ZBCAN: A Zero-Byte CAN Defense System

Khaled Serag*, Rohit Bhatia*, Akram Faqih*, Muslum Ozgur Ozmen*,
Vireshwar Kumar†, Z. Berkay Celik*, and Dongyan Xu*
*Purdue University, {kserag, bhatia13, afaqih, mozmen, zcelik, dxu}@purdue.edu
†Indian Institute of Technology Delhi, viresh@cse.iitd.ac.in

Abstract

Controller Area Network (CAN) is a widely used network protocol. In addition to being the main communication medium for vehicles, it is also used in factories, medical equipment, elevators, and avionics. Unfortunately, CAN was designed without any security features. Consequently, it has come under scrutiny by the research community, showing its security weakness. Recent works have shown that a single compromised ECU on a CAN bus can launch a multitude of attacks ranging from message injection, to bus flooding, to attacks exploiting CAN's error handling mechanism. Although several works have attempted to secure CAN, we argue that none of their approaches could be widely adopted for reasons inherent in their design. In this work, we introduce ZBCAN, a defense system that uses zero bytes of the CAN frame to secure against the most common CAN attacks, including message injection, impersonation, flooding, and error handling, without using encryption or MACs, while taking into consideration performance metrics such as delay, busload, and data-rate.

1 Introduction

Modern vehicles contain hundreds of sensors and actuators, administered by Electronic Control Units (ECUs), including brake, engine, and steering control units. The most central communication channel among ECUs is CAN. Although reliable and robust against electromagnetic interference, CAN lacks any security measures. Researchers have demonstrated the feasibility of remotely compromising an ECU on the CAN bus [4,36,40,41,57]. With the ever-increasing connectivity of today's vehicles, the ease of such compromises is expected to increase. Several works have shown that a compromised ECU can launch a plethora of attacks, including *message injection, impersonation, and flooding* [4,36,40,41,57]. Moreover, recent works have unveiled vulnerabilities in CAN's error handling mechanism [3,5,30,38,43,47,48,60]. These vulnerabilities allow attackers to deliberately inject collisions, map message sources, control the error states of certain ECUs or even persistently disable them [5,38,47].

To secure CAN traffic, two primary approaches have been proposed. One is the *cryptographic approach*, which relies heavily on cryptographic primitives (e.g., encryption, MACs, and hash functions). [1, 2, 20, 24–26, 42, 44, 45, 54, 56, 59]. Unfortunately, this approach suffers from fundamental issues. The first is its *impact on performance* as cryptographic operations incur an unaffordable processing overhead for most commercial ECUs. Even worse, since the maximum payload length of a CAN message is 64 *bits*, these solutions are forced to either carve out a portion of an already-short message to attach authentication information, dropping the effective data rate, or use a completely different message, doubling the busload. Another issue is the *lack of intrusion confinement*. Since most of these solutions use group keys, if one ECU gets compromised, it can impersonate any node in the group. The last downside is the *lack of incremental deployability* or the ability to incrementally secure messages transmitted by a single ECU, without needing to update all ECUs at once.

The second approach is the *intrusion detection (IDS) approach*, which avoids the group-key problems and the computationally expensive cryptographic operations by delegating all security operations to a super-node that may have special equipment [7, 8, 18, 27, 31–33, 39, 49, 50, 63]. This powerful node uses its abilities to detect traffic anomalies and flag them. However, this approach has its problems. First, IDSs take no measure to stop or prevent attacks. Second, most *CAN IDSs* do not achieve *single-message detection*. Instead, they retrospectively detect *flows* of injected messages. This allows intermittent or gradual intrusions to pass unnoticed and contributes to these IDSs' inability to translate their attack detection into prevention, for a flow of messages is composed of a stream of individual messages. Being unable to determine whether an individual message is malicious or not prevents the IDS from taking action against any of the individual malicious messages that constitute the flow.

Additionally, the entire research field suffers from a *hyperfocus* phenomena. A relatively vast amount of research has been dedicated to *message injection* and its variations in comparison with other attacks. Further, many defenses contradict one another and thus cannot be combined to protect against a more extensive attack-set. These problems have prevented any defense from being widely adopted.

Table 1: How ZBCAN compares with other CAN defense systems.

<table>
<tr><th colspan="2" rowspan="2">Defense Approach</th><th colspan="6">Attacks</th><th colspan="3">Features</th><th colspan="3">Cost</th></tr>
<tr><th>Flood</th><th>Injection</th><th>Replay</th><th>Collision Injection</th><th>Error Passive</th><th>Bus Off</th><th>Incremental Deployability</th><th>Single-Msg Detection</th><th>Intrusion Confinement</th><th>Modifies Message</th><th>Increases Busload</th><th>Processing Overhead</th></tr>
<tr><td colspan="2">Cryptographic 1‡</td><td>X</td><td>✓</td><td>✓</td><td>X</td><td>X</td><td>X</td><td>X</td><td>✓</td><td>X</td><td>X</td><td>✓</td><td>●</td></tr>
<tr><td colspan="2">Cryptographic 2†</td><td>X</td><td>✓</td><td>✓</td><td>X</td><td>X</td><td>X</td><td>X</td><td>✓</td><td>X</td><td>✓</td><td>X</td><td>●</td></tr>
<tr><td rowspan="2">Voltage IDS</td><td>Detection</td><td>-</td><td>✓</td><td>✓</td><td>-</td><td>-</td><td>-</td><td rowspan="2">✓</td><td rowspan="2">X</td><td rowspan="2">-</td><td rowspan="2">X</td><td rowspan="2">X</td><td rowspan="2">○</td></tr>
<tr><td>Prevention</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td rowspan="2">Frequency IDS</td><td>Detection</td><td>✓</td><td>✓</td><td>✓</td><td>-</td><td>-</td><td>-</td><td rowspan="2">✓</td><td rowspan="2">X</td><td rowspan="2">-</td><td rowspan="2">X</td><td rowspan="2">X</td><td rowspan="2">○</td></tr>
<tr><td>Prevention</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td rowspan="2">Clk-Skew IDS</td><td>Detection</td><td>✓</td><td>✓</td><td>✓</td><td>-</td><td>-</td><td>-</td><td rowspan="2">✓</td><td rowspan="2">X</td><td rowspan="2">-</td><td rowspan="2">X</td><td rowspan="2">X</td><td rowspan="2">○</td></tr>
<tr><td>Prevention</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td rowspan="2">ZBCAN</td><td>Detection</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td rowspan="2">✓</td><td rowspan="2">✓</td><td rowspan="2">✓</td><td rowspan="2">X</td><td rowspan="2">X*</td><td rowspan="2">◔</td></tr>
<tr><td>Prevention</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td></tr>
</table>

‡: Approach that uses extra messages to send authentication data.
†: Approach that uses message fields to send authentication data.

*: Sometimes, ZBCAN may cause a minute busload-increase (Sec. 6.3).

We present Zero-Byte CAN, a versatile, low-overhead defense system that uses *zero bytes* of the CAN message fields to protect against several attacks and offers *intrusion confinement*, *incremental deployability*, full *backward compatibility*, and *individual message guarantees*. This last feature allows ZBCAN to translate some of its *detection* abilities, into *prevention* ones, since individual malicious messages could be identified and stopped. ZBCAN does not use message fields, authentication messages, or computationally expensive operations such as encryption. Instead, it uses message timing alone to protect against the most common CAN attacks, including *injection, impersonation, fuzzing, flooding, collision injection, voltage corruption, and bus-off*.

ZBCAN is composed of a trusted *officer* node that can interrupt transmission and several software *agents*, installed on ECUs. The *officer* and each *agent* agree on a secret, endless, and dynamically generated sequence of inter-frame spaces, which the *officer* monitors for every message. The *officer* could be set to issue warnings, interrupt messages, or completely suspend violating nodes. Aside from attaching the *officer* to the bus, ZBCAN does not require any hardware changes. Further, since ZBCAN uses no message fields, it could be combined with solutions that do use them. For inclusivity, we evaluated ZBCAN's performance on a testbed using a real vehicle's data, its security and scalability on a testbed using artificial data, and finally, several security and performance aspects of ZBCAN on a real vehicle. Using ZBCAN, we achieved a *detection rate* of 100% for *injection and replay attacks*, and *prevention rates* of 100%, 100%, 99.4%, 99.33%, and 98.5% for *error-passive, bus-off, collision injection, flooding, and injection attacks*, respectively. We summarize our contributions as follows:

- We present ZBCAN, a versatile defense system that uses inter-frame spaces to defend against the most common CAN attacks, offering both detection and prevention abilities.
- We introduce a new method to suspend any ECU as soon as it starts transmitting a frame called *Instant Bus-Off*. This method could be used to suspend intruding nodes.
- We offer worst-case response time analysis to systems with ZBCAN. We apply our analysis on a real CAN bus and show that all messages are guaranteed to be schedulable.
- We offer a probabilistic security analysis of ZBCAN against different attack types.
- To show its applicability, we evaluate different aspects of our system on a CAN testbed, on a real vehicle's traffic, and directly on a real vehicle's CAN bus.

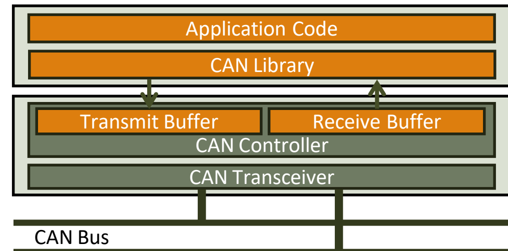

Figure 1: Architecture of an ECU.

2 Background

ECU Architecture. As shown in Fig. 1, an ECU is composed of three components: (1) A micro-controller unit running a vehicle control application. (2) A CAN controller to enable CAN communication and enforce protocol rules. (3) A CAN transceiver to encode the CAN controller's bit-stream into the differential voltage signal used on the CAN bus.

CAN Basics. CAN is a broadcast-based bus that uses a *publish-subscribe* communication model. It uses differential voltage signaling to represent zeros (dominant) and ones (recessive). If a zero and a one are being transmitted by two ECUs at the same time, the zero wins. Messages conclude by sending 7 ones called the End Of Frame (EOF) sequence (Fig. 2). Following EOF, any ECU that wishes to transmit a new frame will have to wait for 3 additional bits called the *Inter-Frame Spacing (IFS)*.

Arbitration and Priority. At the beginning of every CAN message, there is an ID field. Often, an ECU has multiple message IDs, but an ID has only one transmitter. CAN uses the ID as a priority field, with lower ID values having higher priorities. If two ECUs attempt to transmit simultaneously, they first go through an arbitration phase. Each ECU sends one bit at a time and senses the bus; if it is sending one but

Figure 2: CAN frame fields of two back-to-back frames.

senses a zero, it stops transmission. Thus, CAN guarantees that beyond arbitration, there is, at most, one transmitter.

Errors. Whenever a CAN message is being transmitted, every ECU is considered a receiver except for the transmitter. Every ECU keeps two error counters, a *TEC* for errors encountered during transmission, and an *REC* for errors encountered during reception. CAN defines 5 different errors types for which every ECU should watch. If a transmitter encounters an error, its TEC increases by 8. If a receiver encounters an error, its REC increases by 1. Upon encountering an error, a node signals the error by broadcasting an *error frame*.

Error States. For fault confinement, CAN enforces a system of error states. By default, ECUs operate in the *error-active state*. Once their *REC* or *TEC* exceeds 127, they enter the *error-passive state*. If the *TEC* exceeds 255, they enter the *bus-off state*, where they stop communicating with the bus.

Injection Attacks. Attacks where a malicious ECU injects messages into the bus. We broadly consider the following as injection attacks: *(1) Targeted Injection*: Forging and injecting messages that look similar to those transmitted by a specific ECU in charge of certain functions in order to alter those functions. *(2) Replay Attacks:* Replaying one or more messages transmitted by a different ECU. *(3) Random Injection:* Forging IDs randomly or semi-randomly to cause damage or to discover hidden message semantics (e.g. *fuzzing attacks*).

Flooding Attacks. The attacker injects an endless stream of back-to-back high-priority messages to deny other ECUs access to the bus and cause them to drop messages.

Error Handling Attacks. An ever-widening attack surface, these attacks cause a victim ECU to encounter errors using a technique called *simultaneous transmission* in order to achieve various purposes. For instance by accumulating these errors, attackers could push ECUs to the *error passive* or *bus-off* states. These error states could then be exploited to launch persistent DoS attacks, evade voltage intrusion detection systems, or map the network.

Simultaneous Transmission. A technique used by attackers to inject collisions and increase the error counter of a victim. As shown in Fig. 3, the attacker injects a message with the same ID as a target-message exactly at the same time but with a different payload. This causes both the attacker and the victim to experience an error and increase their TEC by 8. For the attacker to synchronize the two messages, they inject one or more *synch messages* with a higher priority slightly before the message, followed by a message with the same ID.

