

# Securing CAN: Introduction to CryptoCAN

*CryptoCAN by Canis Automotive Labs is an encryption scheme for CAN frames. It is designed to meet the requirements of in-vehicle CAN messaging. For example, publish/subscribe communications.*

CAN was created in the mid-1980s to provide a robust atomic broadcast system to connect ECUs (electronic control unit) in passenger cars to replace individual signaling wires and has become a proven technology in applications as diverse as yachts and spacecraft. But CAN was never designed with security in mind – in the mid-1980s there was no notion of embedded systems being connected to the internet. Today the world is very different and there is a need to secure CAN communications because systems built with CAN are cyber-physical systems: there are actuators that move things in the real world based on the contents of CAN frames.

In mainstream computing a common way to secure communications is to use cryptography: to keep secret the contents of messages and to ensure messages have not been tampered with. This can be done for CAN systems too, but there are special requirements for CAN.

being evaluated by the United States Army Combat Capabilities Development Command (DEVCOM) Ground Vehicle Systems Center (GVSC) in the cooperative research and development Agreement “Cyber Security for Military Ground Vehicles Architectures”.

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).

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).



Figure 1: How CryptoCAN encodes and decodes a plaintext CAN frame (Source: Canis Automotive Labs)

## Special requirements for CAN cryptographic schemes

- ◆ CAN is a broadcast network 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.
- ◆ CAN is a real-time control network. The cryptographic scheme must result in messages that have bounded latencies.
- ◆ CAN frames 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 micro-controllers 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 of Canis Labs is designed to meet all these requirements. CryptoCAN is currently

## Basic CryptoCAN messaging

CryptoCAN takes a classical 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). A CryptoCAN message is 128 bit long and contains:

- ◆ The original frame payload (up to 64 bit)
- ◆ The original frame DLC (data length code, 4 bit)
- ◆ A message authentication code (MAC) of 60 bit

A MAC is a bit like a CRC (cyclic redundancy check) but much bigger and practically impossible to forge. CryptoCAN uses the standard AES-CMAC (advanced encryption standard - cipher message authentication code) algorithm to produce the MAC.

The message is encrypted using the standard AES-128 algorithm and 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 (identifier) 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 ▶

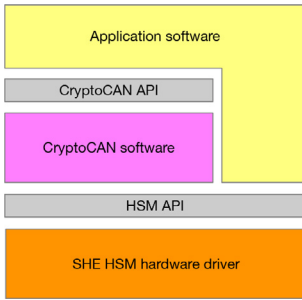


Figure 2: CryptoCAN on a micro-controller with an SHE HSM (Source: Canis Automotive Labs)

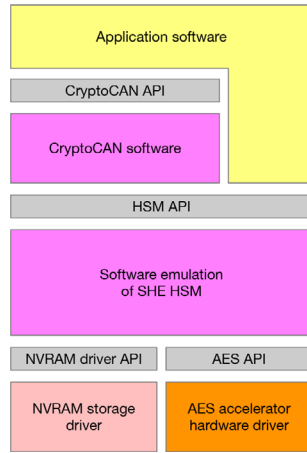


Figure 3: CryptoCAN on a micro-controller with AES accelerator hardware (Source: Canis Automotive Labs)

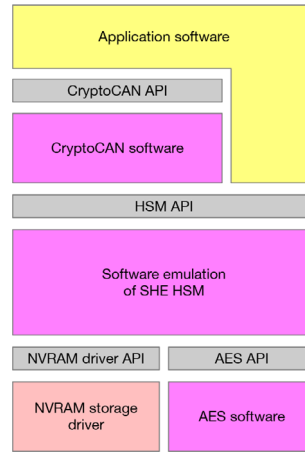


Figure 4: CryptoCAN on a micro-controller with no cryptographic hardware (Source: Canis Automotive Labs)

hardware extensions (SHE) HSM defined by the automotive industry. The SHE HSM standard specifies the AES-128 algorithm (for encrypting blocks of data) and the AES-CMAC algorithm for creating and verifying a MAC. The standard also defines how keys are managed: they are stored in secure non-volatile memory (in a dedicated area of memory that is not directly accessible by the application software), there is a defined protocol for programming them, and keys have defined permissions: they can be used for encryption/decryption or for MAC creation/verification. CryptoCAN uses the SHE HSM functions for encryption and MAC generation and verification (the keys are programmed into the HSM as part of provisioning a device).

