MichiCAN: Spoofing and Denial-of-Service Protection using Integrated CAN Controllers

Mert D. Pesé[*], Bulut Gozubuyuk[*], Eric Andrechek[†], Habeeb Olufowobi[‡], Mohammad Hamad[§], and Kang G. Shin[†]
[*]Clemson University, {mpese, bgozubu}@clemson.edu
[†]University of Michigan, {ericandr, kgshin}@umich.edu
[‡]University of Texas at Arlington, habeeb.olufowobi@uta.edu
[§]Technical University of Munich, mohammad.hamad@tum.de

***Abstract*—The Controller Area Network (CAN) has been the de facto in-vehicle network protocol since the 1980s, despite lacking essential security principles like authenticity, confidentiality, integrity, and availability. CAN is especially vulnerable to Denial-of-Service (DoS) attacks, threatening the availability of safety-critical functions. Existing countermeasures have seen limited adoption due to challenges in real-time detection, prevention, and high overhead on Electronic Control Units (ECUs). To address these issues, we propose `MichiCAN`, a distributed, backward-compatible, real-time defense against DoS and spoofing attacks. `MichiCAN` leverages integrated/on-chip CAN controllers in modern MCUs, enabling bit-level access to CAN messages. This allows `MichiCAN` to detect DoS attacks during the arbitration phase and neutralize them by bussing off the attacker ECU swiftly. Experiments on a CAN bus prototype and a real vehicle demonstrate `MichiCAN`'s effectiveness in enhancing automotive network security.**

I. Introduction

The Controller Area Network (CAN) is an inexpensive, message-based and de-facto in-vehicle network (IVN) protocol serving for over three decades, facilitating communication among various Electronic Control Units (ECUs). Initially designed without security considerations, CAN's vulnerabilities have been exploited as modern vehicles' complexity exposes the driving functions to cyber-physical attacks [1]–[3]. In response, researchers soon started to develop countermeasures to address CAN security, aiming to provide essential security properties such as confidentiality, authenticity, integrity, and availability. However, the foundational CAN protocol lacks inherent security features as all traffic traverses the network in plaintext, without sender or message authentication mechanisms. Consequently, CAN remains highly susceptible to availability-based attacks, including Denial-of-Service (DoS).

Previous research has extensively investigated message authentication codes to ensure message integrity and anomaly detection approaches to identify malicious messages [4]–[6]. However, authentication techniques often require resource-intensive cryptographic computations, potentially introducing performance overhead [7]. Intrusion detection systems (IDS), while capable of identifying attacks, typically operate reactively, detecting anomalies only after disrupting bus communications [8]. Additionally, certain CAN spoofing prevention frameworks [9]–[12], while aiming to mitigate attacks, may impose significant network overhead or lack real-time detection capabilities, limiting their effectiveness in rapidly evolving threat scenarios. Furthermore, Bozdal *et al.* [13] show how an encrypted CAN payload can prevent sniffing attacks. These schemes use cryptography, imposing heavy computation loads on resource-constrained ECUs and incurring a significant computation delay. Furthermore, some standardization for secure onboard communication is provided by the AUTOSAR consortium [14], although it mainly deals with protecting integrity, authenticity and confidentiality.

DoS attacks target the CAN message identifier (CAN ID) by injecting CAN messages with low CAN IDs which indicate higher priority. They will always win arbitration and be allowed to transmit before higher ID messages on the CAN bus. By "continuously" sending CAN messages with a low ID, higher ID messages will always lose arbitration and thus become *unavailable* on the CAN bus. Attackers can choose to make the transmission of all ECUs unavailable (*traditional* DoS) or selectively choose which ECUs to silence (*targeted* DoS). For instance, a traditional DoS attacker continuously injects CAN messages with ID 0x0 and blocks other ECUs' communications on CAN. The major impact of a DoS attack could be safety-critical, mainly when the vehicle can no longer perform certain powertrain control functions. However, vehicles implement a *limp mode*, which still allows certain safety-critical ECUs to work with limited functionality in the event of losing CAN communication [23].

TABLE I: Comparison of countermeasures against CAN DoS

	Backward Compatibility	**Real-Time Capability**	**Network Overhead**	**Prevention Capability**
IDS [15]–[17]	●	○	🟢	○
Parrot+ [18]	●	○	🔴	●
CANSentry [19]	○	○	🟡	●
CANeleon [20]	○	◑	🟢	●
CANARY [21]	○	●	🟢	●
ZBCAN [22]	●	●	🟠	●
`MichiCAN`	●	●	🟡	●

○ No, ◑ Unknown, ● Yes
🟢 None, 🟡 Negligible, 🟠 Medium, 🔴 Very High

Literature on detecting and preventing DoS attacks has been scarce (see Table I). Moreover, all of them come with their own limitations, such as lack of backward compatibility, real-time capability or heavy CAN traffic overhead.

No backward compatibility. Ideally, the countermeasure against DoS attacks should be software-based. Any additional hardware or modifications of existing hardware are not backward-compatible with existing cars. Original Equipment Manufacturers (OEMs) and their supply chain would have to produce or modify their products to add this countermeasure, which is not viable for *cost* reasons. CANSentry [19] is a hardware-based message firewall that can defend against various spoofing and DoS attacks. However, CANSentry introduces a stand-alone device deployed between a high-risk ECU (i.e., the ECU with the highest risk to be compromised) and the CAN bus which limits its backward compatibility. Also, CANeleon [20] is limited in its applicability to the classical CAN protocol with baseline message identifier and payload of 0-8 bytes. Furthermore, CANARY [21] presents a defense system with physical relays on the CAN bus.

No real-time capability. Since DoS attacks may impact driving safety, they must be detected and prevented as quickly as possible, preferably in real time. The surveyed IDS mechanisms [15]–[17] cannot detect attacks in real time. CANSentry [19] incurs an additional propagation time because the intermediate hardware must decode and re-encode the message before passing it to the main bus. Parrot [18] is a distributed spoofing detection and prevention framework where each ECU is responsible for monitoring the bus to detect frames sent by other ECUs that contain its own CAN ID. Parrot will then launch a counterattack to *bus off* the attacking ECU. However, it incurs an additional delay in launching a counterattack (i.e., bus-off time) because it only starts destroying an attacker message after its second instance, with the first instance used for detection. Parrot was not designed for DoS prevention, but can effectively be used as such.

Traffic overhead on the network. The CAN bus load defines how busy the bus is at any given time. It must be kept as low as possible to avoid the difficulty of scheduling messages (with safety-critical implications), with 80% being the recommended upper bound [24] and observed bus load of 40% in real vehicles [18]. When Parrot [18] launches the counterattack, it must start at the same time as the second instance of the attacker's CAN message. As a result, Parrot *floods* the CAN bus with counterattack messages to collide with the attacker's CAN messages in a brute-force fashion. The bus load can reach 100% during those times, making Parrot unusable in real vehicles. In addition, ZBCAN [22], which utilizes zero bytes of the message fields in CAN frames, introduces a minimal increase in bus load due to propagation delays. While seemingly insignificant, this increase may impede the transmission of safety-critical messages.

Eradication. Just detecting a DoS attack is not helpful as all subsequent communications will be halted. It is imperative to counter the DoS attack. This is especially important due to possibly safety-critical consequences of a DoS attack. IDSes usually detect DoS attacks, but do not have any means to eradicate them. Furthermore, even their detection capabilities can be questionable since most IDSes are generally centralized and susceptible to a single point of failure.

To overcome all these limitations, we propose `MichiCAN`, a distributed software solution that can run on any modern ECU. It does not only detect DoS attacks, but can also be used for spoofing prevention. Each ECU equipped with `MichiCAN` stores a list of legitimate CAN IDs from the set $\mathbb{E}$ of all participating ECUs in the IVN. A CAN node $ECU_i \in \mathbb{E}$ can detect a spoofing attack if another node transmits their own CAN ID. ECU_i can also flag CAN messages with lower IDs as a DoS attack if the CAN ID is not part of its stored list $\mathbb{E}$ and thus not a legitimate CAN ID. After marking the incoming CAN message as malicious, ECU_i will start a counterattack. It exploits CAN's error handling flaws to force the attacking ECU into *bus-off*, halting its communication. Forcing ECUs into bus-off state is not new as previous work [25] has shown. In fact, there is a growing literature on bus-off attacks to silence legitimate ECUs [26], [27] which leverage the aforementioned vulnerabilities in the CAN's error handling. The main challenge in busing off an attacker is the timing of counterattack. Since the application software can only send and receive complete CAN frames, the "defender" ECU needs to know *precisely when* to start the counterattack so its message can exactly synchronize with the attacker's CAN message. This was a major deficiency of Parrot [18], as well as others, such as the one proposed by Cho *et al.* [25]. In contrast, `MichiCAN` is leveraging the integrated CAN controller of modern ECUs, allowing the software to gain direct read/write access to each bit of a CAN frame. This technique is called *bit banging*. Recently, bit banging is shown to be accomplishable using such protocols as SPI, UART and I2C [28]. Unlike Parrot, which floods the bus, `MichiCAN` remains synchronized, allowing for precisely timed counterattacks without adding to the bus load. Furthermore, by sampling the CAN ID bit-by-bit, `MichiCAN` will also be able to detect a spoofing or DoS attack before the end of the 11-bit CAN ID field in most cases. This allows quicker counterattacks, reducing the time needed to bus off an attacking ECU. It is the first practical DoS protection for production vehicles. Although it demands more CPU time from ECUs, it is still deployable in modern units. Our evaluations confirm that `MichiCAN` has minimal network overhead and effectively eradicates attackers without affecting critical vehicle communication. Finally, we deployed `MichiCAN` on other MCUs and a real vehicle, corroborating our experimental findings.

The software artifact for `MichiCAN` is publicly available at https://github.com/tigerseclab/DSN25_MichiCAN.