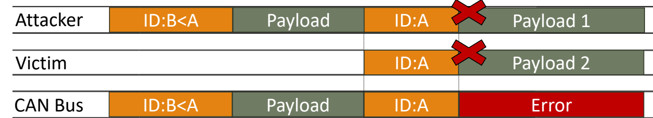

Figure 3: Collision injection.

3 Related Work

Intrusion Detection Approach. To avoid using cryptography, researchers proposed using lightweight Intrusion Detection Systems. Some IDSs rely on traffic features such as message frequencies, lengths, payloads, or clock skews to detect anomalies [6, 27, 39, 49, 50, 63]. Others use physical features such as the unique electrical characteristics of each ECU, manifesting in their transmission voltage levels [7, 8, 18, 32, 33]. Nevertheless, IDSs have their problems. Namely, many of these systems were shown to be evadable [3, 46]. Further, despite the high detection rates they present, most of them do not detect single injections, but flows of N injections, leaving room for low-level attacks to pass unnoticed and preventing them from translating their detection abilities into prevention.

Timing-Based Approach. INCANTA [21] proposed adding secret delays to the expected arrival times of periodic messages, with receivers inspecting the delay of every message. However, the accuracy of such delays degraded significantly for lower-priority IDs. CANTO [22] suggested pre-scheduling bus traffic to avoid unexpected delays of lower priority messages. Unfortunately, both methods use up to 8 message bits and incur processing overhead on the receiving side. Other works [55, 61] used similar techniques with variations such as using authentication messages, a monitor node, the delays between each message and its authentication message, or using multiple covert channels. Similar to INCANTA and CANTO they did not eliminate using frame bits or authentication messages. Additionally, all the aforementioned solutions use a *primitive form of delay* that is vulnerable to an attacker purposely injecting higher priority messages and causing a target message to be late, do not offer prevention, focus only on *injection*, and work only for periodic messages. We propose *a different kind of timing channel*, based on *inter-frame spacing*, which cannot be tampered with and works for periodic as well as aperiodic messages. We use it to defend against an extensive set of attacks and offer both detection and prevention, *without* using any message field.

Other Approaches. Few works have addressed attacks other than *injection*. Namely, to prevent bus flooding, researchers suggested modifying the network's hardware to allow for the isolation of attackers [23, 28]. Such solutions are very expensive to implement. Other works suggested manipulating the ID to bypass targeted flooding [12, 29] or randomizing portions of it to prevent error handling attacks [3]. Nonetheless, these systems cannot be deployed where IDs are used to convey commands, responses, or anything beyond their usage as mere identifiers as in most diagnostic protocols.

4 Threat Model

Similar to prior CAN security papers [3, 4, 19, 34, 36, 40, 41, 47, 58], we assume a remote attacker who has successfully compromised an ECU through Bluetooth, Internet, or any other remote means. The attacker can execute any code but has no control over the protocol controller and cannot alter protocol's rules. The attacker has no physical access to the bus and hence cannot attach devices with special hardware.

5 ZBCAN

5.1 Architecture and Operation Overview

As shown in Fig. 5, ZBCAN consists of a central monitor node, able to stop messages during transmission, called the *officer*, and a set of software *agents*, installed on every ECU. Each ECU privately agrees on a secret, non-repeating, and unique sequence of inter-frame spaces, called the *In BetweeNs* (*IBNs*), with the *officer*, and then enforces these sequences upon outgoing messages. If the *officer* detects a message with the wrong *IBN*, or an unknown ID, it stops it right after the ID portion of the message, thus preventing the message from being received by any ECU. Depending on the *officer's* setting, it may ignore the message, issue a warning, stop it, or disable its transmitter (Sec. 9). This way, several attacks could be prevented at once. Namely, *error handling attacks* rely on a technique called *simultaneous transmission* (Sec. 2), where attackers have to send a message exactly at the same time their victim transmits. With ZBCAN, they need to guess the exact *IBN* value for every message to transmit simultaneously. Similarly, *injection* and *flooding* attackers need to guess the correct *IBN* value for every message. Otherwise, their messages will be stopped by the *officer*.

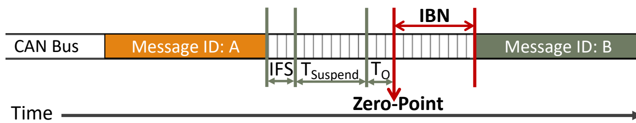

Figure 4: *IBN* basic concept.

The In BetweeN (IBN) As shown in Fig. 4, we use the term (*IBN*) to refer to the spacing between any two consecutive frames, measured from a *zero-point* (explained in Sec. 5.2). Although this definition is similar to that of the *Inter-Frame Spacing* (*IFS*), in most definitions, *IFS* refers specifically to the three bits following the end of a frame. To avoid confusion, we use the term *IBN*. Per the standard [16], CAN controllers initiate transmission after sensing the bus idle. This happens by sampling voltage at time intervals equal to one-bit each. As a result, the spacing between the end of one frame and the beginning of another is not continuous but discrete, meaning, *it is a multiple of a bit's length*, plus a small extra delay caused by clock skews and propagation delay. Therefore, if we can find ways to ignore this small delay, as explained in Sec. 6.3, we can view the spacing as discrete and measure it using bit-length units or simply *bits*.

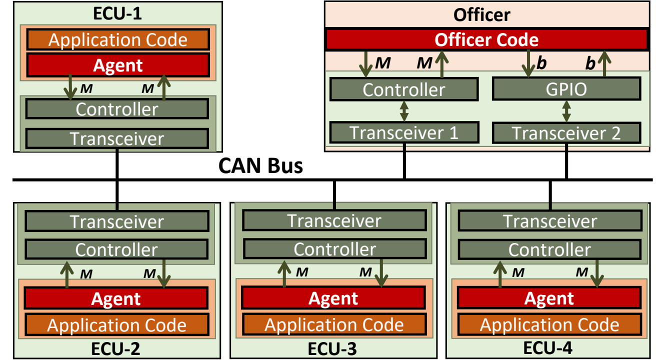

Figure 5: Architecture of a system implementing ZBCAN. Symbol (M) refers to messages. Symbol (b) refers to bits.

IBN Sequence. To illustrate how to use *IBN* as a signature, assume that a generic message is currently being transmitted on the bus. Further, assume messages of $ID = X$ have a sequence of endless, secret, and non-repeating *IBN* values to be followed. As shown in Fig. 6, when wishing to transmit a new instance of X, the *agent* of X waits until the ongoing generic message transmission concludes, counts a distance equal to the scheduled *IBN* (IBN_{sc}) in the sequence, then transmit. The *agent* does not wait until $IBN \geq IBN_{sc}$ but *exactly* $= IBN_{sc}$.

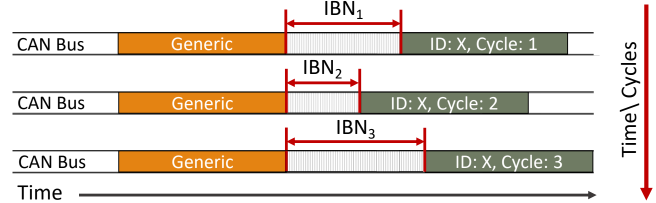

Figure 6: A running *IBN* sequence as a message ID signature.

Officer. As shown in Fig. 5, the *officer* is a trusted node that has the ability to securely store keys and has access to the bus through two channels, one through a CAN controller, and another directly through a GPIO and a CAN transceiver. The GPIO channel serves three purposes: (1) accurately measuring the *IBN* of every message, (2) reading message IDs before their data is delivered, and (3) allowing the *officer* to inject *error frames* on demand to stop any message. The *officer* is connected to the CAN bus in parallel as other nodes. It is not a gate or a bottleneck and causes no delay to messages. Instead, it acts as an observer who can immediately intervene. Its role is to monitor the enforcement of the *IBN* sequence. If it detects a message violating its *IBN* sequence or a message ID that is not allowed, it stops it before being received by any ECU. To be able to do so, the *officer* knows all the allowed IDs on the bus and their secret sequences.

Agent. The *agent* is a software installed on every ECU we wish to protect. It does not require any hardware changes to the ECU. Agent's components are further detailed in Appendix. B.1. The *agent's* role is to apply the *IBN* sequence upon outgoing messages. This sequence is unique per mes-

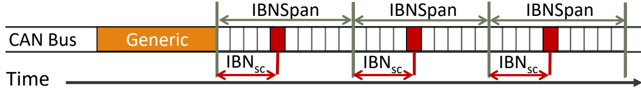

Figure 7: Dividing the timeline into distances $= |IBNSpan|$ allows for using *Modulo IBN* instead of *Absolute IBN*.

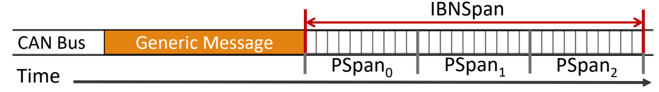

Figure 8: Dividing *IBNSpan* into exclusive priority spans.

sage ID and shared between two parties only: a ZbCAN *agent*, and the ZbCAN *officer*. One *agent* does not know the sequences of any other *agent*.

Message Reception. Upon reception, *agents* do not perform any computations. If a message is received successfully, it is approved by the *officer*. This way, we eliminate any processing overhead on the receiving side.

5.2 IBN Implementation Details

Zero-Point Calibration. The *zero-point* cannot be the same as last frame's last IFS bit for two reasons. First, nodes operating in the *error-passive state* have an additional 8-*bit* suspend-transmission penalty ($T_{Suspend}$), enforced at the protocol controller's level. If IBN_{sc} is 0, and the *zero-point* is the last IFS bit, an *error-passive* node will violate this value. Second, if IBN_{sc} is too low, an ECU with low computational power may not have enough time to initiate transmission in time but after an overhead period (T_O). T_O should be measured empirically for every system. Accordingly, as shown in Fig. 4, we set *zero-point* $\geq IFS + T_{Suspend} + T_O$.

Measuring T_O. *Agents* are composed of a library and three Interrupt Service Routines (ISRs) (Appendix. B.1). T_O is dependent on the end of frame (EOF) ISR, which is responsible for clearing the transmit buffer, updating the sequence index, checking if there are pending messages to be transmitted, then applying the *IBN* value at the next transmission. Ideally, the ISR should be able to execute these tasks by the end of $T_{Suspend}$ in Fig. 4. However, some ECUs may take longer. The effective ISR processing time for an ECU could be measured during the system design phase by sending a test message at the end of the EOF ISR (without any *IBN*), then measuring the distance between the last message on the bus and the test message. The system's T_O, should be set to the *longest* ISR processing time of any ECU.

IBNSpan. If the bus is busy and IBN_{sc} is too long, the *agent* may never find the opportunity to transmit. To prevent this, all *IBN* values should be kept within a span ($IBNSpan$) so that any message with $IBN_{sc} \in IBNSpan$ is guaranteed to transmit within a window not exceeding its deadline (explained in Sec. 6.2). We use the notation $|IBNSpan|$ to refer to the number of elements (*IBN* values) in the $IBNSpan$ set.

Modulo IBN. Since *IBN* is counted from the last frame on the bus, if a message is generated during a long idle period, it will have to wait until a message appears. Even worse, if all ECUs are also waiting for a message to appear, no messages will transmit. To prevent such problems, starting at the last *zero-point*, we divide the timeline into slots of length $= |IBNSpan|$. Instead of having to send the message only at a spacing $= IBN_{sc}$, we send it at any spacing (d) that satisfies the condition: $d \bmod |IBNSpan| = IBN_{sc}$. This way, if a message is generated within an idle period, it waits until the beginning of the next $IBNSpan$, counts a number of bits $= IBN_{sc}$, then initiates transmission, as shown in Fig. 7. Beginning from here, the term *IBN* refers to *Modulo IBN*.

Priority and IBN. Without ZbCAN, if two messages with IDs 0 and 10 are pending transmission at the same time, they will both go through an arbitration phase that ends in $ID:0$ winning and transmitting first. With ZbCAN, if $ID:0$'s scheduled *IBN* is 10 b, while $ID:10$'s scheduled *IBN* is 0 b, $ID:10$ will transmit first, inverting the priority system.

To guarantee that such a scenario does not cause timing deadline violations for time-sensitive messages, we enforce our own priority system. First, we divide $IBNSpan$ further into N_{pri} non-overlapping ranges called *priority spans* ($PSpan$s), each representing one priority level as shown in Fig. 8. Next, we arrange all message IDs in ascending order, based on their deadlines, then group them into $N_{pri} \geq 1$ priority groups (P_{group}s). Each P_{group} contains one or more message IDs sharing the same $PSpan_{group}$, where $PSpan_{group} \in IBNSpan$. P_0 is dedicated $PSpan_0$ and contains the IDs with the shortest deadlines, while $P_{N_{pri}-1}$ is dedicated $PSpan_{N_{pri}-1}$ and contains IDs with the longest deadlines. This way, we guarantee that messages with the shortest deadlines have a higher priority. In Sec. 6.1, we model the worst-case response time (WCRT) for messages in a ZbCAN system, then use this model to map message IDs into *priority groups* and guarantee that time-sensitive messages arrive in time.

