CopyCAN: An Error-Handling Protocol based Intrusion Detection System for Controller Area Network

Stefano Longari*
Dipartimento di Elettronica, Informazione e Bioingegneria,
Politecnico di Milano
Milano, Italy
stefano.longari@polimi.it

Matteo Penco*
Dipartimento di Elettronica, Informazione e Bioingegneria,
Politecnico di Milano
Milano, Italy
matteo.penco@mail.polimi.it

Michele Carminati
Dipartimento di Elettronica, Informazione e Bioingegneria,
Politecnico di Milano
Milano, Italy
michele.carminati@polimi.it

Stefano Zanero
Dipartimento di Elettronica, Informazione e Bioingegneria,
Politecnico di Milano
Milano, Italy
stefano.zanero@polimi.it

ABSTRACT

In the last years, the automotive industry has incorporated more and more electronic components in vehicles, leading to complex on-board networks of Electronic Control Units (ECUs) that communicate with each other to control all vehicle functions, making it safer and easier to drive. This communication often relies on Controller Area Network (CAN), a bus communication protocol that defines a standard for real-time reliable and efficient transmission. However, CAN does not provide any security measure against cyber attacks. In particular, it lacks of message authentication, leading to the possibility of transmitting spoofed CAN messages for malicious purposes. Nowadays, Intrusion Detection Systems (IDSs) detect such attacks by identifying inconsistencies in the stream of information allegedly transmitted by a single ECU, hence assuming the existence of a second malicious node generating these messages. However, attackers can bypass this defense technique by disconnecting from the network the ECU of which they want to spoof the messages, therefore removing the authentic source of information.

To contrast this attack, we present CopyCAN, an Intrusion Detection System (IDS) that detects whether a node has been disconnected by monitoring the traffic and deriving the error counters of ECUs on CAN. Through this process, it flags subsequent spoofed messages as attacks and reacts accordingly even if there is no inconsistency in the stream of information. Our system, unlike many previous attempts to address security issues in CAN, does not require any modification to the protocol or to already installed ECUs. Instead, it only requires the installation of a monitoring unit to the existing network, making it easily deployable in current systems and compliant with required CAN standards.

*Both authors contributed equally to this research.

Unpublished working draft. Not for distribution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACM CPS-SPC, November 11, 2019, London, UK
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Stefano Longari, Matteo Penco, Michele Carminati, and Stefano Zanero. 2019. CopyCAN: An Error-Handling Protocol based Intrusion Detection System for Controller Area Network. In *Proceedings of ACM Workshop on Cyber-Physical Systems Security & Privacy (CPS-SPC) (ACM CPS-SPC).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the last decades, the adoption of electronic components inside vehicles has increased exponentially. Modern vehicles incorporate up to 200 ECUs, which control not only the engine but perform various functions that make driving easier and safer (e.g., Automatic transmission, (Adaptive) Cruise Control, Anti-lock Brake Systems, Autonomous driving technologies). ECUs form complex on-board networks and their communication relies mainly on Controller Area Network (CAN) [12], a bus communication protocol designed by Robert Bosch GmbH in 1983 for automotive applications to provide reliable and efficient in-vehicle communication in real-time between ECUs. These features made CAN the standard for on-board vehicle communication for over 30 years up to today. Moreover, vehicles are widely connected to the outside world, through both local and remote connections (Bluetooth, cellular radio, GPS systems and so on). This tendency keeps increasing through the years, leading to consider them as giant computers moving on the road, able to send and receive data, and to communicate with each other.

However, the growth of on-board network systems and their interconnection with the outside world has increased both the attack surfaces and the vulnerabilities exploitable for malicious purposes. Researchers have already shown that it is possible to gain control of a vehicle, both through local or remote communication, and alter its behavior [11, 15, 20, 27, 31]. The effects of such attacks range from simply changing the audio track played on the radio, to very serious consequences like disabling braking systems or stopping the engine. Since Miller and Valasek's demonstration of being able to control remotely a Jeep Cherokee in 2015 [20], the attention toward automotive security has increased.

Regarding CAN, one of its main weaknesses has proven to be the lack of message authentication. Since nodes in the network do not validate the origin of a message, an attacker can send spoofed CAN messages that receiving nodes are not able to distinguish

Unpublished working draft. Not for distribution.

from authentic ones. Therefore, after an attacker exploits an externally communicating ECU, he or she can proceed to send messages through this ECU to all nodes connected on the same CAN bus and exploit them through forged CAN frames. However, multiple ECUs nowadays check received messages to control whether two different nodes are sending two streams of information with the same identifier, and react by not accepting any of the data received. For this reason, a common approach, after the attacker has made his way inside the network, is to cut off the target ECU from bus communication. To achieve this, a known mechanism, proven possible by Palanca et al. in [22], takes advantage of the error handling protocol of CAN to convince the target ECU to shut itself off the network. At this point the attacker is free to send the forged and spoofed messages without handling the stream of frames sent by the victim ECU. The capabilities of the attacker, once he lays undisturbed on the network, must be of concern. In fact, CAN, which ensures real-time message transmission, is the most commonly used communication bus for ECUs whose function affects driving and, consequently, the safety of people in and around it.

Multiple measures have been proposed to either stop attackers from getting access to CAN networks or to deny their messages to be accepted by the receiving ECUs: they space from segmenting networks to make the access to critical subnetworks more complex, to authentication protocols, to IDSs. All these countermeasures have their pros and cons. For example, authentication protocols usually require significant data and computation overheads or modifications to the hardware of all concerned ECUs, rule-based IDSs can not recognize most powerful attacks, and machine-learning-based IDSs do not have the certainty of not incurring into false positives. Therefore, the automotive environment is still far from being secure and new countermeasures are still required to defend from all those attacks that are not yet covered.

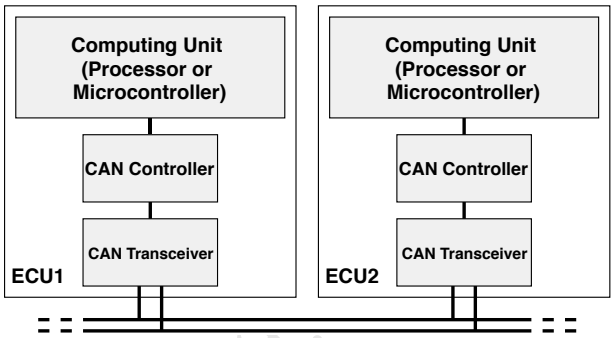
In this paper we present CopyCAN, an error-handling protocol-based intrusion detection system for Controller Area Network (CAN). CopyCAN is able to detect when any ECU has been cut off the CAN bus and to flag every further attempt to transmit messages spoofed from the disconnected ECUs as attacks. Furthermore, CopyCAN requires only to add its monitoring unit to the existing network, making it easily deployable in current systems without the need of modifying other nodes. Finally, since CopyCAN requires a non-standard CAN transceiver, we discuss the possible reactions that can be implemented through it after an attack is detected.

In detail, we make the following contributions:

- We describe an easily deployable IDS which detects if any ECU has been cut off from the communication bus.
- We demonstrate the feasibility of our work by implementing a proof-of-concept testbed.
- We test the performances of our IDS and suggest future works to improve them.
- We analyze the possible reactions that can be implemented once the attack is detected.

The paper is structured as follows: In Section 2 we describe the information regarding CAN required to understand our approach. In Section 3 we describe the related works and subsequentially in Section 4 the threat model of our methodology. In Section 5 we describe our approach. In Section 6 we present the implementation our intrusion detection system and how focusing on its feasibility and performances. Finally, in Section 7 we discuss the possible reactions that can be implemented once our IDS detects an attack. In Section 8 we present our conclusions, the limitations of our approach and suggest some potential future works.

Figure 1: Representation of the basic connections amongst ECU components and CAN bus



2 BACKGROUND CAN PROTOCOL

Since our approach directly relies on the inner workings of CAN, we deem that an overview of its functions is necessary. We will focus on the aspects directly related to our works. For additional details we refer to the official CAN specification [12].

