CryptoCAN MicroPython SDK
Reference Manual

Document number	2213
Version	1
Issue date	2022-08-30

1 Introduction

1.1 Requirements for encrypted CAN messaging

Communication on CAN is not like communication in mainstream computing: CAN is an embedded real-time control bus. The messages are small, containing sensor data and actuator commands, and have strict latency and robustness requirements. From this there are several requirements on any cryptographic system for CAN:

- CAN is a broadcast bus that embodies a publish-subscribe model: messages containing sensor and status information are published periodically and the sender generally doesn't know about the receivers. The cryptographic scheme must not require 1:1 communication, such as peer-to-peer key negotiation.
- CAN is a real-time control bus. The cryptographic scheme must result in messages that have bounded latencies.
- CAN messages are very small by computing standards: just 8-byte payloads. The cryptographic scheme must fit with this limited size.
- CAN systems are usually built from constrained embedded hardware. The cryptographic scheme must work on microcontrollers with limited resources.
- CAN connected devices going through a watchdog reset must return to normal operation quickly to resume control of a piece of physical hardware. The cryptographic scheme must support fast-start communications.

The CryptoCAN scheme from Canis Labs is designed to meet all these requirements.

In the *confidentiality integrity availability* (CIA) model of communications security, CryptoCAN can provide confidentiality (i.e., keep the messages secret) and integrity (i.e., ensure messages came from a legitimate sender).

Before describing CryptoCAN, there is an important caveat to bear in mind:

No cryptographic scheme for CAN ensures availability: attacks such as bus flooding and the Bus Off attack (where a targeted device is driven offline by CAN errors) can prevent communications from taking place (just as a physical attacker can prevent communications simply by cutting the bus).

1.2 Basic CryptoCAN messaging

CryptoCAN takes a standard CAN frame (the *plaintext* frame) and converts it into a CryptoCAN message (the *ciphertext* message) that is sent on CAN then converted back into the original plaintext CAN frame by each receiver (Figure 1).

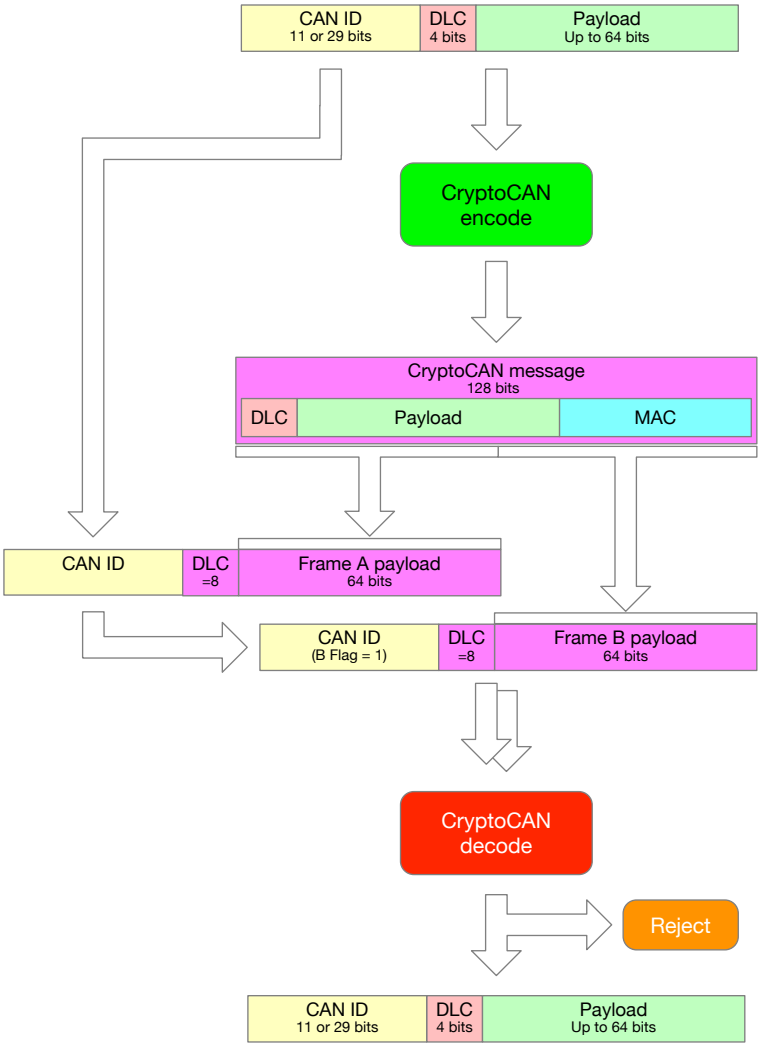

Figure 1: How CryptoCAN encodes and decodes a plaintext CAN frame

A CryptoCAN message is 128 bits long and contains:

- The original frame payload (up to 64 bits)
- The original plaintext frame DLC (4 bits)
- A *message authentication code* (MAC) of 60 bits

A MAC is a bit like a CRC but much bigger and practically impossible to forge. CryptoCAN uses the standard AES-CMAC algorithm to produce the MAC.

CryptoCAN uses a *MAC-then-Encrypt* (MtE) approach: the MAC is formed first then the whole message, including the MAC, is encrypted. Encryption is done using the standard AES-128 algorithm with the cipher feedback (CFB) mode. The result is a 128-bit ciphertext block. This is split into two pieces and put into two 64-bit (8 byte) CAN frames: Frame A and Frame B.

The CAN ID for the pair of frames is the plaintext CAN frame's ID with one bit of the ID used as the *B Flag*: this is 0 for Frame A and 1 for Frame B. The flag is there to ensure that the receiver can reassemble the pair of frames back into the CryptoCAN message before decoding. Under the CAN protocol arbitration rules, Frame A is a higher priority than Frame B and is always sent on the bus ahead of Frame B. The application can choose the

B Flag. For example, in a J1939 system the lowest bit of the priority field (bit 26) might be used, and in a CANOpen system, one of the address bits might be used.

[reference the data sheet and indicate the build options used for the firmware]

[indicate where to get the firmware from]

1.3 Message authentication

The CryptoCAN MAC is computed by using the AES-CMAC algorithm on 128 bits of data that both the sender and receiver know:

- 29 bits containing CAN ID (the ID with the B Flag removed, but with 1 bit set for standard/extended)
- 4 bits containing the plaintext CAN frame DLC
- 64 bits containing the plaintext CAN frame payload (padded if less than 8 bytes)
- A 31-bit *freshness* value: an application-specific value representing when the frame was created (it could be a time or sequence number).

When the receiver decodes a CryptoCAN message, it computes the MAC from these same known values. If the received MAC and the computing MAC do not match exactly then the message is rejected.

The MAC verification will detect any tampering with a message. For example, if the payload is attached to a different CAN frame ID, then the receiver will not compute the same MAC as transmitted. Similarly, a message will be rejected if the payload is altered.

One common attack on encryption systems is a *replay attack*: old messages are copied and then replayed later. An attacker may not know the contents of the message but can guess from context (for example, a message may result in a door being unlocked and therefore the message contains an "unlock door" command) and they can keep copies of messages with known behaviours to replay them later. These messages are genuine (because they were created by the legitimate sender) but are not valid - because they are out-of-date. This is why CryptoCAN has a freshness value included in the MAC: after this value changes, previous messages will no longer verify.

The freshness value is controlled at the application level: it can be a shared global time kept in a real-time clock on each device, or it can be a sequence number incremented each time a message is sent. It could also be partitioned so that the upper bits reflect an operating cycle count, stored in non-volatile memory in each device.

One problem with obtaining the freshness value from a timer is that a message may be created at time t but be received by the receiver at time $t + L$, where L is the latency of Frame B. The freshness value at the receiver is therefore not the same as the one used to create the message, and the MAC verification would normally fail. To address this issue, CryptoCAN has an option to use spare bits in the DLC of Frame A and Frame B: when a CAN frame is 8 bytes long, the lower 3 bits of the DLC are ignored by CAN and can be used to carry information outside the payload. Frame A and Frame B together can are used to carry the least significant 6 bits of the freshness value used to create the frames. CryptoCAN at the receiver uses these 6 bits to work out the original freshness value, determines if it is fresh, and verifies the MAC against it.

