Controller Area Network Intrusion Prevention System Leveraging Fault Recovery

Conference Paper · November 2019

DOI: 10.1145/3338499.3357360

Controller Area Network Intrusion Prevention System Leveraging Fault Recovery

Habeeb Olufowobi
Howard University
Washington, DC
habeeb.olufowobi@howard.edu

Sena Hounsinou
Howard University
Washington, DC
sena.hounsinou@howard.edu

Gedare Bloom
University of Colorado Colorado Springs
Colorado Springs, CO
gbloom@uccs.edu

ABSTRACT

The ever-increasing demand for safety, comfort, and automation in the automobile has increased their vulnerability to cybersecurity risk and attacks. Automobiles now embed several electronic devices to perform these functions, and the complexity in the design of these systems increases along with the functionalities they offer. These devices communicate through the vehicular network—such as controller area network (CAN) and local interconnect network—which are attractive targets for cyber attackers. In this paper, we propose a novel algorithm to detect and recover from message spoofing attacks aimed at distorting the operation of the CAN bus. Using the predictable run-time behavior of CAN message frames in our recovery process, we leverage the error handling capability (bus-off state) of the CAN bus in a reboot-based recovery process of the compromised network node. We implement this algorithm in tandem with a hardware CAN controller as a detector node, and we evaluate its effectiveness and performance in detecting and recovering a compromised node.

CCS CONCEPTS

• **Security and privacy → Intrusion/anomaly detection and malware mitigation**; **Malware and its mitigation**;

KEYWORDS

CAN, Intrusion Detection Systems, Intrusion Prevention System, Automotive Security, Reboot Recovery, Data Injection

ACM Reference Format:
Habeeb Olufowobi, Sena Hounsinou, and Gedare Bloom. 2019. Controller Area Network Intrusion Prevention System Leveraging Fault Recovery. In *ACM Workshop on Cyber-Physical Systems Security & Privacy (CPS-SPC'19), November 11, 2019, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338499.3357360

1 INTRODUCTION

Recent development in the use of embedded devices and Internet connectivity in the automobile has brought vehicles into the Internet of Things (IoT). The vehicles reliant on these electronic devices for primary functions bring concerns about their security. Nowadays, a high-end automobile may embed up to 100 different electronic control units (ECUs) to enhance the functionalities and operations of the vehicle. The attack surface of vehicles continues to grow in proportion to each of these functionalities and their concomitant increase in the complexity of the automobile system. These ECUs communicate through the in-vehicle network which includes the controller area network (CAN), local interconnect network (LIN), FlexRay, and media oriented systems transport (MOST). The CAN bus is the most commonly used network system in the automobile, and it connects ECUs through a message broadcast protocol. The CAN bus was initially designed for automotive applications and has been adopted in other applications such as industrial automation. Unfortunately, the CAN bus implements no security mechanism for the ECUs and their applications.

CAN bus has been shown to be vulnerable to remote and physical attacks [5, 9, 12, 15] as it allows unauthorized node communication and the transmitted message frames between ECUs are not encrypted or authenticated. Physical attacks can be accomplished through the on-board diagnostic (OBD-II) port, while remote attacks are achieved through the use of the wireless connectivity to the network. Koscher et al. [12] were the first to demonstrate and perform possible attacks on the vehicles by reverse engineering of the ECU codes to control a range of vehicle functions. Similarly, Checkoway et al. [5] and Miller and Valasek [15] have demonstrated the evolution of cyber threats against automotive networks by remotely connecting to the vehicles to take over vehicular functions. While other threats are still emerging, experience related to the detection and defense of these threats remains low.

The design of security for the in-vehicle network has been challenging because the requirements for security are not accurately defined while also considering the constraint of the available bandwidth of the vehicular network protocols. Recently, standards such as ISO/SAE 21434 [2] are being developed and the published J3061 cybersecurity guidebook for cyber-physical vehicle systems by Society of Automotive Engineers (SAE) is a much-anticipated standard to fill this gap in security engineering of modern vehicles [17]. An essential requirement in considering a security mechanism for the automotive in-vehicle is the algorithmic requirement of the limited computational power and memory resources of the system. Hence, a security mechanism for the in-vehicle network should be lightweight and computationally efficient while respecting network errors and fault-tolerance.

Prior art has demonstrated a plethora of intrusion detection system (IDS) approaches for CAN bus [21]. However, the response mechanism of such IDSs has been woefully ignored, and designers are left to wonder how to utilize an IDS alert in an automotive system. In this paper, we describe an intrusion prevention system (IPS) that can leverage an existing IDS and extend it with a reboot-based attack mitigation and recovery mechanism. We leverage the inherent error handling capability of the CAN bus called the bus-off state to maneuver a compromised node into a recovery state while it is under an active attack. Bus-off is a state in which the ECU is disconnected from the network and not allowed to transmit messages until in most cases it undergoes a reset.

The main contributions of this paper are:

(1) A practical approach for blocking and terminating malicious message attacks while they transmit on the CAN bus.
(2) A method demonstrating how the error handling capability of CAN bus may recover nodes under attack.
(3) Prototype and evaluation of the effectiveness and performance of the proposed IPS.

The remaining portion of this paper is organized as follows: Section 2 defines the threat and attack model and the underlying assumptions of our proposed approach; Section 3 presents the background on the CAN bus, its error handling capability, and the security challenges. In Section 4, we describe the proposed detection and recovery approach while Section 5 presents the implementation details of the IPS. Section 6 shows the experimental validation and results while Section 7 discusses the related work. Section 8 concludes this paper and describes future work.

2 THREAT MODEL AND ASSUMPTIONS

The attack scenario considered in this paper is a spoofing attack in which a malicious node impersonates another node on the network to launch attacks or infiltrate the network operations. A node represents an ECU connected to the CAN bus that can transmit and receive message frames. We assume that the adversary is able to access the bus to perform receive and transmit operations. Access is achieved by penetrating the CAN bus through remotely accessible nodes on the bus. Typically, these nodes transmit on a low-speed CAN bus, but the adversary is using this medium as a foothold to gain access to the high-speed CAN bus in order to affect the operation of the safety-critical nodes that control driving functionality.

We examine the typical effect of multistep attack scenarios where the adversary's goal is to control the safety-critical nodes on the high-speed CAN bus. The adversary starts by intercepting messages from healthy nodes transmitting on the low-speed bus. An interception operation involves a malicious node reading message frames. When access to this bus is established, the adversary sends masqueraded messages targeting safety-critical nodes on the high-speed bus. Since any node can broadcast messages on the CAN bus and each receiving node now determines if the message is meant for them or not, the adversary can successfully spoof messages targeting a specific node controlling the critical operations of the vehicle without authentication. When these target nodes accept any of the spoofed messages, the adversary has succeeded in the attack.

We assume that:

- All CAN controllers are trustworthy, i.e., the hardware controller behaves correctly with respect to the CAN protocol.

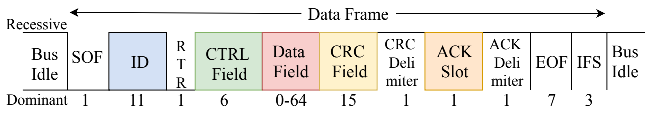


Figure 1: The Standard CAN data frame format

- A network may consist of several separate buses connected through the gateway, and the safety-critical ECUs are connected through the CAN bus.
- ECUs attached to both the CAN bus and a remotely accessible interface are configured to reboot when put in bus-off state.