The Controller Area Network (CAN), developed by Robert Bosch GmbH [12], is a serial communications protocol that efficiently supports distributed real-time control between vehicle's Electronic Control Units (ECUs). A typical ECU, shown in Figure 1, consists of a CAN transceiver to transform physical signals in logic values, a CAN controller, generally implemented in hardware, to enforce the protocol and a computing unit, which is usually a micro-controller running custom firmware and software. CAN is a carrier-sense, multiple-access protocol with collision detection and arbitration on message priority (CSMA/CD+AMP). This implies that each node has to check if the bus is free before trying to transmit data, otherwise it has to wait for a specific signal before trying again. Collision detection and arbitration on message priority is implemented so that when a node starts transmitting data it still checks for collisions during the arbitration phase, and when a collision happens only the message with the highest priority keeps being transmitted.

2.1 Message Transmission

CAN messages are transmitted by sending different voltages between the two wires of a twisted pair cable: when the voltage difference amongst the two wires is "high" (i.e., usually between 3 and 5 volts), the value is defined as "dominant" and is usually translated into a binary 0, while when the difference is "low" (usually close to 0 volts), it is defined as "recessive" and translated into a binary 1. In this way, dominant values overwrite recessive ones: this is called zero-dominance property. Although CAN does not require a clock, it is a synchronous protocol in which time is split into bit-time slots. Therefore, when two ECUs try to transmit a

Unpublished working draft. Not for distribution.

Table 1: Description of the fields of CAN data and remote frames. Length distinguishes between 11/29-bit fields when necessary.

Field	Length	Description
SoF: Start-of-Frame	1	Single dominant bit which signals the start of a message
ID: Identifier	11	Unique Packet Identifier
RTR (11-bit): Remote Transmission Request SRR (29-bit): Substitute Remote Request	1/0 0/1	Dominant for 11-bit data frames, Recessive for 11-bit remote frames Must be recessive.
IDE: Identifier Extension Bit	1	Dominant for 11-bit data frames, recessive for 29-bit ones
ID-extended	0/18	Extended ID field for 29-bit data frames
RTR (29-bit)	0/1	RTR for 29-bit frames: dominant for data frames, recessive for remote ones
RB0,1: Reserved Bits	1/2	Reserved dominant bits for possible future expansions
DLC: Data Length Code	4	Number of data bytes
Data Field	0-64	Data transmitted by data frames
CRC: Cyclic Redundancy Check	15	Check for data sanity
CRC Delimiter	1	Must be recessive
ACK: Acknowledgment slot	1	Sent by transmitter as recessive, the receiver overwrites it as dominant if the message is error-free
ACK delimiter	1	Must be recessive
EoF: End of Frame	7	Sequence of recessive bits

dominant bit and a recessive one in the same slot, all receiving nodes listening on the bus consider only the dominant value.

The CAN protocol includes four types of messages, called frames: data, remote, error, and overload frames. As explained later, only data and remote frames require arbitration. For this reason, each data and remote frame is identified by a message ID which is either 11 or 29 bits long, depending on the employed CAN format (Standard or Extended). When two ECUs start to transmit in the same bit-time slot, an arbitration procedure takes place, where the message ID defines its priority: thanks to the zero-dominance property explained above, 0 bits are considered dominant over 1 bits, hence messages with numerically smaller IDs have a higher priority. For the mechanisms explained above, CAN requires a careful design of the network nodes and of the sent IDs. In fact, although there is not any kind of enforcement of this rule, in CAN there cannot be two different nodes sending messages with the same ID, otherwise the arbitration phase could be resolved with more than one node still writing on the bus. We proceed to explain the four kinds of frames available in CAN.

Data and Remote Frames. As visible in Table 1, are composed of many fields. The first ones, both in the standard and extended formats, mainly regard the arbitration phase. Then, in data frames we find the field related to the data payload. Finally, in both data and remote frames there is the Acknowledgment (ACK) slot, which is sent as recessive (1) by the transmitter, and has to be overwritten with a dominant bit (0) by one of the receivers to validate the message. All the fields described in the table, except the Cyclic Redundancy Check (CRC) Delimiter, the ACK field, and the End Of Frame (which have a fixed-form), are coded with the method of bit stuffing: whenever a transmitting node detects five consecutive bits with the same logical value to be transmitted, it automatically inserts a complementary bit in the actual transmitted bitstream. This complementary bit is also recognized from receiving nodes, and, hence, discarded as not being part of the original payload. Data frames are usually autonomously sent on a fixed time interval, although other nodes can request them by sending remote frames.

Error Frames. They consist of two fields: error flag and error delimiter. Usually the transmission of one error flag leads to a superposition of error flags, followed by the error delimiter. *Error Flag* It can be Active, consisting of six consecutive dominant bits, or Passive, consisting of six consecutive recessive bits. An 'error active' node detecting an error condition signals this by transmitting an active error flag. The error flag's form violates the rule of bit stuffing explained above: as a consequence, all other nodes detect an error condition and start transmitting their own error flag. Therefore the sequence of dominant bits which can be monitored on the bus results from a superposition of different error flags, leading to a sequence long between a minimum of six and a maximum of twelve dominant bits. An 'error passive' node detecting an error condition signals it by transmitting a passive error flag. The 'error passive' node then waits for six consecutive bits of equal polarity: the passive error flag is complete when these six equal bits have been detected. *Error Delimiter* This field consists of eight recessive bits and signals the return to normal bus communication.

Overload Frames. They are obsolete and rarely used. They are sent to delay the transmission of the next data or remote frame, mainly due to nodes requiring more time to compute previous frames. They consist of the overload flag (six dominant bits, like the active error flag) and the overload delimiter (eight recessive bits, like the error delimiter). Overload frames, despite having the same structure of error frames, are not transmitted due to the detection of an error but are sent only in Interframe Space (IFS) (explained in the next paragraph) to delay transmission. Since in the IFS the method of bitstuffing is not implemented they do not raise errors from other nodes.

Finally, data and remote frames are separated from preceding ones, whatever types they were, by the Interframe Space (IFS). This field contains a minimum of three recessive bits (Intermission) plus all the recessive bits representing the bus idle condition, where no

Unpublished working draft. Not for distribution.

node is trying to transmit and the bus is free, ready for the next dominant Start-of-Frame. Overload frames can be transmitted only during the Intermission field.

2.2 Error Detection

As explained above, when a node detects any error during the transmission, it signals it by transmitting an error frame. This leads also the other nodes on the bus to notice that an error occurred and to transmit their own error frame. Here the different types of error (not mutually exclusive) that can occur:

- *Bit error*, raised if a node sending a bit on the bus reads a different bit value than the one being sent. The only exception is when the transmitting node sends a passive error flag and it detects a dominant bit.
- *Stuff error*, raised if six consecutive bits with the same polarity are detected in a message field that should be coded with the method of bit stuffing.
- *CRC error*, raised if the CRC calculated by a receiving node is different from the CRC transmitted in the frame.
- *Form error*, raised when a fixed-form bit field contains one or more illegal bits.
- *ACK error*, raised by a transmitting node when it does not monitor a dominant bit in the ACK slot.

Whenever a bit, stuff, form or ACK error is detected by any node, the transmission of an error flag is started by the respective node at the next bit time. Whenever a CRC error is detected, the transmission of an error flag starts at the bit time following the ACK delimiter. Finally, after a corrupted frame has been detected, such a frame is automatically retransmitted as soon as the bus is idle again, according to arbitration.

2.3 Fault Confinement

In order to handle faulty devices, a node on a CAN bus can be in one of the following three states: 'error active', 'error passive' or 'bus off'. An 'error active' node can normally take part in bus communication and sends an active error flag when it detects an error. An 'error passive' node takes part in bus communication but when it detects an error it sends a passive error flag, which is detected and echoed by other nodes only if the error passive node had already won arbitration at the time when the error occurred. Moreover, after a transmission, the 'error passive' node has to wait eight bit time slots before initiating another transmission. Finally, a 'bus off' node is not allowed to participate in bus communication.