Dummy Messages. To prevent *IBN* inaccuracies due to the inherent clock skew among ECUs, *dummy* messages of zero-byte length may need to be inserted after long idle periods to force all ECUs to resynchronize. We detail this in Sec. 6.3.

5.3 Operation Implementation Details

Registration and Sequence Exchange. Each *agent* starts its operation by an exchange with the *officer* to establish the first sequence for each ID. This exchange (detailed in Appendix B.2) happens only at the beginning of operation.

Sequence Usage and Extension. Every *agent* keeps an $index_{id}$ for each ID's *IBN* sequence. With every transmitted message, the *agent* consumes the bits pointed at by $index_{id}$ from the sequence and then increments $index_{id}$. After the initial sequence exchange, each sequence could be used to generate new sequences throughout the operation without having to re-exchange sequences with the *officer*. We call this operation, detailed in Appendix B.2, *sequence extension*.

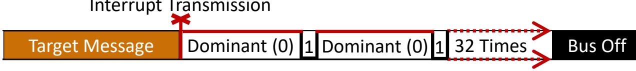

Figure 9: Successively interrupting error frame delimiters 32 times instantly pushes transmitters to the *bus-off* state.

Officer Policing. If the *officer* detects a message with a wrong *IBN*, it interrupts it using an *active error* right after reading its ID field, preventing its payload from appearing on the bus. Right after the interruption, it issues a *warning and resynchronization* message with an ID = ID_{warn}, a system parameter. Only the *officer* is allowed to send this ID. The message contains the violating ID and the $index_{id}$ of the next expected *IBN*. Upon reception of a warning message, ECUs read which ID violated its sequence. If an ECU is a transmitter of the violating ID, it updates its $index_{id}$ to the one in the message. If it is a receiver, it takes note of the possibility of the data being compromised. After N_{warn} successive warnings or if the message has a prohibited or unknown ID, the *officer* suspends the intruding node (Sec. 5.4).

Errors. If an *agent* encounters an error while transmitting a message, it should increment its $index_{id}$ since most errors happen after the ID portion of a message, meaning, after the *officer* has witnessed and approved the ID and *IBN*.

Queuing. Since we group every message on the bus into N_{pri} *priority groups* and consider all IDs within a group to have the same priority, we recommend that within each *agent*, messages in the same *priority group* share a FIFO queue.

5.4 Disabling Transmitter (Instant Bus-Off)

We propose a new technique to push an ECU to the *bus-off state* by targeting a single message. It requires equipment that is able to accurately inject individual bits directly into the bus. Only the *officer* could do that per our threat model (Sec. 4). The method is as follows. (1) We pick a frame on the bus and wait until a one is being transmitted. Once that happens, we inject a zero. (2) After a single bit, the transmitter detects this error and attempts to send an error frame composed of 6 zeros (flag) and 8 ones (delimiter). (3) After the delimiter starts, we release the bus for a single bit, allowing the one to appear. (4) After the one, we re-inject a zero. Consequently, step (2) repeats. We repeat steps $(1-4)$ 32 times, where the transmitter enters the *bus-off state* as illustrated in Fig. 9. This process could take as little time as $(7*32)+(1*31)=255$ b (510 μs on a 500 *kbps* CAN bus).

Although the authors of [47] proposed the *single-frame bus-off* (SFBO), this technique outperforms it in two ways: (1) *SFBO* requires ≈ 5 *ms*. Our technique requires 512 μs, up to $\approx$ $10X$ faster. (2) *SFBO* requires automatic re-transmissions to be enabled. As such, an ECU could protect itself by disabling automatic re-transmissions. Our technique does not rely on re-transmissions and hence *cannot be escaped* once launched.

6 Performance Analysis

6.1 Worst-Case Response Time Analysis

Several works [9–11, 51–53, 62] have analyzed the Worst-Case Response Time (WCRT) in CAN systems. We use the findings of [9] as a starting point. In Equation 1, R_m refers to the WCRT of a message, J_m refers to the queuing jitter or the longest time between initiating queuing and actually queuing a message, w_m refers to the queuing delay or the maximum time a message could wait in the queue before initiating transmission, and C_m refers to the longest transmission time of message m. Every message ID m should have two metrics defined: (1) T_m to represent the period of a periodic message or the minimum inter-arrival time between two instances of an aperiodic message, and (2) D_m to represent the timing deadline or the maximum allowed delay for the message. A message is *schedulable* only if $R_m \leq D_m$.

$$R_m = J_m + w_m + C_m \tag{1}$$

To calculate the worst case queuing delay w_m in Equation 1, we use Equation 2. B_m refers to the blocking delay or the time message m could wait for a lower priority message, currently in transmission, to conclude. T_k refers to the minimum time-interval between successive launches of the queuing task of message k, and τ_{bit} to the bit-time.

$$w_m^{n+1} = max(B_m, C_m) + \sum_{\forall k \in hp(m)} \lceil \frac{w_m^n + J_k + \tau_{bit}}{T_k} \rceil C_k \tag{2}$$

With ZBCAN, messages wait IBN_{sc} before transmission. The effective transmission time for message m then could be viewed as $IBN_{sc,m} + C_m$. Assuming $m \in$ *priority group* (P_n) and $PSpan_n$ starts at point $(Span_{n,beg})$, the maximum $IBN_{sc,m}$ message m could wait is $G_{max,n} = |PSpan_n| - 1 + Span_{n,beg}$. As a result, we define $EC_m = G_{max,n} + C_m$ to represent the *effective maximum transmission time* of m, or the maximum time it could wait *if the bus is available* plus its actual maximum transmission time C_m. Similarly, the *effective maximum blocking delay* EB_m includes the lower priority message's waiting time. Since message m shares the same priority level with a whole group, defining which IDs have higher priorities within the group becomes difficult. The worst case is for the longest message to be currently in transmission, for all messages of higher-priority and same-priority groups $hp(m)$ and $sp(m)$, to be pending transmission at the same time, for all higher-priority messages to receive the maximum *IBN* value for their *PSpans*, for all messages of the same group, including message m to wait for $G_{max,n}$, and for m to lose arbitration to every message and transmit last. The worst queuing delay for our system could be represented by Equation 3.

$$w_m^{n+1} = max(EB_m, EC_m) + \sum_{\forall k \in hp(m) \cup sp(m)} \lceil \frac{w_m^n + J_k + \tau_{bit}}{T_k} \rceil EC_k \tag{3}$$

Equations 1 and 3, give us insight on what factors influence the WCRT of a message. We expect for factors such as the bit-time (baud-rate), message length, jitter, number of messages

Algorithm 1 `Priority Grouping Algorithm`

```
1: AllIDs ← System ID list
2: n ← 0
3: Ratio_safe ← 0
4: while Ratio_safe ≤ 1 do
5:     System ← Schedulable
6:     while AllIDs != Empty && System != Unschedulable do
7:         Create new group P_n
8:         while P_n != Full && AllIDs != Empty do
9:             Add ID to P_n
10:            Remove ID from AllIDs
11:            if !( All P_n IDs are safe) then
12:                Remove ID from P_n
13:                Add ID back to AllIDs
14:                if P_n Empty then
15:                    System ← Unschedulable
16:                    Return IDs from all groups to AllIDs
17:                P_n ← Full
18:                n++
19:    Ratio_safe = Ratio_safe + 0.05
20: if Ratio_safe > 1 && AllIDs != Empty then
21:     System ← Unschedulable
```

and their inter-arrival times, number of priority groups, and $|PSpan|$ to have a direct influence. We also expect for messages in the higher priority groups to have a higher influence on the WCRTs of the system than lower priority groups.

6.2 Priority Grouping

We define the ratio $Ratio_{safe} = R_m/D_m$ to represent the *WCRT* of a message m divided by its deadline. To guarantee that time-sensitive messages will not violate their deadlines (*schedulable*), we map different message IDs to different *priority groups* such that the condition: $Ratio_{safe} \leq 1$ holds for all messages and groups. Grouping should take place during the system design phase and not during operation. To optimize our grouping, we define two objectives. The first is to minimize $Ratio_{safe}$. Assuming the system has a fixed *IBNSpan*, then it is obvious that the security of the system drops with every division of this span. Hence, the second objective is to minimize the number of priority groups. Alg. 1 illustrates how to achieve these objectives. In the algorithm, the term "safe" means $Ratio_{safe} \leq 1$.

6.3 Discretizing IBN Challenges

ZBCAN works by discretizing the spacing between consecutive frames, then controlling this spacing (*IBN*) to achieve its security goals. Nonetheless, the impact of factors such as the *propagation delay* on this "discretization process" should be studied to prevent any *IBN* inaccuracies.

Propagation Delay. Controllers read the value of a bit by taking voltage samples from the bus at bit-lengthed intervals. A bit-time is divided into four segments. The sample point is configurable, and is taken between the third and fourth [16]. For the *propagation delay* to cause *IBN* inaccuracies, it needs to exceed the sampling point. Assuming a typical propagation delay of 5 *ns* [17], a baud rate of 500 *kbps*, and an *officer's* sampling point of 65%, the *round trip propagation delay* needs to exceed 1300 *ns* (cable length of > 130 *m*) to constitute a problem. For a typical CAN bus with a 2 to 15 *m* length, this problem is irrelevant.

Clock Skew and Dummy Messages. Due to the inherent clock skews among ECUs, they gradually lose synch. To counter that, CAN requires all controllers to re-synchronize at the rising edge of every new frame's *SOF*. In ZBCAN, if the clock skew of one ECU causes it to start transmitting past the sampling point of the officer, it will cause a *false positive*. To prevent that, we take advantage of CAN's re-synchronization mechanism. Specifically, we define the metric d_{skew} that refers to the minimum spacing between messages that causes any ECU's clock skew to start causing *IBN* inaccuracies. By inserting a dummy message of *zero-byte* length at an idle spacing $d_{dummy} < d_{skew}$, all clocks re-synchronize before the clock skew causes inaccuracies.

6.4 Overhead Analysis

A *sequence extension* operation for a specific ID happens every $L_{seq}/\log_2(|PSpan|)$ *messages*. On the *officer's* side, the sequences are extended for every message ID in the system. However, On the *agent's* side, sequences are extended only for the message IDs that the *agent* transmits.

To function properly, the *agents* and the *officer* need a minimum amount of memory to hold variables such as keys, sequences, etc. Specifically, *agents* require at least $(2 * L_{sequence}) + ((3 * L_{sequence}) + L_{index}) * N_{ids-agent}$ *bits*, where $L_{sequence}$ refers to the length of the sequence, L_{index} to the length of the index, and $N_{ids-agent}$ to the number of IDs that the *agent* sends. Similarly, the *officer* requires at least $(2 * L_{sequence} * N_{agents}) + ((3 * L_{sequence}) + L_{index}) * N_{ids-system}$ *bits*, where $N_{ids-sys}$ and N_{agents} refer to the number of IDs and number of *agents* in the system, respectively. A more detailed analysis of the memory and processing overhead is provided in Appendix C.1.

7 Security Analysis

Compromised Agent Abilities. Our threat model (Sec. 4) assumes a remote attacker that has all the information of an *agent* but is limited by the ECU's hardware. While ECUs communicate on the CAN bus through a CAN controller, the *officer* can connect directly to the bus. Further, an *agent* knows only its pre-shared key (Sec. 5.1), which is used to agree on an *IBN* sequence. The *officer*, on the other hand, knows the pre-shared keys of all *agents*. Consequently, a compromised *agent* cannot read message IDs during transmission, stop messages on demand, alter protocol rules, or establish *IBN* sequence agreement with other *agents*, since it does not have the necessary hardware or keys. This limits its abilities to receiving and transmitting full messages and controlling their *IBN* values, which could be used to push another ECU's messages off sequence, launch message injection (including impersonation, masquerade, etc.), error handling, or flooding attacks. Here, we discuss these attacks.

7.1 Off Sequence Attack

In ZBCAN, each message ID follows a strict *IBN* sequence. If an attacker is able to guess the scheduled *IBN* for a target ID once, the *officer* will update its $index_{id}$ but the legitimate transmitter will not. Consequently, when it sends its next instance of the message with $IBN = IBN_{sc}$, it will be *off-sequence*, since the *officer* will be expecting an $IBN = IBN_{sc+1}$. However, since each ID has its own sequence, even if one ID is pushed off sequence, the *agent* will be able to transmit the rest of the IDs normally. Further, since the *officer* will stop the first message with unexpected *IBN* then issue a *warning and resynchronization* message containing the expected $index_{id}$, the legitimate *agent* will be able to resynchronize.

This method turns a security weakness into a strength, guaranteeing that injections will be detected in 100% of cases, since even a successful injection always results in the legitimate ECU going off sequence and the *officer* issuing a warning message to all receivers. Further, it is better than having each *agent* monitor whether its messages are being impersonated then automatically resynchronizing in terms of performance, since *agents* have to only watch for messages with ID_{warn}, instead of every ID they transmit.

