Bringing Practical Security to Vehicles

by

Mert Dieter Pesé

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2022

Doctoral Committee:

Professor Kang Geun Shin, Chair
Professor Margit Burmeister
Professor J. Alex Halderman
Assistant Professor Manos Kapritsos

Mert Dieter Pesé

mpese@umich.edu

ORCID iD: 0000-0001-9192-5823

Science is the most reliable guide for civilization, for life, for success in the world. Searching a guide other than the science is meaning carelessness, ignorance and heresy. - Mustafa Kemal Atatürk

ACKNOWLEDGEMENTS

As my time at Michigan is coming to an end, there are several people who impacted my life during the past five years. First and foremost, I would like to express my words of gratitude to my family members. My PhD journey was impacted by the loss of both my grandparents, my grandfather Adem Bulut (1928-2019) and my grandmother Seher Bulut (1931-2021). My grandpa holds a very unique place in my life, being the one who pushed me towards engineering at an early age with his background in electrical engineering. Both losses significantly shook up my time here at Michigan and thus I would like to dedicate my thesis to both of them. May you both rest in peace and be proud of my achievements from heaven where you continue to watch and protect me. I made it, dede and anneanne. I am going to continue making you proud by joining the School of Computing at Clemson University as Assistant Professor.

I would like to continue with my mother, Mualla Pesé, who has raised me as single mother since I was a baby and without whom I would not be here right now. She has always been one of the constants during my time here, especially during the difficult times. She has always wanted me to pursue my studies in the United States, I am happy that I could fulfil your dream. Thank you, anne. Furthermore, my uncle, Mustafa Bulut, played a great role of me pursuing engineering studies due to his technical background. Despite him facing several medical challenges during the past couple years, he always had an open ear and motivated me during all times, even responding to calls at night when he was sleeping. Thank you, dayi.

Continuing on an academic note, this journey would have never been possible

without my advisor, Kang G. Shin. He fought for my admission to the PhD program and helped me in establishing the mindset of a top-tier researcher. He has been extremely patient with me during all these years and gave me the freedom to pursue my research interests independently. I cannot think of a better PhD advisor than him. His perseverance in pushing me to my limits has made me a better researcher. I am convinced that becoming a tenure-track faculty would have not been possible without his guidance. Thank you, Professor Shin.

During the past four years, I had the unique opportunity to work with 16 undergraduate and 2 Master's students. Some of them ended up as my co-authors on papers. It was an extremely important step in my education, as mentoring these students made me realize that I wanted to go to academia. As a result, words cannot express my gratitude to my students Batuhan Akcay, Eric Andrecheck, Kurt Ayalp, Bryan Brauchler, Andrés Campos, Junru Du, Alejandro Fischer, Cassandra Joseph, Junhui Li, Jiaxiang Ma, Murali Mohan, Ashwin Prakash, Osama Saeed, Jay Schauer, Erich Shan, Troy Stacer, Tim Stoldt, Arman Tabaddor, and Alice Ying.

Furthermore, I want to dedicate a paragraph to my other collaborators during my PhD. At Michigan, I had an extremely fruitful collaboration experience with Xiaoying Pu, Arun Ganesan, Dongyao Chen, Eric Newberry and Noah Curran. During my summer internships at General Motors R&D, I worked with Evripidis Paraskevas, Fan Bai, Massimo Osella, Soheil Samii, Prachi Joshi and Kemal Tepe. They helped me with understanding industrial requirements for automotive security, which significantly shaped this dissertation. Next, I would like to thank my collaborators at Harman International, Josiah Bruner and Amy Chu, who supervised me during my summer internship with them. Finally, a big thanks to Sven Bugiel and Nils Ole Tippenhauer who were my supervisors at CISPA during my last summer internship in Germany. You cemented my aspirations to go to academia.

Additionally, thank you to all other RTCL members who overlapped with me

during my time at Michigan, especially, Kassem Fawaz, Eugene Kim, Yu-Chih Tung, Kyong Tak Cho, Youngmoon Lee, Hamed Yousefi, Chun-Yu Chen, Taeju Park, Mert Pese, Duc Bui, Juncheng Gu, Jinkyu Lee, Haichuan Ding, Youssef Tobah, Hsun-Wei Cho, Wei-Lun Huang, Brian Tang, and Mingke Wang. I became close friends with several of you and value this amazing network going forward in the next steps of our careers.