To define in which state a node (ECU) is, each node keeps track of its own two error counters: Transmit Error Count (TEC) and Receive Error Count (REC). These counters are modified according to the following rules (as taken from [12]):

(1) When a receiving node detects an error, its REC is increased by 1, except when the detected error is a bit error during the transmission of an active error flag or an overload flag.
(2) When a receiving node detects a dominant bit as the first bit after sending of an error flag, its REC is increased by 8.
(3) When a transmitting node sends an error flag, its TEC is increased by 8. However there are two exceptions in which the TEC is not changed: if the transmitter is 'error passive' and detects an ACK Error and does not detect a dominant bit while sending a passive error flag, or if the transmitter sends an error flag because of a stuff error that occurred during arbitration whereby the stuff bit is located before the RTR bit, and has been sent recessive but monitored as dominant.
(4) If a transmitting node detects a Bit Error while sending an Active Error Flag or an Overload Flag, its TEC is increased by 8.
(5) If a receiving node detects a Bit Error while sending an Active Error Flag or an Overload Flag, its REC is increased by 8.
(6) Any node tolerates up to seven consecutive dominant bits after sending an Error Flag or Overload Flag. After detecting the 14th consecutive dominant bit (in case of an Active Error Flag or an Overload Flag) or after detecting the 8th consecutive dominant bit following a Passive Error Flag, and after each sequence of additional eight consecutive dominant bits, every transmitter increases its TEC by 8 and every receiver increases its REC by 8.
(7) After the successful transmission of a message (getting the ACK and monitoring no error until End Of Frame is finished) the TEC is decreased by 1 unless it was already 0.
(8) After the successful reception of a message (reception without error up to the ACK Slot and the successful sending of the ACK bit), the REC is decreased by 1, if it was between 1 and 127. If REC was 0, it stays 0, and if it was greater than 127, it is set to a value between 119 and 127.
(9) A node is 'error passive' when its TEC or REC equals or exceeds 128. An error condition letting a node become 'error passive' causes the node to send an Active Error Flag.
(10) A node is 'bus off' when its TEC is greater than or equal to 256.
(11) An 'error passive' node becomes 'error active' again when both its TEC and REC are less than or equal to 127.
(12) A 'bus off' node is permitted to become 'error active' (with its error counters both set to 0) after 128 occurrences of 11 consecutive recessive bits monitored on the bus.

3 RELATED WORKS

CAN has been implemented on vehicles for around thirty years. During this period lots of different countermeasures have been proposed to solve its considerable security flaws: these countermeasures range from the simple insertion of secure gateways to divide the on-board network into subnetworks [34], up to the more complicated implementation of honeypots [32]. There have also been proposals to replace CAN with network protocols on which security is more easily implemented [22, 30], such as automotive Ethernet [13]. However, the real-time properties and the lower costs of CAN and its successor Controller Area Network with Flexible Data-rate (CAN-FD) make them mandatory in some subnetworks.

In the latest years the trend regarding CAN-related security have been mostly focused on two main countermeasure categories: authentication protocols and Intrusion Detection Systems (IDSs).

Regarding **authentication protocols**, the general idea relies on using part of the CAN packet to transmit a hash of the message, encrypted through a secret key, alongside a counter (to defend from replay attacks). Although some state-of-the-art proposals like LeiA [24], VatiCAN [3] and VeCure [33] strengthen the security of

Unpublished working draft. Not for distribution.

CAN, their downsides are still relevant: the decreases in response times and data bandwidth, generated respectively by the calculation and the transmission of the hash, make the usage of authentication protocols an unfeasible solution in most cases. In fact, their implementation is hardly feasible on less powerful ECUs, such as sensors, due to the computation requirements, while busy networks with many ECUs would suffer from the bandwidth overheads.

Regarding **Intrusion Detection Systems (IDSs)**, instead, the general idea relies on reading the bus and detecting whether an attack is being performed. We can divide IDSs for CAN in three categories, depending on the approach they apply to detect attacks. *Frequency-based* IDSs, such as [21, 28], take advantage of the mainly periodic trend of CAN communication (ECUs send messages with the same ID at regular intervals) and detect abnormal behaviors when the frequency of a message changes unexpectedly. *Specification-based* IDSs comprise all those IDSs that detect inconsistencies of data through the analysis of a set of given rules (all the traditional rule-based IDS are included in this category). Two of the latest and less trivial examples are Parrot [9] and the new STINGER CAN transceiver [2] from NXP [1]. Their basic functioning is based on the knowledge of which CAN IDs are transmitted by the ECU they are installed on (as explained in Section 2 each ID is sent by only one CAN node for each network). When one of such IDs is transmitted on the network, if it was not sent by the ECU on which Parrot or STINGER are installed on, they flag it as an attack and react accordingly. Finally, *data-sequence-based* IDSs comprise all those IDSs that detect intrusions by analyzing the changes in the data payload of messages through time (e.g., Markovitz and Wool [17], Taylor et al. [29]) IDSs have fewer downsides compared to authentication protocols but are also less effective. In fact, the final goal of an IDS is only to recognize attacks, and not to prevent them as authentication protocols do. Moreover, the IDS is unaware of the aftermaths of the detection and requires to be installed alongside a "reactional" component in order to be effective. In cyber-physical systems, such as the automotive one, this characteristic of IDSs makes them less suitable to be implemented: in fact, among the proposed IDSs, many of them do not ensure the absence of false positives (especially among those IDS which implement machine learning algorithms). This prevents from being able to react by shutting down the node which is sending the messaged flagged as malicious, since it may lead to the unnecessary lowering of safety features of the vehicle. Furthermore, multiple attacks can often be implemented without surpassing the bounds of preset rules (both frequency or data related), since such rules have to be valid in many vehicular environments.

Our solution targets a specific property of CAN, the Transmit Error Count (TEC) of the controller, which at the best of our knowledge is rarely used as a countermeasure. In fact, the only IDS that works at transceiver level and exploits the TEC, is the one implemented in STINGER [2]. However, STINGER only defends the ECU on which it is installed, while CopyCAN watches over the whole bus. Moreover, CopyCAN cannot be bypassed since it is able to monitor the TEC of the victim and does not require any knowledge about the malicious ECU, making it unfeasible for the attacker to fake or modify the counter even knowing about the implementation of CopyCAN on the bus.

4 THREAT MODEL

The possibility of a remote attacker in automotive networks was not considered when CAN was designed, hence the protocol has not been designed by taking into account security. Furthermore, the necessity for a cost-effective and real-time protocol led to the choice of having short data frames (maximum 8 bytes of payload) and medium bandwidth (in the best case up to 1Mbps), which do not leave a lot of space for implementing security features at a later time. Over the years, with the proliferation of wireless technologies such as Bluetooth, Wi-Fi, and Long Term Evolution (LTE), some ECUs started to be designed with external communication capabilities and are now reachable from remote. However, these ECUs are still connected to all other ECUs inside the vehicle through wired on-board networks. Nowadays, on-board communication does not consist of a single or two CAN networks on which all ECUs are connected, as was the standard 20 years ago. In fact, secure gateways [34] divide it into various sub-networks that limit the capabilities of the attackers.

4.1 Real World Attacks

Researchers implemented many attacks in the last ten years to demonstrate on one side the capabilities that an attacker has once getting access to CAN networks, and on the other side the lack of security in the design of marketed vehicles. Initially, in the early 2010s, Koscher et al. [16] and Checkoway et al. [7] proved first that, by connecting physically to the CAN networks of the vehicle, they could take control of it in some situations, and second that some early stage attack surfaces, such as a CD player or a Bluetooth device, were exploitable if already connected to such CAN networks. Some years later Miller and Valasek published three papers on automotive security: in the first [31] they made an in-depth analysis of the capabilities of an attacker with CAN bus access, proving that more equipped vehicles, with more safety and comfort accessories, were more vulnerable to attacks. In the second [19] they analyzed the topology of multiple vehicles, some newer than others, discussing the security of each one. Finally, in the third [20] they demonstrated an attack performed completely from remote, through the cellular connection, on a one-year-old vehicle, on which they managed to force cyber-physical controls such as park-assist-related steer, brake and acceleration. For this third attack, they required to exploit two ECUs in order to obtain the capability to write on the correct CAN bus. Once on the bus, they were able to perform all the different attacks by spoofing CAN messages, without requiring to control directly the ECU that performed the brake, steer or acceleration. After these works, different researchers proposed similar real-world attacks implementable from remote on newer cars such as on a Tesla S [27] and BMWs [15], demonstrating that although manufacturers have implemented some security features in vehicles, a skilled attacker can still gain access and take control of the vehicle.

