CANdy: Backward-Compatible CAN Message Authentication via Sub-Bit Injection on Commercial Controllers

Yu-Wei Liu[0009−0000−6135−8477], Jan de Voor[0009−0006−4407−1267], Caden Allen[0009−0000−2060−9700], Siddharth Thumsi[0009−0006−1814−8101], and Mert D. Pesé[0000−0001−9192−5823]

School of Computing, Clemson University, Clemson, SC, USA
{yuwei,jdevoor,caden3,sthumsi,mpese}@clemson.edu

Abstract. The Controller Area Network (CAN) is the industry-standard protocol for intra-vehicular communication; however, its prioritization of reliability over security creates significant vulnerabilities in the increasingly connected automotive landscape. While various protocol enhancements have been developed to address these security deficits, they typically require additional hardware, compromise compatibility with existing systems, or increase bandwidth usage. To address these challenges, we propose `CANdy`, a dynamic CAN data hammering technique that embeds Message Authentication Codes (MACs) to protect payload integrity through precise data encoding and verification. `CANdy` can be integrated as a software patch into existing Electronic Control Units (ECUs) or deployed as a standalone node using low-cost commercial hardware. The system exchanges a 128-bit AUTOSAR standard-compliant MAC using AES-CMAC within a single legacy CAN data frame. This is achieved at typical CAN bit rates of 125, 250, and 500 kbps, with 100% accuracy and 0 bit-error rate, including restbus and attacker traffic, without interrupting or interfering with standard protocol transmission, ensuring high compatibility and scalability across diverse hardware platforms. On-vehicle testing achieves 99.8% accuracy with the native prototype and 93.8% with attacker traffic, demonstrating robust defensive capabilities in real-world environments.

Keywords: CAN bus · Authentication · Automotive security.

1 Introduction

The Controller Area Network (CAN), originally developed by Bosch in the 1980s, has become the de facto communication protocol for intra-vehicle and industrial control networking. Data from various sensors is gathered by Electronic Control Units (ECUs), which communicate over an in-vehicle network (IVN). Designed with a strong emphasis on reliability and real-time performance, the CAN protocol was not built with strong security mechanisms [6, 7, 14]. Additionally, the simplified design of the classic CAN protocol limits the maximum data payload

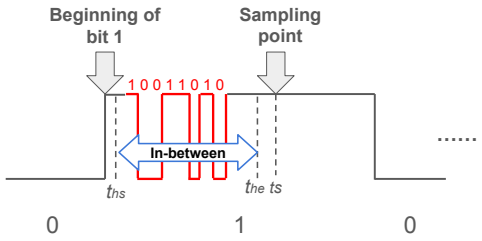


Fig. 1: Sampling point in one CAN-bit and the in-between zone from the beginning and the sampling point

to 8 bytes per frame, which constrains the implementation of modern cryptographic algorithms and security policies. Consequently, increasing network bandwidth while maintaining compatibility with legacy ECUs is challenging since the foundational CAN protocol lacks built-in security features, transmitting all messages in plaintext without any form of sender authentication or message integrity verification.

While widely adopted industry standards such as Automotive Open System Architecture (AUTOSAR) [5] have introduced a cipher-based Message Authentication Code (CMAC) that uses the Advanced Encryption Standard (AES) with a 128-bit key, also called AES128-CMAC, in the Secure Onboard Communication (SecOC) module to address authentication and integrity concerns, integrating these security primitives into legacy CAN systems remains a challenge due to bandwidth constraints and protocol limitations. With a configuration that uses a truncated 24-bit Message Authentication Code (MAC) and no freshness value, there are only 40 bits out of the 64-bit payload of the original CAN message left for real data. Previous research has analyzed real-world production vehicles to show that approximately 40-60% of CAN IDs utilize around 40 bits of their payload [23]. This leaves about 24 bits of free space, which is the minimum required for a truncated SecOC MAC. However, a significant portion — nearly the other half of all CAN IDs — requires more than that amount of space. From these findings, it appears that only roughly + of the ECUs in actual production vehicles could accommodate a SecOC implementation without reducing or altering the original data payload.

Prior work (see Table 1) has explored several CAN bus authentication techniques to prevent attackers from spoofing CAN frames. However, many of these methods either ❶ impact network utilization/bus load by adding additional CAN frames to carry the MAC due to lack of space in the message payload [15,33,35]; ❷ compromise real-time performance by having large latencies to delivery, calculate or verify MACs [13,18]; ❸ lack backward-compatibility by requiring costly and specialized hardware solutions such as Field-Programmable Gate Arrays (FPGAs) or additional circuits [20,34]; or ❹ the use of truncated MACs pro-

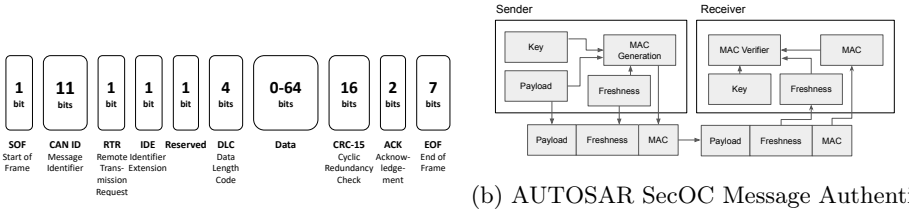
vides only limited security levels [6, 15, 23], with 128 bit entropy being the recommended minimum.

Table 1: Comparison of `CANdy` with existing solutions. The performance metrics and data in this table are derived from results in the respective related works.

Work	Algo.	MAC Length	Type	Bus Load	Latency	Security Level	Layer
CaCAN [15]	SHA256-HMAC	1B	HW+SW	100%	+2.2-3.2μs	2^7	Link
IA-CAN [11]	ID+CMAC	1–4B	SW	0%	+150μs	2^{31}	Link
vatiCAN [18]	SHA3-HMAC	8B	SW	16%	+3.3ms	2^{63}	Link
VeCure [35]	HMAC	4B	SW	100%	+50μs	2^{31}	Link
CANAuth [34]	HMAC	10B	HW+SW	0%	N/A	2^{79}	Physical
S2-CAN [23]	Circular Shift + Internal ID Match	N/A	SW	0%	+75μs	2^{49}	Link
ShadowAuth [13]	HMAC	8B	SW	100%	+60ms	2^{63}	Link
CAN-MM [20]	AES128-CMAC	16B	HW+SW	0%	+128 μs	2^{127}	Physical
CANdy	**AES128-CMAC**	**16B**	**SW**	**0%**	**+112μs**	2^{127}	**Physical**

In this work, we propose `CANdy`, a software-based novel bit-level coding system that enables the secure transmission of a 128-bit AUTOSAR standard-compliant AES-CMAC [4] within a single legacy CAN frame without interfering with standard CAN behavior. We inject additional data into the physical layer, specifically between the beginning of each CAN bit and the actual sampling point, referred to as the *in-between* zone, as shown in Fig. 1 [34]. We refer to this sub-bit data injection as *bit hammering* and call the injected bits *hammered bits*. To overcome the synchronization challenges of hammered bit transmission between the ECUs, we utilize a *state-changing hammering mechanism* that encodes and decodes using bit toggling and the repetition times of the hammering bit. Similar to the principle of Manchester coding [12], data is encoded through transitions that occur within each bit interval. To minimize the limitations of CPU clock resolution in ECUs, we encode the hammered bits based on the transmitted bit's transition and decode them based on the repetition time after the oversampling-receiving method, unlike in the Manchester code, where only one transition or repetition occurs.

To stay compliant with industry standards, we implement SecOC's AES-CMAC integrity tag as the MAC calculation algorithm for data exchanged between ECUs by `CANdy`, allowing the receiver to verify both the origin and the integrity of the data. In legacy CAN, each CAN frame can only transmit a maximum of 64 bits in its payload, making it difficult to transmit the full 128-bit authentication tag without affecting real-time performance, and it is vulnerable



(a) CAN 2.0A Data Frame

(b) AUTOSAR SecOC Message Authentication and Verification

Fig. 2: CAN and AUTOSAR SecOC Fundamentals

to spoofing attacks. Furthermore, `CANdy` maintains full backward-compatibility with legacy CAN frames while sending the MAC by using only lightweight software modifications on existing hardware. This paper makes the following main contributions:

- We propose `CANdy`, a backward-compatible, software-only encoding scheme that exploits the idle *in-between zone* before the CAN sampling point to inject auxiliary data. By "hammering" bits into this physical layer gap, we enable covert data transmission without increasing bus load or violating legacy CAN protocol rules.
- We demonstrate the first practical integration of the full 128-bit AUTOSAR SecOC standard (using AES-CMAC) within a single legacy CAN data frame. Unlike prior solutions that rely on truncated MACs or additional frames due to bandwidth constraints, our approach maintains the full cryptographic strength of the MAC without sacrificing real-time performance.
- We design a state-changing hammering protocol that overcomes ECU clock synchronization challenges by encoding data through bit transitions and repetition times. Extensive evaluation on commercial Renesas hardware achieves 100% accuracy and a 0-bit error rate at bus speeds of 125 kbps (21 bits/frame), 250 kbps (11 bits/frame), and 500 kbps (4 bits/frame), including restbus and attacker traffic, with the full open-source implementation provided for reproducibility at `https://github.com/liu810206/CANdy`. The on-vehicle testing results show 99.8% and 93.9% accuracy without and with an attacker, respectively.

