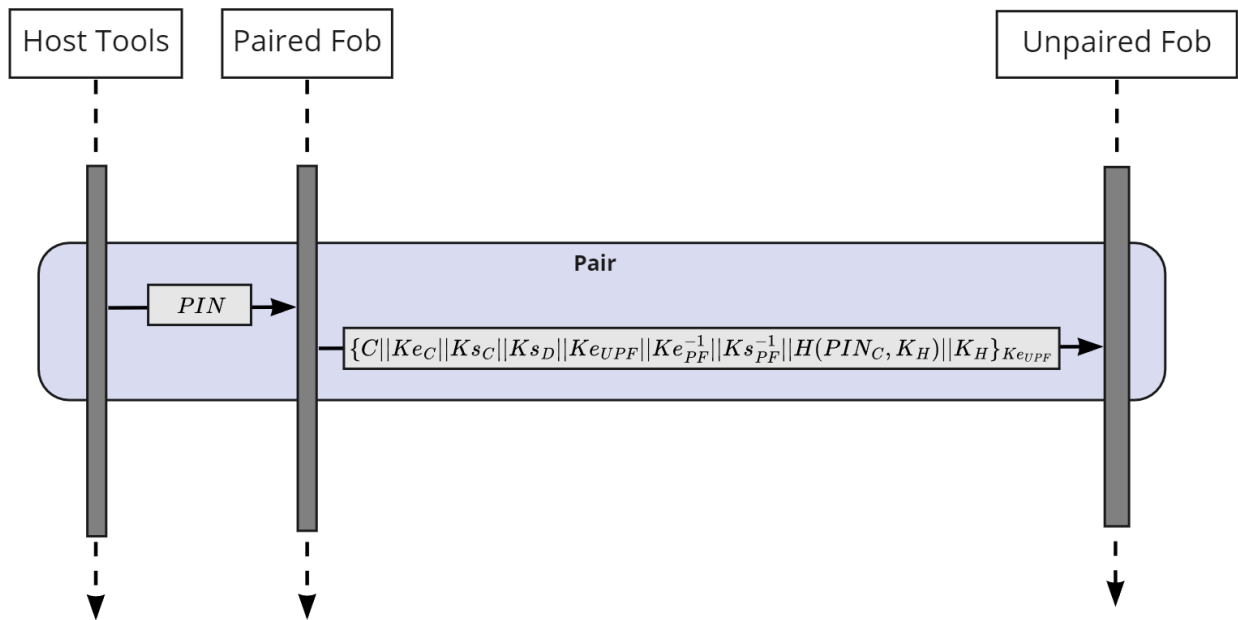


Figure 3: Pair fob mechanism

2.9.1. Security Requirements

SR1: To unlock a car, attackers would need to violate SR4.

SR2: The secrets will be securely stored in the EEPROM, so they cannot be copied from the paired fob to an unpaired fob, except via the pairing process. An attacker with temporary access will not have access to the PIN, so they will be unable to create a working copy.

SR3: Not applicable.

SR4: The pairing pin is required to pair a new unpaired fob. The paired fob will verify that the pin sent from Host Tools matches the pin stored on the paired fob. The paired fob is required since the secrets needed to communicate with the car are transferred in this protocol. If the pin

does not match, the paired fob will not send anything to the unpaired fob. The exact details of this comparison process are listed under the “Pairing PIN Storage” section.

SR5: Not applicable. Features are verified independently of this step.

SR6: Not applicable. Features are verified independently of this step.

2.9.1.1. Notes

The unpaired fob does not authenticate the data coming from the paired fob. This is still secure, because ‘pairing’ an unpaired fob with attacker-controlled data only allows the fob to unlock cars that were already using attacker-controlled keys in the first place.

The paired fob also does not authenticate the unpaired fob. There is no attack scenario in which authentication would provide additional security:

Car 1: Does not apply. There is no paired fob.

Car 2: Does not apply. The attacker is already given a legitimate unpaired fob.

Car 3: Does not apply. The pairing process is never available.

Car 4: Does not apply. There is no paired fob.

Car 5: Does not apply. The attacker is already given a legitimate unpaired fob.