II. Background

A. CAN Primer

Vehicular sensor data is collected from ECUs located within a vehicle. These ECUs are typically interconnected via an in-vehicle network (IVN), with the CAN bus being the most widely-deployed technology in current vehicles. Fig. 1a depicts the structure of a CAN 2.0A data frame — the most common data-frame type used in CAN:

SOF: The *start-of-frame* (SOF) bit indicates the beginning of a new CAN frame/message and is always set to 0.

CAN ID: CAN is a multi-master, message-based bus where frames lack source or destination information, using unique IDs to represent meaning and priority. Lower IDs have higher priority and "win" arbitration when multiple messages compete, thanks to CAN's wired-AND logic, where a dominant "0" overwrites a recessive "1." An ECU can send or receive messages with different IDs. CAN 2.0A supports 11-bit IDs, allowing up to 2,048 unique messages.

RTR, IDE & Reserved: *Remote transmission request* (RTR), *identifier extension* (IDE), *reserved* (r0) bits are 0.

DLC: This field specifies the number of (up to 8) bytes in the payload (data) field of the message.

Data: This is the payload field of a CAN message containing the actual 0–8 bytes of message data.

CRC-15: To detect transmission errors, a *cyclic redundancy check* (CRC) is calculated over all previous fields.

ACK & EOF: The first bit of the *acknowledgment* (ACK) field is called *ACK slot* and the second bit *ACK delimiter*. The ACK slot is always set to 1 for the transmitting ECU. If the receivers do not observe any errors in the frame, they send a 0 during this slot. Due to the wired-AND logic, at least one receiver needs to transmit a 0, acknowledging the correct receipt of the CAN frame. If the transmitter (which reads back this slot) detects that nobody acknowledged this frame by sending a 0, it will retransmit the frame. The ACK delimiter, as well as the *end-of-frame* (EOF) is always 1. After the transmission of a CAN frame, the next CAN frame has to wait another 3 bits (not depicted in Fig. 1a) which is called *inter-frame spacing* (IFS). As a result, the next CAN message can only be transmitted after at least 11 recessive bits.

B. CAN Error Handling

There are 5 CAN error types: (i) bit monitoring, (ii) bit stuffing, (iii) frame check, (iv) acknowledgment check, and (v) cyclic redundancy check. For our purposes, we focus on the first two. A *bit monitoring error* occurs if the bit read on the CAN bus by an ECU is different from the bit level that it has written. Obviously, no bit errors are raised during the arbitration process. A *bit stuffing error* is caused by 6 consecutive bits of the same logic level. According to the CAN protocol, when 5 consecutive bits of the same logic level have been transmitted by a node, it will pad a sixth bit of the opposite level to the outgoing bit stream. The receiving ECUs will remove this sixth bit before passing it to the application.

Each ECU on the CAN bus has a *transmit error counter* (TEC) and a *receive error counter* (REC). In this paper, we focus mainly on transmission errors. A transmission error occurs when a transmitting ECU observes an error frame sent by a different ECU during its transmission of a CAN message on the bus. In such a case, a CAN-compliant node will do one of two things depending on the current value of its TEC. Each ECU starts in the *error-active state*. When the counter is between 0 and 127 (in the error-active state), the node that detects the error will transmit an *active error flag* consisting of 6 dominant (logical 0) bits followed by 8 recessive (logical 1) bits as an indication to all other nodes that the transmitted frame had an error and should be ignored. If the node's TEC is in the *error-passive state* (when the TEC exceeds 127), it will transmit a *passive error flag* consisting of 14 recessive bits. Note that the passive error does not destroy other bus traffic, and hence the other nodes will not hear "complaint" about bus errors. In both cases, the node will increment its TEC by 8 and then retransmit the message. The minimum separation between the original transmission and retransmission are 11 recessive bits (8 bits from error flag + 3 bits from IFS) in the error-active state and 25 recessive bits (14 bits from error flag + 3 bits from IFS + 8 bits from additional transmission suspension) in the error-passive state. When the TEC reaches 256, the node enters *bus-off mode* and will no longer participate in CAN traffic. According to CAN protocol, a device in bus-off mode is allowed to recover into the error-active state after observing at least 128 instances of 11 recessive bits on the bus.

It is also crucial to note that the TEC decreases after each successful transmission, reducing by one for each message that was successfully transferred. This method ensures the robustness of CAN by enabling nodes to recover from momentary errors and return to an error-active state (see Fig. 1b).

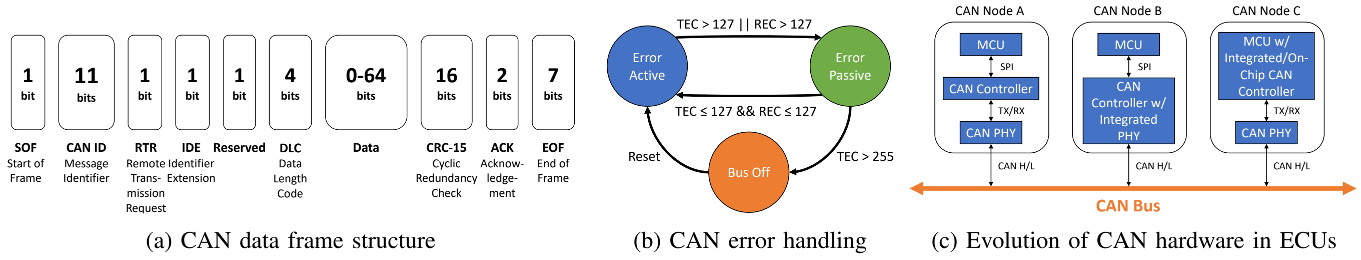
C. CAN Hardware

We refer to ECU as CAN nodes, typically consisting of an MCU, CAN controller, and CAN transceiver. The MCU handles applications, while the controller and transceiver manage communication.

CAN controllers operate at the *data link layer*, building CAN frames (from ID, DLC, Data) and performing bit stuffing, as well as error handling. It interfaces with the physical layer via CAN_TX (outbound) and CAN_RX (inbound).

CAN transceivers, or PHYs, operate at the *physical layer* and convert digital signals to analog voltages (0-5V) and vice versa, using differential signaling (CAN_H and CAN_L).

In the last decade, the internal design of CAN nodes had been gradually changing and an overview of this evolution is depicted in Fig. 1c. In early CAN nodes (A), the MCU, CAN controller and transceiver were separate chips, such as Microchip's MCP2515 [29] and MCP2551 [30]. The MCU/application would send and receive CAN frames from the controller via SPI. CAN node B is a slightly modified version of CAN node A, with the CAN controller and the transceiver combined in a single chip, mainly to reduce cost and space. One example for this is the MCP25625 [31] . The main novelty lies in CAN Node C which represents novel ECUs. It consists of an MCU with an integrated/on-chip CAN controller. The latter are embedded in MCUs and allow memory-mapped access to CAN bus functions. In many MCUs, this involves access to interpreted CAN data, configuration of filters, and access to interrupts on arrival of new messages. Further, MCUs



(a) CAN data frame structure (b) CAN error handling (c) Evolution of CAN hardware in ECUs

Fig. 1: Background on CAN

tend to allow *pin multiplexing* within the hardware at run-time, e.g., allowing the software to directly read and write each bit of the CAN_RX and CAN_TX lines. In Nodes A and B, the application could only pass certain CAN message fields such as CAN ID, DLC and data to the CAN controller which would be responsible for generating a complete CAN frame. The application could also only process the data from an incoming CAN message after successful receipt of the entire frame. MichiCAN exploits the MCUs with integrated CAN controllers to detect and prevent DoS attacks as fast as possible (see Sec. IV). On-chip CAN controllers are already widely used by major ECU manufacturers such as NXP, ST or Renesas [32]–[34]. One example is Renesas V850ES/FJ3 [34] which was used in the Jeep hack [3] in 2015.

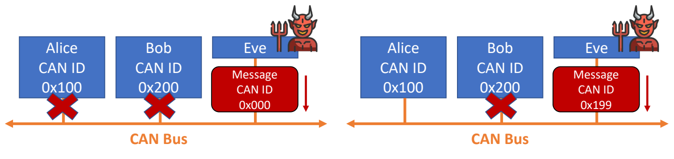
III. Threat Model

Consistent with prior work on CAN security [2], [3], [22], [35]–[38], we assume a remote attacker who has compromised an ECU *remotely* via Bluetooth, WiFi or cellular. The attacker can execute arbitrary code on the compromised operating system of the ECU but cannot modify the protocol controller or violate protocol specifications. The adversary is able to receive and send crafted CAN messages, *fabricating* commands such as unintended acceleration, *suspending* engine control, or *masquerading* sensor data to manipulate critical functions like steering, braking, and engine performance. While physical access might seem improbable, the attacker may also have physical access to the vehicle and therefore connect external hardware to the CAN bus through the OBD-II port [39], [40]. Under the above threat model, the adversary is able to conduct the following attacks:

Fabrication attacks inject spoofed CAN messages with valid IDs but arbitrary data. Without message authentication, ECUs accept them as legitimate. To override real messages, the attacker must transmit at a higher frequency.

Suspension attacks silence victim ECUs by preventing them from sending messages, effectively causing a DoS, which can be traditional (using lowest-priority ID 0x0 to block all ECUs), random, or targeted (disrupting specific ECUs) [15]. We focus on traditional and targeted attacks (see Fig. 2).

Masquerade attacks combine both fabrication and suspension by first suspending a legitimate ECU's CAN broadcast and then fabricating its data. They demonstrate why preventing DoS attacks is of utmost importance for a secure CAN bus.



(a) Traditional DoS (b) Targeted DoS

Fig. 2: Different types of DoS attacks [15]

Attacker Limitations. On the flip side, the ability to use integrated CAN controllers (used in MichiCAN as a defense mechanism) can be exploited to perform CAN injection attacks, since an attacker who compromises the MCU has direct read and write access to the CAN_RX and CAN_TX lines. As a result, it is of utmost importance to prevent a compromised ECU to access its CAN controller/protocol functionality. However, this can be mitigated on modern ECUs since a compromised operating system will not expose the CAN controller to the attacker as explained in the following.