7.2 Injection and Detection Window

Attackers could inject messages with IDs that exist in the network (e.g., masquerade and impersonation) or random IDs that do not necessarily exist (e.g., fuzzing attacks). Assuming a smart attacker that knows the system IDs, their groups and *PSpan*, ZBCAN offers three probabilistic security guarantees for injection attacks: *Individual-Message Detection*, *Individual-Message Prevention*, and *Flow Detection*.

Individual-Message Detection. Assume a periodic message m with a period T belonging to a *priority group* with a $|PSpan| = n$ and a scheduled $IBN = IBN_{sc}$. The probability of guessing IBN_{sc} is $1/n$. Within a time period $\leq T$, the legitimate ECU will send its message with the same $IBN = IBN_{sc}$. Since the *officer* will be expecting an $IBN = IBN_{sc+1}$, the injection will be detected within a time window $\leq T$, except if IBN_{sc+1} is randomly $= IBN_{sc}$. Since the sequence is generated using a *PRF*, the probability of IBN_{sc+1} being equal to IBN_{sc} is also $1/n$. To generalize, let the *detection window* (w) be an integer representing the number of periods/cycles of duration T since the injection, the probability of detecting an injection within a window w is represented by Equation 4. Note that $P(w)_{det}$ always tends to 1 given enough cycles.

$$P(w)_{det} = 1 - \frac{1}{|PSpan|^{w+1}} \tag{4}$$

Individual-Message Prevention. To prevent an injection, the *officer* needs to detect it as soon as it appears on the bus, and before it is delivered to the receivers. This means that the *probability of prevention* $P_{prevent} = P(0)_{det}$, as shown below:

$$P_{prevent} = 1 - \frac{1}{|PSpan|} \tag{5}$$

The expected number of trials before a successful injection is $E(inj) = |PSpan|$, not $|PSpan|/2$, since the *officer* increments $index_{id}$ for the sequence with every observed ID instance whether its *IBN* is accurate or not.

Injection Flow Detection. For a flow of f messages not to be detected, every single message in the flow should pass unnoticed. In other words, an injection flow is detected when any of its messages are detected. Equation 4 could be generalized to quantify this probability to become:

$$P(w,f)_{det} = 1 - \frac{1}{|PSpan|^{w+f}} \tag{6}$$

Random Injections. ZBCAN allows a message to transmit if the message ID is allowed on the bus and the message is following its Seq_{id}. Since random injections violate both conditions, their rate of prevention and detection is $\approx 100\%$.

7.3 Error Handling Attacks

Collision Injection. To inject a collision using *simultaneous transmission* (Sec. 2), one needs to estimate the transmission time of the victim message, send a *synch message* slightly before its expected arrival, followed by a message of the same ID. With ZBCAN, the attacker cannot randomly inject a high priority message for synchronization or it will be stopped by the *officer*. Further, the attacker has to accurately guess the scheduled *IBN* for the victim's message. Finally, with *Modulo IBN*, the attacker has to guess which $|IBNSpan|$ slot to inject its message. Assuming that the attacker only has to guess *IBN*, Equation 5 could be applied to estimate a probabilistic *lower bound* for the *prevention rate*.

Error Passive. The fastest way to push a victim to the *error-passive* state requires the attacker to cause at least 16 successive collisions, the *prevention rate* for this scenario is: $P_{prevent} \geq 1 - (1/|PSpan|^{16})$.

Bus-Off. The fastest *bus-off attack* requires the attacker to cause 32 successive collisions. The *prevention rate* of this scenario is: $P_{prevent} \geq 1 - (1/|PSpan|^{32})$.

7.4 Flooding Attacks

ZBCAN prevents flooding by suspending the attacker using the *instant bus-off* technique. The success of flooding attacks is measured by the *drop rate* ($rate_{drop}$) they cause to messages. We define our *prevention rate* of flooding attacks to be $rate_{prevent} = 1 - rate_{drop}$. $rate_{prevent}$ will differ from a system to another depending on factors such as the busload and the ID allocation of the network more than the $|IBNSpan|$.

7.5 Choosing $|PSpan|$

Looking at Equations 4 through 6, we notice that regardless of the value of $|PSpan|$, injections will always be detected, given enough cycles. Therefore, the value of $|PSpan|$ could be viewed as mainly affecting the *detection speed*, and the *single injection prevention rate*. The system designer should initially define their security objectives. A high *single injection*

prevention rate requires a high $|PSpan|$. However, ZBCAN provides high detection rates, even for small $|PSpan|$ values. For instance, a $|PSpan|$ value as low as 16 *b* will result in a *single injection prevention rate* $\approx 93.75\%$, but a *single injection detection rate* $\approx 99.61\%$ within a single cycle. This is already higher than most of the current IDSs *detection rates* for *flows*. Meanwhile, its *detection rate* of a malicious flow composed of only *two* messages is $> 99.97\%$ within a single *cycle*. Within 5 *cycles*, both the flow and single injection *detection rates* for the aforementioned scenarios become $\approx 100\%$. Outside injection attacks, the same $|PSpan|$ prevents *error passive* and *bus off* attacks at a $\approx 100\%$ rate. To choose a $|PSpan|$ value, a compromise between the security of the system and its performance has to be reached. The busier the system, the smaller the $|PSpan|$ values it could afford.

8 Evaluation

For a thorough analysis, we evaluated ZBCAN's false positive rate, security, performance, and scalability on a testbed using artificial data, on a testbed using a 2011 Chevy-Impala's data, and finally on a 2011 Chevy-Impala's CAN bus.

Trusted Officer Platform. We use a Renesas RA6M5 MCU board as the *officer*. RA6M5 MCU runs on an ARM Cortex M-33 and is equipped with TrustZone. It offers memory and peripheral access isolation, secure boot-loading and processing with TrustZone, a CAN module, and a GPIO module.

Pseudo Random Function. We use Chaskey [37], an open source *PRF* that takes ≤ 0.5 *ms* to generate a $Seqlength = 128$ *b* on an Arduino Uno board and ≈ 1.9 μs on the RA6M5.

Zero-point. As explained in Sec. 5.2, we measured the value of T_O on an Arduino Uno and determined it to be 7 *b*. The *zero-point* is $T_O + T_{Suspend} = 15$ *b* after the IFS.

8.1 False Positive Test

Propagation Delay. This delay is proportional to the bus length. To assess its impact, we attached the *officer* and a *reference* message generator to a breadboard, an *agent* to another breadboard, connected the two with a cable and added a 120 Ω resistance on each board. We set the *agent* to transmit a message immediately after every *reference* message with $IBN = 0$ *b*. We set the cable length to 5 *cm* and measured the average spacing between the *reference* and *agent* messages in nanoseconds. Next, we changed the cable length from 5 *cm* to 30 *m*, and repeated the measurement. The difference between the spacing at 30 *m* and at 5 *cm* was ≈ 340 *ns*, meaning, the *round trip* propagation delay was ≈ 11.33 ns/m (*one way* delay ≈ 5.66 ns/m). At a 500 *kbps* baud rate, for the *round trip* delay to exceed our *officer*'s sampling point of 75%, the cable length has to be ≥ 132.39 *m*.

Impact of Clock Skew. In a system composed of 20 ECUs, the smallest d_{skew} was 1189 *b* and the largest $d_{skew} = 1460$ *b*. The details of these measurements are explained in Appendix C.3. To assess the impact of clock skew on the false positive rate, we connected the *officer* and a traffic source, sending a *reference message* every 6 *ms*, to a 500 *kbps* bus. Next, we connected the ECU with the largest d_{skew} and set it to transmit after every *reference message* with an *ascending IBN* between 0 and 3000 *b*. This means sending the first message with $IBN = 0$ *b*, the second with $IBN = 1$ *b* and so on until 3000 *b*, then rolling over to 0 *b* and repeating. Meanwhile, the *officer* monitored the *IBN* of each message. We ran this test for 30 *min*. Next, we connected the ECU with the smallest d_{skew} and repeated the test. Fig. 10 shows how for both ECUs, the false positive rate is 0% before each ECU's d_{skew}, then increases after exceeding it until it reaches 100%.

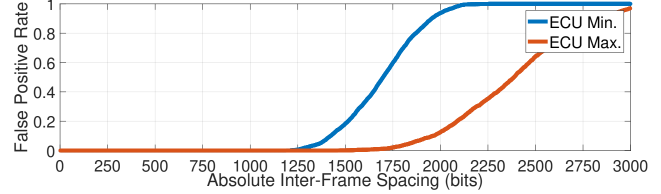

Figure 10: Message spacing vs. *IBN* accuracy.

False Positive Test With Dummy Messages. To assess whether the *dummy message* solution (Sec. 6.3) could keep the *false positive* rate of a network with both propagation delay and clock skews at 0%, we connected the *officer* and a traffic generator to one breadboard and the *agents* to another and connected the two with a 30 *m* cable to maximize the *propagation delay*. We set each *agent* to register and exchange an *IBN* sequence of an $IBNSpan = 128$ *b* with the *officer* as explained in Sec. 5.3. As previously determined, the system's d_{skew} was 1189 *b*. Nonetheless, since the system's $IBNSpan = 128$ *b* and 1189 mod $128 \neq 0$, we chose $d_{dummy} = 1152$ *b*, the biggest value under 1189 whose mod $128 = 0$ *b*. Next, we set up the traffic generator to send a dummy message of length 0 *B* at 1152 *b* idle time and set up each ECU to send 100*K* messages while following its own *IBN* sequence. For all ECUs, the *false positive rate* remained 0% after 100*K messages*.

8.2 ZBCAN Security Evaluation on a Testbed

On a 500-*kbps* CAN bus, We connect the *officer*, 5 ECUs, and a dummy message generator. ECUs are composed of Arduino Uno boards, mcp2515 CAN controllers, and mcp2551 transceivers. One node is used as the attacker. We assume a smart attacker who knows the *IBNSpans* of the system. Therefore, we provide the attacker with an *agent* similar to the one on all nodes with modifications to launch attacks. Below, we report ZBCAN's evaluation results.

Injection. We set the busload to 34% as in our Impala and evaluated ZBCAN under $|IBNSpan|$ values of 16, 32, 64, and 128 *b* and three types of injection: (1) *Random Injections:* Attacker uses the same *PRF* with a random seed to try different IDs and *IBN* values within the *IBNSpan*. (2) *Targeted Injections:* Attacker injects an ID already in use in the network but

Table 2: Observed effectiveness of ZBCAN with different $|IBNSpans|$ against different *single injection* attack types.

Attack	Detection Rate	Prevention Rate Per |IBNSpan|			
		16 b	32 b	64 b	128 b
Random Injection	100%	100%	100%	100%	100%
Targeted Injection	100%	93.6%	96.9%	98.5%	99.1%
Replay	100%	93.8%	96.8%	98.4%	99.3%

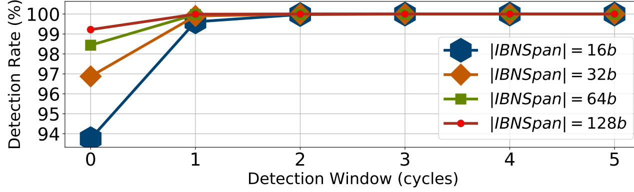

Figure 11: Average observed *detection rates and windows* for *targeted injections* and *replay* attacks.

uses the same *PRF* to randomly select *IBN* values within the *IBNSpan*. (3) *Replay:* Attacker replays a message with the same ID and *IBN* as the last message on the bus.

As shown in Table 2, the *prevention rates* of the *targeted injection* and *replay* attacks were similar to one another and within Equation 5's estimated range. The *random injection attack's prevention rate* was 100%. All attacks were detected at a 100% rate. However, Fig. 11 shows that the *detection window* for the *targeted injection* and *replay* attacks depended on the $|IBNSpan|$, confirming the findings of Equation 4.

Error Handling. With $|IBNSpan|$ values of 16, 32, 64, and 128 b, we set the attacker to inject collisions to a victim ECU using *simultaneous transmission*, to count the number of collision attempts, and the number of successful collisions. We set the victim to record the number of times it is pushed to the *error-passive* or *bus-off* states. As shown in Table 3, *error-passive* and *bus-off* attacks were prevented at a rate of 100%. *Collision injections* were prevented at rates ranging between of 98.8% and 100%. To evaluate the analysis in Sec. 7.3, we disable the *Modulo IBN* feature of our system, rerun the experiment. Fig. 12 plots the *prevention rate* with *Modulo IBN* on and off, as well as the theoretical lower bound.

Flooding. We operated the testbed under the following busloads: 10, 20, 30, and 40%. We set one of the nodes as a receiver to confirm message reception. We set the attacker to send back-to-back messages with $ID = 0_h$, and we measured the message *drop rate* with the *officer* disconnected. For all busloads, the *drop rate* was 100%. We repeated the same test with the *officer* connected and we achieved the following *drop rates*: $0\%, 0\%, 0\%$, and 0.67% for the respective busloads.