2 Background

2.1 CAN Primer

CAN 2.0A. Fig. 2a shows the frame format for the CAN 2.0A protocol. A typical CAN 2.0A frame starts with a Start-of-Frame (SOF) bit, followed by an 11-bit identifier that sets the message's priority level. Next come the control bits and a Data Length Code (DLC), which indicates a data field of up to 8 bytes. To enhance reliability, a 15-bit CRC is used for error detection, and an Acknowledgment (ACK) slot lets receiving nodes confirm that the message was

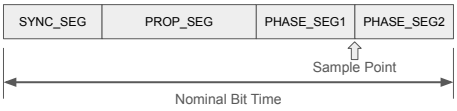


Fig. 3: Nominal Bit Time and Sample Point

correctly received. The frame ends with an End-of-Frame (EOF) sequence and an Inter-Frame Spacing (IFS) interval, which enforces a required idle period between successive bus messages. Please see Appendix A.1 for more details.

CAN Hardware. In modern vehicles, sensor data is gathered and distributed by interconnected ECUs via the CAN bus. Each ECU acts as a CAN node, communicating with other ECUs across the IVN. Each ECU is typically composed of three main components: a microcontroller unit (MCU), a CAN controller, and a CAN transceiver. The MCU runs application-level logic, while the controller and transceiver handle communication tasks. The **CAN transceiver** (PHY) operates on the *physical layer* and is responsible for converting the controller's digital signals into analog voltages (typically 0–5V) for transmission, and vice versa for reception. It uses differential signaling over two wires: CAN_H and CAN_L. The **CAN controller** operates at the *data link layer*. It constructs CAN frames using the ID, DLC, and Data fields, manages bit stuffing, and handles error detection. It connects to the physical layer via two lines: CAN_TX (for outgoing bits) and CAN_RX (for incoming bits). In modern ECUs, the CAN controller is integrated directly into the MCU, enabling memory-mapped access to CAN functions (i.e., *integrated* CAN controller). This setup allows developers to configure filters, handle interrupts for incoming messages, and even directly manipulate CAN_RX and CAN_TX pins at runtime through pin multiplexing. This architecture is now common among major automotive chip manufacturers such as NXP, STMicroelectronics, and Renesas [19, 26, 32].

Sample Point. Each bit is sampled by the CAN controller at a configurable, fixed sampling point to determine whether it is dominant (0) or recessive (1), which is transmitted in one Nominal Bit Time (NBT) that is separated into four distinct segments: Synchronization Segment (SYNC_SEG), Propagation Segment (PROP_SEG), Phase Buffer Segment 1 (PHASE_SEG1), and Phase Buffer Segment 2 (PHASE_SEG2) [27]. Each segment is an integer multiple of a unit of time called a Time Quanta t_Q. The length of a single Time Quantum is directly tied to the CAN system clock, which is generated by dividing the MCU's system clock with a programmable divider, the Baud Rate Prescaler (BRP). The sampling point is located at the end of PHASE_SEG1, near 75%-87.5% of the nominal bit time (see Fig. 3). For a 500 kbps CAN bus, each bit would be sampled at least after 1.5 μs as a result.

2.2 AUTOSAR SecOC

AUTOSAR Secure Onboard Communication (SecOC) is a standardized AUTOSAR module designed to provide integrity and authentication for messages (Protocol Data Units, or PDUs) exchanged within a vehicle's communication network [4]. SecOC is a crucial component of the AUTOSAR Classic Platform, which is widely used in various automotive applications, including engine control, transmission control, and braking systems. It uses cryptographic methods to ensure that messages have not been altered and originate from a trusted source. While the module defines how to apply these security mechanisms, it deliberately leaves key distribution methods unspecified, allowing system designers to choose an appropriate strategy for their specific use case. AUTOSAR created three different profiles with varying sizes for truncated freshness values and truncated MACs. These profiles are all based on CMAC with AES-128. In this paper, we selected SecOC Profile 2 as the development standard, with a 24-bit truncated MAC (the minimum among all profiles) and no dedicated freshness value, since the freshness value is baked into the generation of the MAC tag. For the detailed definition of AUTOSAR SecOC, please see Appendix A.2.

3 Related Work

As the CAN standard was designed primarily for safety and reliability, protecting the CAN bus from adversarial attack has been an important focus in automotive security. Table 1 summarizes key aspects of eight popular CAN authentication solutions that are explained in the following classifications:

Add additional CAN frames for MAC transferring. Existing authentication solutions rely on the same foundational principle: include a MAC to CAN frames to detect adversarial spoofing and protect message integrity. Authentication of CAN messages through a dedicated monitoring node [15] or a hierarchical trust group that relies on members of a higher trust group holding the secret keys for the groups below it [35] minimizes the number of trusted ECUs, but results in increased bus load due to the communication between ECUs and the authentication nodes.

Compromise real-time performance for MAC calculation and verification. By integrating authentication directly into the firmware binary without needing access to the original source code [13], or authenticating only selected safety-critical messages to reduce bandwidth overhead [18], these solutions maintain bus capacity while transmitting authentication data simultaneously. However, they compromise real-time performance due to large latencies in MAC calculation or verification.

Implement additional hardware solutions. To reduce authentication latency and preserve full interoperability with older ECUs, solutions such as integrating a multiplexed signaling scheme via frequency modulation to implant MAC within standard CAN traffic [20], hardware-based overclocked capability to increase bus throughput [34, 38], or depending on a separate node to authenticate the other ECUs [28]. Although they reduce authentication latency, the

requirement for costly, specialized hardware, such as Field-Programmable Gate Arrays (FPGAs) or additional circuits, poses an obstacle in real-world scenarios.
Truncate MAC for authentication. To save the bus load and keep backward compatibility, solutions such as truncating the MAC so that it can be appended to the remaining data bits in a single CAN frame [6, 15], or randomly generated parameters and payload manipulation by using protocol-specific properties [23]. Although they keep the overhead low and ensure real-time performance, they waste bus bandwidth because they must manage data transmission using the CAN 2.0A protocol, which limits frame size and provides only minimal security due to truncated MACs.

4 Threat Model

Adversary Capabilities. In line with existing studies on CAN security [7, 9, 14, 17, 24, 25, 29, 33], we consider an adversary capable of remotely compromising ECUs via Bluetooth, Wi-Fi, cellular links, or the On-Board Diagnostic II (OBD-II) port [23, 36]. We assume the adversary can execute malicious code on compromised ECUs to inject and receive CAN messages, but is restricted to user-level privilege and cannot directly interact with hardware registers or peripherals, such as the CAN controller, CAN transceiver, or General Purpose Input/Output (GPIO) ports. Please see Appendix B for more details.
Attack Goals. The primary objective of the adversary is the transmission of unauthorized commands via the CAN bus, such as engine control, sudden braking or acceleration, or blocking the transmission of certain CAN IDs entirely. Depending on the attack, the adversary will: (1) fabricate CAN messages by injecting frames with valid IDs but malicious payloads, (2) block ECUs on the bus by sending messages with a lower-priority ID, or (3) deceive an ECU with fabricated messages after blocking the transmission from the legitimate ECU.

5 System Design

5.1 Hammering Mechanism

Within each CAN bit, there's a significant idle period before the PHASE_SEG1 segment, ranging from 50% to 75% of the bit time before the bit's sampling point. ECU can send multiple high and low signals in this area without interfering with standard CAN controller operations. In this paper, we refer to this technique as *hammering*. To send as much data as possible during this idle period, we can use on-board GPIO ports supported by most MCUs to read and write directly to the MCU (i.e., to access the physical-layer CAN_H and CAN_L signals), thereby maximizing data transmission speed [8]. The GPIO setup can be found in Appendix C.

The sender and receiver rely on the embedded CPU clock and timer to synchronize the duration of each CAN bit. The duration depends on the CAN bus speed, which is fixed and known a priori. On both sides, a General Purpose

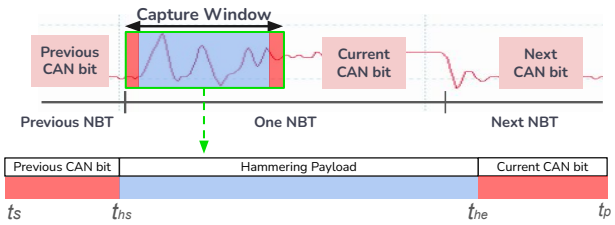


Fig. 4: Receiving hammering data in the capture window

Timer (GPT) is configured with one NBT as the period. On the sender's side, at the beginning of each NBT, the sender starts sending hammering bits at time t_s. Due to the latency of the function call on the sender side, the precise start time of the first hammered bit will be located at t_{hs}. The hammering has to stop before the sampling time to maintain proper CAN communication and backward compatibility, and the sender will set the output port to the current CAN bit until the end of the CAN bit time. Between t_s and t_{hs}, the timer initialization and the GPIO writing process are latent, leaving a gap which consists of the previous CAN bit.

On the receiver side, the receiver starts reading the GPIO port after detecting the first transition at the SOF from 1 to 0. The receiver will trigger the interrupt immediately and synchronize the following receiving bit duration. When the sender starts sending hammered bits, there will be a latency between t_s and t_{hs} due to hardware performance. The previous CAN bit will keep its level until the first hammered bit in the next NBT. On the receiver side, the timer is synced to each NBT start time, so the receiver will receive the previous CAN bit between t_s and t_{hs}. Because the receiver also needs to store the current CAN bit, the capture window W_{cap} is designed to extend from the hammering end time t_{he} to the sampling time t_p. At t_p, the receiver will read the GPIO port as the current CAN bit, and keep the state until the timer stops at the end of NBT. The capture window should be empirically tuned to end before the sampling time.