The application of isolation techniques such as virtualization architectures on ECUs to run multiple operating systems — a current industry trend — is a possible way of solving this problem [41]. For instance, the in-vehicle infotainment (IVI) ECU may run a Virtual Machine (VM) with Android Automotive, which is vulnerable to remote compromise, alongside a separate VM running RTOS to implement CAN functionality (including MichiCAN). The hypervisor allows multiple VMs to run in parallel (see Fig. 3). Only the RTOS VM is allowed to access the CAN bus directly. Both VMs can communicate with each other using inter-VM communication. However, a compromised IVI VM will not be able to access CAN functionality directly to inject CAN data or misuse MichiCAN. Android Automotive, a popular IVI OS, is already supporting virtualization technology [42]. Android does not implement any CAN functionality itself, but exposes sensor data through a generic, hardware-agnostic *Vehicle Hardware Abstraction Layer* (VHAL) using VirtIO drivers [43]. IVI and RTOS VMs communicate with each other over GRPC-vsock. For instance, Android writes the AC fan speed by specifying the abstract sensor name and value which is transmitted to the RTOS VM using GRPC. The CAN logic in RTOS then builds a CAN frame with this information. However, the IVI VM can never access any advanced CAN functionality directly.

Although we highlight hypervisors and Memory Man-

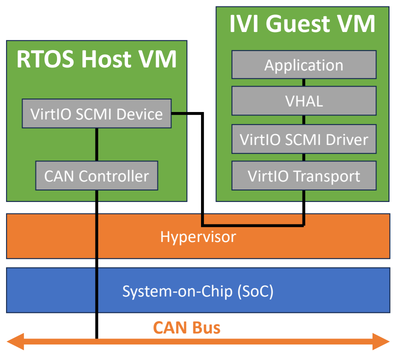


Fig. 3: Isolation between CAN Controller and Application

agement Units (MMUs) as effective tools to isolate CAN functionality from application code in high-end ECUs that run multiple VMs, lower-end ECUs typically rely on Memory Protection Units (MPUs) to separate memory regions. Increasingly, ECUs also support hardware security features such as ARM TrustZone [44]. For example, a Cortex-M3 (e.g., AT91SAM3X8E) uses only an MPU for basic isolation, while upgrading to Cortex-M33 enables TrustZone + MPU for stronger separation. Thus, a range of isolation options exist depending on budget, demonstrating that current automotive ECU technology can already address this challenge.

IV. System Design

`MichiCAN` operates in five phases: *Initial Configuration* is done offline and only once by the OEM at the time of manufacturing, *Synchronization* and *Detection* are performed for each received CAN message, whereas *Pin Multiplexing* and *Prevention* get engaged only for malicious CAN messages.

A. Initial Configuration

As mentioned in the introduction, `MichiCAN` is a distributed solution and can be implemented on every ECU on the IVN. Suppose there are N ECUs on the IVN, all of which are equipped with integrated CAN controllers as described in Sec. II. We define an ordered list of all ECUs as $\mathbb{E} = \{ECU_1, \ldots, ECU_N\}$ where $ECU_i \in \mathbb{E}$ represents its CAN ID. In the scope of `MichiCAN`, it is assumed that each unique CAN ID is tied to a specific ECU and is not transmitted by other nodes. Although not specified by ISO 11898 standard [45], this aligns with technical guidelines highlighting the importance of unique CAN IDs within a CAN bus network for error-free transmission [46], [47]. The reasons for this include ensuring consistent communication, avoiding compatibility concerns (e.g., with legacy ECUs that cannot be updated at all), and avoiding functional safety issues (changing a CAN ID would require a new ISO 26262 impact analysis and potentially re-certification). Additionally, publicly available communication matrices such as OpenDBC [48] which include information about CAN IDs, transmitters and signal definitions, demonstrate that a CAN ID can only be uniquely emitted by one specific ECU.

In this ordered list of ECUs, ECU_1 would have the lowest CAN ID and thus the highest priority, whereas ECU_N has the highest CAN ID and the lowest priority.

Similar to Parrot, $ECU_i \in \mathbb{E}$ detects a spoofing attack if it observes a CAN message with CAN ID ECU_A (injected by the adversary) that is equal to its own CAN ID:

Definition IV.1 (Spoofing Attack)**.** $ECU_i = ECU_A$.

ECU_i detects a DoS attack if it observes a message with a lower CAN ID ECU_A than its own ID that does not originate from any other legitimate ECU:

Definition IV.2 (DoS Attack)**.** $ECU_A < ECU_i, ECU_A \in \mathbb{E} \setminus ECU_j \; \forall j \in [1, N] \wedge i \neq j$.

For instance, if there are $N = 2$ ECUs in the IVN with $\mathbb{E} = \{0x005, 0x00F\}$, the ECU transmitting CAN ID 0x00F will detect all CAN IDs between 0x000 to 0x004 and 0x006 to 0x00F (including its own which would be a spoofing attack) as malicious. It cannot make a detection decision for CAN ID 0x005 since it can be a legitimate transmission from the other ECU. Only the ECU transmitting CAN ID 0x005 can decide whether a message on the CAN bus with its own ID is legitimate or not.

Furthermore, an attacker can inject a message with CAN ID ECU_A higher than ECU_N which is equal to the highest CAN ID in the IVN. This is called a *miscellaneous attack*:

Definition IV.3 (Miscellaneous Attack)**.** $ECU_A > ECU_N$.

If the attacker injects this message at the same time as another ECU, it will lose arbitration. If the message is injected during *bus idle*, i.e., when there are no other CAN messages transmitted on the bus, the attacker will naturally win arbitration and broadcast its message. Since no other ECUs know (or listen to) this CAN ID, there will be no perceivable impact on the vehicle's operation. The only drawback is that a higher-priority CAN message will need to wait until the attacker's message has completed transmission. Given that an average CAN frame consists of 125 bits, the blocking time at a 500 kBit/s bus speed is 250 μs. The higher-priority message which has been buffered by the legitimate ECU will then start its transmission after 11 recessive bits on the bus. Even if the attacker repeats its attack and finds a suitable bus-idle time, the maximum blocking delay for the legitimate ECU is much smaller than the deadline for safety-critical CAN messages which stands around 10 ms [49]. As a result, miscellaneous attacks can never interfere with legitimate CAN communications and thus do not pose a serious threat. Thus, we will focus on spoofing and DoS attacks. Each `MichiCAN`-equipped $ECU_i \in \mathbb{E}$ needs to store the *detection ranges* $\mathbb{D}$ of CAN IDs that it needs to mark as malicious:

Definition IV.4 (Detection Range $\mathbb{D}$)**.** $\mathbb{D} = \{j \mid 0 \leq j \leq ECU_i \wedge j \neq ECU_k \wedge 0 \leq k < i\}$.

Since integrated CAN controllers allow direct read access to every bit of the incoming CAN frame C during its transmission, the detection ranges $\mathbb{D}$ can be encoded as a *finite* *state machine* (FSM). In effect, the FSM is a binary tree since each transition "input" can be either 0 or 1. The root of the tree is the start-of-frame (SOF) bit since the 11-bit CAN ID $C = c_0||\ldots||c_{10}$ will immediately follow that bit. The FSM is run for each bit individually and needs to traverse all 11 bits only in the worst case. If a decision can be made after the 11-th bit or earlier, it will terminate since $C \in \mathbb{D}$ and set the *malicious* flag to true. Alternatively, if $C \notin \mathbb{D}$, the FSM will set the flag to false. The initial setup is a one-time, offline process by OEMs or Tier-1 suppliers. Unique FSMs are generated and patched into each ECU's source code. The patched firmware binaries are then distributed to the respective ECUs via software update which is getting increasingly common in vehicles [50], [51].

While `MichiCAN` can be implemented on all ECUs in the IVN, such a broad application might be practically difficult due to lack of integrated CAN controllers in all ECUs, computational resource limitations, as well as cost restrictions by OEMs. Currently, each $ECU_i \in \mathbb{E}$ will detect both spoofing and DoS attacks. This improves reliability and robustness, since each ECU_i will detect a malicious transmission simultaneously. This is very beneficial in case legitimate ECUs fail. Even if $|\mathbb{E}|-1$ ECUs fail (which is highly unlikely), one ECU can still detect the attack. Alternatively, if the IVN consists of a large number of ECUs, we can split $\mathbb{E}$ equally into two subsets $\mathbb{E}_1$ and $\mathbb{E}_2$ of size $\frac{|\mathbb{E}|}{2}$ each, with the former subset containing the lower half of CAN IDs and the latter the upper half. $\mathbb{E}_2$ will run the original procedure. In contrast, $\mathbb{E}_1$ will only detect spoofing attacks (on their own respective CAN IDs), significantly reducing the computational overhead on the respective ECU. However, the network will still be protected from DoS attacks since all ECUs in $\mathbb{E}_2$ will still run the DoS protection routine. We define this as *light scenario*, whereas every ECU that runs the original FSM is called *full scenario*.

To further save cost, we would like to add that not every ECU necessarily has to be equipped with `MichiCAN`. DoS detection will be provided by any `MichiCAN`-equipped ECU, while spoofing detection requires updating any ECU that wants to implement this feature. If the OEM decides to save cost and only equip ECUs with safety-critical functionality, this is possible and does not break any assumptions in this paper. However, this comes at the expense of the unpatched ECUs not being able to detect spoofing attacks any longer.

B. Pin Multiplexing

MCUs interface outside/peripheral components using their *peripheral I/O* (PIO) controller. Broadly speaking, there are two categories of PIO pins: *System I/O* (SIO) and *general-purpose I/O* (GPIO). For instance, an ECU features SIO pins to connect to the CAN transceiver (also called CAN PHY). By default, these pins are usually only read by the CAN controller (a system component of the MCU package) since the application software does not need access to this low-level bitstream. The application can interact with peripheral I/O using its GPIO pins. Nevertheless, the PIO controllers of modern MCUs have multiplexing capabilities which allow a GPIO pin to be multiplexed to a SIO pin. As a result, the PIO controller can be configured such that the ECU's application software has direct access to the CAN_{TX,RX} lines, which, in turn, allows the ECU to directly read and write every single bit on the CAN bus. Pin multiplexing is depicted in Fig. 4a.