2.10. Unlock/Start Car

- 1) PF -> C: unlock
- 2) C -> PF: challenge $\{nonce || \{nonce\}_{Ks_C^{-1}}\}_{Ke_{PFC}}$
- 3) PF -> C: response

$$\{nonce || A || C || F || \{C || F\}_{Ks_D^{-1}} || C || F || \{C || F\}_{Ks_D^{-1}} || C || F || \{C || F\}_{Ks_D^{-1}} || \{nonce || A || C || F || \{C || F\}_{Ks_D^{-1}} || C || F || \{C || F\}_{Ks_D^{-1}} || C || F || \{C || F\}_{Ks_D^{-1}}\}_{Ks_{PFC}^{-1}}\}_{Ke_C}$$

2.10.1. Security Requirements

We assume that the attacker will not be able to extract the keys out of the car, the paired fob and the unpaired fob.

SR1: When a car and the corresponding paired fob are built securely by the manufacturer they are given the keys needed to be able to securely communicate. These secrets will be stored securely on the device. An unauthentic fob would not be able to read (because it will not be able to decrypt) and respond to the challenge message from the car, so would not be able to unlock it.

SR2: The paired fob signs a response message to the challenge using its own private key. If the car can not verify it then it will reject the message sent to it when unlocking the car. This means that when the fob is not present, the attacker will not be able to forge any messages and hence this requirement is met.

SR3: The protocol for unlocking/starting the car has two mechanisms for preventing future attacks after observing communications. By encrypting the communications, the attacker cannot learn the details of the message, since they will not have the keys needed to encrypt/decrypt it. A random nonce will also prevent replay attacks since an attacker cannot predict the value even after observing multiple messages. When unlocking the car, the signature ensures the car and paired fob will not communicate with an attacker who may be attempting to generate multiple nonces and/or responses to later use for unlocking the car.

SR4: All messages between the car and the paired fob are signed and encrypted using the sender's private key and the receiver's public key respectively. This means that an unpaired fob (which does not have the relevant keys) will not be able to unlock a car by itself. Moreover, to get those keys, the attacker needs to have the correct pin along with a paired fob (that will transfer the keys). Thereby this requirement is met.

SR5: Because the (car id, feature number) pair is signed using the private key of the deployment, a car owner will not be able to sign it (since they do not have the private key) and hence not add new features to the car. This is verified at the car before unlocking the car.

SR6: The car id is sent along with the feature number. This is also signed using the private key of the deployment. The car verifies that it is actually the car that the feature was intended for and hence this requirement is met.