3 CONTROLLER AREA NETWORK (CAN) BACKGROUND

The controller area network is a standard serial communication bus developed for use in automotive applications that interconnects ECUs through a broadcast bus. It implements carrier sense multiple access with collision detection and arbitration on message priority (CSMA/CD+AMP) and it is the most commonly used communication protocol in the modern automobile. CAN efficiently implements static fixed priority non-preemptive scheduling of message frames through bus arbitration. Message frames sent on the bus are broadcast to all nodes, i.e., ECUs, on the network. Every message broadcast contains a unique ID which represents its priority and meaning. Messages with lower ID in the bus have higher priority and get to transmit first. The message frames are of four different types: the data frames for sending data between nodes (depicted in Figure 1), remote frames for requesting transmission of a data frame with the same identifier, error frames used in signifying detected errors, and overload frames used to provide for an extra delay between frames.

CAN bus transmits signals of 0 or 1, where the term dominant bit represents the logical 0 and recessive bit denotes the logical 1 signal. When the voltage difference between the two wires (CAN high and low) is large, the state is dominant. The state is recessive when the voltage difference is small between the two wires. When a node transmits a dominant bit and another transmits a recessive bit, this will result in the transmission of the dominant bit. Automatic arbitration is built into the CAN protocol as all nodes must monitor the state of the bus during transmission and halt transmission if a dominant bit is observed when transmitting a recessive bit.

Typically, CAN buses are either high speed or low speed. High speed CAN bus communicates at a fixed rate of up to 1 Mbps. A high speed bus is terminated with 120-ohm resistors on each end to avoid transmission reflection within the bus, and it is used as a high throughput bus. Low speed CAN bus operates at a fixed rate of 125 Kbps and every node has its own termination. The low-speed bus is often referred to as fault-tolerant CAN bus as it allows communication to continue in case of a wiring failure on the CAN bus lines.

3.1 CAN Error Handling

CAN protocol implements error handling feature for nodes transmitting on the bus in order to monitor the health of the bus. This error handling feature is essential for fault-tolerance, which is vital for maintaining the functionality of system components despite failures and errors. In the CAN protocol, this feature allows nodes on the network to exercise actions like raising error flags, and retransmitting or discarding frames when an error is detected. The CAN protocol defines the following error types for error handling [3]:

- **Bit error:** Nodes transmitting frames on the bus also monitor the bus and compare the bit level to be transmitted with that monitored on the bus. A bit error occurs if these bits are different except during message arbitration.
- **Stuff error:** A stuff error occurs when six consecutive equal bits is observed in the data field which should have been coded by the method of bit stuffing.
- **Cyclic Redundancy Check (CRC) error:** A CRC error is raised when the CRC received for a message frame is different from the one calculated by the receiver for the frame.
- **Form error:** A form error occurs when a fixed-form bit field of a message frame contains illegal bits.
- **Acknowledgment (ACK) error:** ACK error is raised when a transmitting node receives no dominant bit issued by a receiver in the ACK slot.

When a node detects an error on the bus, it flags the corrupted message and transmits an error frame. An error frame initiates the termination of an erroneous data or remote frame. It signals the detection of an error condition by a receiving or transmitting node. If a node is in error active state, an active error frame is transmitted which is six dominant bits signaled on detecting an error on the network. Otherwise, a passive error frame is transmitted, which is six recessive bits transmitted by a node detecting an active error frame on the bus. The transmission of the faulty message will abort and attempt to retransmit at the next bus idle time.

Each node implements two error counters: transmit error counter (TEC) and receive error counter (REC). These counters start at zero, and they increment when an error is observed or decrement whenever the controller successfully transmits or receives a message according to the predefined rules specified by the CAN protocol. When an error is detected at the sending node, a sending node TEC is increased by 8, and the other nodes' RECs are increased by 1. When an error is detected at a receiving node, the receiver node REC is increased by 8. For a successfully transmitted message, the TEC and RECs of both the sending and the receiving nodes are decreased by 1, respectively.

The values of the TEC and the REC affect the error handling of the CAN bus as nodes change their error status. The transitions between the error states is shown in Figure 2. In error active state, the node is said to be in a healthy state when the TEC $\leq$ 127 and REC $\leq$ 127. A node will transition into the error passive state when the $TEC > 127$ or the $REC > 127$. A node in this state can partake in the bus communication but can only transmit a passive error flag when an error is detected. The error flag in this state is changed to 6 consecutive recessive bits to avoid any impacts on the bus operation and must wait before initiating further transmission. In general, a value of an error count that is higher than 96 indicates an extremely disturbed bus.

The bus-off state is an error state of the CAN controller set by the transmitting node when the TEC exceeds 255. In this state, the node is switched off from the bus and can not transmit or acknowledge

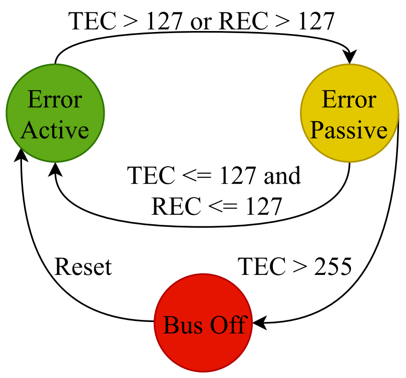


Figure 2: Flowchart of CAN Bus Error Counter

frames compulsorily. This error state is usually a result of critical hardware or software problems. A node in a bus-off state is not allowed to influence the bus operation and can only rejoin the network by transitioning to error active when its error counters are set to zero after monitoring 128 occurrences of 11 consecutive recessive bits on the bus.

3.2 Security Challenges of CAN

CAN architecture was designed to be a closed system of nodes that communicate within the vehicle. Therefore, the CAN bus is implemented with no authentication protocol to allow free flow of messages to all the nodes and these messages are sent in the clear. When a node receives a message, it decides whether the message is for it. The broadcast nature of CAN allows each node connected to the bus to broadcast and receive messages sent with no verification of the source and the destination. An attacker with bus access can eavesdrop on the messages since they are unencrypted. Also, the bus is unsegmented allowing a mix of safety-critical and non-safety-critical nodes to communicate on the same bus. This is a significant concern as it allows for an unauthorized node to transmit spoofed messages which can be used to compromise the safety-critical nodes. Furthermore, the CAN bus is vulnerable to a denial of service attack, which can be realized by leveraging the arbitration process of the bus and transmitting highest priority messages continuously to paralyze the operation of the bus. These vulnerabilities of the CAN bus can be exploited to perform several attacks which include data injection, spoofing, and replay attacks. Also, a malicious node can be placed on the bus to transmit anomalous messages that could compromise the entire operation of the bus.

4 INTRUSION PREVENTION SYSTEM DESIGN

Performance, reliability, and safety are crucial features of safety-critical applications such as automotive networks [11]. A significant challenge for designers is to balance the requirements of safety, security, and functionality. The priority of safety is the passenger's well-being, and to ensure safety, even in the worst of conditions, certain features of the vehicle must remain operational such as airbags and collision avoidance systems. This implies that the safety of critical features and operation have priority over security. However, what follows in situations where one of these safety features is

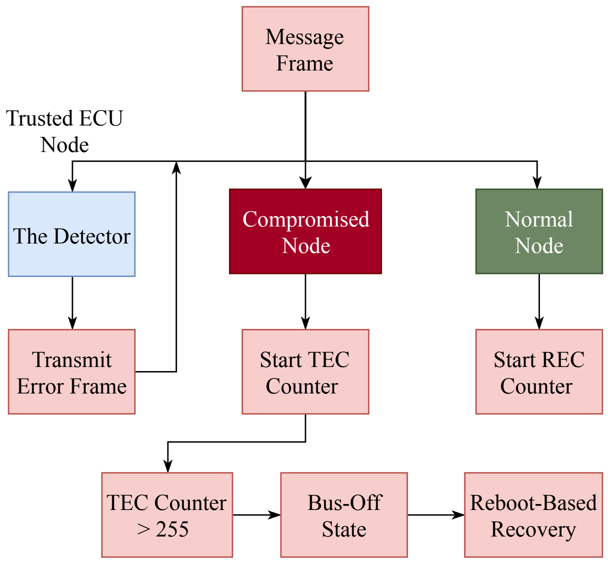


Figure 3: Process steps of the proposed recovery approach

compromised or is the target of an attack? To ensure the security of the vehicle in such situations, some features and operations may need to be limited or brought down into a degraded state while the vehicular systems maintain operational safety.