Pin multiplexing can be configured dynamically in software, i.e., can be done once at boot time or anytime while the MCU is running. `MichiCAN` requires read access to the CAN_RX line once booted up, but write access to CAN_TX only when it starts a counterattack. After the counterattack has been completed, `MichiCAN` will deactivate the multiplexing. Pulling the bus low after the counterattack would destroy all traffic on the bus and pulling it high would cause issues with benign CAN messages, as each CAN controller has to acknowledge the receipt of a well-formed message by writing a dominant bit to the ACK bit in the CAN trailer.

C. Synchronization

To avoid errors, ECUs on the CAN bus must synchronize their clocks for reliable sampling, especially during arbitration. Since all ECUs operate on the same bus speed (e.g., 500 kBit/s), their *nominal bit time* is fixed (e.g., 2 μs). During that bit time, either a logical 0 or 1 will be observed by all ECUs on their CAN_RX pin. Due to bit transitions (e.g., from 1 to 0) and hardware imperfections, sampling the bit right at the beginning of the bit time might result in sampling a wrong logical value. To avoid this problem, CAN controllers usually sample the bit at 70% within the nominal bit time. CAN controllers continuously re-synchronize due to clock drifts. A hard synchronization is done at each SOF bit, i.e., when a transition from 1 to 0 occurs after at least 11 recessive bits during the idle bus.

`MichiCAN` has to replicate the synchronization process in software since we are circumventing the CAN controller. One simple way is to trigger timer interrupts every bit time (e.g., 2 μs) and then read in the value from CAN_RX. However, there are two issues with this straightforward approach: (i) we cannot guarantee *where* each bit is sampled and (ii) due to oscillator drift of the clock that the timer interrupts use, the interrupts will not be triggered at the same location within each bit time. To overcome this, we perform a hard synchronization by attaching an interrupt on the first falling edge on CAN_RX after at least 11 recessive bits (which is the SOF of a new CAN message). When this interrupt is triggered, `MichiCAN` will restart the main timer interrupt to trigger at 70% of the bit time (and thus reset accumulated jitter). For a 500 kBit/s CAN bus, the timer interrupt would first activate after 1.4 μs. Since we also reset the FSM and some other counter variables at the beginning of each CAN frame (which takes a constant number of clock cycles), we need to account for this when we restart the timer interrupts. As a result, we will first trigger the interrupt at a constant delta (called *fudge factor*) less than 1.4 μs. This can be determined empirically since the required clock cycles (and thus execution time) for the fudge factor will always be constant. Since we already know that the current bit is the SOF, we can just skip this bit and restart the timer

Algorithm 1 Main interrupt handler

```
1:  function INTERRUPT_HANDLER()
2:      $value \leftarrow$ Read CAN_RX register with PIO controller
3:      if $sof$ == True then
4:          $cnt \leftarrow cnt + 1$
5:          if $cnt$ < 25 then
6:              if $frame[cnt\text{-}2]$ != $value$ and $stuff$ == 5 then
7:                  $stuff \leftarrow 0$
8:                  $cnt \leftarrow cnt$ - 1
9:              if $stuff$ < 5 then
10:                 $frame[cnt\text{-}1] \leftarrow value$
11:                 if $!start\_counterattack$ then
12:                     $state\_machine\_run(value)$
13:                 if $frame[cnt\text{-}2]$ == $value$ then
14:                     $stuff \leftarrow stuff$ + 1
15:                 else $stuff \leftarrow 0$
16:         if $cnt$ == 20 then
17:             Disable CAN_TX Multiplexing
18:             $sof \leftarrow$ False
19:             $cnt \leftarrow 0$
20:         else if $cnt$ == 13 then
21:             $start\_counterattack \leftarrow$ False
22:             Enable CAN_TX Multiplexing
23:             Pull CAN_TX Low
24:     else
25:         if $value$ == 1 then $cnt\_sof \leftarrow cnt\_sof$ + 1
26:         else if $value$ == 0 and $cnt\_sof$ <11 then $cnt\_sof \leftarrow 0$
27:         if $value$ == 0 and $cnt\_sof \leq 11$ then
28:             $sof \leftarrow true$
29:             $cnt\_sof,\ frame[0] \leftarrow 0$
30:             $stuff,\ cnt \leftarrow 1$
31:             $reset\_state\_machine()$
```

interrupts for the first bit of the CAN ID. When we execute the main interrupt handler for the first time (during the first bit of the CAN ID), we disable the interrupt timer and restart it to trigger every 2 μs since there will be no additional operations.

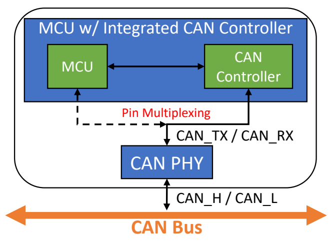
D. Detection

Since the CAN_RX can be read directly and we are properly synchronized to the CAN bus, `MichiCAN` can start with the detection routine. The latter is described in the first half of Algorithm 1. The main interrupt handler will trigger for the first time at the first bit of the CAN ID. At the very beginning of the interrupt handler, we read the bit from CAN_RX. Since we are using a PIO controller for pin multiplexing, we can directly read the value of CAN_RX from the MCU's registers (line 2). This avoids using an external read function from the MCU's libraries, which would add unnecessary computational overhead. Then, we increment a counter to track which bit position within a CAN frame `MichiCAN` is located. Since the interrupt is triggered every bit time, each execution of the interrupt handler will correspond to a new bit in the frame. As mentioned in Sec. II, CAN_RX will contain *stuff bits* which are automatically inserted by the CAN controller if there are more than 5 bits of the same polarity. As a result, we need to detect and identify these stuff bits (lines 6-15). While we are reading the 11-bit CAN ID, `MichiCAN` needs to remove those before appending them to a $frame$ array. For each bit (that is not a stuff bit), `MichiCAN` runs the FSM that is outlined in Sec. IV-A. Once the FSM determines that the CAN ID indicates a spoofing or DoS attack, the malicious flag $start\_counterattack$ will be set to true. To reduce computational overhead, `MichiCAN` will then stop running the FSM for the remaining bits of the CAN ID (line 11) and continue monitoring stuff bits.

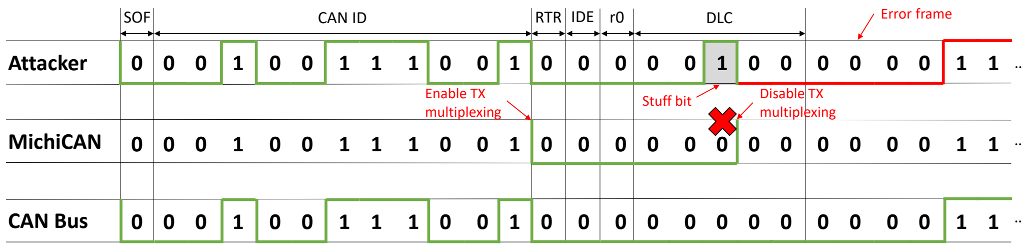
E. Prevention

Once `MichiCAN` sees that the $start\_counterattack$ flag has been raised, it starts its prevention routine. The goal of attack prevention is to bus off the attacker's ECU by triggering an error in its transmission, as per CAN error-handling rules (see Sec. II-B). The two error types we exploit are *bit* and *stuff* errors which can be achieved by transmitting a sequence of dominant bus levels. Note that we are not sending a complete CAN message from the legitimate node, but merely pulling the bus low for a period of time. Thus, the legitimate node's TEC remains unaffected by the counter-attack. Dominant bus levels override recessive ones, causing bit errors for the adversary. If the attacker sends 5 consecutive dominant bits, a recessive stuff bit follows. Since we are pulling the bus low by transmitting dominant bus levels, the stuff bit will be overwritten by another dominant bit which results in a stuff error. Fig. 4b depicts the prevention routine. Two major questions to address are (i) when to start injecting a dominant bit sequence, and (ii) how many dominant bits to inject.

`MichiCAN` cannot inject dominant bits during arbitration, as it would cause the attacker to lose it without generating an error frame. `MichiCAN` injects dominant bits right after the CAN ID field to bus off the attacker effectively, i.e., during the RTR bit (see Fig. 1a). Since this bit (and the following IDE and r0 bits) is already dominant, no bit error will be generated. The following DLC field is usually encoded as "1000" (since CAN messages mostly consist of 8 data bytes), so the earliest bit error can be caused at the fourth bit. Hence, the minimum duration that `MichiCAN` needs to pull the bus low to cause a bit error is four bits. However, if the least-significant bits (LSB) of the CAN ID consist of consecutive dominant levels, a stuff error can be caused as early as in the RTR bit. For this to happen, the five LSBs of the CAN ID need to be dominant. It will be sufficient for `MichiCAN` to just transmit one dominant bit during the RTR slot to raise an error frame. In the worst case, if the CAN data field consists of only one byte, causing a stuff error will require to inject 6 dominant bits if the LSB of the CAN ID is recessive. To sum up, an error frame can be caused by `MichiCAN` injecting 1–6 dominant bits. Since this will depend on several factors (such as the DLC length) that are unpredictable during sampling the CAN frame, `MichiCAN` needs to make sure to inject 6 dominant bits. The worst-case scenario is depicted in Fig. 4b. As described in Sec. IV-B, the CAN_TX pin is disabled by default. At frame position 13 (1 SOF + 11 CAN ID + 1 RTR), `MichiCAN` will enable CAN_TX multiplexing, pull the pin low and set the $start\_counterattack$ flag to false (lines 20-23 in Algorithm 1). At frame position 20, it will



(a) Pin multiplexing



(b) `MichiCAN` prevention routine

Fig. 4: System Design Components of `MichiCAN`

then disable CAN_TX multiplexing which will automatically stop pulling the bus low, set the start-of-frame flag to false and the frame counter to 0 since `MichiCAN` is done processing the frame (lines 16-19). The attacker (who has to comply with the CAN protocol) will immediately raise an *active* error frame consisting of six dominant bits followed by eight recessive bits. Even if `MichiCAN` would have succeeded in only transmitting one dominant bit (in the best-case scenario), five additional dominant bits will not do any harm due to the six-bit dominant error flag. The transmission error counter (TEC) of the attacking ECU will be increased by 8 and it will attempt a retransmission after a total of 11 recessive bits (lines 24-31). At the SOF bit, several variables, as well as the FSM will be reset. `MichiCAN` will repeat the detection procedure and start another counterattack. After a total of 15 retranmissions, the attacking ECU will transition into its error-passive region and start transmitting *passive* error frames. After another 16 retransmissions (summing up to a total of 32 attempts), the attacking ECU will be confined into bus-off state. Additionally, although `MichiCAN` could potentially flag a legitimate node as an attacker due to a bit flip, a node needs to encounter 32 consecutive errors for the TEC to reach a level that would trigger a bus-off condition. In case of sporadic errors, the likelihood of hitting this threshold is near zero, effectively eliminating the chance of false positives.