The work reported in this thesis was supported in part by the Army Research Office under Grant No. W911NF-21-1-0057, the Office of Naval Research under Grant No. N00014-22-2-2622, the National Science Foundation under Grant No. CNS-1646130 as well as a Ford–UM Alliance contract.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF APPENDICES

Appendix

ABSTRACT

Modern vehicles are getting increasingly connected. Together with more automotive electronics and wireless interfaces, the number of possible attack surfaces increases, raising security concerns. Attacks on cars can have multiple implications, ranging from financial incentives or damage to the compromise of human safety. Although attacks vary, all of them have one component in common, namely *CAN bus injection*. The CAN bus is the de-facto technology used inside the in-vehicle network to interconnect automotive controllers. An attacker who compromises the CAN bus can inject arbitrary CAN messages to it, making the vehicle *misbehave*. As a result, countermeasures against the CAN bus attacks need to be implemented by carmakers. Unfortunately, the carmakers have been reluctant to adopt any approach proposed thus far to secure CAN. The main reasons for this reluctance are that (i) CAN injection requires the knowledge of semantics which differs from vehicle to vehicle and is proprietary to the car-makers, as well as (ii) industry-specific functional and cost constraints which have not been reflected in the existing solutions. Any solution that accounts for these constraints has to incur minimal overhead on computational resources and message latency.

I address these two points by first showing that proprietary semantics can be automatically reverse-engineered, effectively removing the barrier for CAN injection attacks. I demonstrated this by developing **LibreCAN** which can quickly and accurately reverse engineer both automotive powertrain- and body-related information in an automated fashion. Adversaries can significantly accelerate their preparation time for a CAN injection attack by obtaining the semantics which car-makers were trying

to keep secret by not disclosing it publicly. Second, to meet the industry-specific constraints, I propose **S2-CAN** and **MichiCAN**. The former adds *confidentiality*, *authenticity* and *integrity* to the CAN bus without the overhead of cryptography, but by leveraging protocol-specific properties. The latter protects the CAN bus against attacks on its *availability*, e.g., Denial of Service, by leveraging novel hardware features of automotive controllers. Its main difference from existing work is practicality. Instead of adapting well-known cryptographic techniques from the realm of computer networks which do not satisfy the aforementioned cost and functional constraints, I propose out-of-the-box solutions that leverage protocol- and hardware-based features of automotive networks and controllers. Furthermore, both S2-CAN and MichiCAN are fully backward-compatible with existing hardware and specifications, as well as incur minimal computation and network overheads. CAN injection attacks can also be conducted *remotely* by accepting malicious data from other cars in vehicle-to-vehicle (V2V) communication scenarios. I propose **CARdea**, a two-phase anomaly detection system that sanitizes incoming data from surrounding vehicles. Compared to computationally-intensive prior work, CARdea combines an in-vehicle light-weight anomaly detection phase with a more resource-heavy machine-learning phase that can be executed on the vehicle, edge or cloud based on available computational resources and manufacturer constraints. Overall, this dissertation demonstrates the omnipresence of CAN injection attacks and develops novel, practical security solutions for CAN and V2X by analyzing the inherent trade-off between security and performance.

CHAPTER I

Introduction

The invention of the car dates back to 1886 when German inventor Karl Benz patented his Benz Patent-Motorwagen. The next decade was accompanied with significant advancements in the field of powertrain engineering and vehicles were mass-produced. Cars began to have electronic features starting in the 1950s with the semiconductor revolution. While automotive electronics consisted of merely 1% of a car's value in 1950, it rose to 30% in 2010 [24].

In recent years, the number of Electronic Control Units (ECUs) inside cars has increased significantly. ECUs are systems embedded in cars and connected to sensors and actuators. Electronic components become increasingly important as the number of functionalities — especially in the domain of advanced driver assistance systems — rises. Modern vehicles can contain up to 150 ECUs [20]. These ECUs have to be interconnected with wires. Nearly all mechanical functions inside modern vehicles are supported by electrical control, even traditional domains such as powertrain. As a result of this development, automotive bus systems were necessary to avoid point-to-point connections between ECUs, and thus to reduce the cable harness.