8.3 Performance with Real Vehicle Data

To evaluate the performance impact ZBCAN could incur if installed on a real vehicle, we emulated the traffic of a 2011 Chevy-Impala containing 4 ECUs, 50 IDs, and a 34% busload, whose data is shown in Appendix Table 8. We applied ZBCAN under three *IBN settings*: (A) 3 Priority groups, each

Table 3: Observed effectiveness of ZBCAN with different $|IBNSpans|$ against different *error handling* attack types.

Attack	Prevention Rate Per |IBNSpan|			
	16 b	32 b	64 b	128 b
Collision Injection	98.8%	99.3%	99.4%	99.6%
Error-Passive	100%	100%	100%	100%
Bus-Off	100%	100%	100%	100%

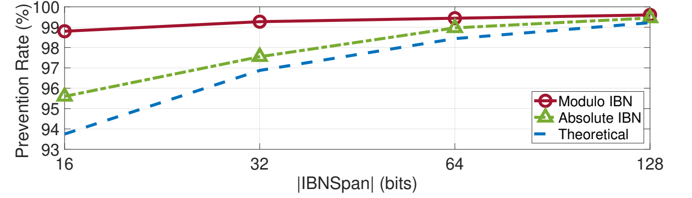

Figure 12: Observed *prevention rates* of the *collision injection* attack with *Modulo IBN* turned on and off.

$|PSpan| = 32\ b$. (B) 3 Priority groups, each $|PSpan| = 64\ b$. (C) 3 Priority groups with ascending priority spans where $|PSpan_0| = 32\ b$, $|PSpan_1| = 64\ b$, and $|PSpan_2| = 128\ b$.

Grouping. We ran Alg. 1 under the three aforementioned *IBN settings*. We assumed that all messages are time-sensitive and that each message's deadline is equal to its period length. Alg. 1 determined $Ratio_{safe}$ to be 0.59, 0.7 and 0.62 for *settings* A, B, and C, respectively. The small value of $Ratio_{safe}$ for all settings guarantees that no time-sensitive messages will violate their deadline. Appendix Table 8 shows the group boundaries for each *setting*.

Observed and Theoretical WCRTs. The top of Fig. 13 shows the observed and the theoretical WCRT of each ID for all three *IBN settings* and without ZBCAN. The bottom shows WCRT/Deadline ratio for each ID. The IDs are arranged from left to right based on their period lengths, with the shortest period being on the left. As shown, no message ID violated its timing deadline. The difference between the theoretical WCRT with and without ZBCAN is small on the left, but gets bigger as we move towards IDs with big periods (right). However, when looking at the WCRT as a ratio of each ID's deadline, this increase becomes trivial. For example, while the theoretical WCRT for ID 120_h at $|IBNSpan| = 64\ b$ approaches 50 ms, it has a period of 5 s, making the theoretical $WCRT/Deadline \leq 1\%$.

Without ZBCAN, the maximum $WCRT/Deadline$ ratio lies at $ID: 1E5_h$ around 0.7, the same as with ZBCAN, while the observed ratio is much smaller. Note that the theoretical WCRT is always greater than or equal to the observed WCRT, and that the difference gets bigger as we move right. This is because our WCRT analysis in Sec. 6.1 is too pessimistic. However, even this pessimistic theoretical WCRT is still below the timing deadline of all messages.

Observed Average and Minimum Response Times. For all *IBN settings*, the average delay for P_0 was $\approx 500\ \mu s$, and < 1 ms in 90% of the cases. For P_1 and P_2, the average delays were $\leq 2\ ms$, and $\leq 3\ ms$, respectively. For all groups, *setting* (B)

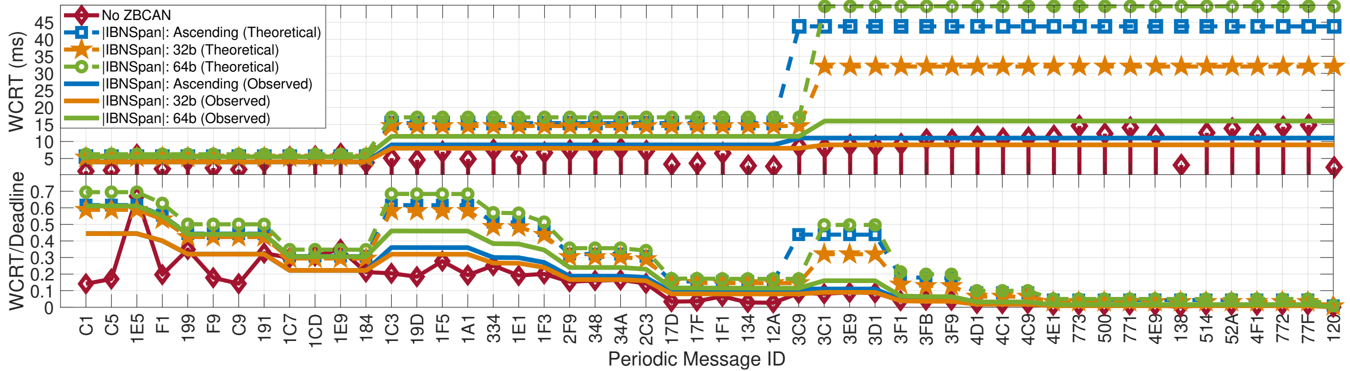

Figure 13: WCRTs (absolute and as ratios of message deadlines) for a 2011 Chevy-Impala traffic with and without ZBCAN.

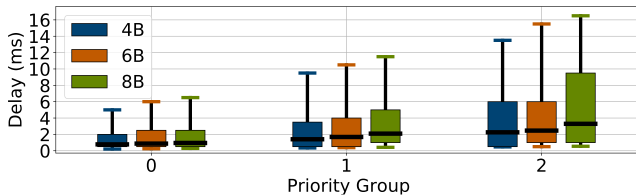

Figure 14: Observed WCRTs vs. message length

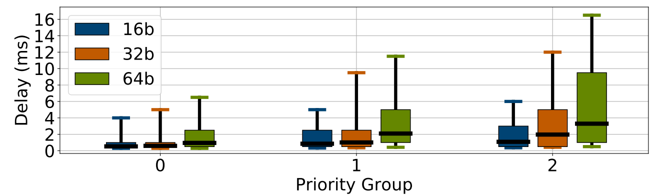

Figure 15: Observed WCRTs vs. $|PSPan|$.

had the largest minimum, average, and maximum response times. This is explained by Equation 3, where the $|PSpan|$ which influences WCRT the most is $|PSpan_0|$. Since *setting* B has the largest $|PSpan_0|$, its delay is also the largest.

Dummy Messages. The average spacing between messages was ≈ 1.6 *ms*, while d_{dummy} was > 2 *ms*. Consequently, the dummy message did not need to be inserted most of the time, resulting in a busload increase of $< 1\%$.

8.4 ZBCAN Scalability Evaluation

To evaluate ZBCAN's performance on a system with a large number of ECUs and safety critical messages, we setup a testbed with 20 ECUs and 100 message IDs. All messages were assumed to be safety critical and time sensitive, with their timing deadlines shown in Appendix Table 9. Note that messages $1, 6$ and F are aperiodic.

Grouping. We ran Alg. 1 under a 3-priority *setting*, a message-length$= 8$ B, a $|PSpan| = 64$ b, and an $|IBNSpan| = 192$ b. The calculated theoretical WCRTs were $7.3, 37.1, 96$ *ms* and the $Ratio_{safe}$ to be $0.73, 0.74$ and 0.38 for groups $0, 1$ and 2. Appendix Table 9 shows the group assignments.

Observed WCRTs. We set the transmission rates of the aperiodic messages to the highest to achieve the worst possible delays on the bus. The observed WCRTs were $\approx 6.5, 11.5$ and 17 *ms* for priority groups $0, 1$ and 2, respectively. No messages violated their deadlines or their theoretically calculated WCRTs as shown in Appendix Table 9, which proves that ZBCAN is safe to use in time-sensitive networks.

WCRT and Message Length. To evaluate the impact of message length on WCRT, we repeated the evaluation with message lengths set to 4 and 6 B and observed the WCRTs. Fig. 14 shows the average, maximum, minimum, 10th and 90th percentile observed WCRTs. As shown, increasing the length clearly increases the delay in every *priority group*.

WCRT and |PSpan|. To evaluate the impact of $|PSpan|$ on WCRT, we reset the message lengths to 8 B and, using the same group assignments, repeated the evaluation with $|PSpan|$ of all groups set to 16 b then 32 b and observed the WCRTs. As shown in Fig. 15, increasing $|PSpan|$ clearly increases the delay of messages in every *priority group*.

WCRT and Number of ECUs. To assess the impact of the number of ECUs on WCRT, we reset all lengths to 8 B, all $|PSpan|$s to 64 b and repeated the evaluation with 10 ECUs instead of 20 (ECU sends 10 IDs instead of 5). As shown in Fig. 16, the impact of increasing the number of ECUs on the observed delay does not show a clear up or down trend. For instance, while the observed WCRT for group 1 slightly increased, it slightly decreased for group 2. Meanwhile, the average WCRTs for all groups remained almost the same.

Per Sec. 6.1, the number of ECUs is not expected to directly impact the WCRTs. Similarly, per Alg. 1, the theoretical WCRTs are the same. Nonetheless, the WCRTs are not identical. These differences are mainly due to the *queuing jitter* that changed as a result of changing the number of IDs per ECU. Specifically, while the maximum jitter in both scenarios was ≈ 750 μs, the average jitter was slightly higher for the 10-ECU scenario, resulting in a very slightly higher average WCRT for the 10-ECU scenario. The jitter's standard deviation, however, was higher for the 20-ECU scenario, resulting in a slightly higher maximum WCRT for priority group 1.

Memory Consumption. We equipped a busy ECU transmitting 20 IDs with a ZBCAN *agent*. We then measured the

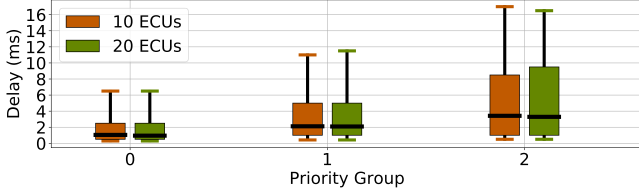

Figure 16: Observed WCRTs vs. numbers of ECUs.

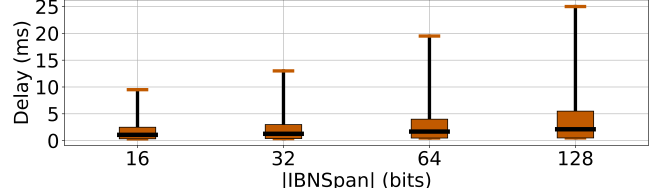

Figure 17: Observed WCRTS on a 2011 Chevy-Impala.

memory consumption of all the ECU's and *agent*'s variables and found it to be $\approx 1.72\ kB$.

Sequence Extension Overhead. With $L_{seq} = 128\ b$ and a $|PSpan| = 64\ b$, a busy *agent* with 20 IDs, each with a period of 25 *ms*, performed a *sequence extension* operation every $\approx 26.25\ ms$ (P_{ext}). Each operation took $\approx 0.5\ ms$ (d_{ext}). The *agent's* $O_{ext} = d_{ext}/P_{ext}$ ratio was $0.5 * 100/26.25 \approx 1.91\%$. On the *officer's* side, each extension operation took $\leq 1.9\ \mu s$. On a busy system with a message observed every 0.5 *ms*, a *sequence extension* operation happened every $\approx 10.5\ ms$, resulting in a an O_{ext} of $0.0019 * 100/10.5 \approx 0.018\%$.

8.5 ZBCAN on a Real Vehicle

Incremental Deployment. Incrementally adding a protected ECU to an unprotected bus is different from adding it to a protected one. Specifically, only the protected ECU's messages will have *IBN* delays. Therefore, we expect such an ECU to have higher delays than one deployed on an already protected bus (Sec. 8.3). To evaluate these delays, we attached the *officer* and an ECU equipped with ZBCAN to the CAN bus of a 2011-Chevrolet Impala. We set the *agent* to transmit at a period = 50 *ms* and under the following $|IBNSpans|$: $16, 32, 64$, and 128 *b*. Fig. 17 shows the average, minimum, maximum, 10th, and 90th percentile response times.

Table 4: Effectiveness of ZBCAN against *flooding* attacks.

Setting	Testbed				Chevy-Impala
	10%	20%	30%	40%	
Prevention Rate	100%	100%	100%	99.33%	100%

Flooding. We attached an attacker to send back-to-back messages of ID= 0_h with the *officer* disconnected. We connected an additional receiver node to confirm the reception of messages. The message drop rate was 100%, which means no messages whatsoever were being received. We repeated the experiment with the *officer* connected and the drop rate becomes 0%. This means that the attack *prevention rate* with the *officer* connected was *100%*. Table 4 shows ZBCAN's prevention rates of flooding in different testing conditions.