We have assumed that retransmissions will not be interrupted by other CAN messages. However, a message with a lower CAN ID could win arbitration during an 11-bit idle period and disrupt the sequence. This comes at the expense of increasing *bus-off time*, an important metric evaluated in Sec. V-C. However, since `MichiCAN` compares the CAN ID of a frame even for a retransmission, it still works as expected, even in the presence of multiple concurrent attackers.

V. Evaluation

A. Experimental Setup

To evaluate `MichiCAN`, we can choose from different evaluation boards that come with integrated CAN controllers and either resemble the specifications of automotive ECUs or are specifically built for it. One prominent and affordable platform is the Arduino Due which features an Atmel SAM3X8E Cortex-M3 CPU [52] clocked at 84 MHz. Since the Arduino only provides CAN_TX and CAN_RX lines and comes without a CAN transceiver, we can use separate CAN PHY breakout boards [53] in conjunction to build a CAN bus.

We evaluate `MichiCAN` under six experiments featuring one or multiple attackers and real-world vehicle CAN traffic (called restbus simulation) to assess key metrics. The experiments will be introduced in Sec. V-C. Fig. 5 shows our testbed consisting of a CAN bus with the `MichiCAN` ECU and four attacker ECUs, as well as benign CAN traffic replayed from a production vehicle using a USB-CAN interface (PCAN) [54]. For certain metrics we need to measure the execution time of `MichiCAN`. To minimize the overhead on the Arduino Due and report accurate numbers, we use ESP8266 as external timer [55]. We have also connected a logic analyzer to the breadboard so as to monitor the CAN traffic and obtain other time measurements for other evaluation metrics.

We evaluate `MichiCAN` using CAN messages collected from real production vehicles of the same OEM manufactured between 2016 and 2019: Veh. A is a luxury mid-size sedan, Veh. B a compact crossover SUV, Veh. C a full-size crossover SUV, and Veh. D a full-size pickup truck. All of these vehicles have two CAN buses each. For the restbus simulation, we randomly selected Veh. D to inject benign CAN traffic from.

In our setup, each CAN controller can be configured to transmit messages at up to 1 Mbit/s, although all ECUs on a CAN bus need to share the same *bus speed*. This is fixed by the OEM at production time and cannot be altered afterwards. Although `MichiCAN` can run at 250 kbit/s and 500 kbit/s, our online evaluation will be based on 50 and 125 kbit/s bus speeds due to the processing power limitations of the used Arduino Dues. However, we implemented `MichiCAN` on different, more powerful hardware from NXP (see Sec. VI-B) to demonstrate that it can indeed run at higher bus speeds. Furthermore, we make direct comparisons of two crucial metrics — bus-off time and bus load — of `MichiCAN` with the closest related prior work, Parrot [18]. Finally, we install `MichiCAN` on a real production vehicle, the 2017 Chrysler Pacifica Hybrid, where we succeed at preventing a DoS attack on the park assist system of the test vehicle.

B. Detection Latency

Detection latency is calculated by multiplying the detection bit position with the nominal bit time. That position is where MichiCAN stops its FSM within the CAN ID and sets the counterattack flag. Although the earliest the counterattack

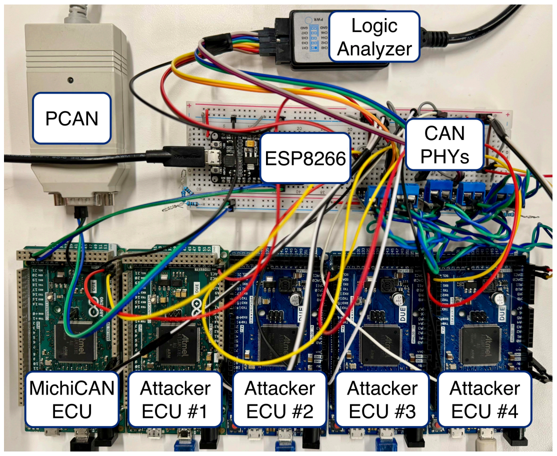


Fig. 5: Experimental CAN bus prototype

can start is *after* finishing the arbitration (as discussed in Sec. IV-E), stopping the FSM early can be beneficial to avoid using additional CPU cycles which will be evaluated in Sec. V-D. As the size of IVN $\mathbb{E}$ grows, the detection bit position rises. Our evaluation with 160,000 random FSMs yielded a mean detection bit position of 9 bits. Furthermore, the evaluation confirmed a 100% detection rate, verifying the correctness of our FSM generation algorithm.

C. Bus-off Time

MichiCAN begins bussing off the attacker's ECU by generating error frames after the CAN ID field, requiring 31 retransmissions after the initial malicious ID transmission. Note that no complete CAN frames are sent since the attacker will retransmit its CAN message after the 14-bit error frame and 3-bit inter-frame space (IFS) in its error-active region and an additional 8-bit suspend period in its error-passive region. The total time from the first bit of a malicious CAN message to the last bit of the passive error frame in the 31st retransmission is called the *bus-off time*. Depending on the attacker's CAN ID, MichiCAN has to inject one dominant bit in the best case, whereas in the worst case, it has to inject 6 dominant bits to trigger an error frame (see Sec. IV-E). In what follows, the best-case and worst-case bus-off time calculations are presented if there is only one attacker ECU transmitting a single CAN ID and one MichiCAN-equipped ECU.

Best-Case Scenario. MichiCAN injects the dominant bit during the RTR bit. As a result, the error frame starts at the 14th bit position within the CAN frame (1 SOF + 11 CAN ID + 1 RTR). The error flag itself consists of 14 bits, in addition to the 3 bit IFS, so the (re-)transmission of an error-active attacker takes 30 bits. Note that this calculation excludes stuff bits which depends on the most-significant bits of the CAN ID. For the error-passive attacker, this number stands at 38 bits including the additional suspend period.

Worst-Case Scenario. MichiCAN injects 6 dominant bits (see Fig. 4b). The error frame starts at the 19th bit within the CAN frame. The bus-off time stands at 35 bits and 43 bits for the error-active and error-passive attacker, respectively.

By busing-off attacking ECUs, MichiCAN intrinsically increases the traffic overhead on the bus. This overhead can be captured by measuring the bus-off time. From a safety standpoint, MichiCAN must not disrupt regular CAN bus communication, especially time-sensitive messages. Communication analysis of production vehicles (see Sec. V-A) shows that the minimum deadline for periodic messages on a 500 kBit/s bus is 10 ms. This translates to a maximum of 5000 bits that the bus-off time might take. In the following, we focus on bit counts rather than time, as bus-off time equals the number of bits multiplied by the nominal bit time.

So far, we only focused on the presence of one attacker and one defender (i.e., MichiCAN-equipped ECU). To generalize the deployment of MichiCAN in the presence of (i) benign vehicular traffic and (ii) multiple attackers in our threat model, we define six distinct experiments that cover all scenarios on the CAN bus. We evaluated the prevention of these six attack scenarios on a MichiCAN-equipped ECU that is configured to send CAN ID 0x173. Table II displays the measurements of the bus-off time for each experiment. During each experiment, we recorded the CAN bus for 2s which includes multiple bus-off attempts, and report the mean μ, standard deviation σ and maximum bus-off time. All experiments were conducted on a 50 kBit/s CAN bus. The maximum bus-off time reported in Table II indicates 2929 bits of added overhead. As a result, MichiCAN is feasible under all experiments.

Table III shows theoretical bus-off times for all experiments, confirming empirical data in Table II. t_a defines the error-active time (in bits), and t_p defines the error-passive time in each row of experiments. $c_{h,a}$ represents the count of high-priority benign messages that win arbitration when the attacker is in an error-active state. In contrast, $c_{h,p}$ denotes high-priority benign messages that win arbitration during the error-passive period. In the error-passive region, low-priority messages may also intervene in the attacker's retransmissions in certain experiments. In this context, $c_{l,p}$ represents low-priority benign messages during this error-passive phase. On the other hand, $z_{h,a}$ defines the count of adversarial high-priority messages that intervenes when the node is error active. When the node is in error-passive mode, $z_{l,p}$ represents the count of low-priority adversarial messages that intervene and $z_{h,p}$ defines the count of high-priority messages that intervene with the current node, respectively.