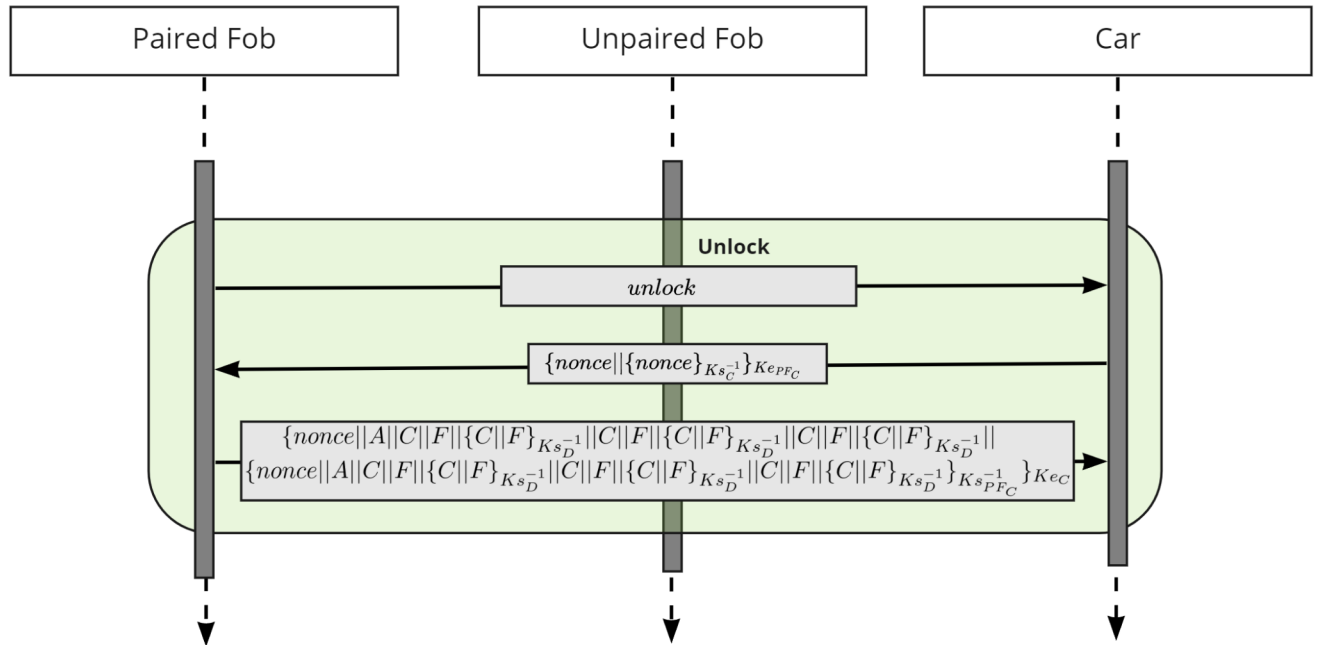


Figure 4: Car unlock mechanism

3. Security Design

3.1. Cryptography

For signatures in the secure communication protocols, we use ED25519. This public key signature scheme uses 32-byte keys and is fast at signing and verifying messages.

For encryption, we use Curve25519, which is an Elliptic Curve cryptosystem. Like the ED25519 signature scheme, this encryption system will have 32-byte keys and functions at a high speed.

3.2. Cryptographic Storage (EEPROM)

The cryptographic data store (CDS), situated in the EEPROM of the Tiva board, holds all of PARED's private keys and hashes generated at compile time which are used in authentication and verification mechanisms.

To elaborate further on the CDS's architecture, the EEPROM on this board is 1792B in size. Teams in the eCTF are unable to use the last 256 bytes as that is reserved for the organizers. The remaining space in the EEPROM will be used to store the keys and hashes.

3.3. EEPROM Encrypt

To prevent an attacker from reading EEPROM memory, keys stored in the EEPROM of all devices are encrypted until a key is requested by the firmware. The encryption is done using a one-time pad which is XORed with the bytes in the keys before placing them in EEPROM.

Generation of the one-time pad and EEPROM encryption are done in the following way:

1. The PARED device binary is provided a space of 110 flash pages (110 * 1024 bytes). The bootloader will not use up the entire space, and hence the remaining pages are filled with random bytes during the system build.
2. There are predefined length keys that need to be stored in the EEPROM, which are a total of X bytes in length. Therefore, the one-time pad needs to be X bytes (the value of X depends on the device).
3. During build time, X random addresses are picked by the host tools. All these addresses point to the unused flash space assigned to the bootloader, which was filled with random bytes. These X addresses are written as macros to a header file, accessible to the `eeprom_otp.h` file.
4. While writing the generated secrets to the EEPROM, host tools encrypt the secrets using the randomly generated addresses. The bytes from those addresses are picked from the bootloader binaries, and the encryption is done using the algorithm:

Python

```
for index in range(0, 288):  
    encr_key[index] = key[index] ^ *(address[index])
```

5. The keys in the EEPROM always stay encrypted in this way. When the firmware requests for a key, the eeprom_read function decrypts that particular key and returns it.

Without the one-time pad, the attacker would not be able to obtain the actual keys even if they are able to read the EEPROM. All the random addresses required to construct the one-time pad are defined as macros and hence are directly embedded into the bootloader code in the flash. The random padding added to the flash makes it unlikely that anyone will predict the exact address of the decrypt instructions, and therefore it is quite difficult to obtain the addresses containing the one-time pad.

3.4. TRNG Design

For dynamic random generation, we exploit the microcontroller's thermal sensor as a difficult-to-control source of entropy. As thermal noise is an example of quantum mechanical physical randomness, and any device above 0K will produce some amount of random noise, it is by nature unpredictable (cannot be modeled).

We sample the thermal sensor with two ADCs with a 16MHz clock on the lowest setting (4 clock cycles per sample), and only keep the LSB.