9 Benchmark Comparison

Since ZBCAN is versatile, it is difficult to compare its effectiveness against its entire set of attacks with systems that may not defend against the same set. Instead, we make three separate comparisons between ZBCAN and other systems.

Compared to IDSs. While most defense systems focus only on *injection attacks*, ZBCAN protects against a wider attack-set, encompassing *injection*, *error handling*, and *flooding*. Further, while most intrusion detection systems are able to detect attacks composed of message flows with a high degree of accuracy, ZBCAN offers security guarantees to single messages as well as flows. These abilities guarantee that, unlike other IDSs, gradual, intermittent, and single-message attacks will not pass unnoticed. They also allow ZBCAN to extend some of its detection abilities to prevent attacks. Table 5 compares ZBCAN's evaluation results with the evaluation results of other IDSs against the same attack set. As shown, ZBCAN's *prevention rate* compares with the best IDS's *detection rates*. Meanwhile, its *detection rate* is 100%.

Table 5: How ZBCAN's evaluation results at $|IBNSpan| = 64b$ compare with other intrusion detection systems.

IDS		Targeted Injection	Replay
Scission [32]		96.8%-98.5%	96.8%-98.5%
Clock-Skew [6]		97%	97%
ZBCAN	Detection	100%	100%
	Prevention	98.5%	98.4%

In conclusion, ZBCAN protects against a bigger attack set than any other defense system, does not require dynamic retraining, offers higher detection abilities of attack flows as well as single attack instances than any other IDS, and finally, attack prevention as well as detection.

Table 6: Comparing the probability of a single injection going undetected with different benchmarks.

Defense	P_{undet}			Message Bytes	Busload
	1 cycle	5 cycles	10 cycles		
Leia [45]	$1/2^{64}$	$1/2^{64}$	$1/2^{64}$	2	+100%
LCAP [26]	$1/2^{16}$	$1/2^{16}$	$1/2^{16}$	2	+0%
CACAN [35]	$1/2^{8}$	$1/2^{8}$	$1/2^{8}$	1	+100%
IA-CAN [24]	$1/2^{32}$-$1/2^{8}$	$1/2^{32}$-$1/2^{8}$	$1/2^{32}$-$1/2^{8}$	1-4	+0%
ZBCAN	$1/2^{14}$-$1/2^{8}$	$1/2^{42}$-$1/2^{24}$	$1/2^{77}$-$1/2^{44}$	0	+0-1%

Compared to Cryptographic Solutions. We define the metric P_{undet}, which quantifies the probability of a single injection going undetected. As shown in Table 6, while P_{undet} within *one cycle* is lower for Leia and LCAP, it is constant with w. As explained by Equation 4, this probability is 0 with ZBCAN, given enough cycles (w). As shown, at $w = 10$ *cycles*, ZBCAN's P_{undet} is the lowest. Further, these systems use message bytes and some of them double the busload. ZBCAN is the only one to use *zero* message bytes and cause

no or a very small busload increase. Finally, while P_{undet} for IA-CAN and CACAN are comparable to ZBCAN's at $w = 1$ *cycle*, an attacker may exhaustively try injecting all different combinations. With ZBCAN, this cannot happen since the *officer* will suspend nodes that try this approach.

Table 7: CANARY and ZBCAN are effective defenses against *error handling* and *flooding* attacks.

Defense System	Response Time	Attacker Isolation	Hardware Changes
CANARY [23]	5ms-100ms	Partial	1 Guardian Node + 8 Relays + Wiring
ZBCAN	22-72 us	Full	1 Officer Node

Compared to Other Solutions. CANARY is one of the few defense systems that addressed *error handling and flooding attacks*. As shown in Table 7, in addition to the expensive costs of wiring and adding relays, relays work by isolating entire sections of the CAN bus, which may result in the isolation of benign nodes, together with the attacker. Moreover, relays often have high relaying times, resulting in attacks taking place for a long time before soliciting a response.

10 Discussion

Intrusion Confinement. ZBCAN provides *intrusion confinement* in two ways. First, since *agents* do not share the same keys or sequences, a compromised ECU *agent* cannot predict the *IBN* sequences of other nodes and hence cannot inject messages impersonating other nodes or reliably inject collisions. Second, a compromised node cannot launch *flooding or error handling* attacks against other nodes, since the *officer* will actively interrupt any such attempt.

Time Triggered CAN. TTCAN systems use *matrix scheduling*, in which each message is expected to be transmitted during a specific interval. No other message is allowed to be sent during this interval. Additionally, TTCAN systems use *reference messages* to provide synchronization between nodes. These two factors make TTCAN systems a good fit for ZBCAN. Namely, *reference messages* eliminate the need for additional *dummy* messages. Similarly, prescheduling messages eliminates the need for *priority groups*, increases the security of the system, eliminates the *interference delay* in Equation 3, and significantly improves the WCRT. The only minor change these systems may need is to increase the acceptance window for each message by a distance $= |IBNSpan|$.

ZBCAN Controller. CAN controllers have all the hardware required to monitor and change message spacing (e.g., *suspend transmission period*). With slight modifications, the functionality of the *agents* could be implemented in the form of controllers, eliminating the overhead on the ECU's side.

Content Authentication. Since we are trying to make ZBCAN as lightweight as possible, we only discussed transmitter authentication. However, ZBCAN could be easily extended to include content authentication by calculating a hash or MAC for each message and then XORing the result with IBN_{sc}. On the *officer's* side, the *officer* will have to read the entire payload as opposed to reading only ID bits, calculate the keyed hash or MAC, then XOR it with the expected IBN_{sc} value to see if the result is equal to the measured *IBN*.

Other Possible Extensions. In addition to *content authentication*, the *officer*'s design could be easily extended to support more sophisticated operations. For example, it could be extended to provide protection to several buses operating at varying levels of security (e.g., confidential, secret, top secret, etc.) similar to monitoring solutions that have been designed for other communication buses [15]. It could also be extended to offer deep packet inspection or content authentication by checking the *DLC*, *RTR*, or payload fields. For example, it could be provided with the used payload fields, signal boundaries, and accepted value ranges specific to each message ID to provide deep packet inspection as has been proposed on other buses [13, 14].

11 Limitations

Officer Failure. Similar to most IDSs, the *officer's* failure removes the security it provides. However, the *officer* is not a filter, a bottleneck, or a gateway that messages go through before reaching the bus, rather, it is connected in parallel as any other ECU. We designed ZBCAN this way so that even if it fails, it *fails safe*. Further, applying *IBN* values, transmission, and reception are the *agent's*, not the *officer*'s, responsibility. Consequently, if the *officer* fails, *agents* continue following their *IBN* sequences normally. This means that while the protection against injection and flooding attacks will be removed, it remains intact against *error handling* attacks, since they do not rely on the *officer*, but the unguessability of the *IBN* that the attacker needs for the collision.

Compromised Officer. The *officer* is assumed to be trusted; a limitation not unique to ZBCAN. Although existing IDSs only monitor the network, they have access to the bus through a CAN controller and/or special pins. Both channels could launch attacks if the IDS is compromised. Nonetheless, to minimize the probability of such a scenario, we used a board that provides *secure processing*, *secure boot-loading*, *secure memory access isolation*, and *peripheral access isolation*. Further, except for the channels connecting the *officer* to the bus, it is assumed to be *air-gapped*.

In-Group Priority Inversions. Although ZBCAN guarantees that a message belonging to *priority group* P_n will always have a higher priority than one belonging to group P_{n+1}, it does not guarantee the priority hierarchy within the same group. Nonetheless, ZBCAN guarantees that even if an in-group priority inversion takes place, every message in the group is guaranteed not to violate any timing deadlines.

Unschedulable Systems. We assume all messages are time sensitive and use Alg. 1 to plan the system's schedulability. However, it is theoretically possible to find unschedulable sys-

tems. In such cases, we have to make the decision of lowering, or even dropping, the security of certain messages, to guarantee the schedulability of all messages. For instance, assuming a 3-priority system with each $|Pspan| = 64\ b$, in Appendix Table 9, if we lower the periods/deadlines of the first five messages to 5 *ms*, the system becomes unschedulable unless we lower the span of all groups to 32 *b*. Further lowering the periods/deadlines of the first five messages to 3 *ms*, renders the system unschedulable again. In this case, we may drop the security of the first 5 messages by adding them to group 0, then setting $|Pspan_0| = 0\ b$, while keeping the protection for other groups to preserve the system's schedulability.

Corrupt Payloads. The discussed design of ZBCAN does not prevent an ECU from corrupting the payload of its own messages. However, the *officer* could be easily extended to check the *DLC*, *RTR*, and payload fields to detect or prevent this scenario, as mentioned in the discussion section.

12 Conclusions

In this paper, we proposed ZBCAN, a novel defense system that exploits inter-frame spacing to protect against the most common CAN attack vectors, offering *attack detection, prevention, individual message guarantees, intrusion confinement, incremental deployability*, and full *backward compatibility* without using any message fields or computationally expensive operations such as encryption. We introduced a novel and instant way to suspend nodes called *the instant bus-off* technique, which we used for defense purposes against intruding nodes. We proved the schedulability of messages on systems implementing ZBCAN by offering a theoretical worst-case response time analysis for such systems. We offered a probabilistic security analysis for ZBCAN against different attack types. Finally, we proved the applicability of our system by evaluating different aspects of it on a testbed using artificial data, then a testbed using a real vehicle's data, and finally on a real vehicle's CAN bus.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by the Office of Naval Research (ONR) under Grants: N00014-22-1-2671 and N00014-18-1-2674. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

[1] CP AUTOSAR. Specification of secure onboard communication. *AUTOSAR CP Release*, 4(1), 2017.

[2] Giampaolo Bella, Pietro Biondi, Gianpiero Costantino, and Ilaria Matteucci. Toucan: A protocol to secure controller area network. In *Proceedings of the ACM Workshop on Automotive Cybersecurity*, pages 3–8, 2019.

[3] Rohit Bhatia, Vireshwar Kumar, Khaled Serag, Z Berkay Celik, Mathias Payer, and Dongyan Xu. Evading voltage-based intrusion detection on automotive CAN. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[4] S. Checkoway, D. Mccoy, B. Kantor, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, pages 77–92, 2011.

[5] K.-T. Cho and K. G. Shin. Error handling of in-vehicle networks makes them vulnerable. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1044–1055, 2016.

[6] K. T. Cho and K. G. Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *USENIX Security Symposium*, pages 911–927, 2016.

[7] K. T. Cho and K. G. Shin. Viden: Attacker identification on in-vehicle networks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1109–1123, 2017.

[8] W. Choi, K. Joo, H. J. Jo, et al. VoltageIDS: Low-level communication characteristics for automotive intrusion detection system. *IEEE Transactions on Information Forensics and Security*, 13(8):2114–2129, 2018.

[9] Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[10] Robert I Davis, Steffen Kollmann, Victor Pollex, and Frank Slomka. Schedulability analysis for controller area network (can) with fifo queues priority queues and gateways. *Real-Time Systems*, 49(1):73–116, 2013.

[11] Marco Di Natale, Haibo Zeng, Paolo Giusto, and Arkadeb Ghosal. *Understanding and using the controller area network communication protocol: theory and practice*. Springer Science & Business Media, 2012.

[12] Shan Ding, Tong Zhao, Ryo Kurachi, and Gang Zeng. Id hopping can controller design with obfuscated priority assignment. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing (DASC)*, pages 94–99. IEEE, 2018.

[13] Josh D Eckhardt, Thomas E Donofrio, and Khaled Serag. System and method of monitoring data traffic on a mil-std-1553 data bus, November 5 2019. US Patent 10,467,174.

[14] Josh D Eckhardt, Thomas E Donofrio, and Khaled Serag. Bus data monitor, June 23 2020. US Patent 10,691,573.

[15] Josh D Eckhardt, Thomas E Donofrio, and Khaled Serag. Multiple security level monitor for monitoring a plurality of mil-std-1553 buses with multiple independent levels of security, June 16 2020. US Patent 10,685,125.

[16] International Organization for Standardization (ISO). Road Vehicles — Controller area network (CAN). *Part 1: Data link layer and physical signalling*, ISO-11898-1, 2015.

[17] International Organization for Standardization (ISO). Road Vehicles — Controller area network (CAN). *Part 2: High-speed medium access unit*, ISO-11898-2, 2016.

[18] M. Foruhandeh, Y. Man, R. Gerdes, et al. SIMPLE: Single-frame based physical layer identification for intrusion detection and prevention on in-vehicle networks. In *Annual Computer Security Applications Conference (ACSAC)*, pages 229–244, 2019.

[19] Ian Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage. Fast and vulnerable: A story of telematic failures. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[20] Bogdan Groza, Stefan Murvay, Anthony Van Herrewege, and Ingrid Verbauwhede. Libra-can: a lightweight broadcast authentication protocol for controller area networks. In *International Conference on Cryptology and Network Security*, pages 185–200. Springer, 2012.