To do so, the sender starts sending the encoded message at the beginning of every CAN bit transmission and stops before the sampling point to avoid overwriting the CAN bit and uphold backward compatibility. Because of the latency t_{hs} - t_s, the receiver captures data from t_s to t_p, includes the previous CAN bit, the hammered data bits, and the current CAN bit at the end of the capture window. The design of the hammering mechanism and capture window is shown in Fig. 4. As a result, the hammering time range should follow this constraint:

$$t_s < t_{hs} < t_{he} < t_p < t_s + T_{bit} \tag{1}$$

where T_{bit} is the NBT, also the total duration of a legacy CAN bit. The definition of each symbol is shown in Table 2.

Table 2: Summary of Timing Parameters used in CANdy

Symbol	Description	Definition / Constraint
T_{bit}	Total duration of a nominal CAN bit (NBT)	1/Baud Rate
t_s	Internal timer start time	0
t_{hs}	Hammering start time	$t_s + L_{\text{gap}}$
t_{he}	Hammering end time	$t_{he} < t_p$
t_p	CAN sampling point	$0.5T_{\text{bit}} \ldots 0.75T_{\text{bit}}$
W_{cap}	Receiver capture window	$[t_s, t_p]$

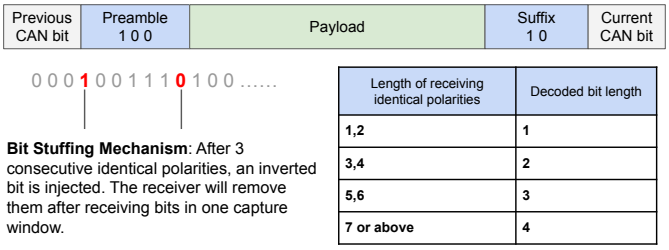


Fig. 5: The encoding and decoding processes rely on preamble and suffix to construct a hammering packet, and high-speed oversampling to capture every bit at the receiver side.

To maximize the number of hammered bits in one NBT, we need to determine the available time window between the start of NBT t_s and the sampling point t_p for hammered bit injection. Also, we have to determine the minimum hammered bit duration δ_{min} (that is achievable by hardware limitations) to maximize the hammering bit rate over the hammering length. As discussed in Section 6.1, the control group ensures that the legacy CAN transceiver continues to capture the CAN bit during hammering.

We define the hammering sequence duration D_h as:

$$D_h = N_h \cdot \delta \tag{2}$$

where δ is the duration of a single hammered bit, N_h is the number of hammered bits. For the legacy CAN transceiver to remain operational, D_h must not exceed the bit sampling point P_s of the controller:

$$D_h < P_s \cdot T_{bit} \tag{3}$$

We will discuss how to empirically determine the sample point t_p and the maximum hammering sequence duration $D_{h,max}$ in Section 6.3.

Algorithm 1: Hammering Encoder

```
Input: M, T      // 128-bit msg, threshold
Output: E[p][*]
p, c, cons, pv ← 0, 0, 0, −1;
foreach byte B ∈ M do
    foreach bit b ∈ B (MSB to LSB) do
        if c = 0 then
            Append 100;
            c ← 4; cons ← 2; pv ← 0;
        if b = pv then
            cons ← cons + 1;
        else
            cons ← 1; pv ← b;
        if cons = 3 then
            Append ¬b;
            c + +, cons ← 1, pv ← ¬b;
        Append b; c + +;
        if c ≥ T − 3 then
            Append 10; Store
            E[p]; p + +, c ← 0;
```

Algorithm 2: Hammering Decoder

```
Input: A, F      // Raw array, length
Output: D[*]   // Restored bitstream
s ← FindFwd(100, A); e ←
  FindBwd(10, A);
if s = NULL ∨ e = NULL then
  return FAIL;
cons ← 2, pv ← 0, i ← s;
while i < e do
    if cons = 3 then
        pv ← A[i]; cons ← 1; i ← i+1;
        continue;
    Append A[i] to D;
    if A[i] = pv then
        cons ← cons + 1;
    else
        cons ← 1; pv ← A[i];
    i ← i + 1;
return D;
```

5.2 Encoding and Decoding

To ensure signal reconstruction accuracy and prevent synchronization loss during transmission, a custom framing protocol was implemented. The encoding process partitions the raw 128-bit AES-CMAC into discrete packets constrained by a specific threshold. To prevent phase errors and ensure a transition within SYNC_SEG at the start of each NBT [16], we use a 3-bit preamble (100_2) to establish a timing baseline for the receiver's decoding. To mitigate clock drift caused by long sequences of identical bits, we used bit stuffing to encode and decode hammered bits. Similar to the CAN bus protocol, the encoder monitors the bitstream and injects an inverted "stuff bit" after five consecutive identical polarities. This forces a signal transition and facilitates edge-based resynchronization at the receiver. Each hammering sequence is terminated with a 2-bit suffix (10_2) to ensure clear boundaries before the sampling point and current CAN bit signal. For data decoding, we directly capture the data within the capture window, identify the preamble and suffix, remove stuffing bits, and extract the payload. Each hammering sequence in one NBT is called a "packet". Its structure and bit-stuffing mechanism are depicted in Fig. 5. The pseudocode of encoding and decoding is demonstrated in Algorithm 1 and 2.

Oversampling in data acquisition. Compared to the sender side, the receiver has to store the incoming bits read from the Port Input Data Register (PIDR) and store them immediately into on-board memory, which takes longer than the sender's writing method. Assume the receiver's fastest sample rate is f_s. According to the Nyquist–Shannon sampling theorem [30], f_s should be at least twice the bus speed to sample all the data. Without hard synchronization, prolonged messages will cause a phase error. We extended the sending bit duration on the sender's side to ensure the f_s was more than twice the sender's hammering bit rate. This modification ensures the receiver will not lose any hammered bit in the capture window W_{cap}. Since the hammered bit duration is extended, we

Table 3: Encoding and decoding logic for 500kbps ($N_h = 4$).

Logical Bits	**No Transition** $(0 \to 0, 1 \to 1)$	**Transition** $(0 \to 1, 1 \to 0)$
00	0 toggles (no change)	1 toggle
01	2 toggles	3 toggles
10	4 toggles	5 toggles
11	2 toggles + first toggle repeat	1 toggle + first toggle repeat

can decode them into four cases after counting the length of receiving identical polarities (see Fig. 5).

500 kbps data encoding and decoding. At 500 kbps, we have limited space to inject hammered bits since each NBT is only 2 μs. Previous methods include a 5-bit preamble and suffix, which nearly exceed the sample point. To successfully send a full 128-bit AES-CMAC in one legacy CAN packet, we should inject at least two logical hammered bits into each NBT, for a total of at most $64 \cdot 2 = 128$ bits. The encoding process generates a 3-bit or 4-bit sequence for the encoded payload based on the combination of the previous CAN bit, hammered bits, and the current CAN bit. The algorithm for data encoding and decoding is shown in Table 3. Decoding is the reverse of the encoding method. For each capture window, we calculate the intercepted bits, determine the times of state changes and bit repetitions, and decode them into two bits based on the encoding logic. The repeating threshold distinguishes between the decoded bits “01" and “11" based on whether the first toggle bit was repeated.

Examples of data encoding and decoding for 125, 250, and 500 kbps are shown in Appendix D.

5.3 Key Establishment and MAC Generation

Following the definition of RFC-4993 [31], The authentication tag T is generated using the AES-CMAC algorithm, which first derives two 128-bit subkeys, K_1 and K_2, by encrypting a zero-block with a primary secret key K and applying bitwise left-shifts and conditional XOR operations with the constant $R_{128} = \texttt{0x87}$. The message M is partitioned into 128-bit blocks and processed through a CBC-style encryption structure where each block M_i is XORed with the previous ciphertext state C_{i-1} before encryption. For the finalization step, a complete last block M_n is XORed with subkey K_1, whereas a partial or empty block is first padded with a 10^j bit-string and XORed with K_2. This final block is then XORed with the cumulative state C_{n-1} and encrypted one last time to produce the secure 128-bit MAC tag T. See Appendix E for more detail.

5.4 Integrity Verification

To follow best industry practices, we implemented AUTOSAR SecOC’s AES-CMAC operations in both the sender and receiver ECUs, allowing the receiver to verify that the message from the sender has not been tampered with. The specific

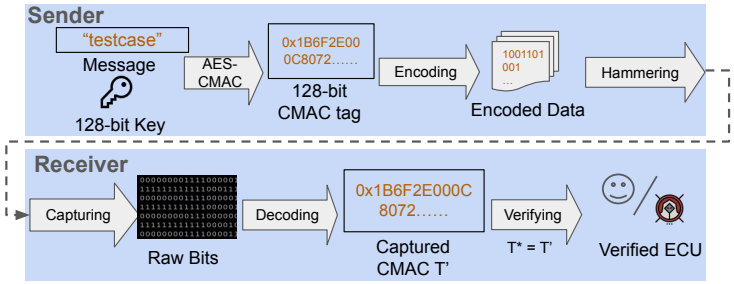


Fig. 6: System architecture of `CANdy` demonstrated with an example

key used for CMAC generation is a 128-bit key K defined as a hexadecimal array. To demonstrate this context, an example is depicted in Fig. 6 where a message, "testcase" (in ASCII), is processed. The function computes the 128-bit CMAC tag T from this message and the specified key. Then, we encoded the CMAC tag as a sequence of raw bits using the method described in the previous section. On the receiver side, the captured signal will be intercepted, decoded, and converted into a bit sequence representing the message. The verification process takes as input the same 128-bit symmetric key K, the decoded message M to be verified, the length of the message $|M|$, and the MAC we captured on the receiving side, denoted as T'. The receiver computes a new CMAC tag, T^*, from the received message M, and then compares T^* with T'. If the two tags are identical, the system returns "VALID", indicating that the message has not been tampered with and originates from a trusted ECU. Otherwise, if T^* and T' differ, the algorithm returns "INVALID", signifying that the message's integrity has been compromised.