We now alternate which ADC we sample from per each von-neumann whitened bit we attempt to generate, adding redundancy in case one ADC malfunctions.


We apply the Von Neumann Extractor to 'whiten' (aka, de-bias) the output. Von Neumann's method looks at the change in bits, rather than the instantaneous read bits themselves, to determine noise - Therefore allowing an infrequent, yet still random, alternating source of noise to be used.

Von Neumann Extractor - for Removing Bias

- Removes the bias from a stationary and otherwise uncorrelated bit stream (i.e., bits are independent), using the mapping:

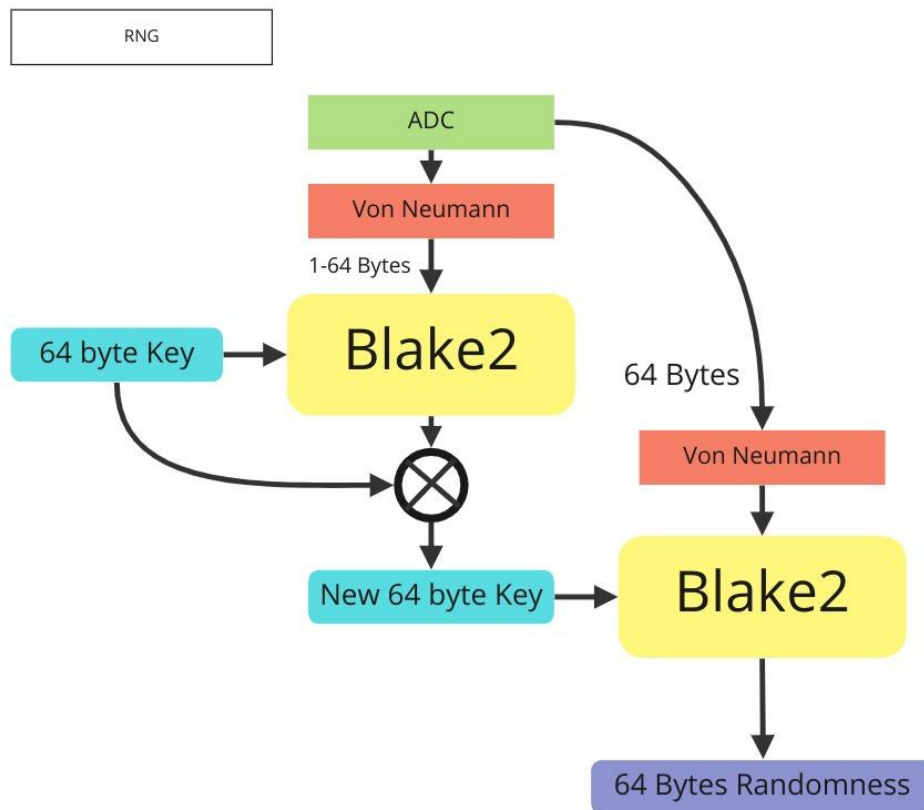
00 → Λ, 01 → 0, 10 → 1, 11 → Λ

where Λ indicates that no bit is output for that input pair
- The efficiency is defined as the number of output bits per input bit
 - The efficiency is fairly low: $\leq \frac{1}{4} \approx p_1 \cdot p_0$ (where p_1 is the probability of a one and p_0 is the probability of a zero)
- It can be used as a test by measuring the actual ratio of output to input bits (i.e., measure the "efficiency")
 - It approaches the upper limit of $\frac{1}{4}$ for input sequences with no bias
 - The frequency test seems simpler and more direct for measuring bias

 Actel Corporation © 2009 11

We use two standard methods to validate the TRNG's output - Robert G. Brown's [DieHarder suite](#), and the [NIST SP 800-22 RNG Evaluation Tool](#). These tests cannot prove an entropy source is random, but they can rule out very weak RNGs.

Additionally, we use keyed BLAKE2 hashing as a stronger debiasing function, combined with a strong-entropy key.