[21] Bogdan Groza, Lucian Popa, and Pal-Stefan Murvay. Incanta-intrusion detection in controller area networks with time-covert authentication. In *Security and Safety Interplay of Intelligent Software Systems*, pages 94–110. Springer, 2018.

[22] Bogdan Groza, Lucian Popa, and Pal-Stefan Murvay. Canto-covert authentication with timing channels over optimized traffic flows for can. *IEEE Transactions on Information Forensics and Security*, 16:601–616, 2020.

[23] Bogdan Groza, Lucian Popa, Pal-Stefan Murvay, Yuval Elovici, and Asaf Shabtai. CANARY-a reactive defense mechanism for controller area networks based on active relays. In *30th USENIX Security Symposium*, 2021.

[24] Kyusuk Han, André Weimerskirch, and Kang G Shin. A practical solution to achieve real-time performance in the automotive network by randomizing frame identifier. *Proc. Eur. Embedded Secur. Cars (ESCAR)*, pages 13–29, 2015.

[25] Oliver Hartkopp and R Schilling. Message authenticated CAN (MaCAN). In *Escar Conference, Berlin, Germany*, 2012.

[26] Ahmed Hazem and HA Fahmy. Lcap-a lightweight can authentication protocol for securing in-vehicle networks. In *10th escar Embedded Security in Cars Conference, Berlin, Germany*, volume 6, page 172, 2012.

[27] T. Hoppe, S. Kiltz, and J. Dittmann. Security threats to automotive CAN networks–practical examples and selected short-term countermeasures. *Reliability Engineering & System Safety*, 96(1):11–25, 2011.

[28] Abdulmalik Humayed, Fengjun Li, Jingqiang Lin, and Bo Luo. Cansentry: Securing can-based cyber-physical systems against denial and spoofing attacks. In *European Symposium on Research in Computer Security*, pages 153–173. Springer, 2020.

[29] Abdulmalik Humayed and Bo Luo. Using id-hopping to defend against targeted dos on can. In *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles*, pages 19–26, 2017.

[30] K. Iehira, H. Inoue, and K. Ishida. Spoofing attack using bus-off attacks against a specific ECU of the CAN bus. In *IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 1–4, 2018.

[31] Sungwoo Kim, Gisu Yeo, Taegyu Kim, Junghwan "John" Rhee, Yuseok Jeon, Antonio Bianchi, Dongyan Xu, and Dave (Jing) Tian. Shadowauth: Backward-compatible automatic CAN authentication for legacy ECUs. In *the ACM on Asia Conference on Computer and Communications Security*, 2022.

[32] M. Kneib and C. Huth. Scission: Signal characteristic-based sender identification and intrusion detection in automotive networks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 787–800, 2018.

[33] Marcel Kneib, Oleg Schell, and Christopher Huth. EASI: Edge-based sender identification on resource-constrained platforms for automotive networks. In *Network and Distributed System Security Symposium (NDSS)*, pages 1–16, 2020.

[34] K. Koscher, A. Czeskis, F. Roesner, et al. Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy (S&P)*, pages 447–462, 2010.

[35] Ryo Kurachi, Yutaka Matsubara, Hiroaki Takada, Naoki Adachi, Yukihiro Miyashita, and Satoshi Horihata. Cacan-centralized authentication system in can (controller area network). In *14th Int. Conf. on Embedded Security in Cars (ESCAR 2014)*, 2014.

[36] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015:91, 2015.

[37] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. Chaskey: an efficient mac algorithm for 32-bit microcontrollers. In *International Conference on Selected Areas in Cryptography*, pages 306–323. Springer, 2014.

[38] Pal-Stefan Murvay and Bogdan Groza. Dos attacks on controller area networks by fault injections from the software layer. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, pages 1–10, 2017.

[39] M. Müter, André Groll, and Felix C Freiling. A structured approach to anomaly detection for in-vehicle networks. In *Sixth International Conference on Information Assurance and Security (IAS)*, pages 92–98, 2010.

[40] S. Nie, L. Liu, and Y. Du. Free-fall: Hacking Tesla from wireless to CAN bus. *Briefing, Black Hat USA*, 2017.

[41] S. Nie, L. Liu, Y. Du, and W. Zhang. Over-the-air: How we remotely compromised the gateway, BCM, and autopilot ECUs of Tesla cars. *Briefing, Black Hat USA*, 2018.

[42] Stefan Nürnberger and Christian Rossow. –vatican–vetted, authenticated can bus. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 106–124. Springer, 2016.

[43] Andrea Palanca, Eric Evenchick, Federico Maggi, and Stefano Zanero. A stealth, selective, link-layer denial-of-service attack against automotive networks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 185–206, 2017.

[44] Mert D Pesé, Jay W Schauer, Junhui Li, and Kang G Shin. S2-can: Sufficiently secure controller area network. In *Association for Computing Machinery*, ACSAC, page 425–438, 2021.

[45] Andreea-Ina Radu and Flavio D Garcia. Leia: A lightweight authentication protocol for can. In *European Symposium on Research in Computer Security*, pages 283–300. Springer, 2016.

[46] S. U. Sagong, X. Ying, A. Clark, et al. Cloaking the clock: Emulating clock skew in controller area networks. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 32–42, 2018.

[47] Khaled Serag, Rohit Bhatia, Vireshwar Kumar, Z Berkay Celik, and Dongyan Xu. Exposing new vulnerabilities of error handling mechanism in CAN. In *30th USENIX Security Symposium*, 2021.

[48] Khaled Serag, Vireshwar Kumar, Z Berkay Celik, Rohit Bhatia, Mathias Payer, and Dongyan Xu. Attacks on can error handling mechanism. *International Workshop on Automotive and Autonomous Vehicle Security (AutoSec)*, 2022.

[49] H. M. Song, H. R. Kim, and H. K. Kim. Intrusion detection system based on the analysis of time intervals of CAN messages for in-vehicle network. In *International Conference on Information Networking (ICOIN)*, pages 63–68, 2016.

[50] A. Taylor, N. Japkowicz, and S. Leblanc. Frequency-based anomaly detection for the automotive CAN bus. In *World Congress on Industrial Control Systems Security (WCICSS)*, pages 45–49, 2015.

[51] K Tindell, Alan Burns, and Andy Wellings. Calculating controller area network (can) message response times. *IFAC Proceedings Volumes*, 27(15):35–40, 1994.

[52] Ken Tindell and Alan Burns. Guaranteeing message latencies on control area network (CAN). In *Proceedings of the 1st International CAN Conference*, 1994.

[53] Ken Tindell, Alan Burns, and Andy J Wellings. Calculating controller area network (can) message response times. *Control engineering practice*, 3(8):1163–1169, 1995.

[54] Anthony Van Herrewege, Dave Singelee, and Ingrid Verbauwhede. Canauth-a simple, backward compatible broadcast authentication protocol for can bus. In *ECRYPT Workshop on Lightweight Cryptography*, volume 2011, page 20, 2011.

[55] Stien Vanderhallen, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. Robust authentication for automotive control networks through covert channels. *Computer Networks*, 193:108079, 2021.

[56] Qiyan Wang and Sanjay Sawhney. Vecure: A practical security framework to protect the can bus of vehicles. In *2014 International Conference on the Internet of Things (IOT)*, pages 13–18. IEEE, 2014.

[57] Haohuang Wen, Qi Alfred Chen, and Zhiqiang Lin. Plug-n-pwned: Comprehensive vulnerability analysis of obd-ii dongles as a new over-the-air attack surface in automotive iot. In *29th USENIX Security Symposium*, pages 949–965, 2020.

[58] Samuel Woo, Hyo Jin Jo, and Dong Hoon Lee. A practical wireless attack on the connected car and security protocol for in-vehicle can. *IEEE Transactions on intelligent transportation systems*, 16(2):993–1006, 2014.

[59] Samuel Woo, Daesung Moon, Taek-Young Youn, Yousik Lee, and Yongeun Kim. Can id shuffling technique (cist): Moving target defense strategy for protecting in-vehicle can. *IEEE Access*, 7:15521–15536, 2019.

[60] Fuyu Yang. A bus off case of CAN error passive transmitter. *EDN Technical paper*, 2009.

[61] Xuhang Ying, Giuseppe Bernieri, Mauro Conti, and Radha Poovendran. Tacan: Transmitter authentication through covert channels in controller area networks. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 23–34, 2019.

[62] Wang Yong. A scheduling algorithm for CAN bus. Master's thesis, National University of Singapore, 2004.

[63] Clinton Young, Habeeb Olufowobi, Gedare Bloom, and Joseph Zambreno. Automotive intrusion detection based on constant CAN message frequencies across vehicle driving modes. In *Proceedings of the ACM Workshop on Automotive Cybersecurity*, pages 9–14, 2019.

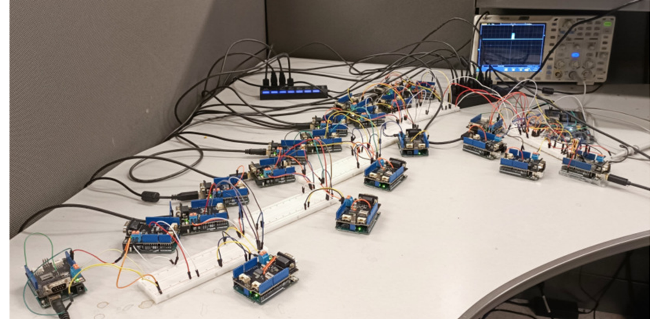

Figure 18: Testbed with 20 ECUs *(agents)* and the *officer*.

A Evaluation Details

Table 8 shows the messages of a 2011 Chevy-Impala. It also shows their theoretical WCRTs under four different settings. The first is without ZBCAN applied. The rest are after running Alg. 1 to find the optimum grouping and to calculate the WCRT for each group under three different $|IBNSpan|$ $settings$. Table 9 shows the messages we used for our scalability evaluation. In the table, (G) refers to the priority group assignment, (Type) specifies whether the message is periodic or aperiodic, and (R) refers to the theoretical WCRT. In both tables, we used a queueing jitter $J_m = 750\ \mu s$. This number is based on the empirical observation of our testbed. Finally, Fig. 18 shows our scalability testbed.

B System Design

B.1 Agent Components

As shown in Fig. 19, the agent is composed of four blocks: a Start Of Frame (SOF) ISR, an End Of Frame (or receive/transmit/error) (EOF) ISR, a $Timer$ ISR, and a *buffering and IBN sequence extension* library.

B.2 Sequence Exchange and Extension

Sequence Exchange Details. Each *agent* has a secret key, *pre-shared only with the officer*. *Agents* do not know each others' keys. However, the *officer* knows the *pre-shared keys* of all *agents*. The details of pre-sharing this data are outside the scope of this paper. Using these keys, each *agent* starts its operation by securely and randomly generating a seed SR then exchanging it with the *officer*. Both the *agent* and *officer* use the seed, the *pre-shared key* and an agreed-upon *pseudo-random function (PRF)* to generate a *session key*. Next, based on the number of IDs per ECU ($Nids_{ecu}$), both the *agent* and *officer* generate $Nids_{ecu}$ seeds, each separated by an agreed-upon offset Off. The first ID's seed is $= SR + Off$ and the last ID's seed is $= SR + (Off * Nids_{ecu})$. Finally, using the

Table 8: WCRTs and *groups* for a 2011 Chevy-Impala.

Without ZBCAN				With ZBCAN					
ID	Length (B)	Period (μs)	WCRT (μs)	(32,32,32)		(64,64,64)		(32,64,128)	
				WCRT (μs)		WCRT (μs)		WCRT (μs)	
C1	8	9000	1274	P0	5290	P0	6250	P0	5546
C5	8	9000	1536						
1E5	8	9000	6026						
F1	4	9976	1962						
199	8	12477	4348						
F9	8	12486	2224						
C9	7	12488	1778						
191	8	12495	4086						
1C7	7	17980	5356						
1CD	5	17981	5560						
1E9	8	17981	6288						
184	6	18025	3824						
1C3	8	24986	5114	P1	14538	P1	17090	P1	15398
19D	8	25005	4610						
1F5	8	25014	6958						
1A1	7	25016	4852						
334	2	29977	7530						
1E1	5	30113	5764						
1F3	2	33278	6696						
2F9	5	47939	7384						
348	4	47954	7714						
34A	4	47956	7898						
2C3	6	50025	7180						
17D	8	98891	3340						
17F	8	98920	3602						
1F1	8	99694	6550						
134	4	99841	2874						
12A	8	99842	2690			P2	49676		
3C9	8	99848	8422					P2	43786
3C1	8	99938	8160	P2	32002				
3E9	8	99988	8946						
3D1	8	100052	8684						
3F1	8	233192	9208						
3FB	2	249738	10586						
3F9	8	249805	10440						
4D1	8	499492	11372						
4C1	8	499713	10848						
4C9	8	499872	11110						
4E1	8	997963	11634						
773	7	998184	14562						
500	4	998189	12284						
771	7	998237	14078						
4E9	5	998391	11838						
138	5	998424	3078						
514	8	998942	12546						
52A	8	999229	13836						
4F1	8	999307	12100						
772	7	999347	14320						
77F	8	999751	14824						
120	5	4992122	2428						

seed, *session key*, and *PRF*, we generate a number of length *SeqLength*. This number, per ID, will act as the ID's first *IBN* sequence (Seq_{id}).