We present a reboot-based intrusion prevention approach for security covering arbitrary faults of network nodes that are under attack and also considering the effect of attack propagation in the CAN bus. Presently, our focus is on message spoofing attacks that impersonate or masquerade as a functional ECU on the bus. The reboot-based recovery is a practical recovery method for ECUs that have been compromised by a remote adversary. The goal of this approach is to prevent adversaries from propagating attack messages further into the network and control the safety-critical CAN bus operations. The high-level overview of the proposed recovery method is illustrated in Figure 3. The figure shows the proposed architectural model of the recovery approach. When a message is released on the bus, it is broadcast to all the nodes on the network, including the detector node. The detector node, which represents the IDS placed strategically to monitor messages broadcasted on the bus, performs checks and transmits an error frame if the sent message is anomalous. The compromised node increments its TEC while the healthy node also increments its REC. Eventually, incrementing the error counter leads to the process of reboot recovery as illustrated in the figure.

4.1 ECU Architecture and General System Model

The architecture of a typical ECU node contains a hardware CAN controller and transceiver that interfaces the ECU software to the CAN bus—the controller manages digital bits in frames, and the transceiver implements the physical bus access including bus arbitration. We add IPS logic in parallel to the CAN controller that processes commands and may generate CAN messages as depicted in Figure 4. This logic monitors the message frame at the controller level and contains a hardware implementation of an IDS and our proposed recovery approach, which together comprise a CAN bus IPS.

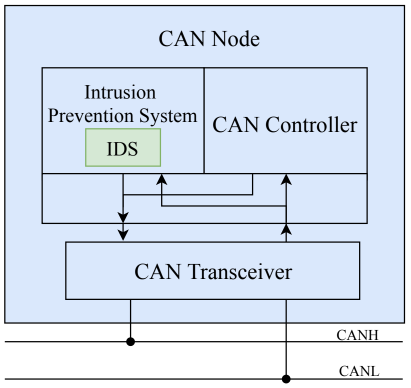


Figure 4: ECU Architecture

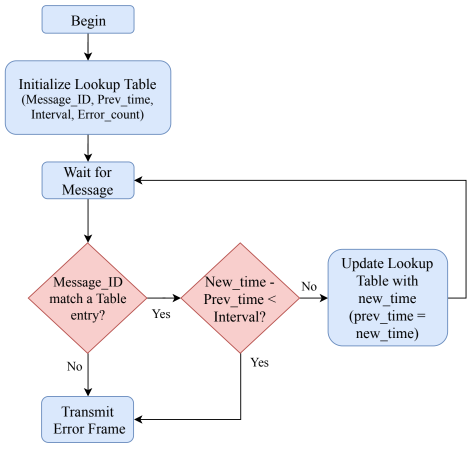
The primary functionality of the IPS logic is to monitor data transfers of the bus to protect against unauthorized access. The IPS has two operating modes: monitor and react. In the monitoring mode, the IDS component observes the bus operations while receiving message frames without meddling on the bus activities. When the IDS detects an attack, the IPS enters a reactive mode and sends an error frame to defend the system.

4.2 Detectors

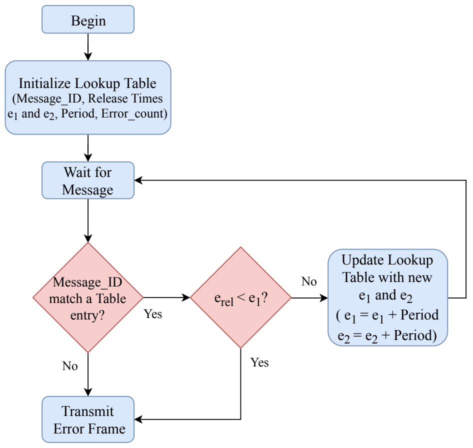
Our proposed IPS can use any kind of IDS satisfying the requirement that it can detect an attack message before that message finishes transmission on the bus. We encapsulate such an IDS in a *detector* node, which is responsible for triggering the IPS mechanism. In this work, we investigate two IDS algorithms for implementation as a detector: message interval [18] and message response time analysis [16]. In the following we briefly describe each of these.

4.2.1 Message Interval IDS. Song et al. [18] describe an IDS using the interval between messages as a feature. By examining the time interval between messages of the same ID, they evaluate how message injection attacks affect the individual time interval of each message ID. The authors determine that the time interval is a feature capable of detecting message injection attacks in CAN bus traffic by computing the time difference in the arrival of every new message of the same ID transmitted on the bus, and label a message as injected if the time interval is shorter than the predefined normal.

The detector node operation for the message interval IDS is described in Figure 5a. The controller maintains a lookup table of the message IDs, their intervals, and the transmission time of the previous instance of the message. We assume the lookup table contains the list of messages transmitting on the bus as the detector. When a new message frame is transmitted, the IDS checks the CAN ID and computes the time interval from the arrival time of the previous message with the same ID. If the time interval of the new message frame is as expected, the previous transmission time of the ID is updated in the lookup table. Else, if the calculated interval


Begin
Initialize Lookup Table (Message_ID, Prev_time, Interval, Error_count)
Wait for Message
Message_ID match a Table entry?
Yes
No
New_time - Prev_time < Interval?
No
Yes
Update Lookup Table with new_time (prev_time = new_time)
Transmit Error Frame

(a) Message Interval IDS.


Begin
Initialize Lookup Table (Message_ID, Release Times e1 and e2, Period, Error_count)
Wait for Message
Message_ID match a Table entry?
Yes
No
erel < e1?
No
Yes
Update Lookup Table with new e1 and e2 (e1 = e1 + Period e2 = e2 + Period)
Transmit Error Frame

(b) Message Response Time IDS.

Figure 5: Flowcharts of the *detector* nodes for each IDS.

is shorter, i.e., the message frame arrives sooner than expected, the IDS indicates the message is anomalous and transmits the error frame. By updating the previous transmission time, the controller can compute the difference between the received and previous frame transmission time with the stored interval.

4.2.2 Message Response Time Analysis IDS. Olufowobi et al. [16] introduced an IDS based on estimating the real-time model parameters of a set of CAN messages and using response time analysis to derive best- and worst-case response times for each message. These response times are then used to predict the arrival window of periodic messages, and the IDS triggers an attack signal if messages arrive too soon. An attack is detected when a message with an unknown ID is transmitted, the release time falls outside of the acceptable range, or more than one message is received at a period or in an interval.