Finally, all the attacks presented up to now consist mainly on spoofing the messages of other ECUs. The last real world attack we present, implemented by Palanca et al. [22], shows the possibility for the attacker to target one ECU and disconnect it from the CAN bus through a targeted Denial of Service (DoS) attack.

Unpublished working draft. Not for distribution.

4.2 Attacker Capabilities

We consider an attacker who has already obtained control of (or installed) an ECU on the CAN bus. The attacker has complete knowledge of the system and the functionalities of each ECU on the network, and he also knows which CAN messages are required to control each functionality. The attacker can either be connected from remote or from inside the vehicle, and he is aware of CopyCAN being installed on the network. The only requirement that we have is that the attacker does not have physical access to the ECU on which our IDS is installed. However, in all literature regarding CAN related countermeasures, this is considered an assumption since if the attacker has such capabilities, he has non strictly "cyber-related" methods to achieve his goals [7].

4.3 Attack Model

After explaining the attacks that have been proven to be feasible throughout the years and the strengths of our attacker, we describe how we can classify attacks to CAN.

Sniffing. Sniffing is not usually considered a threat. Since the nodes are connected to the bus network, reading the messages from and to other nodes just requires to be physically connected. Furthermore, the authentication protocols mentioned in Section 3 does not encrypt the whole message. The reason for this is dual: encryption would require an even higher computation time, and usually CAN messages are not interesting to be sniffed for an attacker since they do not carry data meaningful to steal.

Denial of Service (DoS). DoS is instead usually considered a threat, but not to the safety of the people in and around the vehicle. In fact, since ECUs may fail due to malfunctions while the vehicle is on the road, the whole system has been designed to ensure safety anyway. However many features, comprising the possibility to drive the vehicle, could be compromised due to the malfunctions. For this reason, an attacker with financial goals or that targets the reputation of the manufacturer may still have interest in DoS attacks.

To implement a DoS the attacker has two options: either he floods the CAN bus with packets with ID "0" to win all arbitrations, although this approach is easily detectable by an IDS, or he exploits the same characteristic of CAN exploited by Palanca et al. [22], creating a so-called **targeted DoS attack**:

This attack takes advantage of the rules for fault confinement provided by the CAN protocol. Essentially, apart from rare exceptions (as explained in Section 2.3), when a transmitting node sends an error flag its TEC is increased by 8: this means that after 16 invalid transmissions an 'error active' node with $TEC = 0$ will become 'error passive' ($TEC = 128$), and after 16 more invalid transmissions it will go in 'bus off' state ($TEC = 256$), disconnecting itself from the bus. The goal of the attacker is therefore to convince the target node of being defective by triggering victim's fault confinement protocol enough times.

There are mainly two ways to implement this attack: the first one, presented by Palanca et al. [22] and represented visually in Figure 2, requires a non-standard CAN controller and can be divided in two phases: The attacker first has to detect the ID of one of the frames sent by the target ECU. Second, the attacker waits for the victim to win an arbitration, then waits for the transmission

Figure 2: Representation of Targeted DoS as in [22]

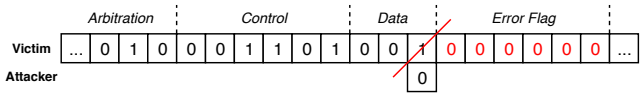
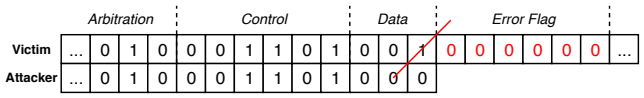


Figure 3: Representation of frequency-based targeted DoS



of the payload: at this point he overwrites one of the recessive bits of the victim's message with a dominant one. This causes the detection of a bit error by the victim, signaled by the transmission of an error flag and a subsequent new attempt to transmit the frame. The TEC of the victim increases by 8 for each time the frame is overwritten. Therefore, the attacker only needs to perform 32 straight bit overwrites of a frame sent by the victim node in order to block that node in 'bus off' state.

This attack is not counterable by the victim. However, given the requirement of a physical modification of the CAN controller of the attacking device, this attack is feasible only if the attacker had previous physical access to the vehicle. For this reason, in the literature it is usually considered a threat mainly for car-sharing services, in which case the attacker has physical access to multiple vehicles in a relatively short amount of time, while being less dangerous for traditionally owned cars.

On the other hand, the second kind of targeted DoS is not as reliable as the first one, but does not necessarily require a modified CAN controller, which makes it much more feasible to implement in remote attacks.

Figure 3 shows how this attack works: In this case the attacker requires to detect the ID of one of the frames sent by the victim, and the frequency at which messages with that ID are transmitted. Obtained this information, the attacker sends a message with the same ID but with an 8 bytes long sequence of "0" as payload, synchronized with the message of the victim. If the attack is timed precisely the ID sent by the victim and the one sent by the attacker overlap, hence convincing both the victim and the CAN controller of the attacker (which has not been modified) that they won arbitration. However, since the data payload of the attacker is composed only of dominant bits, and making the suitable assumption that the victim is sending meaningful data and not only dominant bits as the attacker does, at least one recessive bit of the victim is going to be overwritten by the attacker, hence triggering its fault confinement protocol and sending an error active flag on the bus (which triggers also the fault confinement protocol of the attacker, increasing its TEC of 8 points). Both ECUs repeatedly try to send their own message, not realizing that they are being overwritten by the other and both repeatedly increase their TEC. However, as explained in [9], when both switch to "error passive" state, the victim can only send error passive flags, which are not detected by the attacker who keeps transmitting successfully his frames, hence only the victim increases its TEC until switching to "bus off" state.

Unpublished working draft. Not for distribution.

The unreliability of such an attack comes from the unpredictability of the precise instant in which the victim sends the first bit of its message. If the attacker anticipates or delays the dispatch of its frame of a "bit time slot," the attack fails.

Spoofing. Finally, spoofing attacks are the most threatening ones. Amongst the real-world attacks explained in Section 4.1, all those that affect the safety of people in and around the vehicle comprise the use of spoofed CAN messages.

Spoofing attacks are easier than targeted DoS to implement, since the only requirements for the attacker are to know the ID of the frame that he wants to spoof and to know how the data is encoded in the payload (i.e., how the receiving ECU translates the bits of the payload into meaningful information). After the attacker obtains this knowledge, whether it is public or through reverse engineering of a similar vehicle, he can proceed to send the packet to the receiving ECU, which will consider it as being transmitted from the spoofed victim.

However, through the years countermeasures have been applied to protect the network from such attacks. There are three countermeasures that the attacker needs to bypass to spoof the message: (a) the attacker needs not to cross the boundaries of rule-based IDS. To obtain this goal he just requires to know the rules and respect them when implementing the attack: in fact, as mentioned in Section 3, not all attacks require to cross the boundaries set by rule-based IDS. (b) The attacker needs to ensure that the receiving ECU considers his data over the ones of the authentic transmitting node. In order to do so, Miller and Valasek [31] increased their frequency of transmission: however, this may nowadays trigger frequency-based IDS. (c) The attacker has to avoid to be detected by Parrot [9] or similar countermeasures.

However (a) is feasible to bypass, knowing the rules of the IDS, and in order to take care of (b) and (c) the attacker can implement a targeted DoS against the transmitting ECU of which he wants to spoof the messages.

As we explained up until now, this is the most threatening kind of attack to automotive networks, since it threatens the safety of people in and around the vehicle. The implementation of the attack through a previous DoS is also currently the only one that cannot be detected.

CopyCAN focuses on detecting this implementation of the attack by knowing which ECUs should be in "bus off" state.

5 APPROACH

The final goal of CopyCAN is to detect when any ECU has been disconnected from the network through the exploitation of the fault confinement mechanism of CAN, as explained in Section 4.3. Specifically, CopyCAN keeps a copy of the Transmit Error Count (TEC) of the protected ECU. In fact, the core concept behind our approach is that, since the attacker abuses of the fault confinement protocol of CAN to shut down the victim ECU, we can exploit the same protocol to detect when said ECU switches to "bus off" state.

We proceed to explain in detail the reasoning behind CopyCAN's behavior: first we explain the basic assumptions required for its functioning, then we proceed to explain how the rules for modifying the TEC are implemented. Finally, we describe its model through the use of an "extended finite state machine". To conclude the description of our approach, we discuss potential reactions to the detection of an attack.