CryptoCAN creates a *context* for each message source: this stores data to encode and decode CryptoCAN messages, including key numbers of the encryption and MAC keys, the bit number of the B Flag, and the previous CryptoCAN message ciphertext (i.e., the payloads of Frame A and Frame B). The previous ciphertext is used by the CFB mode of encryption (a mode that allows a receiver to start receiving messages very quickly after starting or re-starting) but when a context is initialized, the previous ciphertext is unknown and set to a random value. This results in an important CryptoCAN property: *the first CryptoCAN message after initialization will always be rejected*. For a periodic message this is usually not a problem. But it could be a problem for a sporadic message because there may be no previous ciphertext. In this case, a simple solution is to always send the sporadic message twice.

1.4 MicroPython API

Firmware for the Canis Labs CANPico hardware contains a MicroPython API for CryptoCAN. The firmware runs on an RP2040 microcontroller, which has no cryptographic hardware, so the software emulation of a SHE HSM is included, and where keys stored in external flash memory. This is of course not resilient to physical attacks (where the flash memory is de-soldered and the keys read out) but is primarily intended to be used as an evaluation kit for CryptoCAN.

There is further development support built into CryptoCAN: an option to disapply the encryption of CryptoCAN messages so that they are transmitted as plaintext (but still with the MAC to protect against tampering). This helps a developer locate set-up problems (for example, failing to set the same key values at the sender and receivers). They can also continue to debug applications: Frame B contains the original payload and existing CAN bus analyzer tools can simply process the unencrypted Frame B. The processing time with or without the encryption applied is identical so that it can be switched on later in deployment without invalidating previous testing.

2 CryptoCAN Python API

CryptoCAN is provided via two classes: `CryptoCAN` and `HSM`.

2.1 `HSM` — Hardware Security Module

class HSM([*secret_key=None*])

Initializes the HSM. On the CANPico board this will attempt to load the HSM with keys from a flash memory block. If the parameter *secret_key* is set to 16 bytes then it will factory reset the HSM (erasing all keys) and set the SHE SECRET_KEY value.

The ability to reset the HSM is only possible because it is a software emulation: real HSMs cannot be reset. The feature is provided only for experimentation and early development: it is not suitable for a deployment system and **may be removed in a future release of this API**.

Raises

- `ValueError` – if *secret_key* is not *None* and not 16 bytes
- `RuntimeError` – if the key storage is corrupted or uninitialized

Methods

get_id([*challenge=None*])

Gets the unique ID of the HSM.

This method implements SHE CMD_GET_ID. The function returns a 120-bit unique ID, the 8-bit HSM status register, and a 128-bit MAC.

If *challenge* is not set, then a challenge of 0 is issued.

If the key for `MASTER_ECU_KEY` is empty, then the MAC returned is 0.

The return values are bytes in big-endian format.

Parameters

- **challenge** (*bytes*) – 16 bytes as a challenge value to the HSM

Raises

- `ValueError` – if the HSM returned an error

Returns A 3-tuple of the HSM unique ID, HSM status register, HSM MAC

backdoor_set_key(*key*, *key_value* [, *authentication_key=False*] [, *store_keys=False*])

A function for easily setting a key value.

This function is deprecated and will be removed in a future release of the API. It is provided only for experimentation and early development: it is not suitable for a deployment system and is present only to temporarily avoid the complexity of the SHE secure key loading process (which requires tool support for generating the necessary key programming values).

Parameters

- **key** (*integer*) – the key number
- **key_value** (*bytes*) – 16 bytes of key value
- **authentication_key** (*bool*) – if *True* then the 'key flag' is set and the key can only be used for MAC generation and verification
- **store_keys** (*bool*) – if *True* then all the key values are flushed back to flash memory

Return type bytes

enc_ecb(*plaintex*t, *key*)

Encrypt a block of plaintext using AES-128 in Electronic Code Book mode.

This method implements SHE CMD_ENC_ECB. The function returns 16 ciphertext bytes, encrypted using the specified valid key. Valid key numbers are `SHE_KEY_1` to `SHE_KEY_10` and `SHE_RAM_KEY`.