Experiments 1 & 2. These experiments assume a single attacker that sends a CAN message with CAN ID 0x173. MichiCAN on the legitimate ECU that is transmitting 0x173 will detect a spoofing attack and raise a flag for the prevention routine to kick in. Experiment 1 considers restbus simulation, i.e., benign CAN traffic from other ECUs in the IVN $\mathbb{E}$. This is a more realistic and generalized version of Experiment 2 which only analyzes the presence of a single attacker, but no other ECU on the bus. We injected real CAN traffic from Veh. D (see Sec. V-A) using a PCAN-USB interface [54] with SocketCAN [56]. Assuming the worst-case scenario from above and lack of any benign traffic, each error-active (re)transmission will consist of 35 bits and each error-passive

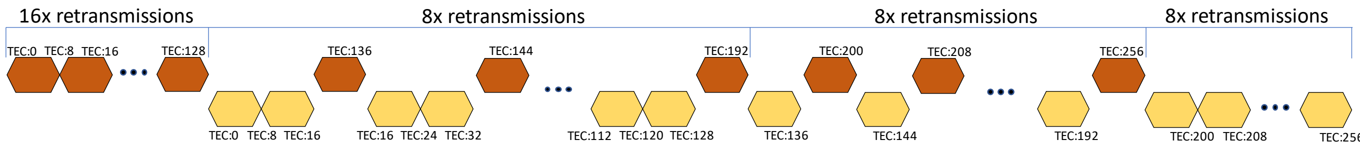


Fig. 6: Pattern on CAN network in Experiment 5 (brown: 0x066, yellow: 0x067)

TABLE II: Empirical Bus-off Time for All 6 Experiments

Exp. Number	**Attacker CAN ID**	**Restbus Simulation**	**μ of Bus-off Time (in ms)**	**σ of Bus-off Time (in ms)**	**Max. Bus-off Time (in ms)**
1	0x173	✓	24.6	2.64	58.6
2	0x173	×	24.2	0.27	25.2
3	0x064	✓	25.1	1.39	38.3
4	0x064	×	24.9	0.45	25.2
5	0x066	×	39.0	0.79	48.6
	0x067	×	35.4	0.60	44.0
6	0x050 0x051	×	24.9	0.01	25.4

TABLE III: Theoretical Bus-off Time Calculation

Exp. Number	**Scen.**	**Error-Active Time (in bits t_a)**	**Error-Passive Time (in bits t_p)**	**Total Bus-off Time (in bits)**
1, 3	All	35+$s_f \cdot c_{h,a}$	43+$s_f \cdot (c_{h,p} + c_{l,p})$	$\sum_{i=1}^{16}(t_{a,i} + t_{p,i})$
2, 4, 6	All	35	43	1248
5	H.P.	35	43+$s_{f,a} \cdot z_{l,p}$	560+$\sum_{i=1}^{16} t_{p,i}$
	L.P.	35+$s_{f,a} \cdot z_{h,a}$	43+$s_{f,a} \cdot z_{h,p}$	$\sum_{i=1}^{16}(t_{a,i} + t_{p,i})$

retransmission of 43 bits. There are 16 retransmissions for each error mode. The total bus-off time can be calculated as $16 \cdot (t_a + t_p) = 1248$ bits using the aforementioned nomenclature. Considering restbus simulation, benign CAN messages can interrupt retransmissions and thus extend the total bus-off time by the length of a CAN frame. Each CAN frame consists of 125 bits on average (including stuff bits), i.e., $s_f = 125$. In the error-active region, only higher-priority CAN messages can interrupt a retransmission attempt since they would win arbitration. On the other hand, in the error-passive region, any CAN message can interrupt a retransmission due to the additional suspend period and passive error flag totaling 25 bits. Recall that a CAN message can be retransmitted on a CAN bus after seeing 11 recessive bits. As a result, the error-active time of a bus-off attempt will increase to $35{+}s_f{\cdot}c_{h,a}$ and the error-passive time to $43{+}s_f{\cdot}(c_{h,p}{+}c_{l,p})$. Theoretically, the total bus-off time can go to infinity in this case if every retransmission attempt gets interrupted by a benign message. However, our actual measurements in Table II confirm that only few benign messages from the restbus simulation interrupt the bus-off attempt. As a result, the mean bus-off time for Experiments 1 and 2 are comparable, with a slightly higher variance in Experiment 1 as expected.

Experiments 3 & 4. These experiments are very similar in nature to the previous experiments. A single attacker transmits 0x064 which constitutes a DoS attack. Only the detection routine is different and the prevention is identical to a spoofing attack. As a result, the theoretical bus-off time calculations in Table III are identical to the previous experiments. The actual measurement in Table II confirms that the mean and standard deviation values are comparable to Experiments 1 & 2 and within the worst-case bound of 1248 bits.

Experiment 5. This experiment features 2 attacking ECUs that transmit 2 distinct CAN IDs 0x066 and 0x067, respectively. As a result, this experiment constitutes 2 DoS attacks on the CAN bus. We selected 0x066 and 0x067 randomly and confirmed that choosing other CAN IDs did not make a difference to our evaluation since the prevention routine only depends on the malicious flag set by the detection part of `MichiCAN`. There are 2 distinct patterns that can emerge in this experiment: (i) CAN ID 0x067 does not interrupt the bus-off attempt of 0x066 (or vice versa) because their periods are large. The same behavior as in Experiment 4 can be observed in this case. (ii) However, if 0x067 is scheduled to transmit shortly after the start of 0x066 (or vice versa) which is getting bused-off, both bus-off attempts will get intertwined.

The behavior observed in this experiment is shown in Fig. 6. The brown signal denotes the transmission of attack message with CAN ID 0x066 while yellow signals are representing attack messages with CAN ID 0x067. 0x066 goes error-passive after its active error flag is sent for the 16th time, and because it was transmitting the current frame, it is suspended for an additional 8 bits after IFS. Since it cannot assert a SOF bit after IFS due to the suspend period, this lets 0x067 win arbitration. 0x067 then gets attacked by `MichiCAN` for the first time and raises an active error flag. 0x066 will still have its TEC at 128. Since 0x066 was the transmitter of the previous frame, it still has a suspend transmission active according to the ISO 11898 standard [45]: *"An error-passive node, which is the transmitter of the current frame or has been transmitter of the previous frame, may access the bus as soon as its suspend transmission time is finished, provided that no other node has started transmission meanwhile"*. The next time, the suspend transmission time is lifted, and it can assert SOF after IFS, the same as 0x067, and it then wins arbitration. This frame is destroyed, and it then has suspend transmission activated again. As a result, there will be only 8 error-passive retransmissions for 0x066 during the 16 error-active retransmissions of 0x067. After this, both 0x066 and 0x067 are error-passive, with the TEC of 0x066 at 192 and TEC of 0x067 at 128, respectively. Since both have an additional suspend period, both retransmissions will toggle from now on. After another 8 retransmissions, 0x066 will go bus-off and 0x067 will finish its remaining 8 retransmissions. Looking at the results from Table II, we observe that the mean bus-off time grows by around 50% due to the retransmissions getting intertwined. Since 0x067 can transmit during the error-passive region of 0x066, the bus-off time does not double. The bus-off time of 0x067 is slightly smaller since its bus-off finishes 8 retransmission times less than that of 0x066 (see Fig. 6). Table III summarizes our findings and distinguishes an additional case to cover all possibilities. In our example, the higher-priority (HP) CAN message starts its transmission first. In this case, it cannot lose arbitration to the lower-priority (LP) message in the error-active region. However, this changes in the error-passive region as explained above. We named this scenario *HP* in Table III. If the LP message starts its transmission before the HP message, the LP message can get interrupted since it will lose arbitration to the HP message. This scenario is summarized as *LP* in Table III and has a higher total bus-off time compared to the *HP* scenario.

Experiment 6. This experiment features one attacker again. However, the attacker node is sending two different CAN IDs consecutively, e.g., toggling between 0x050 and 0x051. An ECU adds each message that it schedules for transmission in a buffer until it is successfully transmitted. After 32 (re)transmissions of either 0x050 or 0x051, the attacking ECU will go into bus-off. Bused-off ECUs will recover after observing 128 instances of 11 recessive bits. After its recovery, the other CAN message will be transmitted (and the ECU will be bussed-off again). Since both CAN messages are bused-off separately, the bus-off time is identical to Experiment 4.

Experiments with more than two attackers: Since previous experiments only analyzed a maximum of $A = 2$ attackers (behavior with more attackers is analog to Experiment 5), we wanted to determine the maximum number of attacking ECUs before the CAN bus becomes fully inoperable due to deadline misses. We already know that the bus-off time does not double with the number of attackers. We repeated Experiment 5 with $A = 3$ and $A = 4$ (see Fig. 5) attacking ECUs. The total bus-off time consists of 3515 and 4660 bits, respectively. `MichiCAN` is effective against up to four attackers, as $A \geq 5$ would render the CAN bus inoperable. This is a realistic limit given the difficulty of compromising even one ECU.

D. CPU Utilization

We envision `MichiCAN` as a software patch to existing application software running on MCUs; hence, a minimal overhead is desired. We measure the overhead by measuring the execution time of the interrupt handler and dividing it by the nominal bit-time. Although there is an external interrupt for re-synchronization at the SOF, the CPU cycles consumed by this step are negligible compared to the main interrupt. With a 160 MHz clock, the ESP8266 captures the start and end of `MichiCAN`'s interrupt handler by monitoring toggled pins. It counts clock cycles between these events, multiplied by the 6.25 ns resolution gives us the handler's execution time.

CPU utilization for *full* and *light* scenarios varies in two key periods during CAN_RX reading. In the bus-idle state, only line 23 of Alg. 1 executes, resulting in low and constant execution time. CPU utilization consists of *idle load* when no CAN frame is processed (lines 3-21) and *active load* during frame processing. The *combined load* is the average CPU overhead on the Arduino, varying between these two states.

The evaluation was conducted using the eight CAN buses $\mathbb{E}$ of the four production vehicles from Sec. V-A. For each $\mathbb{E}$, we deployed the FSM for ECU_N on the Arduino for maximum testing coverage and then calculated `MichiCAN`'s CPU utilization overhead by measuring the execution time of the interrupt handler. We make the following observations:

CPU load depends on bus speed. Higher bus speeds elevate CPU usage; a 125kbit/s bus averages 40% CPU load, implying an 80% load for a 250kbit/s bus, not accounting for jitter. This explains why `MichiCAN` does not always reliably work on higher bus speeds than 125kbit/s on Arduino Dues.