5.1 Assumptions and Physical Placement

CopyCAN does not use a standard CAN controller. In fact, since it reads every single bit transmitted on the bus, it requires to retrieve the data directly from the CAN transceiver.

CopyCAN requires to be on the same physical network of the protected ECUs and the messages cannot be transmitted through a gateway since this would not relay the CAN errors, making it impossible to detect when the victims switches to "bus off" state. There are no theoretical limitations to the number of ECUs simultaneously analyzed by our IDS.

We assume that CopyCAN knows all the IDs "owned" by each protected ECU (i.e., all the IDs of messages that each ECU transmits) since it needs to understand, while an error occurs, whose TEC should be incremented. However, CopyCAN does not require to know any other information about the victim.

Lastly, we assume CopyCAN to be listening on the network since the moment in which the first packet is sent, since otherwise it may not be aware of previous events that triggered a TEC change in one of the protected ECUs.

Similar assumptions are also done by the majority of other state-of-the-art IDSs for CAN.

5.2 Fault Confinement Rules Analysis

CopyCAN detects the increment or decrement in the Transmit Error Count (TEC) of the protected ECU through the analysis of the fault confinement rules listed in Section 2.3. We refer to them to explain how we implemented them.

As we already mentioned, our goal is to know when the defended ECU gets disconnected from the network, switching to "bus off" state. Rule (10) explains that this happens only when the TEC of the ECU reaches 256. Therefore we are only interested in counting the TEC, and not the REC, of the protected ECU. For this reason, rules (1),(2),(5) and (8) are not necessary. Since the IDS does not require to know when an ECU goes in "error passive"state, rules (9) and (11) are also not of our concern.

The basic definition of (3) ("when a transmitting node sends an error flag, its TEC is increased by 8") can be implemented in our IDS by increasing the TEC of the protected ECU if it is the current transmitter and if we monitor an error flag during its transmission.

The first exception of (3) regards a corner case where a transmitter is error passive and it detects an ACK error. In this case, if no other node sends an active error flag, the transmitter does not increase its TEC. This is easily implemented since, from the point of view of the bus, no error flag is detected: in fact the passive error flag consisting of six recessive bits is not recognized as an error by other nodes since the bit stuffing rule has been deactivated since the CRC delimiter. Therefore, we just avoid increasing the TEC of the transmitter already when we detect the missed ACK and wait for the subsequent flag instead.

The second exception regards events happening during arbitration, which are not of our concern since they are not exploitable by the attacker (i.e., the attacker cannot know yet whether its target is writing on the bus).

Unpublished working draft. Not for distribution.

(4) represents the corner case in which the CAN transceiver of the protected ECU fails in a specific way: reading a bit that should be dominant as recessive during the transmission of an active error flag. This event is not visible by our IDS, since there is no way to detect it on the bus. However, since the bit trasmitted on the bus is dominant, the attacker cannot exploit this to bypass CopyCAN. Also, as explained later in the implementation in Section 6.2, we never found occurrences of this corner case in our tests.

(6) represents another corner case: the seven dominant bits are composed by the active error flag of other nodes (*stuff* error caused by the first error flag composed of six equal bits) and one more dominant bit (which we assume to be comprised in the count just to double-check the failure of an ECU, since there seems to be no justification for it in the official CAN specification [12]) .

After these seven dominant bits the node expects the error delimiter (eight recessive bits). In case this does not happen, this means that an error occurred and an ECU is flooding the network with "0," or the node is reading incorrectly from the bus. In order to try to disconnect the faulty node from the network, the TEC of all transmitters is increased by 8 for every time they read eight consecutive dominant bits after they send an error flag.

We are interested in this rule only if it is triggered when a protected ECU is transmitting. Since our IDS is only aware of the transmissions on the bus, and it does not know the error type that triggered the first error flag, this rule creates ambiguity. In fact, in a sequence of "0," we are unaware of when the error flag of the protected ECU has started: the first bit of the error flag could be from the first of the sequence up to the seventh. If it is the first, the protected ECU will consider the fourteenth "0" as a trigger of rule (6). If, instead, the seventh "0" bit of the sequence is the first one of the error flag of the protected ECU, the rule will be triggered only at the twentieth consecutive "0".

Is it important to notice, although, that in a functioning network only 12 consecutive "0" can appear consecutively: in fact after six consecutive "0," independently from the reason behind them being transmitted, every error active node should detect a *stuff* error and start sending its own flag. After this event, every node on the network should be waiting for eight recessive bits (error delimiter) monitored on the bus. Therefore, there is no reason to detect another "0" after the two sequences of six dominant bits justified by the error flags. In fact, the chances for this faulty behavior to repeat itself enough times to trigger the IDS into considering the protected ECU in "bus off" state, without it actually being in that state, are considerable zero in real-world scenarios, and this is supported by our tests, since this behavior was never triggered.

On the other hand, waiting for the twentieth bit would enable an extremely skilled attacker to trigger the "bus off" state of the protected ECU without being detected by the IDS. For this reasons, we choose to count the increasing by 8 of the TEC after the thirteenth bit detected on the bus and each eight consecutive dominant bits after that.

(7) is implemented by decreasing the TEC every time the protected (ECU successfully transmits a message.

(12) is implemented by resetting the TEC of the protected ECU after 128 occurrences of 11 consecutive recessive bits on the bus. Although it may be possible for the protected ECU to "lose" one of said occurrences, the attacker has no capability of controlling this event.

5.3 Algorithm Modelization

As mentioned before, CopyCAN requires to read the bus bit by bit, retrieving data directly from the CAN transceiver. We proceed to explain the algorithm behind CopyCAN, represented through an extended finite state machine in Figure 4, which explains how we parse messages and how we update the counters of the protected ECUs. For the sake of comprehensibility, we describe the algorithm required to parse frames with 11-bit IDs, although it can be easily be adapted to parse frames with 29-bit IDs by following Table 1.

In order to handle the parsing of the frame we require a set of variables that we proceed to list: **BC** (Bit Counter), used to check which frame's field we are processing: this counter is increased by 1 whenever we read a bit. **PC** (Polarity Counter), used to keep track of how many subsequent bits of the same polarity we read: in this way we can both handle *stuff* errors and update the bit counter after bit stuffing takes place. This counter is increased by 1 if the current monitored bit is identical to the previous one, reset to zero otherwise. **STUFF** is used to store the number of stuffing bits inserted during a frame transmission: after we read five subsequent bits with the same polarity, we increase this variable by 1. **DL** is used to store the number of data bytes of the frame. **formErr**, which can assume 0 or 1 values, is used to handle form errors, as explained later in details. Finally, we use **TEC** to keep track of the TEC of the protected ECU.

We divide the analysis in error-free parsing, which describes the situation in which all ECU behave correctly and no error is detected, and error handling, in which we describe what CopyCAN does in case an error occurs.

Error-free parsing. The process starts in idle state during the ignition of the vehicle, therefore knowing that the TEC of each ECU is currently equal to 0. All variables mentioned above are also initialized to 0. As long as we read recessive bits, the bus is in idle state. Once we monitor a dominant bit, representing the Start of Frame (SoF), we move to the SoF state. Since from now on bit stuffing is implemented, we start updating the *Polarity Counter (PC)* as explained above.

The algorithm can now process the ID that is being written on the bus and store it in order to handle the TEC of the corresponding ECU. The ID is composed by the second to twelfth bits of the packet, unless bit stuffing is required, in which case we need to add some bits and update properly the bit counter (e.g., in case of ID 0x16, which is transmitted as 00000010000, we expect to read 000001010000).

Then we memorize the next bit, which is the Remote Transmission Request (RTR) bit as expected by the protocol, to distinguish between data and remote frames.

Now, in the case of 11-bit ID frames, the Identifier Extension (IDE) bit and reserved bit are both expected to be dominant. Note that, when considering 29-bit ID frames, also the IDE should be stored to handle the distinctions between the two different employed standards.

Once we parse the Data-Length Code (DLC) and store its value, we either move to CRC state in case the RTR bit was recessive,

Unpublished working draft. Not for distribution.