In 1983, Bosch started developing the *Controller Area Network* (CAN) which was released in 1986. CAN has since been the most widely in-vehicle network (IVN) protocol, despite many alternatives being developed and adapted in the following

decades, such as FlexRay or Automotive Ethernet. (A detailed introduction of CAN will be provided in Sec. 1.2.)

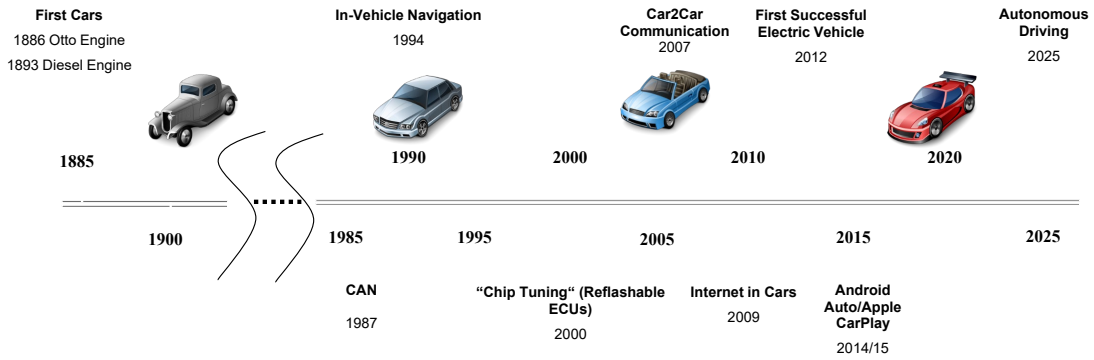


Figure 1.1: Evolution of Cars

Since the introduction of CAN, several elementary automotive features have been developed as depicted in Fig. 1.1, e.g., built-in navigation systems, possibilities of software updates by reflashable ECUs or Internet connection. Newer developments are (a) vehicle-to-everything (V2X) communication and (b) autonomous driving. V2X communication allows vehicles to talk to each other (e.g., cooperative driving in a platoon), as well as infrastructure (e.g., roadside units) and can be seen as a necessary precursor of autonomous driving.

Cars used to be *closed* entities, with no exposed interfaces to the *outside*. Some of the aforementioned developments require vehicles to communicate with *external* entities, which can be a substantial benefit to drivers, especially in terms of safety (V2X) and convenience (Internet connectivity). According to a United Nations Economic Commission for Europe (UNECE) analysis, modern cars contain 100 million lines of code, which will increase to 300 million by 2030 to account for autonomous driving, Internet connections, and other advanced capabilities [20]. However, this increasing connectivity comes with the risk of security.

1.1 Evolution of Automotive Security

Although concerns about automotive security had already been raised in the 2000s [56, 130, 186], it was not until 2010 when these concerns started to get increasing attention [58, 109], especially in academic circles. The compromise of the CAN bus can have grave consequences on the functioning of the car since an attacker can take control of it (e.g., by braking or steering it) [124]. This is done by so-called *CAN injection* attacks, i.e., the attacker who has compromised the CAN bus will inject a well-formed CAN message to the bus and every ECU listening to this message will operate with the data in the malicious CAN message.

The field of automotive security has changed since its inception, due mainly to rising connectivity and more exposed *external* interfaces. Attacks on the CAN bus (or interchangeably any in-vehicle network) can be grouped into three generations. Fig. 1.2 shows my proposed taxonomy.