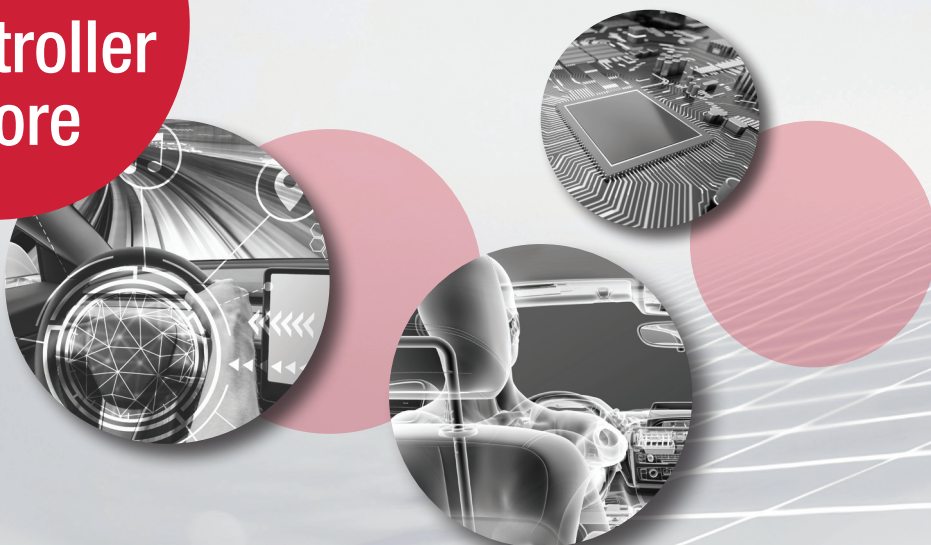
decoding. Under the CAN protocol arbitration rules, Frame A is a higher priority than Frame B and is always sent on the network ahead of Frame B. In a J1939 system the lowest bit of the priority field (bit 26) could be used. In a CANopen system, one of the address bits could be used.

### CryptoCAN implementation

The cryptographic algorithms used are the ones provided by a particular hardware security module (HSM): the secure

Not all embedded micro-controllers have an SHE HSM: some have AES-128 accelerators, some have true random number generators (TRNG) and some have no cryptographic hardware. To allow CAN devices using these micro-controllers to participate in secure communications, CryptoCAN has a layered architecture (see Figures 2 to 4).

# CAN Controller IP Core



# CAST

Learn more at: [www.cast-inc.com](http://www.cast-inc.com) or email [info@cast-inc.com](mailto:info@cast-inc.com)



**COMPLETE**  
CAN 2.0, CAN FD, CAN XL plus TTCAN AUTOSAR & SAE optimization

**RELIABLE**  
Plugfest-verified & proven in hundreds of customer systems

**FLEXIBLE**  
ASICs or FPGAs; Works with any Transceiver

**SAFE**  
Certified Functional Safety ASIL D

**SECURE**  
Available CANsec

In the first situation (Figure 2), the CryptoCAN messaging software uses SHE HSM hardware. The application accesses the HSM for key management functions (setting and updating key values).

In the second situation (Figure 3), CryptoCAN is running on a micro-controller without an HSM but with an AES-128 accelerator. In this case the CryptoCAN software includes an SHE HSM emulator that uses the AES-128 accelerator hardware via a driver API (application programming interface) and to access target-specific non-volatile memory storage (typically on-chip flash or EEPROM) to store keys.

In the third situation (Figure 4), CryptoCAN software is running on a micro-controller with-out any cryptographic hardware. There is a software emulation of an SHE HSM with a software implementation of AES-128.

A pure software implementation allows CryptoCAN to run on a wide range of CAN-connected devices. The AES-128 encrypt operation is the most compute-intensive part of CryptoCAN, and on the RP2040 micro-controller (used in the Canis Labs CANPico board) it takes approximately 13  $\mu$ s. The creation of a CryptoCAN frame requires two AES-128 encrypt operations and the decode of Frame A and Frame B, each requires one. The RP2040 micro-controller uses execute-in-place (XIP) external flash and there can be very large cache fetch delays for cache misses. Cryptographic operations must have constant execution time so the cryptographic functions in the RP2040 implementation of CryptoCAN are placed in RAM (random access memory).