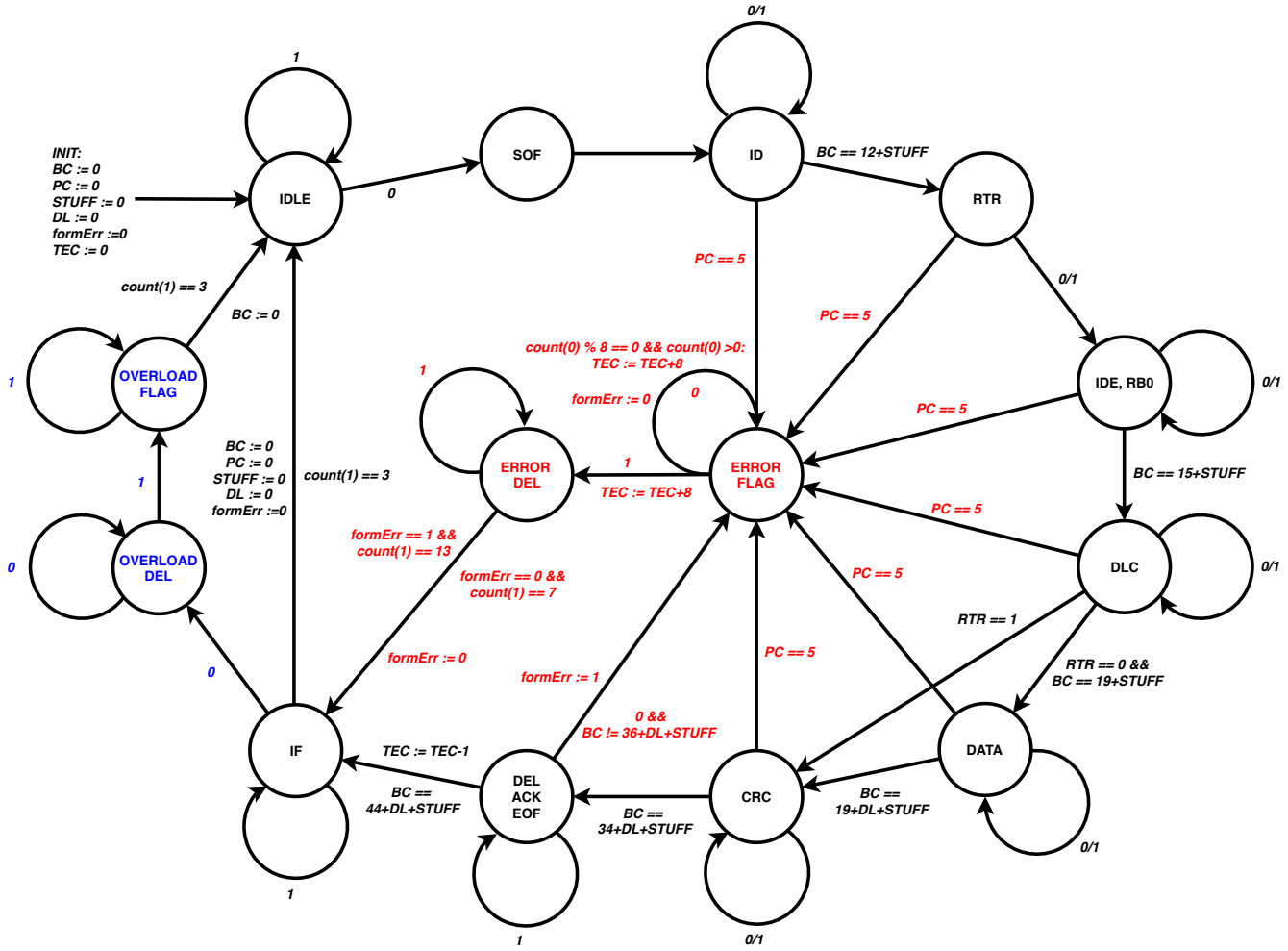


Figure 4: Graphical representation of the state machine model. In black, the transitions to error free states. In red, transitions to error related states. In blue, transitions to overload related states. When only a number is used to describe the transition between two states (arrow), we consider it to be the bit monitored on the bus.

meaning that we are parsing a remote frame, or we read the data payload in case the RTR bit was dominant (data frame): in this last case we read $8 \times DL$ bits plus potential bit stuffing.

After we parse the CRC, the polarity counter is deactivated since the bit stuffing rule is not applied from now on.

We proceed to read the last ten bits of the frame: we expect them to be all recessive except the ACK slot in case of a positive acknowledge of the frame. If no error occurs we move directly from DEL/ACK/EoF state to Intermission Field (IF) state: during this change of state we decrease by 1 the *TEC* of the protected ECU since we parsed an error-free frame.

Finally, after three more recessive bits representing the IF, the algorithm moves back to idle state, waiting for the next transmission.

Moreover, the algorithm handles the transmission of overload frames, signaled by a dominant bit monitored during IF state. In this case we process up to twelve consecutive dominant bits, due to the propagation of the overload flag, followed by eight consecutive recessive bits, representing the overload delimiter. After this we move back to idle state, as expected by the protocol.

Error Handling. If at any moment during the transmission the polarity counter signals that six consecutive bits with the same polarity have been read ($PC == 5$, which is feasible only since we read the ID until we read the CRC, since the *PC* is deactivated after the latter field), the algorithm switches to a state that handles the error flag. As explained in Section 5.2, the error flag should be from six to twelve bits long due to error propagation. If it is longer, the exception of rule (6) is triggered. Once the algorithm detects a recessive bit, it switches to error delimiter state. Both when the exception is raised both when the algorithm switches to error delimiter state, the *TEC* of the protected ECU is increased by 8. After eight consecutive recessive bits, representing the error delimiter, we go back to IF state as expected by the protocol.

The only error case that is not taken into consideration by the previous procedure happens when, during the last twelve bits (CRC delimiter, ACK slot, ACK delimiter, End of Frame (EoF)), a *form error* arises. In this case, even if an error happens we may not detect an error flag: since the bit stuffing rule is currently not applied, if all the ECUs on the network are in error passive state, even if they detect the error, they will write six consecutive recessive bits on the bus, which is the same sequence of bits that we detect in an error-free transmission. In order to handle this event, if we detect a dominant bit among these fields (except for the ACK slot, where the dominant bit represents an ACK), we set *formErr* to 1 and move to the error flag state. Here we expect one of two situations: either we monitor six consecutive dominant bits, representing the active error flag of the transmitting ECU, followed by the eight recessive bits of the error delimiter, or we monitor fourteen consecutive recessive bits in case of a passive error flag plus its delimiter. Through *formErr* we can handle properly both situations: after reading the first bit of the error flag, if this is dominant we set the variable back to zero to signal that we expect only eight consecutive recessive bits once moved in error delimiter state, before moving to IF state. If instead the first bit is recessive, we move immediately to error delimiter state with *formErr* equal to one, signaling that we expect 14 consecutive recessive bits before moving to IF state. Whenever we go back to idle state, in case of both an error-free transmission and an error handling situation, we reset all variables except *TEC*.

Finally, the situation after the ECU goes "bus off" is handled in parallel: when the TEC reaches 256, a check starts for all 11 bits long sequences of recessive bits and a counter is increased. When the counter reaches 128 we reset the TEC and assume the ECU to be connected again.

6 EVALUATION

We describe the environment created to test our approach, both under a feasibility and a performance point of view. After that, we discuss the results of our proof-of-concept implementation. We run two different tests. The first to understand the feasibility of our methodology and its limitations, while the second to analyze the performances required to sustain high speed CAN communication.

6.1 Testbed

To represent a real-world environment we implement four different components in our testbed: the attacker (*A*), which performs the bitbanging attack described in Palanca et al. [22] by overwriting the recessive CRC delimiter of the frame with a dominant bit. The protected ECU (*P*), on which the attack is performed. A traffic generator node (*T*), which generates frames with different IDs from the ones sent by *P*. Finally, the last node on the testbed is the proof-of-concept of our IDS, *CopyCAN*.