For keys other than the RAM key, the key permission must be set to encryption (i.e. the SHE 'usage' flag for the key must be clear, indicating it is not used for generating or verifying a MAC). If the key value is set to a key defined as an authentication key, then the HSM will return an error and an exception will be raised.

Parameters

- **plaintext** (*bytes*) – 16 bytes of plaintext to be encrypted
- **key** (*integer*) – the key number

Return type bytes

Raises

- `ValueError` – if the HSM returns an error

dec_ecb(*ciphertext*, *key*)

Decrypt a block of ciphertext encrypted using AES-128 in Electronic Code Book mode.

This method implements SHE CMD_DEC_ECB. The function returns 16 plaintext bytes, encrypted using the specified valid key. Valid key numbers are `SHE_KEY_1` to `SHE_KEY_10` and `SHE_RAM_KEY`.

For keys other than the RAM key, the key permission must be set to encryption (i.e. the SHE 'usage' flag for the key must be clear, indicating it is not used for generating or verifying a MAC). If the key value is set to a key defined as an authentication key, then the HSM will return an error and an exception will be raised.

Parameters

- **plaintext** (*bytes*) – 16 bytes of ciphertext to be decrypted
- **key** (*integer*) – the key number

Return type bytes

Raises

- `ValueError` – if the HSM returns an error

generate_mac(*message*, *key*)

Generates a 128-bit MAC for a given message.

This method implements SHE CMD_GENERATE_MAC. The function returns 16 bytes in big-endian format for the specified message using the specified valid key. Valid key numbers are `SHE_KEY_1` to `SHE_KEY_10` and `SHE_RAM_KEY`.

For keys other than the RAM key, the key permission must be set to authentication (i.e. the SHE 'usage' flag for the key must be set, indicating it is not used for encryption). If the key value is set to a key defined as an encryption key, then the HSM will return an error and an exception will be raised.

Parameters

- **message** (*bytes*) – a multiple of 16 bytes
- **key** (*integer*) – the key number

Return type bytes

Raises

- `ValueError` – if the HSM returns an error or if *message* is not a multiple of 16 bytes in length

verify_mac(*message*, *mac*, *key* [, *mac_length*=128])

Verifies a MAC for a given message. Returns *True* if the verification failed and *False* if the MAC was verified.

This method implements SHE CMD_VERIFY_MAC. The function returns 16 bytes in big-endian format for the specified message using the specified valid key. Valid key numbers are `SHE_KEY_1` to `SHE_KEY_10`, `SHE_RAM_KEY` or `SHE_BOOT_KEY`.

For keys other than the RAM key, the key permission must be set to authentication (i.e. the SHE 'usage' flag for the key must be set, indicating it is not used for encryption). If the key value is set to a key defined as an encryption key, then the HSM will return an error and an exception will be raised.

Parameters

- **message** (*bytes*) – a multiple of 16 bytes
- **mac** (bytes) – 16 bytes
- **mac_length** (integer) – a value between 1 and 128
- **key** (*integer*) – the key number

Return type bool

Raises

- `ValueError` – if the HSM returns an error or if *message* is not a multiple of 16 bytes in length or if *mac_length* is not between 1 and 128 or if *mac* is not 16 bytes

load_key(*m1*, *m2*, *m3*)

Securely set a specified key to a defined value.

This method implements SHE CMD_LOAD_KEY. The function takes values for M1, M2 and M3 and returns a 2-tuple of 32 and 16 bytes, corresponding to the defined values M4 and M5 respectively (the meaning of these values is detailed in the SHE HSM specification).

Parameters

- **m1** (*bytes*) – 16 bytes
- **m2** (*bytes*) – 32 bytes
- **m3** (*bytes*) – 16 bytes

Return type 2-tuple

Raises

- `ValueError` – if the HSM returns an error or if *m1* is not 16 bytes or if *m2* is not 32 bytes or if *m3* is not 16 bytes

load_plain_key(*key_value*)

Set the RAM key to the given value.

This method implements SHE CMD_LOAD_PLAIN_KEY. The function takes the value for the RAM key and sets it to the defined value. It returns *None*.

Parameters

- **key_value** (*bytes*) – 16 bytes

Raises

- `ValueError` – if the HSM returns an error or if *key_value* is not 16 bytes

export_ram_key()

Securely export the RAM key.