Sequence Usage. With every transmission, the *agent* extracts $\log_2 |PSpan_{group}|$ *bits* from Seq_{id}. The value of the bits act as the scheduled *IBN* value for the next message. For example, if a message belongs to a *priority group* whose $PSpan_{group} = [32,63]$, then $|PSpan_{group}| = 32$, whose $\log_2$ is 5. If we extract the 5 *bits*, pointed at by $index_{id}$, and find their value $= 15$, then $IBN_{sc} = 32+15 = 47$ *bits*. Once transmission is initiated at bit 47, we increment $index_{id}$.

Sequence Extension. To keep a running sequence, we recommend using a fast *PRF*. As shown in Fig. 20, the *agent and officer* start with a *session key* and a different seed per ID.

Table 9: Scalability evaluation dataset.

ID	Type	T (ms)	D (ms)	R (ms)	G	Safety Ratio	ID	Type	T (ms)	D (ms)	R (ms)	G	Safety Ratio
1	A	10	10	7.3	0	0.73	33	P	250	250	96	2	0.384
2	P	10	10	7.3	0	0.73	34	P	250	250	96	2	0.384
3	P	10	10	7.3	0	0.73	35	P	250	250	96	2	0.384
4	P	10	10	7.3	0	0.73	36	P	250	250	96	2	0.384
5	P	10	10	7.3	0	0.73	37	P	500	500	96	2	0.192
6	A	25	25	7.3	0	0.292	38	P	500	500	96	2	0.192
7	P	25	25	7.3	0	0.292	39	P	500	500	96	2	0.192
8	P	25	25	7.3	0	0.292	3A	P	500	500	96	2	0.192
9	P	25	25	7.3	0	0.292	3B	P	500	500	96	2	0.192
A	P	25	25	7.3	0	0.292	3C	P	500	500	96	2	0.192
B	P	25	25	7.3	0	0.292	3D	P	500	500	96	2	0.192
C	P	25	25	7.3	0	0.292	3E	P	500	500	96	2	0.192
D	P	25	25	7.3	0	0.292	3F	P	500	500	96	2	0.192
E	P	25	25	7.3	0	0.292	40	P	500	500	96	2	0.192
F	A	50	50	37.11	1	0.7422	41	P	500	500	96	2	0.192
10	P	50	50	37.11	1	0.7422	42	P	500	500	96	2	0.192
11	P	50	50	37.11	1	0.7422	43	P	500	500	96	2	0.192
12	P	50	50	37.11	1	0.7422	44	P	500	500	96	2	0.192
13	P	50	50	37.11	1	0.7422	45	P	500	500	96	2	0.192
14	P	50	50	37.11	1	0.7422	46	P	500	500	96	2	0.192
15	P	50	50	37.11	1	0.7422	47	P	500	500	96	2	0.192
16	P	50	50	37.11	1	0.7422	48	P	500	500	96	2	0.192
17	P	50	50	37.11	1	0.7422	49	P	500	500	96	2	0.192
18	P	100	100	37.11	1	0.3711	4A	P	500	500	96	2	0.192
19	P	100	100	37.11	1	0.3711	4B	P	500	500	96	2	0.192
1A	P	100	100	37.11	1	0.3711	4C	P	500	500	96	2	0.192
1B	P	100	100	37.11	1	0.3711	4D	P	500	500	96	2	0.192
1C	P	100	100	37.11	1	0.3711	4E	P	500	500	96	2	0.192
1D	P	100	100	37.11	1	0.3711	4F	P	500	500	96	2	0.192
1E	P	100	100	37.11	1	0.3711	50	P	500	500	96	2	0.192
1F	P	100	100	37.11	1	0.3711	51	P	500	500	96	2	0.192
20	P	100	100	37.11	1	0.3711	52	P	500	500	96	2	0.192
21	P	100	100	37.11	1	0.3711	53	P	500	500	96	2	0.192
22	P	100	100	37.11	1	0.3711	54	P	500	500	96	2	0.192
23	P	100	100	37.11	1	0.3711	55	P	500	500	96	2	0.192
24	P	100	100	37.11	1	0.3711	56	P	500	500	96	2	0.192
25	P	100	100	37.11	1	0.3711	57	P	500	500	96	2	0.192
26	P	100	100	37.11	1	0.3711	58	P	500	500	96	2	0.192
27	P	100	100	37.11	1	0.3711	59	P	1000	1000	96	2	0.096
28	P	100	100	37.11	1	0.3711	5A	P	1000	1000	96	2	0.096
29	P	100	100	37.11	1	0.3711	5B	P	1000	1000	96	2	0.096
2A	P	100	100	37.11	1	0.3711	5C	P	1000	1000	96	2	0.096
2B	P	100	100	37.11	1	0.3711	5D	P	1000	1000	96	2	0.096
2C	P	100	100	37.11	1	0.3711	5E	P	1000	1000	96	2	0.096
2D	P	100	100	37.11	1	0.3711	5F	P	1000	1000	96	2	0.096
2E	P	100	100	37.11	1	0.3711	60	P	1000	1000	96	2	0.096
2F	P	100	100	37.11	1	0.3711	61	P	1000	1000	96	2	0.096
30	P	100	100	37.11	1	0.3711	62	P	1000	1000	96	2	0.096
31	P	100	100	37.11	1	0.3711	63	P	1000	1000	96	2	0.096
32	P	100	100	37.11	1	0.3711	64	P	1000	1000	96	2	0.096

Using the *PRF*, they generate an initial sequence of length *SeqLength* for each ID and start drawing bits from it with every transmission. A circular buffer holding two sequences (a current one and a future one) should be kept to avoid interruptions. Once a sequence is consumed, a new one should be extended to replace it. We also recommend using a counter of length *SeqLength*, to be incremented and *XORed* with the session key with every extension for augmented security. *SeqLength* needs to be small enough for a limited-space ECU to be able to store it. For a typical configuration of $SeqLength = 128b$ and $|IBNSpan| = 64b$, a message draws $6b$, an extension happens every $128/6 \approx 21$ messages.

Sequence Exchange Frequency. Without counters, sequences could be extended until the result of one extension operation repeats or equals the initial seed. If that happens, all the following sequences will also repeat. For a $128b$ sequence, the probability of an extension generating such a number is very low ($1/2^{128}$ with every extension). Further, with a counter, both the output and the counter values need to be the same as a previous combination for sequences to start repeating. The probability of that is even lower ($1/2^{256}$).

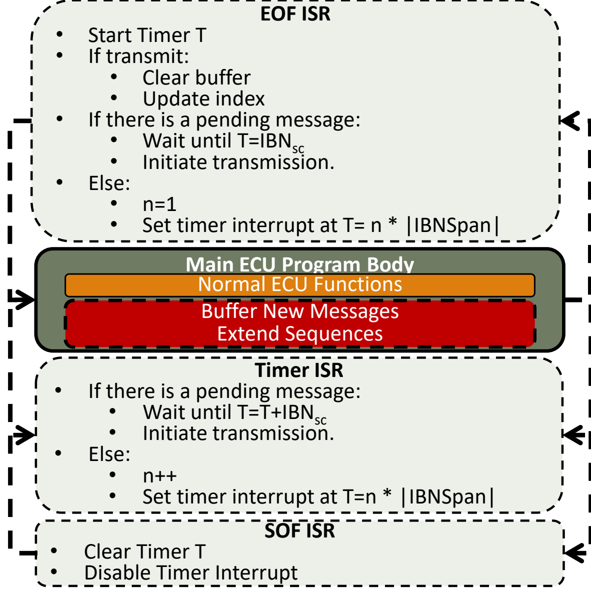

Figure 19: *Agent* components (dashed) within an ECU.

For an *agent* with 16 IDs, if we do not want two IDs to have the same counter values for stricter security, we can divide the counter values into exclusive ranges for each ID and only extend the sequence until the counter reaches the end of its range ($2^{128}/16$ *extensions*). Assuming an extension covers 21 *messages*, then for a fast transmitting ID, with a 10 *ms* period, we could perform sequence extensions for $21*10*2^{128}/16 = 4.46*10^{39}$ *ms*, before a counter repeats. Alternatively, *agents* could perform an exchange once at the beginning of operation.

C Security and Performance Analysis

C.1 Agent's Overhead Analysis

Memory. To function properly, *agents* require a minimum amount of memory as follows: (1) $L_{preshared}$ *bits* for the preshared key. (2) $L_{session}$ *b* for the session key. (3) $L_{counter}$ * $N_{ids-agent}$ *b* for the counter value of every ID transmitted by the *agent*. (4) $L_{sequence}$ * $N_{ids-agent} * 2$ *bits* to hold the *IBN* sequence for each of the IDs transmitted by the *agent*. Note that we store *two* sequences for each ID (Appendix B.2). (5) L_{index} * $N_{ids-agent}$ *bits* for the index of the next *IBN* within each sequence for each ID. Assuming $L_{preshared} = L_{session} = L_{counter} = L_{sequence}$, the minimum amount of memory required is: $(2*L_{sequence}) + ((3*L_{sequence}) + L_{index}) * N_{ids-agent}$ *bits*.

Sequence Extension Processing Overhead. An *agent* performs a *sequence extension* operation every time a sequence for a specific message ID runs out. This means that a *sequence extension* operation for a specific ID happens every $L_{seq}/\log_2(|PSpan|)$ outgoing *messages*. This number should be multiplied by the average period of message transmission for all IDs in the ECU (P_{av}) to calculate the average time between extensions (P_{ext}). We use (d_{ext}) to refer to the time needed to perform the extension operation itself. We recommend picking a fast *PRF* to perform the extension and minimize the overhead ratio $O_{ext} = d_{ext}/P_{ext}$. In our evaluation (Sec. 8), we achieved $O_{ext} \approx 1.91\%$ on the *agent's* side.

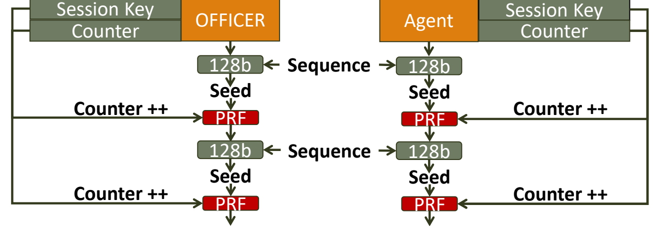

Figure 20: Extending a 128*b* sequence.

C.2 Officer's Overhead Analysis

Memory. The minimum amount of memory required by the *officer* is as follows: (1) $L_{preshared} * N_{agents}$ *b* for the preshared keys of all agents. (2) $L_{session} * N_{agents}$ *b* for the session keys of all agents. (3) $L_{counter}$ * $N_{ids-system}$ *b* for the counter value of every ID in the system. (4) $L_{sequence}$ * $N_{ids-system} * 2$ *b* for every *IBN* sequence in the system. (5) L_{index} * $N_{ids-system}$ *bits* for the index of the next *IBN* within every sequence in the system. Assuming $L_{preshared} = L_{session} = L_{counter} = L_{sequence}$, the minimum amount of required memory is: $(2*L_{sequence} * N_{agents}) + ((3*L_{sequence}) + L_{index}) * N_{ids-system}$ *bits*.

Sequence Extension Processing Overhead. The *officer* performs a *sequence extension* operation whenever a sequence for a specific message ID runs out, meaning, a *sequence extension* operation for a specific ID happens every $L_{seq}/\log_2(|PSpan|)$ observed *messages*. In our evaluation (Sec. 8), we achieved $O_{ext} \leq 0.018\%$ on the *officer's* side.

C.3 Measuring d_{skew}

d_{skew} is related to the clock skews of every ECU, the clock skews of their CAN controllers, the ISR of each *agent*, its timer settings, and age. To measure d_{skew} of the system, several methods could be used. In our system, we measured it empirically. We ran the *Span Scan False Positive Test*, in which we connect a traffic source, sending a *reference message* every 6 *ms*. Next, we connect the first test ECU and set it up to send a message after every *reference message* with an *ascending IBN* between 0 *b* and 3000 *b*. This means that the test ECU sends the first message with an $IBN = 0$ *b* and the second with $IBN = 1b$ and so on until it reaches 3000 *b*, then it rolls over to 0*b* again. Meanwhile, the *officer* monitors the *IBN* of each message and verifies their order. Once a message has an unexpected *IBN*, it is flagged as a false positive. We repeated the scan for 5 *min* for each ECU. For each ECU, we marked the smallest *IBN* value after which inaccuracies start to occur as $d_{skew(ECU)}$. For all ECUs, the smallest d_{skew} was 1189 *b*, and the largest d_{skew} was 1460 *b*. We select $d_{skew(sys)}$ as the smallest $d_{skew} = 1189$ *b* for all ECUs.