The hardware specifications for the attacker *A* and for *CopyCAN*, visible alongside their connections in Figure 5, comprise an ATMega328P microcontroller [4] (Arduino Uno rev3) connected to an MCP2551 [14] CAN transceiver. The CAN controller in both cases has been implemented in software due to the necessity of making it non-compliant to CAN specifications. The traffic generator *T* and the protected ECU *P* are composed of a CANTact [10] USB to CAN interface that works as CAN controller and transceiver connected to a computer.

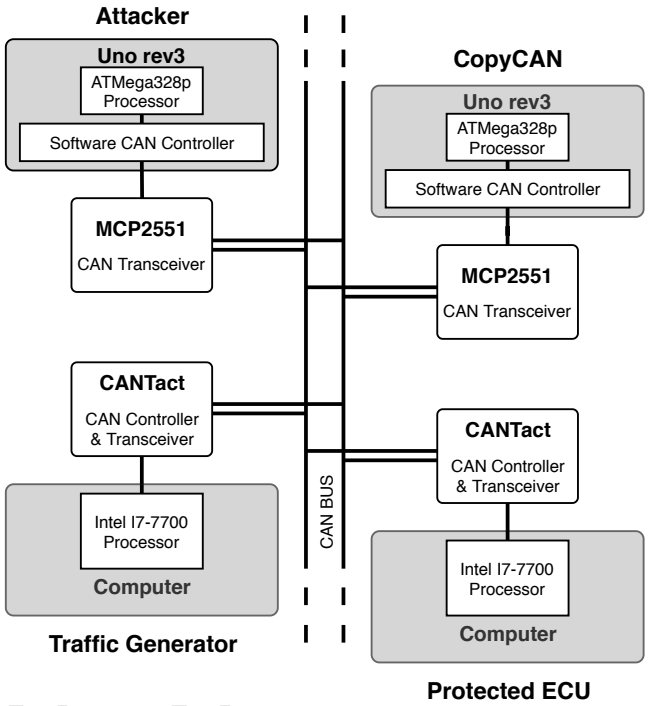


Figure 5: Overview of the testbed

6.2 Feasibility Tests

We run two different feasibility tests.

The first test analyses a basic attack recognition that does not require *T*. When *P* tries to send a frame, *A* flips the CRC delimiter bit of the frame from recessive to dominant, generating the error that increases the TEC of *P* and triggers a new attempt to send the message. After 32 attempts of transmission "denied" by *A*, *P* switches to bus off state. We ran this test 50 times and *CopyCAN* always detected the disconnection of the victim.

The second test recognizes whether the exceptions described in Section 5.2 regarding rule (4) and (6) of the fault confinement protocol affect the bus and are relevant for the correct behavior of *CopyCAN*. To stress the IDS for this test, the traffic generator *T* is connected to the bus and writes frames with different IDs. While *T* writes frames, the protected ECU *P* does the same. In the first stage of the test the attacker *A* is switched off: the goal of the test is in fact to check if any of the exceptions mentioned above occur due to some possible collisions between frames written by *P* and *T*, causing the TEC of *P* to increase. After a prescribed period of time *A* performs the DoS attack against *P*: if *CopyCAN* detects the disconnection from the bus on *P* it means that even if some collisions happened, none generated an exception that was not detected by *CopyCAN*. In fact, in case some undetectable exceptions occurred, *P* would stop transmitting frames before *CopyCAN* could signal *P*'s transition to "bus off" state. Hence, *CopyCAN* would never detect the disconnection of *P* from the bus. We ran the test 50

Unpublished working draft. Not for distribution.

Table 2: Performance test results on different processors

Processor	Frame processing time	Bit processing time	Increment	Exp. CAN rate
ATMega328P (Arduino Uno rev3)	678μs	7.2μs	0 (base)	100Kbps
Broadcom BCM2836 (Raspberry Pi2)	18μs	0.2μs	37×	>1Mbps
Intel i7-7700HQ (Dell XPS 15 9560)	1,87μs	0.02μs	362×	>1Mbps

times, with up to 15 thousand frames sent per test. In all the tests the IDS detected the attack at the exact moment it happened.

6.3 Performance Analysis

We choose the ATMega328P for the feasibility tests due to its real-time capabilities that derive from the lack of an operating system. In fact, this property enables us to trust the interrupts, required to read and write bits, to be precise to the microsecond and not lose synchronization with the bus. Other devices with non-real-time operating systems may have faster processors, but do not ensure reliable timings in reacting to interrupts. This may lead the IDS to skip bits and lose synchronization between the bus and the state machine. However, the ATMega328P microcontroller does not have a fast enough clock rate to keep up with the maximum baud rate of CAN, which is 1Mbps. In fact, after multiple tests, we detected that the controller is not able to process all bits (and therefore to update correctly the counters) if the baud rate surpasses 50Kbps. Therefore, we proceeded to execute the same code on other systems with higher-end processors in order to check how much improvement we can obtain. Since our goal is to evaluate only the execution speed of the algorithm, without taking into account the variable delays necessary for reading each bit on the bus (which would be unreliable due to the lack of real-time operative systems on top of the processors we used), we produced a sample CAN data frame of 94 bits (5 bytes of payload) and fed it directly to the code as it was the CAN bus output.

In Table 2 we show the timing results. The processing time is calculated as an average of 100 tests to process the whole 94 bit frame. The results highlight that with higher-end processors the IDS can easily sustain the maximum CAN rate of 1Mbps. In the case of the Intel i7-7700HQ it should theoretically sustain the CAN FD [6] maximum baud rate of 12Mbps during the transmission of the data payload. Due to the lack of an actual interrupt and retrieval of each bit on the bus, the results are definitely optimistic. In fact, in case of the ATMega328P, they justify the maximum 50Kbps obtained in our tests. The bit reading time of Arduino Uno rev3 is 4μs that, added to the 7.2μs processing per bit gives us 11.2μs or a maximum baud rate of 89Kbps, requires us to step down to the lower CAN compliant baud rate of 50Kbps. Depending on the specifications of the hardware we can then conclude that the increment may be lower than calculated, but given that the minimum time between one bit read and the other should be 1μs, we have respectively 0.8μs and 0.98μs to read the bit, which should definitely be sufficient with hardware more performing than the Arduino Uno connected to the MCP2551 CAN transceiver.

7 DISCUSSION ON REACTIONS

Considering the properties of CopyCAN explained in Section 5 and the threat model previously described in Section 4, we proceed to discuss two reactions that we consider feasible and suitable to implement. As explained in in Section 4, the severity of the attacks that CopyCAN protects from is high: a skilled attacker can exploit it to affect the safety of people in and around the vehicle. Moreover, in case of vehicles connected to the external environment, similar attacks have already been proven feasible from remote. Even if we cannot consider our methodology free from false positives, due to the exceptions explained in Section 5.2, the probability of generating false attack detections is close to zero in real-world scenarios, as shown by our tests in Section 6. Therefore, due both to the dangerousness of the attacks and to the extremely low chances of false positives, we claim that the reactions in case of an attack detection can be as aggressive as necessary.

The first reaction consists in alerting other ECUs and eventually the driver of the attack attempt. In this case, the ECUs can switch to a downgraded mode, less reliant on CAN communication for safety purposes, until the vehicle is reset. The driver, likewise, could for example send the vehicle to a repair shop to investigate the problem and detect which ECU has been compromised.

The second and more feasible reaction is implementable thanks to the fact that CopyCAN already requires a modified CAN controller. In fact the same device can implement both CopyCAN and a system to perform targeted DoS attacks such as the one from Palanca et al. [22]. Hence, we suggest that the most immediate reaction, once CopyCAN flags a message as an attack, should consist in a targeted DoS attack against the attacker.

The outcome of this DoS falls in one of three cases: The first case happens if the compromised ECU from which the attack is being carried out can be switched to "bus off" mode, such as in the second targeted DoS attack presented in the threat model (in Section 4.3) where the attacker uses a standard CAN controller. In this case, the attack is denied completely and the attacker has no chance of success in spoofing the message. This case has no downsides. The second case happens if the compromised ECU cannot be switched to "bus off" mode. In this case, the attacker can keep trying to send spoofed messages, that will be nullified by CopyCAN. However, if the attacker tries to send the spoofed messages too fast the bus may become unavailable. We would like to point out that in this case the attacker already has the capabilities to implement a DoS attack against the whole bus, hence we are still blocking him from achieving his goal. Moreover, as explained in Section 4.3, the vehicle already implements measures to ensure safety in case of a network-level DoS attack. The third case represents the situation in which the IDS flags a false positive. In this case, the reaction shuts the ECU down before it would actually go "bus off". However, even supposing that one of the exceptions presented in Section 5.2 leads

Unpublished working draft. Not for distribution.

to CopyCAN counting a higher TEC than the actual one, we claim that the event of the ECU going "bus off" was already destined to happen, since the only cases in which the exceptions could happen are related to either the protected ECU or another node generating too many errors, showing a faulty behavior.