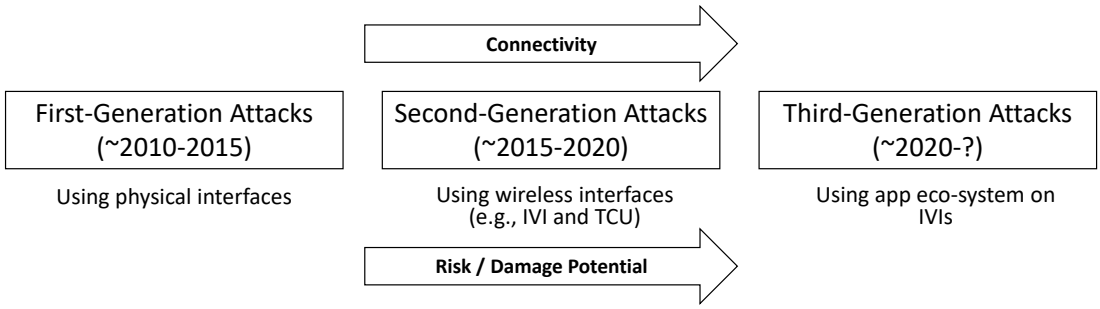


Figure 1.2: Generations of automotive security

First-generation attacks that started with the rise of automotive security literature in the early 2010s were mostly targeting physical interfaces, i.e., the attacker needed to have physical access to their victim vehicle. Once the attacker was *inside* the vehicle, they could simply access the in-vehicle network (IVN) through a physical connector called OBD-II port. This interface is mandated in all US gasoline vehicles manufactured after 1996.

Second-generation attacks went further and tried to gain IVN access without being

physically inside a vehicle. For this purpose, attackers would exploit vulnerabilities in the wireless interfaces of ECUs, e.g., the WiFi or cellular connectivity of Telematic Control Units (TCUs). The most comprehensive and impressive attack of this generation was the famous Jeep hack that happened in 2015 [127], with the white-hat hackers exploring these remote interfaces already a year earlier [125]. These hackers were able to obtain CAN bus access through several vulnerabilities in the TCU's software and hardware and could kill a running Jeep Cherokee on the highway (or steer it into a ditch). As a result of this hack, 1.4 million vehicles had to be recalled and a lawsuit that had been filed against the OEM and Tier-1 [85] was just dismissed in 2020 [187]. This sophisticated attack also highlighted one issue with automotive security, namely the dependence of OEMs on Tier-1s which supply various ECUs. Compared to the first-generation of attacks, this generation is more feasible to be conducted since no physical access is required. As a result, the risk and damage potential increases. Furthermore, these attacks are also more scalable as the high number of recalls proved.

Finally, third-generation attacks take the scalability and damage potential even further. As of the time of this writing, there are no known attacks yet, although the technology required for it is slowly maturing. A new in-vehicle infotainment (IVI) operating system called Android Automotive (AAOS) was announced by Google in 2017. A custom flavor of the popular Android mobile operating system, its most distinct feature is its ability to connect to the IVN and read, as well as write data to it. Third-party apps will be gradually supported in a custom Play Store, but are limited to media, messaging, navigation, parking, and charging apps at the moment [27]. With an increasing number of third-party apps, as well as OEMs heavily customizing AAOS, I predict serious security risk coming from this platform. Now, malicious entities will be able to access the vehicle and its IVN from anywhere, opening the doors for significant damage potential. In fact, I was the first to conduct a first high-

level security analysis of AAOS in 2020 [137] and showed possible attacks on driver safety, privacy and financial incentives.

All three generations have one thing in common: The end goal is to access the IVN, e.g., the CAN bus, although they differ in the *approach* they take to compromise it. Once the CAN bus is accessible by an attacker, they can conduct *CAN injection* attacks and thus ultimately compromise the operation of the vehicle. This is the reason why CAN injection attacks pose a serious threat that needs to be prevented.

1.2 Background on CAN Bus

Since most of this thesis proposal deals with CAN bus security, all relevant concepts need to be introduced. Several chapters rely on this background knowledge and its repetition in each respective chapter will thus be avoided. Only concepts that are specific to a chapter will be presented in the background section of that chapter.

1.2.1 CAN Primer

Vehicular sensor data is collected from ECUs located within a vehicle. These ECUs are typically interconnected via an on-board communication bus, or in-vehicle network (IVN), with the CAN bus being the most widely-deployed technology in current vehicles. Fig. 1.3 depicts the structure of a CAN 2.0A data frame — the most common data-frame type used on the CAN bus. The fields in each CAN frame are depicted in Fig. 1.3.