CPU load depends on MCU. On repeated testing with more powerful hardware, namely the NXP S32K144, the bus load was much lower for a 500kbit/s bus, standing at 44%. Most production ECUs in real vehicles use MCU like the aforementioned platform. In fact, NXP is currently the second largest Tier-2 supplier to the automotive industry [57]. These results underline the practical deployability of `MichiCAN`.

CPU load depends on FSM complexity. A larger FSM increases clock cycle usage; 125kbit/s full scenario uses about 40% CPU, while the light version uses 30%.

E. Bus Load

The bus load b is calculated as $b = \frac{s_f}{f_{baud}} \sum_{m \in M} p_m^{-1}$ [58], where s_f is the frame length, f_{baud} the bus speed, and p_m the period of a CAN message m. The bus-off time introduced by `MichiCAN`'s prevention routine affects the overall bus load. One CAN message at 50 kBit/s is transmitted within 2.5 ms. Table II shows that if this message is counterattacked by `MichiCAN`, it will be on the bus for 25 ms in the worst case including retransmissions and in the absence of higher-priority CAN messages. In theory, we increase the bus load by 10x. Since the attacking ECU can recover from bus-off, the bus-off attempt will repeat. Using a persistent bus-off attack [38], the attacker will be bused off once and the remaining CAN communications will continue normally. As a result, there will only be a short spike in the bus load during the counterattack for around 25 ms. As discussed in Sec. V-C, the bus-off time during which the bus load will peak is smaller than typical message deadlines. Low-priority messages have deadlines standing at 500 or 1000 ms (at 50 kBit/s). As a result, the bus-off attempt will incur a bus load overhead of 2.5–5%. For high-priority messages with deadlines around 100 ms (at 50 kBit/s), the bus load overhead stands at 25%. Given that the bus load will never exceed 80% [24] (real observed bus load 40% in real vehicles [18]), the overhead is considered negligible. Parrot [18] transmits frequent defensive messages to collide with an attacker, requiring a 3-bit gap equal to the IFS, thus incurring a bus load overhead of $\frac{125}{128} \approx 97.7\%$. In contrast, `MichiCAN` has no such overhead, and its bus load is at least 2x lower than Parrot's during bus-off attempts.

F. On-Vehicle Testing

To both stress-test `MichiCAN` with real CAN traffic and demonstrate how a DoS attack can be prevented in practice,

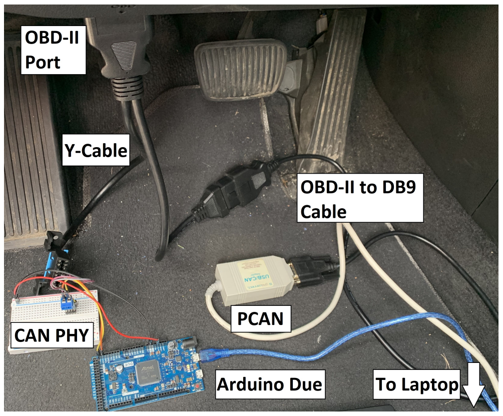


Fig. 7: Attack device and MichiCAN connected to OBD-II

we deployed MichiCAN in a 2017 Chrysler Pacifica Hybrid. We launched a targeted DoS attack to disable the park assist system called *ParkSense* [59]. Our DoS attack is similar to the one of Palanca *et al.* [27] who also targeted the park assist system of a 2012 Alfa Romeo Giulietta. To launch the DoS attack on the test vehicle, we extracted all relevant CAN IDs for this feature from a publicly available CAN communication matrix [48]. The lowest CAN ID that was relevant for ParkSense was 0x260. As a result, we injected CAN messages periodically with a CAN ID of 0x25F using a USB-CAN interface (PCAN [54]) from the OBD-II port. Without the Arduino Due running MichiCAN attached to the OBD-II port, the DoS attack will disable ParkSense in accordance with the observed results from the related Alfa Romeo Giulietta [27]. The dashboard of the vehicle was stating "PARKSENSE UNAVAILABLE SERVICE REQUIRED" after we started injecting CAN messages, indicating the success of the DoS attack. This attack might also have safety-critical implications since "automatic brakes will not be available if there is a faulty condition detected with the ParkSense Park Assist system" according to the vehicle manual [59]. After connecting an Arduino Due with MichiCAN using an OBD-II splitter cable (also called *Y-cable*), as depicted in Fig. 7, the DoS attack was eradicated within 32 transmission attempts, restoring the park assist system. A DoS attack never disables the park assist if the Arduino Due with MichiCAN is connected to the car at the same time as PCAN.

VI. Discussion

A. Prevalence of integrated CAN controllers

MCUs with on-chip CAN controllers have been in production vehicles for years, but MichiCAN is the first to use them for attack detection and prevention. While researchers have explored bypassing CAN controllers for bus-off attacks since 2017, these approaches lacked integrated controllers. Palanca *et al.* [27] used an Arduino Uno with an MCP2551 PHY, re-implementing the CAN protocol due to the absence of a built-in controller. Murvay *et al.* [26] advanced this with a more sophisticated attacker model using an NXP S12XD512 MCU but still without an integrated CAN controller.

An ECU enters a bus-off state when its TEC or REC reaches 256. Although meant for fault confinement, this can be exploited by attackers to disable benign ECUs [25], [60]. Defenses are limited, often relying on detecting the attack and busing off the attacker first [61], [62]. Since these attacks generate error frames, tracking their frequency and associated CAN IDs may help identify malicious patterns [63].

The most sophisticated work that actually mentions integrated CAN controllers and uses them to launch a stealthy bus-off attack was proposed by Kulandaivel *et al.* [64]. Their CANnon attack can inject single bits and force the victim to generate error frames until it is bused off. Instead of pin-multiplexing, they deploy a *peripheral clock gating* technique to arbitrarily pause and resume the clock of the CAN controller. While their method manipulates the CAN controller and is hard to detect, MichiCAN adds software redundancy for CAN features, ensuring full backward compatibility.

Finally, CANflict [28] utilizes pin conflicts on an MCU to inject arbitrary bits to the CAN network using other existing peripherals such as SPI, UART and I2C. In contrast, MichiCAN is a defense tool that is using regular GPIO pins without underlying protocol assumptions.

B. Replicability on other MCUs

In this paper, we used an Arduino Due based on the AT91SAM3X8EA MCU. Kulandaivel *et al.* [64] present an overview of automotive MCUs with integrated CAN controllers, including the Microchip SAM V71 Xplained Ultra (150 MHz) and the STMicro SPC58EC Discovery (180 MHz). MichiCAN was implemented on a comparable NXP S32K144 [65] (112 MHz) with integrated CAN controllers. We could confirm that MichiCAN works as intended on this MCU as well, and even exceeds the Arduino by fully working on a 500 kbit/s CAN. Due to high overhead (i.e., additional CPU cycles) to enter and exit the interrupt handler on the Arduino Due compared to other comparable MCUs [66], a more optimized code together with a more powerful MCU will run MichiCAN on bus speeds up to 1 Mbit/s.

VII. Conclusion

In this paper, we have developed MichiCAN which is a practical and backward-compatible spoofing and DoS detection and prevention framework for CAN. By using integrated CAN controllers deployed in many modern ECUs, we can solve various issues of prior work, such as lack of real-time detection and prevention, as well as reduce network overhead. Our evaluation demonstrated the efficacy of MichiCAN using multiple metrics. Since DoS attacks are an overlooked but important threat vector in automotive security standards [14], we envision MichiCAN to be a compelling and practical security enhancement for OEMs.

Acknowledgments

The work in this paper was supported in part by the US NSF under Grant CNS-2245223 and the Office of Naval Research under Grant N00014-22-1-2622. We thank Bryan Brauchler and Junru Du for their contributions to the implementation.

References

[1] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, *et al.*, "Experimental security analysis of a modern automobile," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 447–462, IEEE, 2010.

[2] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, *et al.*, "Comprehensive experimental analyses of automotive attack surfaces," in *Proceedings of the 20th USENIX Security Symposium (USENIX Security '11)*, pp. 77–92, USENIX, August 2011.

[3] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, vol. 2015, no. S 91, pp. 1–91, 2015.

[4] W. A. Farag, "Cantrack: Enhancing automotive can bus security using intuitive encryption algorithms," in *2017 7th International Conference on Modeling, Simulation, and Applied Optimization (ICMSAO)*, pp. 1–5, IEEE, 2017.

[5] H. Olufowobi, C. Young, J. Zambreno, and G. Bloom, "Saiducant: Specification-based automotive intrusion detection using controller area network (can) timing," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 2, pp. 1484–1494, 2019.

[6] O. Ikumapayi, H. Olufowobi, J. Daily, T. Hu, I. C. Bertolotti, and G. Bloom, "Canasta: Controller area network authentication schedulability timing analysis," *IEEE Transactions on Vehicular Technology*, vol. 72, no. 8, pp. 10024–10036, 2023.

[7] O. Ikumapayi, H. Olufowobi, J. Daily, T. Hu, I. C. Bertolotti, and G. Bloom, "Work in progress: Schedulability analysis of can and can fd authentication," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 349–352, IEEE, 2023.

[8] C. Young, J. Zambreno, H. Olufowobi, and G. Bloom, "Survey of automotive controller area network intrusion detection systems," *IEEE Design & Test*, vol. 36, no. 6, pp. 48–55, 2019.

[9] R. Kurachi, Y. Matsubara, H. Takada, N. Adachi, Y. Miyashita, and S. Horihata, "Cacan-centralized authentication system in can (controller area network)," in *14th Int. Conf. on Embedded Security in Cars (ESCAR 2014)*, 2014.