5.5 Detection and Prevention

Typically, the ECUs implementing `CANdy` will all have hammering data in the AES-CMAC integration. The receiving message can be categorized as having or not having hammering data in the CAN data field.

CAN message without hammered bits. If the receiver captures a CAN message without hammered data, the receiver will read the hammering duration as the current CAN data bit; this will cause "INVALID" in the verification process described in the previous subsection.

CAN message with hammered bits. If the receiver captures a CAN message with hammered bits containing the MAC, the message will only be validated if it is coming from a legitimate sender. Since the attacker does not know the 128-bit symmetric key for AES-CMAC, it makes brute-force guessing computationally infeasible and secures the authenticity of generated MACs. For a successful attack, the adversary might only replay the same message as the previously captured one if no freshness value (FV) is used. In AUTOSAR SecOC, the FV is a counter or timestamp used to prevent replay attacks. The FV is also used for

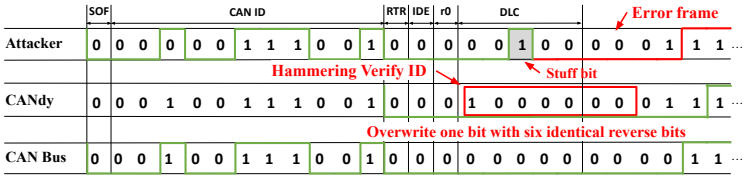


Fig. 7: Bus off the malicious ECU by overwriting one bit with six identical reverse bits, which will continue to cause the bit stuffing error

MAC generation, so if the receiver receives a replay message, the FV embedded in its MAC will be lower than the receiver currently expects. The message will be rejected as invalid or stale, effectively preventing the replay attack. Currently, SecOC Profile 2 does not use FVs which would allow replay attacks to happen.

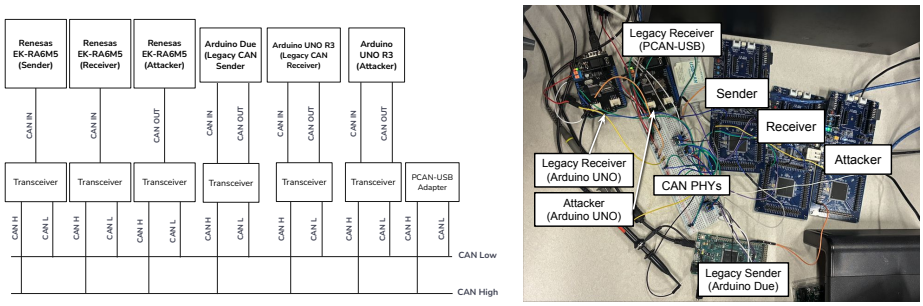
Prevention. Once an attacking node has been identified, the CAN protocol error-detection mechanism can be exploited to force an ECU into a bus-off state so that the ECU stops sending and receiving messages. This strategy uses pin multiplexing or instantaneously overwrites the signal on GPIO lines connected to the CAN bus, allowing a compromised ECU's MCU to overwrite data on the physical CAN bus lines. In the current prototype of `CANdy`, the hammered data starts from the first CAN bit in the data field. We can use the first subsection of the hammered bits to transfer verification information for the sender's CAN ID, thereby enhancing the safety of the hammering method. Also, instead of using the data field in one CAN frame, we can hammer data into the DLC field. We can at least send $4 \cdot 2 = 8$ logical hammered bits in the DLC field, since we can only encode 2 raw bits per CAN bit at 500 kbps. Previous research shows that we can send six identical CAN data bits to cause a Bit Stuffing Error [25], as shown in Fig. 7. Repeated injection of these corrupted frames can quickly increase the target ECU's internal Error Counter past the critical threshold (usually 255), triggering a permanent bus-off state and rendering the ECU unable to send or receive any further messages.

6 Evaluation

6.1 Experimental Setup

The testbed comprises three components: the sender, the receiver, and the control group. The sender and the receiver both use the Renesas EK-RA6M5 board, which can be developed with the RA Flexible Software Package (FSP). The system structure and device implementation are shown in Fig. 8a and Fig. 8b, respectively.

Sender. Each port on the EK-RA6M5 microcontroller, such as PORT5, utilizes key registers to control its pins: the Port Output Data Register (PODR) sets a pin's output level when configured as output, the Port Direction Register (PDR) defines a pin as either an input (0) or output (1), and the Port Mode Register



(a) Schematic of testbed (b) `CANdy` prototype

Fig. 8: Experimental Setup of `CANdy`

(PMR) switches a pin between GPIO and its peripheral functions. For instance, to output a high or low signal on pin P53 (bit 3 in PORT5), the sender must first set it to GPIO mode (PMR3 = 0) and configure it as an output (PDR3 = 1); then, the third PODR (PODR3) controls the actual output level.

Receiver. Similar to the sender, to determine the current state (high or low) of an input pin, we used the Port Input Data Register (PIDR) rather than the native framework to read the port on the EK-RA6M5 board. The PIDR is read to obtain the pin's real-time input status. On the receiver side, reading the PIDR bit 3 of Port 5 would indicate the input state of pin P53.

Control Group. To evaluate the compatibility of the `CANdy` hammering method with legacy CAN protocol transmissions, we implemented two reference devices as control receivers: Arduino UNO R3 with ATmega328P MCU and MCP2515 CAN Controller [1], the PCAN-USB Adapter (model IPEH-002022) with SJA1000 CAN Controller [21] and PEAK-System's Linux driver [22]. These devices ensure that the proposed method does not interfere with standard CAN communication.

Attack Simulation. We analyzed CAN messages collected from four diverse production vehicles (Vehicles A–D) from the same OEM, built between 2016 and 2019. The vehicle types include a luxury sedan (Vehicle A), a compact SUV (Vehicle B), a full-size SUV (Vehicle C), and a full-size pickup truck (Vehicle D). Our analysis used data from the two distinct CAN buses in each vehicle. We send simulated CAN messages with an Arduino Due based on the AT91SAM3X8EA MCU [2], and PCAN-USB Adapter [21]. For the restbus simulation, benign CAN traffic was injected using data collected from Vehicle A. The evaluation is conducted at different typical bus speeds (125, 250 and 500 kbps), transmitting 1,000 CAN packets, and without influencing the legacy CAN transceiver in the Arduino Uno R3 CAN controller and the PCAN-USB Adapter.

6.2 Evaluation Metrics

The evaluation of `CANdy` is structured into four components: Hammering sequence duration measurement, system accuracy, operational latency, and mem-

ory usage. We varied the number of hammered bits N_h to evaluate their impact on both system accuracy and legacy CAN transmissions. On the sending side, the sender first computes an AES-CMAC of the message using a pre-shared key K, encodes this authentication tag T into a hammering message, and transmits it onto the CAN bus. On the receiving side, the receiver acquires the raw bits present on the bus within a capture window W_{cap}, decodes and validates the result by comparing it to an AES-CMAC recomputed locally with the same pre-shared key K. A confusion matrix for the experiments is depicted in Table 11. We measured performance primarily using *Accuracy*, *Recall*, *F-1 Score*, and *Bit Error Rate (BER)*, with and without the presence of an attacker and restbus traffic. The attacker simulates a compromised ECU without hammering ability that sends a fabricated message with the same CAN ID as the sender.

6.3 Hammering Sequence Duration Measurement

To optimize the hammering duration (D_h) while avoiding synchronization errors, we analyzed the impact of signal flipping on legacy CAN receivers across different sample points. Based on hardware limitations, we empirically determined a minimum hammered bit duration of $\delta = 96\ ns$, which allows for stable signal distinction and provides a baseline for incrementally expanding D_h while maintaining compatibility with standard legacy ECUs. We found that the optimal D_h is close to 50% of the NBT in the current control group. This result aligns with the testbed devices setting: the Arduino Uno with MCP2515 maintains its default 50% sample point, and the PCAN adapter's sample points are set to 50% and 75%. (as shown in Table 4). The detail of measure hammering sequence duration can be found in Appendix F.

Table 4: Tuning hammering sequence duration by sending alternating bits incrementally in 125 and 250 kbps data rate. If PCAN-USB and Arduino are both able to capture legacy CAN bits, which represent the D_h is available. Noted that various encoding method deployed at 500 kbps, which are compatible with legacy CAN receivers.

Speed	D_h	**PCAN**	**Arduino**
125 kbps	3.648 μs (45.6%)	✓	✓
	3.840 μs (48.0%)	✓	✓
	4.032 μs (50.4%)	✓	✓
	4.183 μs (52.3%)	✓	✗
	4.320 μs (54.0%)	✓	✗
250 kbps	1.791 μs (44.8%)	✓	✓
	2.000 μs (50.0%)	✓	✓
	2.080 μs (52.0%)	✗	✗
500 kbps	1.000 μs (50.0%)	✓	✓

Table 5: Hammering Results: Detailed Accuracy, Recall, and BER across scenarios.