- **SOF:** The *start-of-frame* (SOF) bit indicates the beginning of a new CAN frame/message and is always set to 0.
- **CAN ID:** CAN is a multi-master, message-based broadcast bus. Unlike better-known socket-based communication protocols like Ethernet, CAN is message-oriented, i.e., CAN message frames do not contain any information concerning

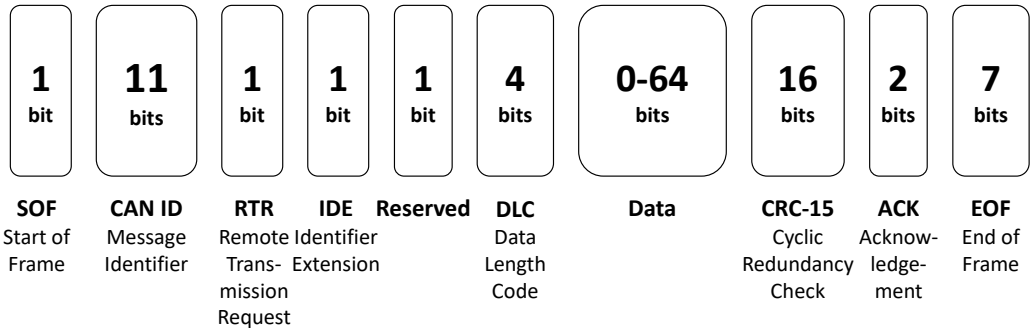


Figure 1.3: CAN data frame structure

their source or destination ECUs, but instead each frame carries a unique message identifier (ID) that represents its meaning and priority. Only one CAN message can be broadcast on the bus at a time. If multiple messages want to transmit, the CAN message with a higher priority will ”win” the distributed *arbitration* process and be allowed to transmit on the bus. Lower CAN IDs have higher priority. The reason behind this is the *wired-AND* logic of CAN. If two ECUs would like to transmit at the same time, the dominant "0" bit of one sender will always overwrite the recessive "1" bit of the other sender. As a result, the sender which transmits the dominant bit will win arbitration and allowed to continue transmitting their CAN message on the bus, while the other sender needs to abort and retry later. It is possible for the same ECU to send and/or receive messages with different CAN IDs. The basic CAN ID in the CAN 2.0A specification is 11 bits long, (compared to 29 bits in CAN 2.0B, a.k.a. the extended format) and thus allows for up to 2048 different CAN IDs.

- **RTR & IDE & Reserved:** The *remote transmission request* (RTR) bit is always set to 0 for data frames. The *identifier extension* (IDE) bit is set to 0 for 11-bit CAN IDs. Finally, the *reserved* (r0) bit is always set to 0.

- **DLC:** This field specifies the number of bytes in the payload (data) field of the message. The DLC field is 4 bits long and can specify a payload length from 0

to 8 bytes.

- **Data:** This is the payload field of a CAN message containing the actual message data and can contain 0–8 bytes of data depending on the value of the DLC field.

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

- **ACK & EOF:** The first bit of the *acknowledgment* (ACK) field is called *ACK slot* and the second bit *ACK delimiter*. The ACK slot is always set to 1 for the transmitting ECU. If the receivers do not observe any errors in the frame, they send a 0 during this slot. Due to the wired-AND logic, at least one receiver needs to transmit a 0 and thus acknowledge the correct receipt of the CAN frame. If the transmitter (which reads back this slot) detects that nobody acknowledged this frame by sending a 0, it will retransmit this frame. The ACK delimiter, as well as the *end-of-frame* (EOF) is always 1. After the transmission of a complete CAN frame, the next CAN frame has to wait another 3 bits (not depicted in Fig. 1.3) which is called *inter-frame spacing* (IFS). As a result, the next CAN message can only be transmitted after at least 11 recessive bits.

Next, we will describe the structure of the data payload field, which consists of one or more "signals." A "signal" is a piece of information transmitted by an ECU, such as vehicle speed. Messages transmitted with the same CAN ID usually contain related signals (within the same domain) so that the destination ECU needs to receive and process fewer messages. For instance, a message destined for the Transmission Control Module (TCM) might contain both the vehicle speed (m/s) and engine speed (RPM) signals in one CAN message. The length and number of signals vary with CAN ID and are defined in the aforementioned DBC file for the corresponding vehicle. This translation file specifies the start position and length of a signal, allowing it to be easily retrieved from the payload using a bitmask if the DBC file is available.