8 CONCLUSIONS AND FUTURE WORKS

In this paper, we proposed CopyCAN, a novel anti-spoofing Intrusion Detection System for Controller Area Networks. CopyCAN monitors the traffic on the bus and keeps track the error counter of the protected nodes to detect when they are disconnected from the network, therefore being able to flag subsequent messages belonging to disconnected nodes as attacks. We demonstrated the feasibility of CopyCAN by implementing it in a proof-of-concept testbed and we tested the hardware requirements to implement it in real-world scenarios. Finally, we discussed the feasible reactions that can be implemented once CopyCAN detects an attack.

The only limitation regarding the CAN protocol fault confinement rules comes from the impossibility to detect rule number (4), as explained in Section 5.2. In fact, this may lead to the protected ECU switching to "bus off" state without CopyCAN detecting it. However, this particular case does not invalid the IDS since it is nor detectable nor reproducible by the attacker: since the attacker himself can only monitor the bus as CopyCAN does, he has at most the same information that we have about the protected ECU. Also, our tests show that rule (4) does not apply in the high majority of cases since we always detected the ECU going "bus off".

Future works will be focused on improving CopyCAN, by analyzing possible solutions for its limitations, and on extending the same algorithm to CAN-FD.

ACKNOWLEDGMENTS

This project has been partially supported by BVTech SpA under project grant UCSA, and by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement nr. 690972.

REFERENCES

[1] NXP Semiconductors Website. Online, https://www.nxp.com/. Accessed: 2019-06-18.
[2] NXP TJA115x secure can transceiver family. Online, https://www.nxp.com/docs/en/fact-sheet/SECURCANTRLFUS.pdf. Accessed: 2019-06-18.
[3] *VatiCAN - Vetted, Authenticated CAN Bus* (August 2016), vol. 9813.
[4] Atmel Corporation. Atmega328p datasheet. Online, sparkfun.com/datasheets/Components/SMD/ATMega328.pdf. Accessed: 2019-06-03.
[5] Berg, J., Pommer, J., Jin, C., Malmin, F., Kristensson, J., and Semcon Sweden, A. Secure gateway-a concept for an in-vehicle ip network bridging the infotainment and the safety critical domains. *13th Embedded Security in Cars (ESCARâĂŹ15)* (2015).
[6] Bosch. Can with flexible data-rate specification version 1.0. Online, https://can-newsletter.org/assets/files/ttmedia/raw/e5740b7b5781b8960f55efcc2b93edf8.pdf. Accessed 2019-06-23.
[7] Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., and Kohno, T. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), USENIX Association.
[8] Cho, K.-T., and Shin, K. G. Error handling of in-vehicle networks makes them vulnerable. CCS '16, ACM, pp. 1044–1055.
[9] Dagan, T., and Wool, A. Parrot, a software-only anti-spoofing defense system for the can bus. In *ESCAR Europe* (2016).
[10] Evenchick, E. Cantact, the open source can tool. Online, https://linklayer.github.io/cantact/. Accessed 2019-06-23.
[11] Foster, I., Prudhomme, A., Koscher, K., and Savage, S. Fast and vulnerable: A story of telematic failures. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)* (Washington, D.C., 2015), USENIX Association.
[12] GmbH, R. B. Can specification, version 2.0. Online, http://esd.cs.ucr.edu/webres/can20.pdf. Accessed 2019-06-23.
[13] Hank, P., Suermann, T., and Mueller, S. Automotive ethernet, a holistic approach for a next generation in-vehicle networking standard. In *Advanced Microsystems for Automotive Applications 2012*. Springer, 2012, pp. 79–89.
[14] Inc., M. T. Mcp2551, high-speed can transceiver datasheet.
[15] KeenLab Security. Experimental security assessment of BMW cars: A summary report, 2018.
[16] Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., and Savage, S. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy* (May 2010), pp. 447–462.
[17] Markovitz, M., and Wool, A. Field classification, modeling and anomaly detection in unknown can bus networks. *Vehicular Communications 9* (2017), 43–52.
[18] Matsumoto, T., Hata, M., Tanabe, M., Yoshioka, K., and Oishi, K. A method of preventing unauthorized data transmission in controller area network. In *2012 IEEE 75th Vehicular Technology Conference (VTC Spring)* (2012), IEEE, pp. 1–5.
[19] Miller, C., and Valasek, C. A survey of remote automotive attack surfaces, August 2014.
[20] Miller, C., and Valasek, C. Remote exploitation of an unaltered passenger vehicle, August 2015.
[21] Moore, M. R., Bridges, R. A., Combs, F. L., Starr, M. S., and Prowell, S. J. Modeling inter-signal arrival times for accurate detection of can bus signal injection attacks: a data-driven approach to in-vehicle intrusion detection. In *Proceedings of the 12th Annual Conference on Cyber and Information Security Research* (2017), ACM, p. 11.
[22] Palanca, A., Evenchick, E., Maggi, F., and Zanero, S. A stealth, selective, link-layer denial-of-service attack against automotive networks. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings* (2017), M. Polychronakis and M. Meier, Eds., vol. 10327 of *Lecture Notes in Computer Science*, Springer, pp. 185–206.
[23] Quarta, D., Pogliani, M., Polino, M., Maggi, F., Zanchettin, A. M., and Zanero, S. An experimental security analysis of an industrial robot controller. In *2017 IEEE Symposium on Security and Privacy (SP)* (May 2017), pp. 268–286.
[24] Radu, A., and Garcia, F. D. Leia: A lightweight authentication protocol for CAN. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part II* (2016), I. G. Askoxylakis, S. Ioannidis, S. K. Katsikas, and C. A. Meadows, Eds., vol. 9879 of *Lecture Notes in Computer Science*, Springer, pp. 283–300.
[25] Ring, M., Frkat, D., and Schmiedecker, M. Cybersecurity evaluation of automotive e/e architectures. In *ACM Computer Science In Cars Symposium (CSCS 2018)* (2018).
[26] Ring, R., and Kriesten. Evaluation of vehicle diagnostics security - implementation of a reproducible security access. In *SECURWARE 2014* (2014).
[27] Sen Nie, Ling Lie, Y. D. Free-fall: Hacking tesla from wireless to can bus. In *Black Hat USA 2017* (2017).
[28] Song, H. M., Kim, H. R., and Kim, H. K. 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* (2016), IEEE, pp. 63–68.
[29] Taylor, A. *Anomaly-based detection of malicious activity in in-vehicle networks*. PhD thesis, Université d'Ottawa/University of Ottawa, 2017.
[30] Tuohy, S., Glavin, M., Hughes, C., Jones, E., Trivedi, M., and Kilmartin, L. Intra-vehicle networks: A review. *IEEE Transactions on Intelligent Transportation Systems 16*, 2 (2014), 534–545.
[31] Valasek, C., and Miller, C. Adventures in automotive networks and control units, August 2013.
[32] Verendel, V., Nilsson, D. K., Larson, U. E., and Jonsson, E. An approach to using honeypots in in-vehicle networks. In *2008 IEEE 68th Vehicular Technology Conference* (2008), IEEE, pp. 1–5.
[33] Wang, Q., and Sawhney, S. Vecure: A practical security framework to protect the can bus of vehicles. In *2014 International Conference on the Internet of Things (IOT)* (Oct 2014), pp. 13–18.
[34] Wolf, M., Weimerskirch, A., and Paar, C. Security in automotive bus systems. In *Proceedings Of The Workshop On Embedded Security In Cars (ESCAR) '04* (2004).
[35] Zanero, S. When cyber got real: Challenges in securing cyber-physical systems. In *IEEE SENSORS 2018* (New Delhi, India, 2018), IEEE.

Unpublished working draft. Not for distribution.