Figure 5b shows the system flow diagram of the detector node operation for the message response time analysis IDS. The controller maintains a lookup table of message IDs, their earliest and

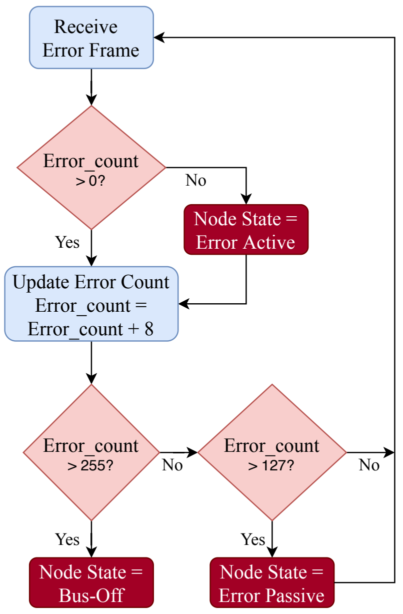


Figure 6: Flowchart of the behavior of the victim and malicious ECUs

latest release times of the next frame, and their period or interarrival time. We assume that the lookup table contains the list of all messages that transmit on the same bus as the detector node. By our attack model, these are non-safety-critical nodes (message IDs) that can be accessed remotely by an adversary to gain access to the safety-critical nodes. The inputs to the monitoring node are the lookup table and the received frame observed through the controller. When a valid frame is received and is transmitted successfully, the next expected release time of the ID is updated in the lookup table. Otherwise, the error frame is transmitted if the message violates its periodicity or sporadicity. By incrementing the counter, the controller can compare the next received message with the sequence of the message in the updated table. The sequence helps in validating the authenticity of the message as the counter should be consistent with the received message.

4.3 Attack Mitigation and Recovery

When the IDS detects an attack, the IPS immediately enqueues an error frame for transmission. This frame starts with six consecutive dominant bits, which will have the highest priority during the next bus arbitration. The nodes in receipt of the error frame will discard the message they received.

Each time a message is flagged as an attack, the IPS will transmit the error frame causing the sending node to increase its TEC by 8, and every other node on the bus will increase its REC by 1. If the attack continues until the TEC of the compromised node is higher than 255, it enters the bus-off state. This method is similar to the attack proposed by Cho and Shin [6] to drive a node to bus-off. Figure 6 shows the process of steps the compromised node goes through before entering the bus-off state.

ECUs in the bus-off must observe 128 times 11 consecutive recessive bits on the bus before they can transition back into the error active state. Prior to this transition, an ECU may also reset or reboot itself. We suggest that all ECUs with remote interface capability undergo a reboot process when they reach bus-off. A reboot process represents a recovery procedure that provides a way to restore the initial system state. The reboot process does not depend on the correct functioning of the rebooted system, is easy to implement and automate, and returns the software to its initial state, which is often its best understood and best-tested state [4]. Power-cycling is fast with minimal impact on the time it takes for the system to recover, and can provide high availability of these ECUs even when a detection algorithm is prone to false alarms or when it is unknown if the reboot process can correct the failure. Also, it should be noted that these ECUs are not the safety-critical ones, so the impact of rebooting them will not affect system safety, but perhaps will negatively affect user experience.

The reboot process plays a pivotal role in keeping the node in a bus-off state alive along with the reboot policy that performs the actual restart. The reboot policy specifies how the node in the bus-off state will be restarted, and it will be initiated in the application layer of the CAN controller. There are different kinds of reset policies to decide when and how the application layer should reset the CAN controller. These policies include automatic reset policy, wait-then-reset policy, and frequency-limited-reset policy [10].

In automatic reset, the reset is initiated immediately after the CAN controller enters the bus-off state while the wait-then-reset requires the application layer to observe a predefined wait period to recover from the fault condition before starting the reset. In the frequency-limited-reset, the time between any two subsequent resets must be greater than a predetermined time interval. When the CAN controller is in a bus-off state, the reset vector of the application layer is triggered to initialize the reset, and the CAN controller observes the predetermined number of recessive bits before rejoining the network.

4.4 Discussion and Limitations

In a transient attack scenario where the adversary established a network connection and compromised the ECU in the RAM (i.e., malware is RAM-resident), resetting the ECU will neutralize the attack process. However, perhaps the exploit still exists, so the attacker can relaunch the same attack. For a persistent attacker, this process increases the burden on the attacker as there will be a need to reestablish the attack by re-infecting the ECU every time a reboot occurs. Resetting the compromised node represents a good step in a remediation process even if not foolproof.

If the attacker can modify the ROM or flash the ECU, then a reset will not evict the attacker necessarily but it will disrupt the attack process. This persistent attack requires a sophisticated process to accomplish. If the attacker is persistent and the ECU needs to reset itself several times, a crude way to handle the repeated faults would be to have an indicator light such as the check engine light to notify the driver of a potential vehicular malfunction.

Currently, it is not typical for ECUs connected to the CAN bus with remotely accessible interfaces to have reboot capability when in a bus-off state. The reboot process is implementation-dependent. Some ECUs may reboot after a driver-initiated power cycle or a visit to a service station. Our proposed approach may require the use of controllers that have this particular feature built-in, and it is in the manufacturer's purview to decide whether or not to include this feature. Also, our assumption about trustworthy CAN controller can be violated, e.g., in case an attacker can reprogram the (software/firmware) controller.

5 IPS IMPLEMENTATION

To demonstrate the effectiveness of our approach, we developed a proof of concept implementation using the Xilinx LogiCORE IP CAN v5.0 as a reference [20]. This core conforms to the ISO 11898-1, CAN 2.0A, and CAN 2.0B standards. It is particularly suited for automotive applications and has user-configurable options that provide flexibility for multiple ECU applications. It also supports message prioritization via its high priority buffer (TX HPB) and readable error counters. As such, it allows for seamless integration of our detector functionalities. Our implementation targeted the Zynq-7000 SoC.

The CAN nodes in our network are connected to the CAN bus via the physical interface (CAN PHY). Each Xilinx CAN node can operate in stand-alone mode or connected to a Control block or processor using its AXI4-Lite Interface located inside the CAN Controller. The Controller has an Object Layer for message storage, filtering, and status updating. It also has a transfer layer where the CAN protocol engine resides.

The CAN protocol engine consists primarily of the bit timing logic (BTL), the bit stream processor (BSP) and the clock prescalar modules. The BTL synchronizes the operation of the CAN bus and the BSP. At the appropriate clock tick, the BTL captures a received bit or places a transmitted data bit on the CAN bus. It also produces a sampling clock signal for the BSP. The BSP analyzes bus traffic during transmission and reception, updates the error counters and the error state when necessary, and manages operations dealing with CAN message transmission and reception. It captures message frames from the high priority buffer (TX HPB) or from the transmission queue. It also inserts the error flags (bit, stuff, form, CRC and ACK errors). The frame's bits are serialized and constructed into fields per the CAN core messaging protocols at this stage. The reverse operation is also completed by the BSP when data is received. Message frames are deconstructed and stored in the receive queue (RX FIFO) where the identifier of the received message (IDR) can be accessed by the detector node. The IDR register is four bytes long, and its 11 most significant bits store the message ID from the message frame. The arrival time stamp (T_arrival) is generated by a system clock counter which is started after the system's initial reset. For each message, T_arrival is also recorded and submitted to the IPS inside the detector node along with the clock signal (clk), which is derived from the main CAN engine protocol clock. In our approach, the IPS module sits at the base of the BSP in the CAN protocol engine as shown in Figure 7.

The implementation features three modules: Message ID Check, Check Message Feature and Update State. After the initial system reset, the Message ID Check module is activated once a message is received and ready to be read from the RX FIFO. The sender node

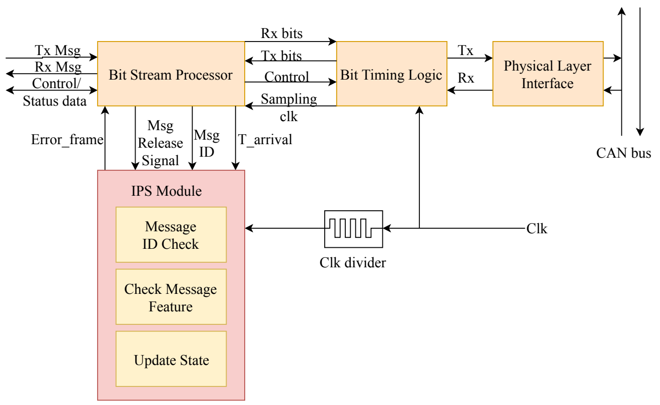


Figure 7: CAN controller with IPS module

identification Msg ID is extracted and compared against entries of a look-up table containing all known nodes in the network. If a match is not found in the table, the error_frame signal is activated to indicate that the node from which the message is received is unknown so invalidate the message. Upon a successful match, the detector examines the received message inside the Check Message Feature, which is where we implement the IDSs used for evaluation.