## CryptoCAN MAC

The CryptoCAN MAC is computed by using the AES-CMAC algorithm on 128 bits of data that both the sender and receiver are expected to know: 29 bits containing the 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), and 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 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 behaviors 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

```

ken@ubuntu-linux-20-04-desktop:~$ ssh -p 2222 pi@192.168.1.100
pi@raspberrypi:~$ micropython
Retrieving sysname ... rp2
Testing if ubinascii.unhexlify exists ... y
Retrieving root directories ... /canis.py /canpico.py /main.py /pinger.py /t
est.py
Setting time ... Aug 08, 2022 12:34:52
Evaluating board_name ... pyboard
Retrieving time_epoch ... Jan 01, 1970
Welcome to rshell. Use control-D (or the exit command) to exit rshell.
Entering REPL. Use control-X to exit.
MicroPython as5a247f-dirty on 2022-08-08; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>> from rp2 import *
>>> h = HSM()
>>> cc = CryptoCAN(transmit=True)
>>> f = CANFrame(CANID(0x120), data=b'hello')
>>> frames = cc.create_frames(f)
>>> c = CAN()
>>> c.send_frames(frames)
>>> frames = cc.create_frames(f)
>>> f.send_frames(frames)
>>>

ken@ubuntu-linux-20-04-desktop:~$ ssh -p 2222 pi@192.168.1.100
pi@raspberrypi:~$ micropython
MicroPython as5a247f-dirty on 2022-08-08; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>> from rp2 import *
>>> h = HSM()
>>> cc = CryptoCAN()
>>> c = CAN()
>>> frames = c.recv()
>>> frames[0]
CANFrame(CANID(0x120), dlc=8, data=49989a3d0f14c5e, timestamp=259414044)
>>> frames[1]
CANFrame(CANID(0x121), dlc=8, data=f24da8390e4910f, timestamp=259414208)
>>> cc.receive_frame(frames[0])
>>> cc.receive_frame(frames[1])
>>> frames = c.recv()
>>> frames[0]
CANFrame(CANID(0x120), dlc=8, data=8081e24b0b1e971a, timestamp=298892220)
>>> frames[1]
CANFrame(CANID(0x121), dlc=8, data=d3a5b0199dc1a5, timestamp=298894440)
>>> cc.receive_frame(frames[0])
>>> cc.receive_frame(frames[1])
>>> CANFrame(CANID(0x120), dlc=5, data=d856cc6cf, timestamp=298894440)
>>>

```

Figure 5: Interactive Micropython session on two CANpico boards (Source: Canis Automotive Labs)

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 EEPROM 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 the least significant 3 bits of the CAN DLC fields of Frame A and Frame B to encode 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, determine if it is fresh, and then verifies the MAC against it.

## CryptoCAN contexts

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: there may be no previous ciphertext. In this case, a simple solution is to send the message twice.

## Development support

CryptoCAN software is supplied as source code with a C API. Also provided is a Micropython API to CryptoCAN in firmware for the Canis Labs CANpico hardware. This uses an RP2040 micro-controller without any cryptographic

hardware so the software emulation of a SHE HSM is included, with 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. Figure 5 shows a simple interactive Micropython session on two CANpico boards, creating and sending encrypted CAN frames (left) and receiving and decoding them (right). The HSM on each of the CANpico boards has been pre-provisioned with the encryption and authentication keys. Note how the first CryptoCAN message is discarded.

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). This helps a developer locate set-up problems (for example, failing to set the same key values at the sender and receivers) and application problems: Frame B contains the original payload and existing CAN analyzer tools can simply process the unencrypted Frame B.

## Summary

CryptoCAN is an encryption scheme specifically designed for CAN. It fits the publish-subscribe paradigm common to CAN systems, where a sender is not coupled to receivers. It also supports the fast start of a receiver to participate in encrypted communication.

CryptoCAN replaces a plaintext CAN frame with a pair of ciphertext CAN frames with the same real-time properties, and where the latency of Frame B is the latency of the message, allowing existing scheduling analysis tools for CAN to continue to be used to calculate worst-case frame latencies. CryptoCAN has also been carefully designed to run efficiently on micro-controllers with no cryptographic hardware, and the extra bandwidth used by CryptoCAN is one extra CAN frame per original frame. The issue of replay attacks has been directly addressed, with support for automatically detecting and dropping replayed messages.

The CryptoCAN Micropython firmware is free to use for the Canis Labs CANpico hardware. ◀



## Author

Ken Tindell  
Canis Automotive Labs  
[ken@canislabs.com](mailto:ken@canislabs.com)  
[canislabs.com](http://canislabs.com)

# Landing soon: a more personal touch



Push to open



**SafeLine**   
Your partner in lift safety