Speed (kbps)	N_h	**S & R Acc/Rec**	**+Restbus Acc/Rec**	**+Attacker Acc/Rec**	**A+R F-1**	**A+R BER**	**Arduino 50%**	**PCAN 75%**
125	19 bits	1.000/1.000	1.000/1.000	1.000/1.000	1.000	0.000	✓	✓
	20 bits	1.000/1.000	1.000/1.000	1.000/1.000	1.000	0.000	✓	✓
	21 bits	**1.000/1.000**	**1.000/1.000**	**1.000/1.000**	**1.000**	**0.000**	✓	✓
	22 bits	1.000/1.000	1.000/1.000	1.000/1.000	1.000	0.000	✗	✓
	23 bits	1.000/1.000	0.837/0.837	0.833/0.828	0.906	0.143	✗	✓
	24 bits	1.000/1.000	0.939/0.939	0.946/0.943	0.970	0.044	✗	✓
	25 bits	1.000/1.000	1.000/1.000	1.000/1.000	1.000	0.000	✗	✓
	26 bits	0.882/0.882	0.886/0.886	0.897/0.891	0.942	0.102	✗	✓
250	10 bits	1.000/1.000	1.000/1.000	1.000/1.000	1.000	0.000	✓	✓
	11 bits	**1.000/1.000**	**1.000/1.000**	**1.000/1.000**	**1.000**	**0.000**	✓	✓
	12 bits	0.765/0.765	0.719/0.719	0.692/0.692	0.817	0.312	✗	✓
	13 bits	0.001/0.001	0.001/0.001	0.005/0.000	0.000	1.000	✗	✓
	14 bits	0.000/0.000	0.000/0.000	0.025/0.000	0.000	1.000	✗	✓
500	**4 bits**	**1.000/1.000**	**1.000/1.000**	**1.000/1.000**	**1.000**	**0.000**	✓	✓

6.4 System Accuracy

Using the definitions from Table 11, a high recall rate indicates that the receiver is effectively capturing the raw bits and correctly decoding them from the capture window, and a high accuracy represents that the system can verify the incoming message correctly without the influence of the restbus and attacking traffic. Our analysis focuses on how variations in the hammering rate, the sender's hammered bits, and the receiver's sampling width affect this recall. At a 125 kbps bus speed ($T_{bit} = 8\ \mu s$), the system demonstrated robust performance with N_h below 21 and maintained perfect recall as N_h increased up to 25 (see Table 5). If we consider the 50% sample point, the proper choice is $N_h = 21$, since it will not overlap with the sampling point. The restbus and the attacker have a small influence on this speed, as the BER is below 0.15 in testing with most N_h.

When using a bus speed of 250 kbps ($T_{bit} = 4\ \mu s$), the parameter N_h is reduced to approximately 11 bits. For values beyond this threshold, recall performance begins to decline and ultimately reaches zero when D_h is set to 13 bits (see Table 5). Furthermore, the legacy CAN transceiver cannot correctly receive frames when D_h is set to 12 bits, since the hammering sequence duration reaches the sample point.

At the highest tested bus speed of 500 kbps ($T_{bit} = 2\ \mu s$), the system's resilience to hammering was significantly reduced; the maximum effective N_h was only 4 bits. In this case, we are strictly sending bits according to the data encoding we discussed in Section 5.2. `CANdy` can still reach 100% recall, accuracy, and F-1 Score without bit error, as shown in Table 5, which demonstrates that `CANdy` works flawlessly under all typical CAN bus speeds.

6.5 Operation Latency

To characterize the temporal performance of `CANdy`, we decompose the end-to-end latency (T_{total}) into constituent components representing the sender, network, and receiver stages. Given the CPU frequency $f_{cpu} = 200$ MHz for the Renesas EK-RA6M5, the latency in microseconds (μs) is derived from the measured CPU cycles (C) as follows:

$$t = \frac{C}{f_{cpu}} \times 10^6 \tag{4}$$

The total end-to-end latency is modeled as:

$$T_{total} = T_{enc} + T_{tx} + T_{bus} + T_{net} + T_{rx} + T_{ver} \tag{5}$$

where T_{enc} is the MAC calculation and data encoding, T_{tx} is the transmission time, T_{bus} is the duration required for a single CAN frame to be transmitted on the bus, which is determined by the CAN bus bitrate and the length of the frame. T_{net} is the physical propagation delay, T_{rx} is the receiving and decoding duration, and T_{ver} is the MAC verification stage.

The MAC calculation and encoding latency on the sender side was computed as the median latency measured over 1,000 frames, which was approximately 98,992 CPU cycles, equivalent to 494.97 μs. The transmission latency is determined by the CAN Bus rate and data length; for a 500 kbps CAN Bus, we use a 64-bit data frame as an example. The network latency was relatively negligible at 165 ns. On the receiver side, the receiving and decoding latency was 20,369 cycles (101.84 μs), and the MAC verification step took 2,097 cycles, translating to 10.48 μs. The breakdown of the latency components is shown in Table 6.

Table 6: Decomposed Latency of `CANdy` Protocol Components. Note that 500 kbps was selected as an illustrative example.

Stage	**Component**	**Cycles (C)**	**Time (μs)**	**Total (%)**
Sender	MAC Calc. & Encoding (T_{enc})	98,992	494.97	60.1%
Bus	Transmission (T_{bus})	43,200	216.00	26.2%
Network	Propagation (T_{net})	–	0.17	<0.1%
Receiver	Decoding (T_{rx})	20,369	101.84	12.4%
	Verification (T_{ver})	2,097	10.48	1.3%
Total		**164,658**	**823.46**	**100%**

Based on a median of 1,000 transmitted frames, T_{enc} and T_{tx} combined (Total Sender Latency) account for 267,868 cycles ($1,339.34\ \mu s$). On the receiver side, T_{rx} and T_{ver} require 101,271 cycles (506.35 μs) and 96,843 cycles (484.21 μs), respectively. As a result, the receiver-side latency can be calculated as $T_{rx} + T_{ver} = 112.32 \mu s$. A detailed breakdown of these components is provided in Table 6.

The real-time feasibility of `CANdy` is assessed by comparing T_{total} to the network transmission time (T_f) of a standard 108-bit CAN 2.0 frame:

$$T_f = \frac{N}{R} \tag{6}$$

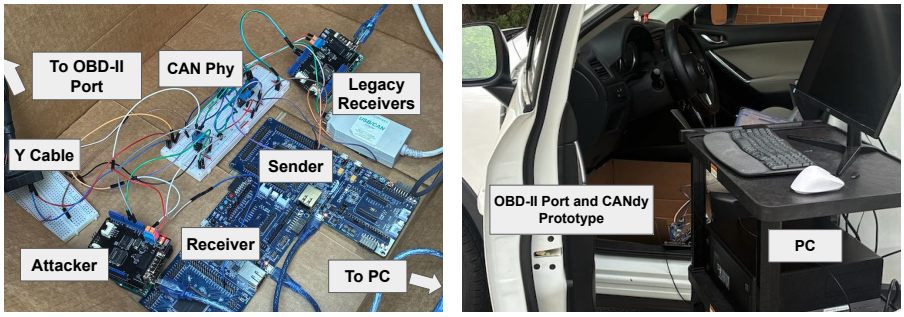
where $N = 108$ and R is the bus speed. At $R = 500$ kbps, the total frame duratoin T_f is 216 μs. Since our total receiver-side processing latency ($T_{rx} + T_{ver} \approx 112$ μs) is lower than T_f, the system achieves line-speed verification. This ensures that each message is authenticated before the subsequent frame arrives, meeting strict real-time deadlines without requiring extensive buffering. We acknowledge that related works (see Table 1) employ varying definitions of latency, including ❶ MAC processing latency [15, 34] ❷ Decoding and signal processing delay [6] ❸ Delivery delay [18, 20, 35] ❹ End-to-End latency [23] ❺ Worst-case detection latency for Instruction Detection System (IDS) [13]. We use the processing latency of `CANdy` as the metric for comparison with related work, as it represents the critical path for per-frame MAC verification on the receiver side. Furthermore, while our current results rely on a software-based implementation, modern automotive microcontrollers (e.g., Infineon Aurix, NXP S32K) feature dedicated Hardware Security Modules (HSM) or AES accelerators. Implementing `CANdy` on such hardware would reduce the cryptographic overhead from 112 μs to negligible levels, making the system viable for high-speed real-time applications.

6.6 Memory Usage

The memory usage of the hammering and legacy methods differs significantly, particularly on the sender side. Data hammering requires more RAM and flash memory to temporarily store encoded and decoded data and to provide space for AES-CMAC calculations. However, when we consider the resources of the Renesas EK-RA6M5 boards—which have 2048 kB of FLASH and 512 kB of RAM—the memory overhead is manageable. The legacy method uses only a fraction of these resources, as detailed in Table 7.

Table 7: Memory Usage Comparison Between Hammering and Legacy Implementations

Component	**RAM (KB)**	**Flash (KB)**
Sender with `CANdy`	20.12 / 512	12.72 / 2048
Receiver with `CANdy`	7.69 / 512	16.02 / 2048
Sender with legacy CAN	1.63 / 512	5.55 / 2048
Receiver with legacy CAN	1.60 / 512	4.74 / 2048



(a) Prototype connected to OBD-II Port (b) `CANdy` tested with 2014 Mazda CX-5

Fig. 9: On-Vehicle Testing of `CANdy` prototype

6.7 On-Vehicle Testing

To verify that the `CANdy` system's accuracy meets real-world conditions and withstands stress testing with real CAN traffic, we connected the prototype to the CAN bus via the OBD-II port of a 2014 Mazda CX-5 (See Fig. 9a and Fig. 9b). In the baseline sender scenario, the system achieved near-perfect performance, with an F1 Score of 99.8%, successfully processing 1,000 frames with minimal noise interference. Under the attacker scenario, the system maintained robust defensive capabilities, with an Accuracy of 93.3% and an F1 Score of 96.7% (See Table 8).