Moreover, signals can not only contain physical information, but also other types of information [120, 122], such as:

- **Constants:** Values that do not change over time.
- **Multi-Values:** Values with a domain consisting of only a few constant values. [122] reported 2–3 changing values within these types of signals. An example of a 2-value field could be the status of a specific door (e.g., open or closed).
- **Counters:** Signals that behave as cyclic counters within a specific range. These signals could serve as additional syntax checks or be intended to order longer signal data at the destination ECU(s).
- **Checkcodes:** Besides the CRC-15 field at the tail of every CAN frame, the payload can also contain additional checkcodes, typically as the last signal in the payload.

A contrived example is given in Fig. 1.4 showing multiple signals of different types (physical signals, multi-values, counters, CRCs, etc.) embedded in the 8-byte payload of a CAN message. For instance, the orange-colored entity represents a 2-byte physical signal and the yellow one depicts a 12-bit counter, whereas the blue region is another 1-byte long physical signal. Several CAN IDs also contain 1-bit signals that are multi-values, i.e., booleans that describe a body-related event (e.g., door is open/closed). Three status flags are depicted in byte 7 of this example. The remaining green signal is a 4-bit checksum. White regions are unused, i.e., no signals are defined in the DBC file. CAN signals are defined by the OEM and can thus have arbitrary lengths. Some OEMs also decide not to include specific signal types. For instance, none of our evaluation vehicles (all from the same OEM) contain checksums.

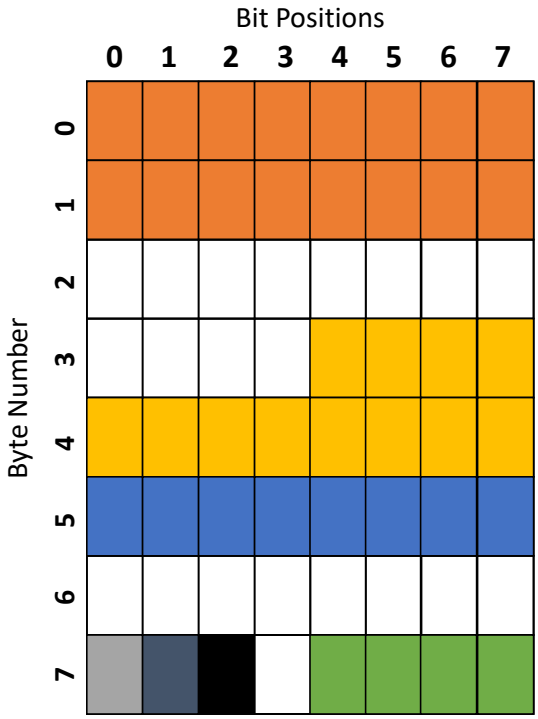


Figure 1.4: Example of CAN signals

1.2.2 DBC Files

All recorded CAN data can only be interpreted if one possesses the translation tables for that particular vehicle. These tables can come in different formats, as there is no single standard. Examples are KCF (Kayak [98]) and ARXML (AUTOSAR [1]) files. However, the most common format used for this purpose is DBC [76], a standard created by German automotive supplier company Vector Informatik.

DBC files contain a myriad of information. However, to understand this thesis, one must be aware of the following information stored in these files:

- Message structure by type: CAN ID, Name, DLC, Sender;
- Signals located within messages, containing Name, Start Bit, Length, Byte Order, Scale, Offset, Minimum/Maximum Value, Unit, Receiver

The representation of translation data in DBC files can be confusing [74]. CAN data can be represented in either *big endian* (Motorola) or *little endian* (Intel) byte-

order. The bits can also be numbered using either MSB0 (most significant bit first) or LSB0 (least significant bit first). However, most DBC files use the Intel format with LSB0 numbering. Therefore, the start bit included in the signal information does not describe the actual start bit. Since we need to know the actual signal boundaries, we need to calculate the true start bit s so that we can, combined with the signal length l, obtain the signal end bit e:

$$\begin{aligned} s &= \lfloor \frac{s}{8} \rfloor + 7 - (s \ \% \ 8), \\ e &= s + l - 1. \end{aligned} \tag{1.1}$$

Note that DBC files are kept secret by OEMs and are not disclosed to the public.