[10] K. S. Han, S. D. Potluri, and K. G. Shin, "Real-time frame authentication using id anonymization in automotive networks," Mar. 15 2016. US Patent 9,288,048.

[11] S. Nürnberger and C. Rossow, "–vatican–vetted, authenticated can bus," in *International Conference on Cryptographic Hardware and Embedded Systems*, pp. 106–124, Springer, 2016.

[12] A. Van Herrewege, D. Singelee, and I. Verbauwhede, "Canauth – a simple, back-ward compatible broadcast authentication protocol for can bus," *ECRYPTWorkshop on Lightweight Cryptography*, 2011.

[13] M. Bozdal, M. Samie, S. Aslam, and I. Jennions, "Evaluation of can bus security challenges," *Sensors*, vol. 20, no. 8, p. 2364, 2020.

[14] AUTOSAR, "AUTOSAR specification of secure onboard communication protocol." https://www.autosar.org/fileadmin/standards/R21-11/FO/AUTOSAR_PRS_SecOcProtocol.pdf.

[15] S. Ohira, A. K. Desta, I. Arai, H. Inoue, and K. Fujikawa, "Normal and malicious sliding windows similarity analysis method for fast and accurate ids against dos attacks on in-vehicle networks," *IEEE Access*, vol. 8, pp. 42422–42435, 2020.

[16] M. D. Hossain, H. Inoue, H. Ochiai, D. Fall, and Y. Kadobayashi, "Lstm-based intrusion detection system for in-vehicle can bus communications," *IEEE Access*, vol. 8, pp. 185489–185502, 2020.

[17] P. Kalyanasundaram, V. Kareti, M. Sambranikar, N. K. SS, and P. Ranadive, "Practical approaches for detecting dos attacks on can network," tech. rep., SAE Technical Paper, 2018.

[18] T. Dagan and A. Wool, "Parrot, a software-only anti-spoofing defense system for the can bus," *ESCAR EUROPE*, p. 34, 2016.

[19] A. Humayed, F. Li, J. Lin, and B. Luo, "Cansentry: Securing can-based cyber-physical systems against denial and spoofing attacks," in *European Symposium on Research in Computer Security*, pp. 153–173, Springer, 2020.

[20] K. Cheng, Y. Bai, Y. Zhou, Y. Tang, D. Sanan, and Y. Liu, "Caneleon: Protecting can bus with frame id chameleon," *IEEE Transactions on Vehicular technology*, vol. 69, no. 7, pp. 7116–7130, 2020.

[21] B. Groza, L. Popa, P.-S. Murvay, Y. Elovici, and A. Shabtai, "Canary-a reactive defense mechanism for controller area networks based on active relays.," in *USENIX Security Symposium*, pp. 4259–4276, 2021.

[22] K. Serag, R. Bhatia, A. Faqih, M. O. Ozmen, V. Kumar, Z. B. Celik, and D. Xu, "{ZBCAN}: A {Zero-Byte}{CAN} defense system," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 6893–6910, 2023.

[23] T. Dagan, M. Yuval, M. Marchetti, D. Stabili, M. Colajanni, and W. Avishai, "Vehicle safe-mode, concept to practice limp-mode in the service of cybersecurity," *SAE International Journal of Transportation Cybersecurity and Privacy*, 2020.

[24] C. Academy, "Can bus load calculator." http://www.canbusacademy.com/resources/can-bus-load-calculator/.

[25] K.-T. Cho and K. G. Shin, "Error handling of in-vehicle networks makes them vulnerable," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1044–1055, 2016.

[26] P.-S. Murvay and B. 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*, pp. 1–10, 2017.

[27] A. Palanca, E. Evenchick, F. Maggi, and S. 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*, pp. 185–206, Springer, 2017.

[28] A. de Faveri Tron, S. Longari, M. Carminati, M. Polino, and S. Zanero, "Canflict: Exploiting peripheral conflicts for data-link layer attacks on automotive networks," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pp. 711–723, 2022.

[29] Microchip, "Mcp2515," 2018. https://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf.

[30] Microchip, "Mcp2551," 2006. https://ww1.microchip.com/downloads/en/DeviceDoc/21667E.pdf.

[31] Microchip, "Mcp25625," 2018.

[32] NXP, "Nxp automotive processors — product map." https://www.nxp.com/docs/en/product-selector-guide/BRAUTOPRDCTMAP.pdf, 2020.

[33] STMicroelectronics, "Spc58 b/c/g-lines product selector guide." https://www.st.com/content/ccc/resource/sales_and_marketing/promotional_material/selection_guide/group0/34/0e/4f/b2/1a/49/4f/b6/SPC58%20selection%20guide/files/SGSPC58C.pdf/jcr:content/translations/en.SGSPC58C.pdf, 2020.

[34] Renesas, "V850es/fx3." https://www.renesas.com/us/en/products/microcontrollers-microprocessors/other-mcus-mpus/v850-family-mcus/v850esfx3-32-bit-microcontrollers-non-promotion.

[35] I. Foster, A. Prudhomme, K. Koscher, and S. Savage, "Fast and vulnerable: A story of telematic failures," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[36] 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*, vol. 91, pp. 1–19, 2018.

[37] M. D. Pesé, T. Stacer, C. A. Campos, E. Newberry, D. Chen, and K. G. Shin, "Librecan: Automated can message translator," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2283–2300, 2019.

[38] K. Serag, R. Bhatia, V. Kumar, Z. B. Celik, and D. Xu, "Exposing new vulnerabilities of error handling mechanism in {CAN}," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 4241–4258, 2021.

[39] H. Wen, Q. A. Chen, and Z. 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 (USENIX Security 20)*, pp. 949–965, 2020.

[40] M. D. Pesé, J. W. Schauer, J. Li, and K. G. Shin, "S2-can: Sufficiently secure controller area network," *Annual Computer Security Applications Conference (ACSAC'21)*, 2021.

[41] H. Dutta, "Understanding the growing trend of ecu virtualization." https://blog.sasken.com/understanding-the-growing-trend-of-ecu-virtualization.

[42] Android, "Android open source project." https://source.android.com/docs/automotive/virtualization/architecture.

[43] OpenSynergy, Jul 2023. https://www.opensynergy.com/why-hypervisor-technology/.

[44] ProvenRun, "Trusted execution environment in automotive critical ecus." https://provenrun.com/wp-content/uploads/2021/10/Leaflet_CTI_TEE_Automotive_final.pdf, March 2021. Version 1.0.

[45] ISO, “Iso 11898-1:2015,” Jun 2021. https://www.iso.org/standard/63648.html.
[46] T. HIL. https://www.typhoon-hil.com/documentation/typhoon-hil-software-manual/References/can_bus_protocol.html.
[47] I. Technology, “Can bus: Basics,” Jun 2021. https://www.influxtechnology.com/post/can-bus-basics.
[48] Commaai, “Commaai/opendbc: Democratize access to car decoder rings.” https://github.com/commaai/opendbc.
[49] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, “Controller area network (can) schedulability analysis: Refuted, revisited and revised,” *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.
[50] T. Karthik, A. Brown, S. Awwad, D. McCoy, R. Bielawski, C. Mott, S. Lauzon, A. Weimerskirch, and J. Cappos, “Uptane: Securing software updates for automobiles,” in *International conference on embedded security in car*, vol. 11, 2016.
[51] S. Halder, A. Ghosal, and M. Conti, “Secure over-the-air software updates in connected vehicles: A survey,” *Computer Networks*, vol. 178, p. 107343, 2020.
[52] A. Corporation, “32-bit cortex-m3 microcontroller sam3x/sam3a datasheet.”
[53] Waveshare, “Sn65hvd230 can board.” https://www.waveshare.com/sn65hvd230-can-board.htm.
[54] P.-S. T. GmbH, “Pcan-usb.” https://www.peak-system.com/PCAN-USB.199.0.html.
[55] E. Systems, “Esp8266.” https://www.espressif.com/en/products/socs/esp8266.
[56] T. L. Kernel, “Socketcan - controller area network.” https://docs.kernel.org/networking/can.html.
[57] C. Systems, “Top 10 automotive semiconductor companies,” 2024. Accessed: 2024-09-20.
[58] vector, “Can bus load calculation.” https://kb.vector.com/entry/1519/.
[59] FCA, “2018 pacifica,” 2018. https://pictures.dealer.com/tomahlchryslerdodgecllc/429af8240a0e0ca21b13dc012566e7b3.pdf.
[60] G. Bloom, “Weepingcan: A stealthy can bus-off attack,” in *Workshop on Automotive and Autonomous Vehicle Security (AutoSec)*, vol. 2021, p. 25, 2021.
[61] D. Souma, A. Mori, H. Yamamoto, and Y. Hata, “Counter attacks for bus-off attacks,” in *International Conference on Computer Safety, Reliability, and Security*, pp. 319–330, Springer, 2018.
[62] M. Takada, Y. Osada, and M. Morii, “Counter attack against the bus-off attack on can,” in *2019 14th Asia Joint Conference on Information Security (AsiaJCIS)*, pp. 96–102, IEEE, 2019.
[63] S. Longari, M. Penco, M. Carminati, and S. Zanero, “Copycan: An error-handling protocol based intrusion detection system for controller area network,” in *Proceedings of the ACM Workshop on Cyber-Physical Systems Security & Privacy*, pp. 39–50, 2019.
[64] S. Kulandaivel, S. Jain, J. Guajardo, and V. Sekar, “Cannon: Reliable and stealthy remote shutdown attacks via unaltered automotive microcontrollers,” in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 195–210, IEEE, 2021.
[65] N. Semiconductors, “S32k144-q100 general purpose evaluation board.” https://www.nxp.com/design/development-boards/automotive-development-platforms/s32k-mcu-platforms/s32k144-q100-general-purpose-evaluation-board:S32K144EVB.
[66] manitou48, “Duezoo/isrperf.txt at master · manitou48/duezoo.” https://github.com/manitou48/DUEZoo/blob/master/isrperf.txt.