The Blake2 Hash function receives a strong entropy key derived during compile-time of the hardware's firmware. This will be different per each car/fob image. The ADC generates 64 bytes of Von-Neumann debiased entropy, plus an additional 1-64 implementation-defined bytes of entropy to mix into the initially seeded key. This mixing is performed by hashing the random bytes into BLAKE2 with the old key, then xoring the hashed output with the key in order to hash the next 64 bytes of entropy.

By continually refreshing the key, even if the 'secret' BLAKE2 hash key is leaked, since it is xor'd with random hashed bytes it will be unpredictable to the attacker. Additionally, as the key is continually changed with random noise, forward secrecy is preserved. By nature of the BLAKE2 hash design, an attacker will not be able to replicate the behavior of our hash without knowing its key.

Our current implementation passes the dieharder tests, and the NIST suite.

Running ‘ent’ on our output bitstream has the following result:

Unset

Entropy = 1.000000 bits per bit.

Optimum compression would reduce the size
of this 1916878832 bit file by 0 percent.

Chi square distribution for 1916878832 samples is 0.00, and
randomly
would exceed this value 94.67 percent of the times.

Arithmetic mean value of data bits is 0.5000 (0.5 = random).
Monte Carlo value for Pi is 3.141657958 (error 0.00 percent).
Serial correlation coefficient is 0.000023 (totally uncorrelated
= 0.0).

4. Defenses

4.1. Pairing PIN Storage

SR4 requires that an attacker cannot leak the pairing PIN when only given a paired fob. To mitigate potential risks of memory leakage or similar attacks, we store the BLAKE2b hash of the PIN, and the plaintext is never loaded onto any exposed device. During the Pair Fob step, we compute a hash of the PIN given by the user and compare it to the one stored. This is not a perfect defense: if the hash was leaked, it could still potentially be brute-forced by a sufficiently-well-equipped attacker.

4.2. Compiler Countermeasures

We are using Clang/LLVM as our compiler suite. We rely on the following compiler flags for compile time hardening of our system.

```
-D_FORTIFY_SOURCE=2 -Wformat -Wformat-security -Werror=format-security
```

Fortify source adds preprocessor checks to prevent buffer overflows and the other part alerts the developers for format string type vulnerabilities. We also strip the ELF to remove debug symbols using llvm-strip.

4.3. Fault Injection Defenses

Because an attacker is expected to have physical access to tamper with our hardware, we harden our critical sections against fault-injection attacks. This is done by repeating critical calculations multiple times, along with computing known “padding” computations in between

these calculations. These calculations are checked for self-consistency, and if the system detects any self-inconsistency (i.e. adding two known numbers not producing the correct result) then it will assume that there has been a hardware failure; and enter into an error state until it is reset or power-cycled. **The only condition that will trigger this error state is if an internal hardware failure is detected. It is not possible to enter this error state through only external inputs (i.e. no sequence of data over GPIO, UART, or USB can cause the system to enter such a state).** Small, randomized time delays (generated using our TRNG) are also inserted immediately before important calculations in order to significantly increase the difficulty of corrupting the given calculations via a fault attack. Finally, cryptographic signing operations are immediately verified (after signing) using the same set of keys in order to detect if there was any hardware-level corruption or failure.

4.4. Protections Against Brute force

To prevent brute force attacks, the system has protection measures in place for both the car and the paired fob. Both devices use a counter value which is stored in flash memory. After we have detected 3 failed attempts, we delay for at most 5 seconds on each further attempt.. This counter value is reset to 0 if the tool is run successfully. Since it is stored in flash memory, the count will persist even when the device is reset.

The car checks for for following in the challenge response message it receives, and if any checks fail it will increase the count:

- Message type is correct
- Message decrypts successfully
- Message is signed by the cars paired fob
- Nonce matches nonce sent in challenge
- All features were packaged by manufacturer
- All features were packed for that car
- Feature number is valid
- Set of enabled features is valid

The fob will check for failed attempts in both the pair and the enable tools.

When enabling a feature the fob will check the following:

- The feature was packaged by the manufacturer
- The feature was not packaged for a different car

When pairing, the fob will check the following:

- The pairing pin must be 6 characters, and only contain hexadecimal values
- The hash of the input pairing pin matches the hash of the pin stored in EEPROM

If any of the checks above have failed, it indicates that a potential attack is happening so the device will add delays to protect itself as discussed above.