Table 8: On-Vehicle Testing Results: Comparison of Sender/Receiver and Attacker Scenarios

Scenario	Total	Accuracy	Recall	F1-Score
Sender and Receiver	1,000	0.998	0.998	0.998
Attacker	1,000	0.939	0.937	0.967

7 Discussion

Evaluation results confirm that `CANdy` successfully injects AES-CMAC payloads into a single CAN frame with high fidelity. We observed a direct trade-off between bus speed and hammering capacity: the optimal hammering sequence duration for 100% accuracy under concurrent restbus and attacker traffic was 21, 11, and 4 bits at 125, 250, and 500 kbps, respectively. To mitigate capture-window shifts caused by cycle delays at 500 kbps, future work will focus on dual-timer synchronization, migration to higher-frequency CPU platforms, or implementation of 8b/10b encoding [37] to improve DC balance and code rates. By integrating

with AUTOSAR SecOC, CANdy ensures the authenticity and integrity of messages exchanged between sender and receiver ECUs, providing these properties on low-cost commercial hardware. The software-based implementation achieves a competitive processing latency of 112 μs, ensuring real-time constraints are met by offloading security overhead from the CAN Bus bandwidth to the CPU.

While CANdy protects against adversaries sending malicious CAN 2.0A frames from compromised ECUs, we do not prevent attacks against adversaries capable of accessing sensitive memory and hardware peripherals directly. If an attacker compromises the ECU's CAN or GPIO interfaces used by CANdy, they could directly access the CAN bus via the CAN_RX and CAN_TX lines and inject arbitrary CAN signals. With access to the ECU's memory, an attacker could obtain the cryptographic key used by CANdy or the memory space of the compiled binary data. However, modern ECUs have implemented various virtualization techniques to mitigate this threat and prevent exposure of the CAN controller and sensitive memory space. For example, the Renesas RH850/U2x MCU family, a high-performance automotive processor line, uses "Virtual Device Extensions" (VDEs) enabling hypervisor-level access control to physical peripherals in virtualized applications [10]. In low-cost hardware lacking the computational resources to support virtualization, ECU designers have relied on hardware-supported isolation for access-control systems. Memory Protection Units (MPUs) and the ARM TrustZone technology provided in cost-effective ARM processors, such as the Cortex-M33, are two such features that provide strict memory and hardware isolation control from user-level applications [3]. These strict access-control systems protect sensitive memory and hardware peripherals from user-space applications, mitigating bit-level manipulation of CAN frames while restricting an adversary's ability to access the keys needed to compute valid authentication codes.

8 Conclusion

In this paper, we introduced CANdy, a physical-layer encoding technique that enables the retrofitting of robust message authentication onto legacy CAN systems. By exploiting the sub-bit timing "in-between zone" before the sampling point, CANdy successfully injects full AUTOSAR-compliant AES-CMACs into standard CAN frames without altering the bus topology or violating protocol standards. Our extensive evaluation on commercial Renesas hardware demonstrates that CANdy achieves 100% decoding accuracy and a zero Bit Error Rate (BER) across standard automotive bus speeds of 125, 250, and 500 kbps. Unlike existing solutions that require additional frames or truncated MACs, our approach incurs 0% bus load overhead, preserving valuable network bandwidth for critical vehicle operations. CANdy also achieves 99.8% and 93.9% accuracy while implementing without and within attacker ECU during on-vehicle testing, proving the possible implementation in the real-world environment. Ultimately, CANdy proves that high-fidelity security and legacy compatibility are not mutually exclusive, offering a scalable path for securing the next generation of connected vehicles.

Acknowledgments

This project is partially supported by the Clemson University Creative Inquiry program, as well as a gift from the BMW Group.

References

1. Arduino: Arduino UNO R3 Documentation. Arduino (2024), https://docs.arduino.cc/hardware/uno-rev3/, accessed: 2026-02-22
2. Arduino AG: Arduino Due Board. https://docs.arduino.cc/hardware/due/ (2024), accessed: 2026-02-22
3. Arm Limited: Cortex-M33. https://developer.arm.com/Processors/Cortex-M33 (2024), accessed: 2026-07-20
4. AUTOSAR: AUTOSAR Secure Onboard Communication (SecOC). Tech. Rep. R20-11, AUTOSAR (2020), https://www.autosar.org/fileadmin/standards/R20-11/FO/AUTOSAR_PRS_SecOcProtocol.pdf
5. AUTOSAR: AUTOSAR – Automotive Open System Architecture (2025), https://www.autosar.org/, accessed: 2025-07-30
6. Bozdal, M., Samie, M., Aslam, S., Jennions, I.: Evaluation of can bus security challenges. Sensors **20**(8), 2364 (2020)
7. Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T.: Comprehensive experimental analyses of automotive attack surfaces. In: 20th USENIX security symposium (USENIX Security 11) (2011)
8. de Faveri Tron, A., Longari, S., Carminati, M., Polino, M., Zanero, S.: Canflict: Exploiting peripheral conflicts for data-link layer attacks on automotive networks. In: Proceedings of the 2022 ACM SIGSAC conference on computer and communications security. pp. 711–723 (2022)
9. Foster, I., Prudhomme, A., Koscher, K., Savage, S.: Fast and vulnerable: A story of telematic failures. In: 9th USENIX Workshop on Offensive Technologies (WOOT 15) (2015)
10. Gold, S., Buttle, D.: Zone-ECU Virtualization Solution Platform: Integrated Solution for Automotive Applications. White paper, Renesas Electronics Corporation and ETAS (2022), accessed: 2024-05-20
11. Han, K., Weimerskirch, A., Shin, K.G.: A practical solution to achieve real-time performance in the automotive network by randomizing frame identifier. Proc. Eur. Embedded Secur. Cars (ESCAR) pp. 13–29 (2015)
12. Khateeb, F., Saleh, H., Abdulqader, A.: Manchester coding and decoding generation: Theoretical and experimental design. https://www.researchgate.net/publication/335991655 (2019), available on ResearchGate
13. Kim, S., Yeo, G., Kim, T., Rhee, J.J., Jeon, Y., Bianchi, A., Xu, D., Tian, D.J.: Shadowauth: Backward-compatible automatic can authentication for legacy ecus. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security. p. 534–545. ASIA CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3488932.3523263
14. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., et al.: Experimental security analysis of a modern automobile. In: 2010 IEEE symposium on security and privacy. pp. 447–462. IEEE (2010)

15. Kurachi, R., Matsubara, Y., Takada, H., Adachi, N., Miyashita, Y., Horihata, S.: Cacan-centralized authentication system in can (controller area network). In: 14th Int. Conf. on Embedded Security in Cars (ESCAR 2014). p. 10 (2014)
16. Microchip Technology Inc.: MCP2515 Stand-Alone CAN Controller with SPI Interface. Microchip Technology Inc. (2019), https://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf, dS20001801J
17. Nie, S., Liu, L., Du, Y., Zhang, W.: Over-the-air: How we remotely compromised the gateway, bcm, and autopilot ecus of tesla cars. In: Briefing, Black Hat USA. vol. 91, pp. 1–19 (2018)
18. Nürnberger, S., Rossow, C.: –vatican–vetted, authenticated can bus. In: International Conference on Cryptographic Hardware and Embedded Systems. pp. 106–124. Springer (2016)
19. NXP: Nxp automotive processors — product map. https://www.nxp.com/docs/en/product-selector-guide/BRAUTOPRDCTMAP.pdf (2020)
20. Oberti, F., Savino, A., Sanchez, E., Casasso, P., Parisi, F., Carlo, S.D.: Can-mm: Multiplexed message authentication code for controller area network message authentication in road vehicles. IEEE Transactions on Vehicular Technology **73**(10), 14661–14673 (2024). https://doi.org/10.1109/TVT.2024.3402986
21. PEAK-System Technik GmbH: PCAN-USB: CAN Bus Interface for USB. https://www.peak-system.com/products/hardware/external-pc-interfaces/pcan-usb/ (2024), accessed: 2025-07-29
22. PEAK-System Technik GmbH: PEAK-System Linux Driver. https://www.peak-system.com/fileadmin/media/linux/index.php (2025), accessed: 2025-12-04
23. Pesé, M.D., Schauer, J.W., Li, J., Shin, K.G.: S2-can: Sufficiently secure controller area network. In: Proceedings of the 37th Annual Computer Security Applications Conference. pp. 425–438 (2021)
24. Pesé, M.D., Stacer, T., Campos, C.A., Newberry, E., Chen, D., Shin, K.G.: Librecan: Automated can message translator. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 2283–2300. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3363190
25. Pesé, M.D., Gozubuyuk, B., Andrechek, E., Olufowobi, H., Hamad, M., Shin, K.G.: Michican: Spoofing and denial-of-service protection using integrated can controllers. In: 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (2025)
26. Renesas: V850es/fx3, https://www.renesas.com/us/en/products/microcontrollers-microprocessors/other-mcus-mpus/v850-family-mcus/v850esfx3-32-bit-microcontrollers-non-promotion
27. Robert Bosch GmbH: CAN Specification Version 2.0, Part A (1991), https://www.port.de/fileadmin/user_upload/Dateien_IST_fuer_Migration/CAN20A.pdf, accessed: 2024-05-22
28. Serag, K., Bhatia, R., Faqih, A., Ozmen, M.O., Kumar, V., Celik, Z.B., Xu, D.: ZBCAN: A Zero-Byte CAN defense system. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 6893–6910. USENIX Association, Anaheim, CA (Aug 2023), https://www.usenix.org/conference/usenixsecurity23/presentation/serag
29. Serag, K., Bhatia, R., Kumar, V., Celik, Z.B., Xu, D.: Exposing new vulnerabilities of error handling mechanism in CAN. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 4241–4258. USENIX Association (Aug 2021), https://www.usenix.org/conference/usenixsecurity21/presentation/serag