When Check Message Feature detects an anomaly, the message is invalidated by signaling the error frame and the error count update is triggered at the BSP level. In the case T_arrival's range is valid, the Update State module becomes active. It updates the stored state depending on the IDS in use, for example programming the next expected arrival time of a message or incrementing message counters.

6 EXPERIMENTAL VALIDATION

Here, we describe the evaluation criteria for our IPS to illustrate its effectiveness and performance. First, we focus on the latency required for the detector node to issue a decision after a message frame has been received at the controller level. We compare the implementation results of two different IDSs, the interval-based and response time analysis-based approaches. We also offer an analysis of the minimum operating speed suitable for various CAN bus speeds, using different data lengths for a standard message frame. The experimental network is composed of 6 nodes and a detector node communicating on the bus. Each node is capable of transmitting a message to one or more nodes in the network by broadcasting it through the bus. Each node may also send one or multiple messages of the same type. However only one message can be sent at a particular instance. In the case of time response IDS, the periodicity of each message as well as the expected arrival time range of the first message originating from each node is known to the supervisory node. Similarly, the minimum interval threshold for each message is saved for the detector node during setup.

6.1 Area

Table 1 shows that the IPS only requires 0.03% and 0.14% of slice registers and LUTs, respectively. The total number of occupied slices is 6 out of the 4400 available on the platform and only 18 LUT-flip flop pairs are utilized. The table also shows the area requirements of the message interval and message response time IDSs. When compared against each other, the difference in resource usage is noticeable. The message response time IDS required 18% more slice registers, 8.5% more slice LUTs, approximately 37% more occupied slices, and an additional 22.7% LUT-flip flop pairs. However, when the available resources are taken into consideration, both designs require very little hardware for implementation. For the message response time analysis IDS, 0.3% of available slice registers, 0.6% of slice LUTs and 1.04% occupied sliced were used. In the case of the message interval IDS, the usage was 0.23%, 0.55% and 0.66% respectively.

We also compared the resources used in both types of IDS to the original CAN controller design. As described in [20], the number of slice LUTs and registers necessary to implement the CAN controller core increases as the depth of receiver or transmitter FIFOs increases. Valid values for RX/TX FIFO depth range from 2 to 64. Thus, integrating an IPS module in the CAN core would have a higher impact on controllers with an RX/TX FIFO depth of 2. Table 2 shows the usage of the unmodified CAN controller (with a FIFO depth of 2) depending on the number of acceptance filters available. It also shows the overhead generated by integrating each IDS. As can be seen, for a CAN controller with no acceptance filter, the response time detection approach yields a 14.83% and 18.74% increase in slice LUTs and registers utilization respectively. On the other hand, for the same size controller, the interval based detection approach generated a 13.57% and 15.30% overhead respectively.

Table 1: Synthesis area results

	IPS		Interval IDS		Response Time IDS		
	Resources Used	% of Available Resources	Resources Used	% of Available Resources	Resources Used	% of Available Resources	Total Available Resources
Slice Registers	10	0.03%	70	0.20%	88	0.25%	35200
Slice LUTs	18	0.14%	79	0.63%	88	0.70%	12600
Occupied Slices	6	0.14%	23	0.52%	40	0.91%	4400
LUT-Flip Flops	18	-	84	-	114	-	-

6.2 Detection Latency

This evaluation considers only the computation time for a single message at the detector level. The clock signal of the detector block is derived from the CAN controller clock.

For each message received, the detector node performs various checks which include message instance ID checks and a check based on the arrival time. When the message instance passes both checks, the detector node updates the arrival time of the next instance of

Table 2: Overhead of detector

	Slice LUTs Utilization			Slice Registers Utilization		
Acceptance Filters	Original CAN Controller	Response Time IDS Overhead	Interval IDS Overhead	Original CAN Controller	Response Time IDS Overhead	Interval IDS Overhead
0	715	14.83%	13.57%	523	18.74%	15.30%
1	788	13.45%	12.31%	617	15.88%	12.97%
2	794	13.35%	12.22%	620	15.81%	12.90%
3	802	13.22%	12.09%	623	15.73%	12.84%
4	808	13.12%	12.00%	626	15.65%	12.78%

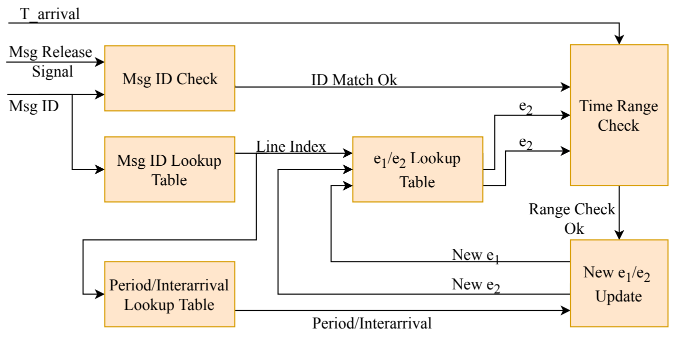


Figure 8: Process check of the *detector* node using the message response time analysis IDS

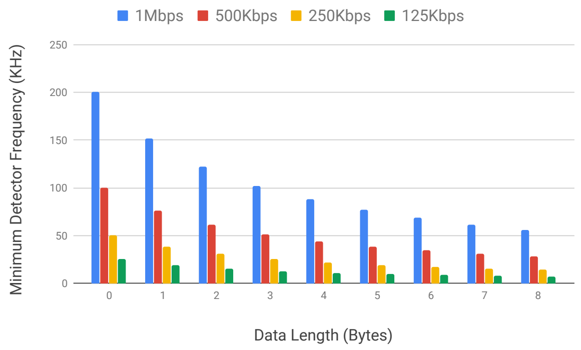


Figure 9: Detector Behavior for Different Bus Speeds

the message. This process is depicted in Figure 8 in detail for the message response time analysis IDS. If the message fails one of the checks, the detector node transmits an error frame and increases its REC by one while the TEC of the transmitting node is increased by 8.

Regardless of the IDS in use, if a message frame is received with an ID that is not in the detector's lookup table, it takes the detector five clock cycles to check for validity. The detector node queues the error frame for transmission, which will occupy the bus immediately after the injected message completes, thus invalidating the message frame of that unknown ID.

For each IDS, the time needed for checking the message feature for an anomaly and updating the state may differ. The following characterizes these costs for each IDS we consider.

- **Message Interval IDS.** This IDS can run at $417MHz$ maximum frequency or $2.4ns$ cycle time and uses five clock cycles to check the difference between the previous message transmission time and the transmission time of the new message frame compared with the saved interval. The lookup table is updated with the record of the new transmission time denoting the previous time for the next arriving message. Therefore, it takes approximately $12ns$ to complete the entire check.
- **Message Response Time Analysis IDS.** This IDS can run at $400MHz$ maximum frequency or $2.5ns$ cycle time and also uses five clock cycles to check whether the interarrival time of the message fits in the expected arrival time range of the next message for the matching ID. The interarrival time of the next message in the lookup table is updated with the record of the next arrival time. Therefore, the detection time is $12.5ns$.

Thus, both of the considered IDSs require at most $12.5ns$ to complete detection.