1.2.3 In-Vehicle Network Architecture

There are four major bus systems used in cars: CAN, FlexRay, LIN, and MOST. MOST is used for multimedia transmission, whereas the other bus types are mostly used for control tasks, e.g., in the powertrain domain. The most widely used In-Vehicle Network (IVN) architecture is the *central gateway architecture*. An overview of the buses and their interconnection within a vehicle is shown in Fig. 1.5.

The major point of entry into a vehicle for data collection (and diagnostics) is the on-board diagnostics (OBD-II) interface. This connector is mandatory for all vehicles sold in the US after 1996.

Emission-related sensors such as vehicle speed, engine speed, intake temperature, mass airflow, etc., are universally available in all vehicles (after 1996) via the standardized OBD-II protocol [9]. Apart from the standardized OBD-II protocol (called SAE J/1979), this port can also be used to both read and *write* raw CAN data. Note that the OBD-II protocol and OBD-II interface are different and should not be confused.

Electric vehicles (EVs) are not mandated to either have an OBD-II connector

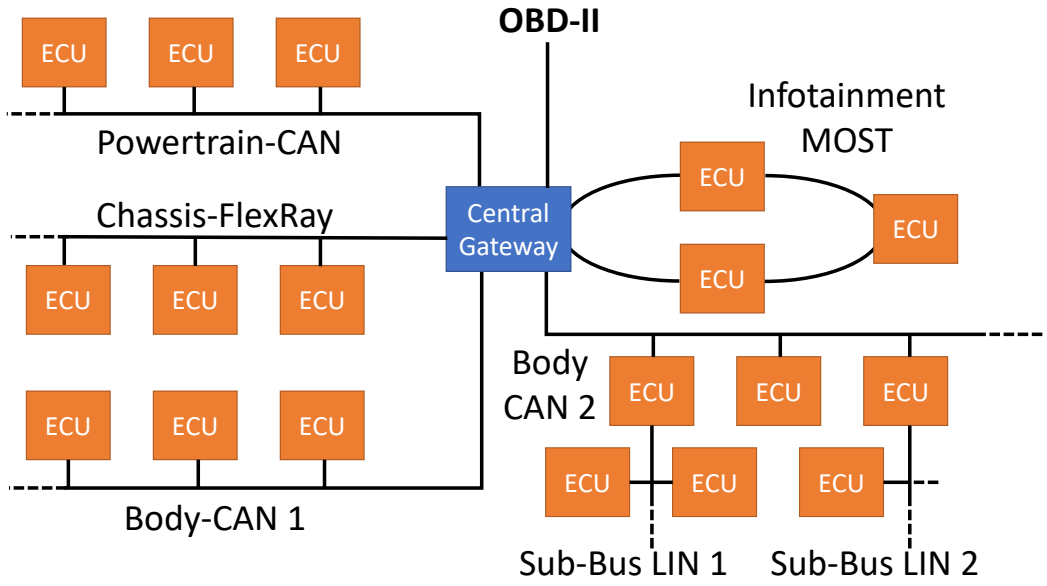


Figure 1.5: Common automotive E/E architecture (adapted from [195])

nor support the OBD-II protocol. The latter would not contain a lot of information anyway due to the lack of mechanical powertrain components (the OBD-II protocol provides emission-related information [9]). Since there is no standard for EV diagnostics, EV OEMs can use any interface they desire. For instance, older Tesla Model S and X still carry a traditional OBD-II port, whereas the newer Model 3 has its proprietary hardware interface [44]. Furthermore, proprietary diagnostic protocols are used in EVs (instead of SAE J/1979).