30. Shannon, C.: Communication in the presence of noise. Proceedings of the IRE **37**(1), 10–21 (jan 1949). https://doi.org/10.1109/jrproc.1949.232969
31. Song, J., Poovendran, R., Lee, J., Iwata, T.: The AES-CMAC Algorithm. RFC 4493, RFC Editor (June 2006), https://www.rfc-editor.org/rfc/rfc4493.html
32. STMicroelectronics: Spc58 b/c/g-lines product selector guide. https://www.st.com/content/ccc/resource/sales_and_marketing/promotional_material/selection_guide/group0/34/0e/4f/b2/1a/49/4f/b6/SPC58%20selection%20guide/files/SGSPC58C.pdf/jcr:content/translations/en.SGSPC58C.pdf (2020)
33. Valasek, C., Miller, C.: Remote exploitation of an unaltered passenger vehicle. Technical White Paper, IOActive (Jul 2015), available at https://www.ioactive.com/wp-content/uploads/pdfs/IOActive_Remote_Car_Hacking.pdf
34. Van Herrewege, A., Singelee, D., Verbauwhede, I.: CANAuth-a simple, backward compatible broadcast authentication protocol for CAN bus. In: ECRYPT workshop on Lightweight Cryptography. vol. 2011, p. 20. ECRYPT (2011)
35. Wang, Q., Sawhney, S.: Vecure: A practical security framework to protect the can bus of vehicles. In: 2014 International Conference on the Internet of Things (IOT). pp. 13–18. IEEE (2014)
36. Wen, H., Chen, Q.A., Lin, Z.: Plug-N-Pwned: Comprehensive vulnerability analysis of OBD-II dongles as a new Over-the-Air attack surface in automotive IoT. In: Proceedings of the 29th USENIX Security Symposium (USENIX Security 20). pp. 949–965. USENIX Association, Boston, MA (Aug 2020)
37. Widmer, A.X.: 8B/10B encoding and decoding for high speed applications. Tech. Rep. RC23408, IBM Research, Yorktown Heights, NY, USA (October 2004), https://dominoweb.draco.res.ibm.com/reports/rc23408.pdf
38. Ziermann, T., Wildermann, S., Teich, J.: Can+: A new backward-compatible controller area network (can) protocol with up to 16× higher data rates. In: 2009 Design, Automation & Test in Europe Conference & Exhibition. pp. 1088–1093 (2009). https://doi.org/10.1109/DATE.2009.5090826

A CAN Primer and AUTOSAR SecOC

A.1 CAN Primer

Start-of-Frame (SOF): This single bit marks the beginning of a CAN message and is always set to 0.
CAN ID: The CAN bus operates as a multi-master, message-oriented system where messages are identified by unique IDs rather than sender/receiver addresses. Lower numerical IDs take priority due to CAN's arbitration method, which relies on wired-AND logic: a dominant bit (0) overrides a recessive bit (1). This allows the highest-priority message to proceed. ECUs can transmit or receive messages associated with different IDs. CAN 2.0A supports 11-bit identifiers, allowing for up to 2,048 distinct message types.
RTR, IDE, and Reserved: These control bits — Remote Transmission Request (RTR), Identifier Extension (IDE), and a reserved bit (r0) — are all set to 0 in CAN 2.0A frames.

Table 9: Comparison of Standard AUTOSAR SecOC Profiles

Profile	MAC Size	Freshness Value Size
Profile 1 (`24Bit-CMAC-8Bit-FV`)	24 bits	8 bits
Profile 2 (`24Bit-CMAC-No-FV`)	24 bits	0 bits
Profile 3 (`JASPAR`)	28 bits	4 bits

Data Length Code (DLC): This field indicates the size of the payload, which can range from 0 to 8 bytes.
Data: This is the payload, containing up to 8 bytes of data.
CRC-15: For error detection, a 15-bit Cyclic Redundancy Check (CRC) is computed over the earlier parts of the frame.
Acknowledgment (ACK) and End-of-Frame (EOF): The ACK field has two bits: the ACK slot and the ACK delimiter. The transmitting ECU sets the ACK slot to 1. If receivers detect no errors in the frame, they respond with a dominant 0 during this slot. Due to wired-AND logic, the transmitter sees a 0 and knows the message was successfully received. If not, it retransmits. The ACK delimiter and the EOF bits are always set to 1.
Inter-Frame Spacing (IFS): After a frame is sent, a brief idle period of at least 3 bits (not shown in Fig. 1a) must pass before the next message, resulting in a minimum of 11 consecutive recessive bits between frames.

A.2 AUTOSAR SecOC

The broad SecOC standard requires three components for secure CAN message generation:
Secret Key: 16-byte symmetric key for AES-CMAC.
Freshness Value: Counter or timestamp to prevent replay attacks, includes Trip Counter, Reset Counter, Message Counter, and Reset Flag.
Message payload: An Authentic I-PDU is a type of AUTOSAR I-PDU that requires protection from unauthorized manipulation and replay attacks. The payload of a Secured I-PDU is created by combining the Authentic I-PDU with an Authenticator, such as a MAC. This Secured I-PDU may also optionally include the Freshness Value that was used to create the Authenticator, which helps prevent replay attacks.

AUTOSAR SecOC provides three standard profiles (as shown in Table 9). Profiles 1 and 3 include dedicated freshness values in the payload. To maximize the utilization of the hammering bits, we choose Profile 2 as our MAC generation standard, which sends payloads without freshness values but supports the inclusion of freshness values during MAC generation.

To compute the MAC, the sender first concatenates the CAN ID, payload, and freshness value, then uses AES128-CMAC to generate the MAC. The MAC will be truncated and transmitted along with the payload, including the truncated freshness value. The receiver reconstructs the full freshness value from its

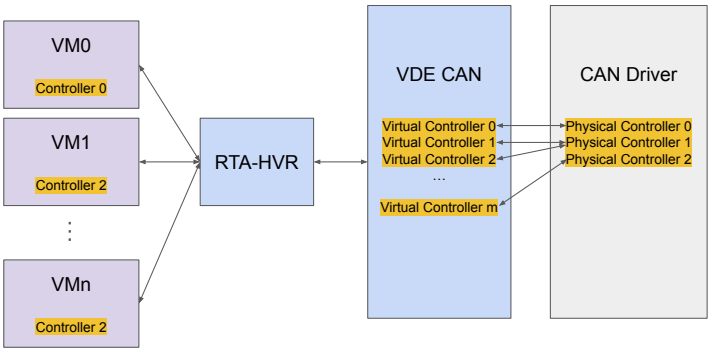


Fig. 10: RTA-HVR (Real-Time Application Hypervisor for Hardware Virtualization) is an automotive software hypervisor by ETAS, paired with Renesas RH850/U2x MCUs. The hypervisor can create multiple isolated Virtual Machines (VMs) on a single ECU.

state and the received bits, then recalculates and verifies the MAC. The message authentication and verification procedure of the AES-CMAC based on the AUTOSAR SecOC standard is shown in Fig. 2b.

B Adversary Constraints

If an attacker compromised one of the integrated CAN Controllers or General Purpose Input/Output (GPIO) interfaces on the ECUs used by `CANdy`, they could directly access the CAN bus via the CAN_RX and CAN_TX lines and inject arbitrary CAN signals. Moreover, an attacker could access the cryptographic key or the memory space of the compiled binary data. However, modern ECUs have implemented various virtualization techniques to mitigate this threat and prevent exposure of the CAN controller and sensitive memory space. For example, contemporary ECU structures, such as the Renesas RH850 MCU, use “Virtual Device Extensions" (VDE) to enable the hypervisor to customize how virtual peripherals are bound to physical hardware, such as additional CAN channels (shown in Fig. 10) [10]. For a low-end solution, the hardware may lack the computational resources to support virtualization. However, for memory isolation, the ARM Cortex-M33 CPU supports Memory Protection Unit (MPU) and TrustZone, a hardware-based security technology that separates a single CPU core into normal and secure zones [3]. This provides a cost-effective approach to protecting sensitive data, including key management, freshness value calculation, and CAN bus data encoding and decoding. To the attacker, computing a valid authentication code without the information in the secure zone is computationally infeasible.

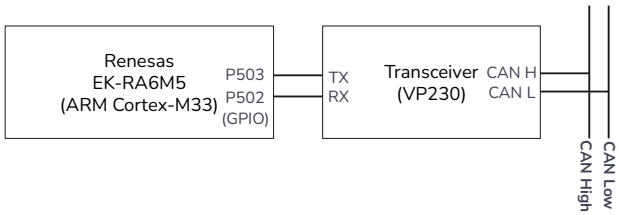


Fig. 11: Example GPIO configuration of the sender

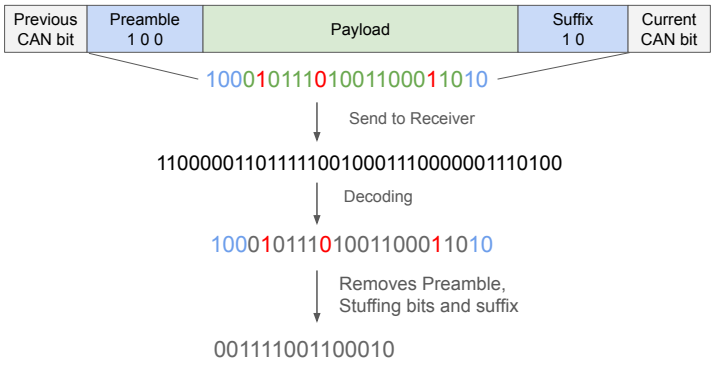


Fig. 12: Example of data encoding and decoding for 125kbps and 250kbps

C GPIO Setup

Fig. 11 illustrates the hardware interface between the `CANdy` sender/receiver platform and the physical CAN bus, and shows the sender configuration as an example. The Renesas EK-RA6M5 board, which integrates an ARM Cortex-M33 CPU core, is connected to a VP230 (SN65HVD230) CAN transceiver through the GPIO pins P502 and P503. Pin P503 is configured as the transmit line (TX) on the sender and the receive line (RX) on the receiver, used to deliver or receive the hammering signals to/from the transceiver. On the sender side, pin P502 is configured as the RX and is used to sync to the start of the CAN frame from the bus. The VP230 transceiver converts the digital GPIO-level signals into differential voltages on the CAN High (CAN_H) and CAN Low (CAN_L) lines of the physical CAN bus.