The static timing analysis of the post-route implementation shows that the detector node can operate with a minimum clock period of $2.33ns$. However, we have only been able to achieve a clock period of $2.5ns$. In other words, the maximum operating clock frequency achieved without violating any time constraints is $400MHz$, which is about 16 to 50 times faster than the IP CAN controller clock of 8 to $24MHz$. In addition, per [20], the characterization results of the IP Core runs between $160MHz$ to $220MHz$ for a Zynq board running at a speed of -3. Thus, our detector block can run fast enough to generate the necessary signals for the node.

To understand the time to transmit a bit of the message frame when the message ID bits are received and decide on its validity, we assume a bus speed of $1Mbps$. This bus is the fastest and gives the quickest time the detector has to be. The transmission time for a single bit, τ_{bit}, in this bus is $1/1000000 = 1 \times 10^{-6} seconds$. In a frame with 0-bytes of data, a maximum time of $25 \times \tau_{bit}$ is available to decide on the message frame after the entire arbitration field is received. Therefore, depending on the number of data bytes transmitted, the shortest time available to decide on the message from the end of the ID to the end of the message frame is about $25000ns$, excluding the end-of-frame bits. This minimum occurs when 0-bytes of data are included in the message frame, and for each additional data byte, the decision window increases by 8-bit transmission time — effectively decreasing the required operating frequency imposed on the detector node.

Figure 9 shows the minimum frequency of the detector at different bus speeds while varying the number of data bits (0 to 8 bytes) being transmitted. The minimum operating frequency is desirable to reduce power costs, while the minimum data length, overall (valid) messages of the bus have to be considered. Thus, if any message uses 0-bytes data length, then the 0-bytes analysis must be used in determining the minimum operating frequency. As can be seen in the chart, for different bus speeds, the required minimum operating frequency decreased gradually to approximately a third when the number of data bytes increased from 0 to 8 bytes. Although the required operating frequency varies according to the bus speed, the detector must issue a decision within 25-bit transmission time. In the worst case on a $1Mbps$ data bus, when the frame contains no data byte, the detector must operate at a minimum of $200KHz$ or $5000ns$ cycle time. As described above, with the implemented IDS, the detector is able to operate as fast as $400MHz$ and complete the message validation in $12.5ns$. At this speed, it will have completed its operation and decided on the still-queued data frame (bit 19 to 83) before the completion of the 13th bit which takes $1000ns$. In an attack scenario where the message ID is not in the lookup table or the timing is not as expected, the detector node will have decided on the message, and an error frame will have been queued well before the end of the transmission of the spoofed message. In the best case scenario on a $1 Mbps$ bus and the the messages transmitted have 8-bytes of data length, the detector can operate at a minimum frequency of $56 KHz$. With this, it takes $17800 ns \times 5$ clock cycles $= 89000 ns$ to validate the message frame. This is still enough time to issue the error frame right before the last bit in the end-of-frame and before the message completes.

Additionally, in many vehicles, the non-safety-critical ECUs are attached to the medium and lower speed buses in which the transmission time is even longer. This also implies that a detector node operating in the worst case frequency of $56 KHz$ has much more time to complete its operations and send an invalidation message. By this, our approach to mitigating anomalous behavior in the vehicular network is practical, and the proposed algorithm is lightweight and computationally efficient.

6.3 Time to Error Passive and Bus-Off States

We calculate the time for the compromised node to transition to error passive and bus-off state. As noted by Davis et al. [8], using $1 Mbps$ bus speed, the transmission time, C_m for a single bit is $\tau_{bit} = \frac{1}{busspeed} = \frac{1}{1,000,000}$, and the error frame has a maximum of 23 bits. Assuming the message is of the highest priority and the error frame experiences no interference during its transmission when the anomalous message is detected, the total time consumed is calculated by:

$$T_{EP} = 16(C_m + 23 \times \tau_{bit}) \tag{1}$$

where 16 represents the total number of anomalous messages required to transition into another state, and C_m is the maximum transmission time of a CAN message including the stuff bits and the inter-frame space. C_m of a message with an 11-bit identifier containing s_m data bytes including the stuff bits and inter-frame space is given by:

$$C_m = (55 + 10 s_m)\tau_{bit}. \tag{2}$$

Similarly, if the anomalous message is not of the highest priority transmitting in the bus, we incorporate the notion of message interference which is due to higher priority messages that may win arbitration and get transmitted before the message. The recurrence relation (equation 3) gives the interference where k are messages with higher priority than message i, and T_k is their respective periods. The starting value for $w_i^0 = 0$ and terminates when $w_i^{n+1} = w_i^n$.

$$w_i^{n+1} = \sum_{k<i} \left\lceil \frac{w_i^n + \tau_{bit}}{T_k} \right\rceil C_k \tag{3}$$

Therefore, from these calculations, it takes 2.53 ms for a message with the highest priority to transition into the error passive state, and approximately the same time to transition into the bus-off state. The need to transition the compromised node into a bus-off state as fast as possible is to prevent the attack from propagating into the entire network. This propagation can affect the safety-critical nodes which can affect the operation of the entire system. Also, we want to assure a seamless transition of the anomalous node to bus-off without impeding the performance of the vehicle before the system is compromised further. Figure 10 shows the time it takes for a message with the highest priority and different data bytes to transition into error passive state for different bus speeds.

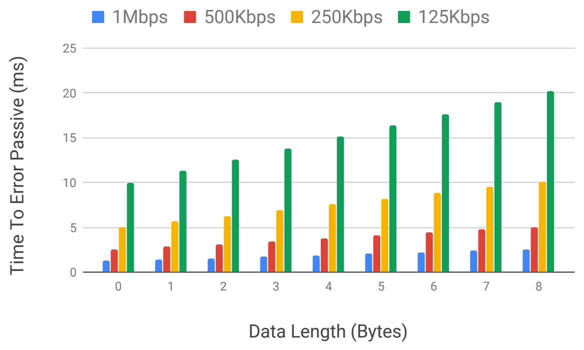


Figure 10: Time to error passive for different bus speeds

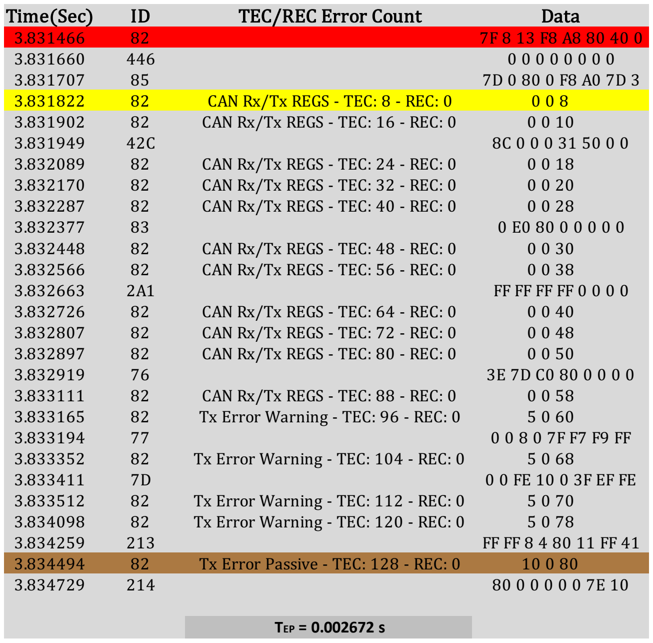
6.4 Case Study

In this section, we confirm the calculation of Equation 1 on the high speed CAN bus of a vehicle with 500 $kbps$ bus speed by measuring how long it takes for a node to transition into an error passive and bus-off states. Figure 11 and 12 show the logs from our sniffer. The anomalous message (ID 0x82) is the 6th highest priority message transmitting on the bus. As shown in Figure 11, it takes approximately 2.672 ms for this node to transition into the error passive state. This number represents the difference between the time the node enters the error passive state (highlighted in brown) and the time the first error flag was transmitted (highlighted in yellow) for message ID 0x82 (highlighted in red). Similarly, as shown in Figure 12, it takes approximately 3.732 ms for the same message ID to transition into the bus-off state (highlighted in magenta). After observing the required number of recessive bits, the TEC and REC for this ID go through reset (highlighted in orange) and the next successful transmission (highlighted in green) shows the transition of this node from bus-off to error active state.