This method implements SHE EXPORT_RAM_KEY. The function takes no parameters and returns a 5-tuple of 16, 32, 16, 32 and 16 bytes, corresponding to the defined values M1, M2, M3, M4 and M5 respectively (the meaning of these values is detailed in the SHE HSM specification).

Return type 5-tuple

Raises

- `ValueError` – if the HSM returns an error

init_rng()

Initializes the random number generator.

This method implements SHE CMD_INIT_RNG.

It takes no parameters and returns *None*.

Raises

- `ValueError` – if the HSM returns an error

extend_seed(*entropy*)

Extends the entropy of the random number generator.

This method implements SHE CMD_EXTEND_SEED. The function takes 16 bytes of entropy value. It returns *None*.

Parameters

- **entropy** (*bytes*) – 16 bytes

Raises

- `ValueError` – if the HSM returns an error or if *entropy* is not 16 bytes

rnd(*as_int=False*)

Return a cryptographically secure random number.

This implements SHE CMD_RND. It returns a 128-bit random number as bytes in big-endian format or as a Python integer.

Parameters

- **as_int** (*bool*) – if *True* then the value returned is a 128-bit integer

Return type bytes

Raises

- `ValueError` – if the HSM returns an error

Symbols

The following symbols are defined that correspond to the standard SHE error codes:

SHE_ERC_NO_ERROR
ERC_SEQUENCE_ERROR
ERC_KEY_NOT_AVAILABLE
ERC_KEY_INVALID
ERC_KEY_EMPTY
ERC_MEMORY_FAILURE
ERC_BUSY
ERC_GENERAL_ERROR
ERC_KEY_WRITE_PROTECTED
ERC_KEY_UPDATE_ERROR
ERC_RNG_SEED

The following symbols are defined that correspond to the standard SHE key numbers:

SHE_SECRET_KEY
SHE_MASTER_ECU_KEY
SHE_BOOT_MAC_KEY
SHE_KEY_1
SHE_KEY_2
SHE_KEY_3
SHE_KEY_4
SHE_KEY_5
SHE_KEY_6
SHE_KEY_7
SHE_KEY_8
SHE_KEY_9
SHE_KEY_10
SHE_RESERVED
SHE_RAM_KEY

2.2 CryptoCAN

2.3 `CryptoCAN` — CryptoCAN context

class CryptoCAN([*transmit=False*] [, *encryption_key=HSM.SHE_KEY_*1] [, *authentication_key=HSM.SHE_KEY_*2] [, *b_flag*=0] [, *anti_replay=False*] [, *alt_freshness=False*] [, *no_encrypt=False*])

Creates a CryptoCAN context and makes it ready to generate or receive CryptoCAN frames. A transmission context is created if *transmit* is set to *True*.

The key numbers are set in the call to refer to the keys loaded into the HSM. A receiver and a transmitter can used different key numbers, but the key values loaded into the key respective key slots in each HSM must be the same for the sender and the receiver. The keys must have their permission set in the HSM appropriately: keys to be used for authentication must have the authentication permission.

The B Flag is set to indicate which bit of the CAN ID will be used to indicate Frame B. The CAN ID of the plaintext frame must not have this bit set. Both the transmit and all receive contexts must use the same B Flag value. The B Flag is measured from the least significant bit of the arbitration ID (either 11-bit or 29-bit identifiers).

The Anti-Replay option can be enabled but it must be enabled for the transmitter and at every receiver that wishes to use it. When anti-replay is disabled, the freshness value at the sender and the receivers must match exactly. When anti-replay is enabled, there are two rules:

- The transmitter and the receivers must keep their freshness values synchronized together within a range: the freshness value at a receiver must be no more than 32 ahead of the freshness value used at the transmitter and no more than 31 behind.
- The transmitter must use a freshness value higher than the previous value (using modulo arithmetic: the freshness value is a 31-bit unsigned integer and rolls over to 0).

The Alternative Freshness option applies to receive contexts and allows the application to supply two freshness values, with the message being accepted if either value causes the message to verify.