Table 10: Encoding and decoding logic for 500 kbps ($N_h = 4$).

Logical Bits	**No Transition** ($0 \to 0, 1 \to 1$)	**Transition** ($0 \to 1, 1 \to 0$)
00	0 toggles (e.g. 0-0000-0)	1 toggle (e.g. 0-1111-1)
01	2 toggles (e.g. 0-1111-0)	3 toggles (e.g. 0-1100-1)
10	4 toggles (e.g. 00-1010-0)	5 toggles (e.g. 0-1010-1)
11	2 toggles + first toggle repeat (e.g. 0-0011-0)	1 toggle + first toggle repeat (e.g. 0-0000-1)

D Encoding Example

For 125 kbps and 250 kbps, the end-to-end data encoding and decoding pipeline consists of three distinct stages:

- The input payload is structured by inserting a fixed preamble (100) and suffix (10). Stuffing bits are interleaved inside consecutive three-bit runs of identical bits.
- The framed sequence is converted into physical bit pulses and transmitted via the CAN transceiver. The receiver will capture the hammering bits with a higher data rate and receive the raw bits.
- The receiver then decodes the raw bits based on the consecutive identical bits, strips the 3-bit preamble, 2-bit suffix, and stuffing bits to restore the original payload.

The example of data encoding and decoding is shown in Fig. 12. For 500 kbps, the encoding and decoding will be based on the toggle time of hammering bits in the capture windows. We can encode 4 types of 2-bit logical states based on the transitions of CAN bits and the toggle time of the hammering bits. Examples are shown in Table 10.

E AES-CMAC

The AES-CMAC algorithm begins by deriving two 128-bit subkeys, K_1 and K_2, from the primary secret key K. Let $E_K(0^{128})$ denote the AES encryption of a 128-bit block of zeros. First, an intermediate block L is computed:

$$L = E_K(0^{128}) \tag{7}$$

The first subkey K_1 is derived by a left-shift of L. If the most significant bit (MSB) of L is 1, an XOR operation with a constant R_{128} is applied to ensure the result remains within the finite field $GF(2^{128})$:

$$K_1 = \begin{cases} (L \ll 1) & \text{if } MSB(L) = 0 \\ (L \ll 1) \oplus R_{128} & \text{if } MSB(L) = 1 \end{cases} \tag{8}$$

where R_{128} is the constant value `0x87`. The second subkey K_2 is derived from K_1 using the same shifting and conditional XOR logic:

$$K_2 = \begin{cases} (K_1 \ll 1) & \text{if } MSB(K_1) = 0 \\ (K_1 \ll 1) \oplus R_{128} & \text{if } MSB(K_1) = 1 \end{cases} \tag{9}$$

The message M is partitioned into n blocks $M_1, M_2, \ldots, M_n$, each of 128-bit length. The algorithm processes these blocks iteratively. Let C_i represent the state after processing the i-th block:

$$C_0 = 0^{128} \tag{10}$$

$$C_i = E_K(M_i \oplus C_{i-1}) \quad \text{for } i = 1, \ldots, n-1 \tag{11}$$

Let $|M|$ denote the bit length of the message M. The message is partitioned into n blocks, where $n = \lceil |M|/128 \rceil$. If the message is empty, it is treated as a single partial block ($n = 1$). The treatment of the final block M_n is defined by the following conditional logic:

If $|M| > 0$ and $|M|$ is a multiple of 128, the last block is complete. The block is XORed with subkey K_1 before the final encryption:

$$M_{last} = M_n \oplus K_1 \tag{12}$$

If $|M|$ is not a multiple of 128 (including the case of an empty message), the last block is partial. It is first padded with a bit-string 10^j, where $j = 127 - (|M| \mod 128)$. This padded block, M_n^*, is then XORed with subkey K_2:

$$M_{last} = (M_n \parallel 10^j) \oplus K_2 \tag{13}$$

The final Tag T is then generated by XORing M_{last} with the output of the previous iteration C_{n-1} and performing the final AES encryption:

$$T = E_K(M_{last} \oplus C_{n-1}) \tag{14}$$

F Hammering Sequence Duration Measurement

Tuning hammering sequence duration by inverting CAN bit. We started inverting the current CAN bit and tuning the sample point by incrementally expanding the hammering duration (the red lines in Fig. 13). We can invert the current CAN bit until 48% of NBT without interfering with the legacy CAN receiver (PCAN-USB) when the sample point is set to 50%. However, when the sample point is set to 75% in the PCAN Adapter, it fails to capture the legacy CAN frame at the same inverting length. This occurs because the start bit is changed when the CAN controller fails to detect a signal edge at the beginning of the NBT. If the edge is detected in SYNC_SEG, the sample point and the NBT will not be changed. If the edge is detected before the sample point (in PROP_SEG or PHASE_SEG1), which will cause resynchronization, the start of the bit is changed to the edge position, and the PHASE_SEG1 is extended with δ, the difference of the Synchronization Jump Width (SJW) and the duration between the start of PROP_SEG and t_{he} [16]. If the sample point is 50%, the new sample point remains within the NBT and is read by the CAN controller. If the sample point is 75%, the new sample point will be located at the next CAN bit, and the CAN controller will incorrectly read the next NBT data. Note that

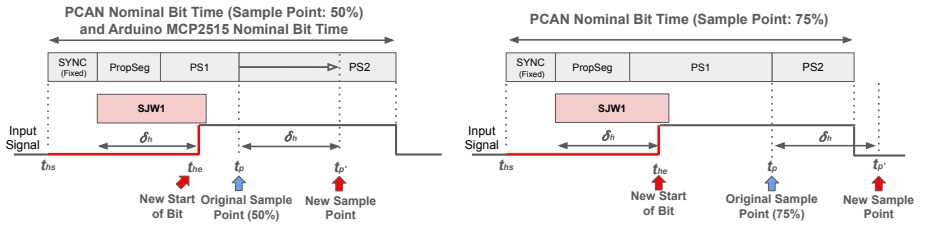
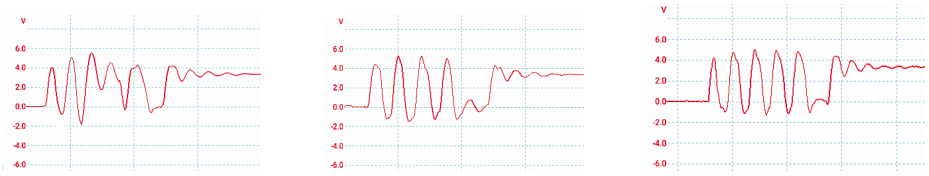


Fig. 13: Flipping the signal caused the synchronization error. The red lines are the flipping duration from the start of NBT to t_{he}. This shows the cases of PCAN-USB Adapter with 50% and 75% sample points, and Arduino Uno R3 MCP2515 with 50% sample points.

the sample point for the Arduino Uno with the MCP2515 CAN Controller is also 50%, which yields the same result as the PCAN adapter with a 50% sample point.

To address this issue, we manually established an initial edge at the start of the NBT by transmitting an alternating bit sequence. This approach ensures that the positions of the edges located in the SYNC period and the sampling points remain invariant during synchronization.

Tuning hammering sequence duration by sending alternating bits. To accurately measure D_h, we need to determine the minimum duration δ_{min} of each hammering bit. From the sender's side, we can manually write bit values to the Port Output Data Register (PODR) port with a sequence of NOP instructions to control the bit duration. As Fig. 14 shows, when sending 10 alternating bits, the signal can be distinguished correctly only for larger than 96 ns due to hardware limitations. Using the same method as for sending an inverted signal, we empirically determined the hammered bit duration as $\delta = 96$ ns, started sending the alternating bits, and tuned the sampling point by incrementally expanding the D_h.



(a) 81ns duration (✗) (b) 91ns duration (✗) (c) 96ns duration (✓)

Fig. 14: Waveform comparison for hammering bit durations of 80 ns, 90 ns, and 96 ns during transmission of a 10-bit alternating pattern (1010101010). The previous CAN bit is 0, and the current CAN bit is 1, which decides the signal level before and after hammering bits. The 96 ns one shows a perfect alternating hammering signal.

G Confusion Matrices

See Table 11.

Table 11: Confusion Matrix for Hammering Length Measurement

Actual	**Predicted / Outcome**	**Notation**
Hammered Valid AES-CMAC	Verified success and the CAN frame is from the sender	TP
	Verified failed, but the CAN frame is from the sender	FN
Hammered invalid AES-CMAC or Non-Hammered	Verified success but the CAN frame is from the attacker	FP
	Verified failed, and the CAN frame is from the attacker	TN