7 RELATED WORK

The related work falls in two categories. First, there is a lot of prior work in automotive IDS, from which we have chosen just two IDSs to consider in our approach. Other automotive IDSs can also be used to realize the detector node if they satisfy the basic requirements of signalling a detected attack prior to the completion of message transmission (i.e., before the inter-frame space), so that the error frame can be transmitted in time to thwart the attack message. Second, the most closely related work uses a similar mechanism to try to push an attacking ECU into a bus-off state, which we review in the following.

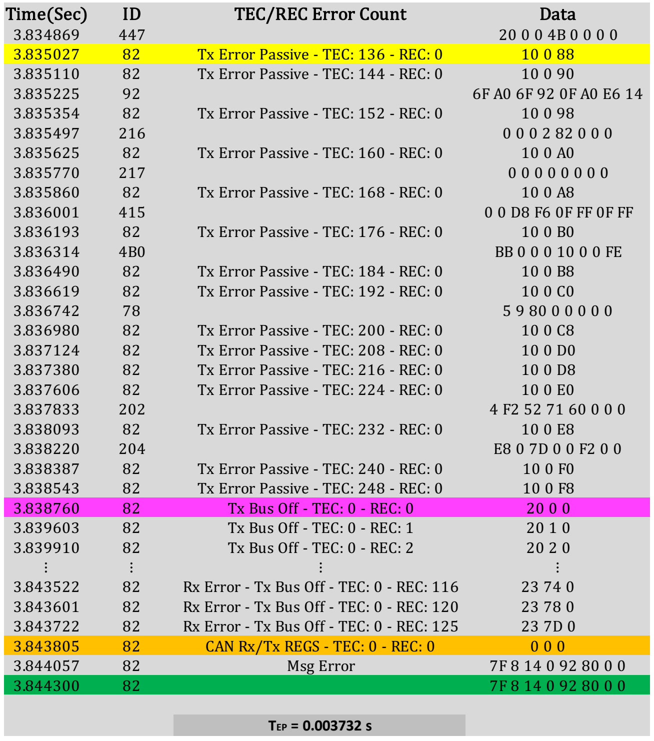
Kurachi et al. [13] proposed a centralized authentication system for preventing malicious message transmission in CAN using the error frame. The authors used a single centralized ECU to verify the MAC of all CAN messages and send out an error frame if the MAC attached is invalid. Their method requires a modification to all ECUs to be able to share keys and generate MACs when messages are transmitted. This approach will be very costly to implement and



Time(Sec)	ID	TEC/REC Error Count	Data
3.831466	82		7F 8 13 F8 A8 80 40 0
3.831660	446		0 0 0 0 0 0 0 0
3.831707	85		7D 0 80 0 F8 A0 7D 3
3.831822	82	CAN Rx/Tx REGS - TEC: 8 - REC: 0	0 0 8
3.831902	82	CAN Rx/Tx REGS - TEC: 16 - REC: 0	0 0 10
3.831949	42C		8C 0 0 0 31 50 0 0
3.832089	82	CAN Rx/Tx REGS - TEC: 24 - REC: 0	0 0 18
3.832170	82	CAN Rx/Tx REGS - TEC: 32 - REC: 0	0 0 20
3.832287	82	CAN Rx/Tx REGS - TEC: 40 - REC: 0	0 0 28
3.832377	83		0 E0 80 0 0 0 0 0
3.832448	82	CAN Rx/Tx REGS - TEC: 48 - REC: 0	0 0 30
3.832566	82	CAN Rx/Tx REGS - TEC: 56 - REC: 0	0 0 38
3.832663	2A1		FF FF FF FF 0 0 0 0
3.832726	82	CAN Rx/Tx REGS - TEC: 64 - REC: 0	0 0 40
3.832807	82	CAN Rx/Tx REGS - TEC: 72 - REC: 0	0 0 48
3.832897	82	CAN Rx/Tx REGS - TEC: 80 - REC: 0	0 0 50
3.832919	76		3E 7D C0 80 0 0 0 0
3.833111	82	CAN Rx/Tx REGS - TEC: 88 - REC: 0	0 0 58
3.833165	82	Tx Error Warning - TEC: 96 - REC: 0	5 0 60
3.833194	77		0 0 8 0 7F F7 F9 FF
3.833352	82	Tx Error Warning - TEC: 104 - REC: 0	5 0 68
3.833411	7D		0 0 FE 10 0 3F EF FE
3.833512	82	Tx Error Warning - TEC: 112 - REC: 0	5 0 70
3.834098	82	Tx Error Warning - TEC: 120 - REC: 0	5 0 78
3.834259	213		FF FF 8 4 80 11 FF 41
3.834494	82	Tx Error Passive - TEC: 128 - REC: 0	10 0 80
3.834729	214		80 0 0 0 0 0 7E 10

T_{EP} = 0.002672 s

Figure 11: Log showing node 0x82 (in red) transition into error passive state. The yellow and brown lines indicate the transmission of the first error flag and the transition into error passive state respectively. T_{EP} represents the time to error passive



Time(Sec)	ID	TEC/REC Error Count	Data
3.834869	447		20 0 0 4B 0 0 0 0
3.835027	82	Tx Error Passive - TEC: 136 - REC: 0	10 0 88
3.835110	82	Tx Error Passive - TEC: 144 - REC: 0	10 0 90
3.835225	92		6F A0 6F 92 0F A0 E6 14
3.835354	82	Tx Error Passive - TEC: 152 - REC: 0	10 0 98
3.835497	216		0 0 0 2 82 0 0 0
3.835625	82	Tx Error Passive - TEC: 160 - REC: 0	10 0 A0
3.835770	217		0 0 0 0 0 0 0 0
3.835860	82	Tx Error Passive - TEC: 168 - REC: 0	10 0 A8
3.836001	415		0 0 D8 F6 0F FF 0F FF
3.836193	82	Tx Error Passive - TEC: 176 - REC: 0	10 0 B0
3.836314	4B0		BB 0 0 0 10 0 0 FE
3.836490	82	Tx Error Passive - TEC: 184 - REC: 0	10 0 B8
3.836619	82	Tx Error Passive - TEC: 192 - REC: 0	10 0 C0
3.836742	78		5 9 80 0 0 0 0 0
3.836980	82	Tx Error Passive - TEC: 200 - REC: 0	10 0 C8
3.837124	82	Tx Error Passive - TEC: 208 - REC: 0	10 0 D0
3.837380	82	Tx Error Passive - TEC: 216 - REC: 0	10 0 D8
3.837606	82	Tx Error Passive - TEC: 224 - REC: 0	10 0 E0
3.837833	202		4 F2 52 71 60 0 0 0
3.838093	82	Tx Error Passive - TEC: 232 - REC: 0	10 0 E8
3.838220	204		E8 0 7D 0 0 F2 0 0
3.838387	82	Tx Error Passive - TEC: 240 - REC: 0	10 0 F0
3.838543	82	Tx Error Passive - TEC: 248 - REC: 0	10 0 F8
3.838760	82	Tx Bus Off - TEC: 0 - REC: 0	20 0 0
3.839603	82	Tx Bus Off - TEC: 0 - REC: 1	20 1 0
3.839910	82	Tx Bus Off - TEC: 0 - REC: 2	20 2 0
⋮	⋮	⋮	⋮
3.843522	82	Rx Error - Tx Bus Off - TEC: 0 - REC: 116	23 74 0
3.843601	82	Rx Error - Tx Bus Off - TEC: 0 - REC: 120	23 78 0
3.843722	82	Rx Error - Tx Bus Off - TEC: 0 - REC: 125	23 7D 0
3.843805	82	CAN Rx/Tx REGS - TEC: 0 - REC: 0	0 0 0
3.844057	82	Msg Error	7F 8 14 0 92 80 0 0
3.844300	82		7F 8 14 0 92 80 0 0