The No Encrypt option disapplies the encryption wrapper so that the contents of Frame A and Frame B are sent on the bus unencrypted (the MAC still applies to the message). This allows existing CAN tools to decode the message contents during a debugging phase: Frame B contains the plaintext payload. This option must be set consistently for the transmitter and the receivers.

Parameters

- **transmit** (*bool*) – Set to *True* if the context is for generating frames, or *False* if the context is used for receiving frames
- **encryption_key** (*integer*) – The key number in the HSM used for encryption
- **authentication_key** (*integer*) – The key number in the HSM used for authentication

- **b_flag** (*integer*) – The bit number of the CAN ID that indicates a B Frame
- **anti_replay** (*bool*) – If *True* enables the anti-replay feature
- **alt_freshness** (*bool*) – If *True* then a second freshness value can be supplied (receive contexts only)
- **no_encrypt** (*bool*) – If True then the encryption wrapper is not applied

Raises

- `ValueError` – if *encryption_key*, *authentication_key* or *b_flag* are out of range or if the key numbers are the same or if the HSM returns an error or if *transmit* is *True* and *alt_freshness* is *True*
- `RuntimeError` – if the key storage is corrupted or uninitialized

Methods

create_frames([*challenge=None*])

Gets the unique ID of the HSM.

This method implements SHE CMD_GET_ID. The function returns a 120-bit unique ID, the 8-bit HSM status register, and a 128-bit MAC.

If *challenge* is not set, then a challenge of 0 is issued.

If the key for `MASTER_ECU_KEY` is empty, then the MAC returned is 0.

The return values are bytes in big-endian format.

Parameters

- **challenge** (*bytes*) – 16 bytes as a challenge value to the HSM

Raises

- `ValueError` – if the HSM returned an error

Returns A 3-tuple of the HSM unique ID, HSM status register, HSM MAC

3 Quick start on the CANPico

3.1 Setting up the HSM

Connect to the board using a terminal emulator (minicom, PuTTY, rshell, etc.) to get a REPL prompt (use the first serial port – typically /dev/ttyACM0 on Linux – not the second one, which is reserved for MIN).

At the REPL prompt, bring in the CryptoCAN API and do a factory reset of the HSM:

```
>>> from rp2 import *
>>> h = HSM(secret_key=bytes([0] * 16))
```

This obviously isn't a very good secret key, so use a proper random key from an appropriate source. The HSM after a reset will have no keys in it, so program a key using the simple backdoor method:

```
>>> h.backdoor_set_key(key=HSM.SHE_KEY_1, key_value=bytes([0] * 16),
store_keys=True)
```

This will set the value of user key 1 and define it as an encryption key.

Reset the CANPico with CTRL-D (or power cycle it) and try initializing the HSM:

```
>>> from rp2 import *
>>> h = HSM()
```

This should succeed, and user key 1 will be set. We can check this by trying to encrypt something with it:

```
>>> m = bytes([0] * 16)
>>> c = h.enc_ecb(m, HSM.SHE_KEY_1)
>>> c
b'f\xe9K\xd4\xef\x8a,;\x88L\xfaY\xca4+.'
```

To see the hex value for the ciphertext we can use some Python:

```
>>> from binascii import hexlify
>>> hexlify(c)
b'66e94bd4ef8a2c3b884cfa59ca342b2e'
```

That particular ciphertext is very common: an internet search for the string gives many results because it's zero encrypted with zero.

Just to check everything is working the reverse process can be done by decrypting that ciphertext with the key:

```
>>> p = h.dec_ecb(c, HSM.SHE_KEY_1)
>>> hexlify(p)
b'00000000000000000000000000000000'
```

3.2 Basic CryptoCAN messaging

This simple example will encrypt then decrypt CryptoCAN CAN frames on the same CANPico board to keep things simple.