T_{EP} = 0.003732 s

Figure 12: Log showing node 0x82 transition into bus-off state and resetting the error counters. The yellow and magenta lines indicate the transmission of the first error flag from the error passive state and the transition into bus-off state, respectively. The orange and green lines indicate the error counter reset and the first successful message transmission after the reset, respectively. T_{EP} represents the time to bus-off.

deploy for modern vehicles that may contain over 100 ECUs. Also, adoption of this scheme is unlikely since these ECUs are sourced from different original equipment manufacturers. Since ECUs need software modification to execute the node authentication and key exchange, the ECUs might not have enough computational power or memory to execute such an algorithm.

Matsumoto et al. [14] proposed an approach for preventing unauthorized message transmission in CAN bus using the error frame. In their approach, each ECU detects unauthorized data transmission using its message ID by monitoring the data on the bus. The ECU transmits an error frame to override the message if it detects that the message is unauthorized before it finishes transmission.

Dagan and Wool [7] proposed the Parrot system to mitigate spoofing attacks in CAN bus. In their approach, the Parrot defense launches a counter-attack of carefully crafted collisions to damage the spoof message and drive the compromised ECU into a bus-off state. This solution can be implemented as a software patch to each ECU.

Abbott-McCune and Shay [1] proposed an intrusion prevention system that monitors the CAN bus to detect invalid messages by matching the message start-of-frame field with the one preprogrammed in the ECUs connected to the CAN bus. When a match is detected while the connected ECU is not transmitting, the ECU identifies a replay attack and sends an alert to the detector to signal a replay attack. In this approach, each segment of the network requires a device that can be implemented in the gateway to monitor the network activities and compare the message IDs transmitted to the valid ID, then flag non-matching ones as anomalous. The authors briefly mention the possibility for the detector to emit a burst of dominant bits in case of a detected attack to cause an error that will eventually cause the attacker to shut down. However, they do not provide details or evaluation of this mechanism, which may cause unintended negative side effects on the CAN bus as it does not conform to CAN specifications. Souma et al. [19] proposed a countermeasure to bus-off attacks in the CAN bus using a similar approach of a burst of dominant bits.

The prior approaches require modification to the software stack interfacing the CAN controllers with the bus, which implies modified software and hardware for each ECU in the vehicle. In contrast to existing centralized IDSs that have misdetection errors, prior work [1, 7, 14] that rely on the authentic ECUs to detect an attack have the potential for perfect classification when the authentic ECU is not compromised. Our approach, while similar in nature, does not require authentic ECUs to act as part of the defense scheme. Thus, our method has a lower adoption cost and greater practicality.

8 CONCLUSION

We presented a novel IPS for CAN bus that can prevent remote message injection attacks from succeeding and can trigger reboot-based recovery of a remotely-compromised ECU. We synthesized previously proposed CAN IDSs and measured their ability to detect attacks at latencies within bus line speeds, and we analyzed the effectiveness of using the CAN error handling mechanisms to drive an active attacker off the bus. Our detector node is capable of deciding on the message frame being broadcasted between the transmission of the last bit of the arbitration field and the end of the message frame. We provided a case study with message ID 0x82 which has an authentic period of 20 *ms*. In this case study, the recovery mechanism transitions the anomalous node into a bus-off state within approximately 6*ms* which is less than the periodicity of the legitimate message frame. Future work will integrate the IPS in a real automotive system, determine the impact of false positives on attack mitigation, and measure the performance degradation of reboot-based recovery.

ACKNOWLEDGMENTS

This material is based upon work supported by Northrop Grumman and the NSF under Grant CNS-1646317 and OAC-1839321.

REFERENCES

[1] S. Abbott-McCune and L. A. Shay. 2016. Intrusion prevention system of automotive network CAN bus. In *2016 IEEE International Carnahan Conference on Security Technology (ICCST)*. 1–8. https://doi.org/10.1109/CCST.2016.7815711

[2] Angela Barber. 2018. Status of work in process on ISO/SAE 21434 Automotive Cybersecurity Standard. *presentation, ISO SAE International, April* 10 (2018).

[3] Robert Bosch et al. 1991. CAN specification version 2.0. *Rober Bousch GmbH, Postfach* 300240 (1991), 72.

[4] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. 2004. Microreboot–A Technique for Cheap Recovery. *arXiv preprint cs/0406005* (2004).

[5] Stephen Checkoway, Damon Mccoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. 2011. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *USENIX SECURITY*. USENIX.

[6] Kyong-Tak Cho and Kang G Shin. 2016. Error handling of in-vehicle networks makes them vulnerable. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1044–1055.

[7] Tsvika Dagan and Avishai Wool. 2016. Parrot, a software-only anti-spoofing defense system for the CAN bus. *ESCAR EUROPE* (2016).

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

[9] Tobias Hoppe, Stefan Kiltz, and Jana Dittmann. 2008. Security threats to automotive CAN networks–practical examples and selected short-term countermeasures. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 235–248.

[10] Shengbing Jiang, Mutasim A Salman, Michael A Sowa, and Katrina M Schultz. 2017. Approach for controller area network bus off handling. (March 21 2017). US Patent 9,600,372.

[11] John C Knight. 2002. Safety critical systems: challenges and directions. In *Proceedings of the 24th international conference on software engineering*. ACM, 547–550.

[12] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. 2010. Experimental Security Analysis of a Modern Automobile. In *2010 IEEE Symposium on Security and Privacy*. 447–462. https://doi.org/10.1109/SP.2010.34

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

[14] Tsutomu Matsumoto, Masato Hata, Masato Tanabe, Katsunari Yoshioka, and Kazuomi Oishi. 2012. A method of preventing unauthorized data transmission in controller area network. In *2012 IEEE 75th Vehicular Technology Conference (VTC Spring)*. IEEE, 1–5.

[15] C. Miller and C. Valasek. 2015. Remote exploitation of an unaltered passenger vehicle. *Unknown Journal* (2015).

[16] Habeeb Olufowobi, Gedare Bloom, Clinton Young, and Joseph Zambreno. 2018. Work-in-Progress: Real-Time Modeling for Intrusion Detection in Automotive Controller Area Network. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 161–164.

[17] Christoph Schmittner, Zhendong Ma, Carolina Reyes, Oliver Dillinger, and Peter Puschner. 2016. Using SAE J3061 for automotive security requirement engineering. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 157–170.

[18] Hyun Min Song, Ha Rang Kim, and Huy Kang Kim. 2016. Intrusion detection system based on the analysis of time intervals of CAN messages for in-vehicle network. In *Information Networking (ICOIN), 2016 International Conference on*. IEEE, 63–68.

[19] Daisuke Souma, Akira Mori, Hideki Yamamoto, and Yoichi Hata. 2018. Counter Attacks for Bus-off Attacks. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 319–330.

[20] Xilinx. 2016. CAN v5.0 LogiCORE IP Product Guide. (August 2016). https://www.xilinx.com/support/documentation/ip_documentation/can/v5_0/pg096-can.pdf

[21] Clinton Young, Joseph Zambreno, Habeeb Olufowobi, and Gedare Bloom. 2019. Survey of Automotive Controller Area Network Intrusion Detection Systems. *IEEE Design & Test* (2019).