First, create an instance of the HSM. This is necessary even if no further use of the HSM is made: the class constructor initializes the HSM and loads the keys from flash memory.

```
>>> from rp2 import *
>>> h = HSM()
```

Program a pair of keys into the HSM:

```
>>> from binascii import unhexlify
>>> ek = unhexlify('92b91441bc219b140c325d34a3d3c73f')
>>> ak = unhexlify('0bb491e6e5809f960959b2fc662e70e2')
>>> h.backdoor_set_key(key=HSM.SHE_KEY_1, key_value=ek)
>>> h.backdoor_set_key(key=HSM.SHE_KEY_2, key_value=ak,
authentication_key=True, store_keys=True)
```

The first key is the encryption key, the second is the authentication key. These key values need to be programmed into all the CANPico boards that will receive CryptoCAN frames (for simplicity, only one board is being used). Note that this uses the backdoor API: this will be removed in the future and then the SHE load key API must be used to set keys.

Next, create a sending context on the sending CANPico board:

```
>>> tx = CryptoCAN(encryption_key=HSM.SHE_KEY_1,
authentication_key=HSM.SHE_KEY_2, transmit=True)
```

All receiving CANPico boards need to create a receive context:

```
>>> rx = CryptoCAN(encryption_key=HSM.SHE_KEY_1,
authentication_key=HSM.SHE_KEY_2)
```

Then create a plaintext CAN frame using the regular CAN API:

```
>>> f = CANFrame(CANID(0x122), data=b'hello')
>>> f
CANFrame(CANID(id=S122), dlc=5, data=68656c6c6f)
```

Create a CryptoCAN pair of these frames:

```
>>> frames = tx.create_frames(f)
>>> frames
[CANFrame(CANID(id=S122), dlc=8, data=ec363347ad12172f),
CANFrame(CANID(id=S123), dlc=8, data=9d99ff5d4228e0fa)]
```

The values of the payloads will be different each time the example is run because CryptoCAN contexts are initialized using random numbers.

These frames could now be transmitted over CAN with the `transmit_frames()` method of the `CAN` class, and then picked up at receivers by the `recv()` method. But for

simplicity we can just use the `frames` variable directly on the same CANPico board. Frames are received in sequence by the receive context:

```
>>> rx.receive_frame(frames[0])
>>> rx.receive_frame(frames[1])
```

This produces no frame: with CryptoCAN *the first message after creating a receive context is always dropped*. Subsequent messages are received though:

```
>>> frames = tx.create_frames(f)
>>> rx.receive_frame(frames[0])
>>> rx.receive_frame(frames[1])
CANFrame(CANID(id=S122), dlc=5, data=68656c6c6f)
```

The call returns the decrypted and verified CAN frame.

3.3 Anti-replay protection

The simple communication example left the freshness value and anti-replay protections in the application domain. This example demonstrates using the built-in anti-replay feature of CryptoCAN.

CryptoCAN contexts are created as before, but also enable the anti-replay feature:

```
>>> tx = CryptoCAN(transmit=True, anti_replay=True)
>>> rx = CryptoCAN(anti_replay=True)
```

The encryption and authentication key numbers aren't specified above because they default to application key 1 and application key 2 anyway.

The following sequence illustrates the anti-freshness rules:

```
>>> frames = frames = tx.create_frames(f, freshness=12345678)
>>> rx.receive_frame(frames[0], freshness=12345670)
>>> rx.receive_frame(frames[1], freshness=12345670)
>>> frames = frames = tx.create_frames(f, freshness=12345679)
>>> rx.receive_frame(frames[0], freshness=12345670)
>>> rx.receive_frame(frames[1], freshness=12345670)
CANFrame(CANID(id=S122), dlc=5, data=68656c6c6f)
>>> frames = frames = tx.create_frames(f, freshness=12345679)
>>> rx.receive_frame(frames[0], freshness=12345670)
>>> rx.receive_frame(frames[1], freshness=12345670)
>>> frames = frames = tx.create_frames(f, freshness=12345680)
>>> rx.receive_frame(frames[0], freshness=12345670)
>>> rx.receive_frame(frames[1], freshness=12345670)
CANFrame(CANID(id=S122), dlc=5, data=68656c6c6f)
```

Note how the receiver uses the same freshness value for all calls (in a real system this might be broadcast from a central time source and be updated only periodically). This is acceptable because the value is within -31 to 32 of the freshness value used at the transmitter.

The first message is dropped, as always, and the second message is received. The third message doesn't advance the freshness value and to a receiver it looks like a replayed old message and is dropped. The fourth message advances the freshness value and received.