OBD-II data can be accessed by anyone through aftermarket dongles [77]. The OBD-II protocol uses the CAN bus at the physical layer in all newer vehicles. It is a request-response protocol that sends requests on CAN ID 0x7E0 and obtains responses on 0x7E8. For instance, to obtain the vehicle speed, a dongle connected to the OBD-II port sends a CAN message with ID 0x7E0 and payload 0x02010D5555555555. The first byte (0x02) indicates that 2 more bytes will follow, the second byte (0x01) corresponds to the OBD mode of getting live data, and 0x0D indicates vehicle speed. Unused bytes are set to 0x55 ("dummy load") and ignored. A complete specification is available in Wikipedia [9].

Note that the OBD-II protocol is public and does not make any use of DBC files at all. As stated in [9], only certain emission-related sensors can be read. Body-related signals are not part of the OBD-II specification. Nevertheless, signals in the aforementioned specification are still available in the raw CAN protocol. However, we would still like to locate the CAN IDs and signal positions of emission-related signals on the CAN bus. For CAN injection attacks, we need to know this information because the OBD-II protocol does not allow *writing* arbitrary values to these sensors.

Since any node can tap into the unencrypted CAN bus and start broadcasting data without prior authentication, a malicious entity can gain access to the in-vehicle network by using an OBD-II dongle as a CAN node and send messages (e.g., through a mobile app). Note that it is also possible to physically tap into any CAN bus domain (after removing plastic compartments) by using an Arduino with a CAN bus shield [197]. If the message semantics (i.e., the DBC file(s) or portions thereof) are known to the attacker because they reverse-engineered the CAN bus, they can cause the vehicle to misbehave by affecting the operation of receiver ECUs. This can range from displaying false information on the instrument cluster [109] to erroneously steering the vehicle [124]. The latter impacts vehicle safety and, therefore, poses greater risk. Furthermore, it is also possible to cause certain ECUs to fail, possibly incurring operational/financial damage to the vehicle.

Theoretically, it is possible to monitor the traffic on all in-vehicle buses through the OBD-II interface. In practice, however, not all buses are mirrored out by the central gateway, which is responsible for routing CAN messages between buses or domains. This can be explained by access control [140] that OEMs implement. Nevertheless, previous literature [124] has shown that CAN injection through the OBD-II port is possible in numerous cars. Furthermore, the OBD-II connector has only 16 pins, with some pins already assigned [10], and thus only up to three CAN buses can be monitored through the OBD-II port.

1.3 Background on Vehicle-to-Everything (V2X) Communication

Cars are becoming increasingly connected to support an increasing number of convenience and safety functions. The future of intelligent transportation systems (ITS) will be spearheaded by V2X (vehicle-to-everything) communication which can complement and enhance Advanced Driver-Assistance Systems (ADAS) and autonomous vehicles (AVs). Among others, V2X allows connected vehicles to talk to other vehicles (V2V), smart infrastructure (V2I) and pedestrians (V2P). V2X can be expected to not only help AVs (e.g., cooperative adaptive cruise control), but also benefit traditional (i.e., human-driven) cars by avoiding traffic congestion and preventing collisions.

Vehicles exchange Basic Safety Messages (BSMs) in the US which are defined in the SAE J2735 standard [149]. BSMs contain state information about a vehicle, such as its location, speed or acceleration. An overview of all these sensors is provided in Fig. 1.1. Note that other territories have different formats, such as Cooperative Awareness Messages (CAMs) in the European Union which share a similar format to BSMs [107].

1.4 State-of-the-Art Defenses

As mentioned before, the Controller Area Network (CAN) bus is the de-facto standard in contemporary vehicles and has been around for more than three decades. Since CAN has not been designed with security in mind, most of my thesis deals with enhancing it by the key cyber-security properties of *confidentiality*, *authenticity*, *integrity* and *availability*. The main threat and key part of every cyber-attack against vehicles to date are CAN injection attacks, which can lead to serious malfunctioning of the vehicle [58, 109, 127]. CAN injection can also result in attacks on the *availability* of the CAN bus, e.g., by Denial-of-Service (DoS) attacks [133].

Table 1.1: Format of Basic Safety Messages

Message	**Content**
BSM Part I	Message Count Temporary ID Time Position (latitude, longitude, elevation) Position accuracy Transmission state Speed Heading Steering wheel angle Acceleration Yaw rate Brake system status Vehicle size (width, length)
BSM Part II	Event flags Path history Path prediction RTCM package

There is no one-size-fits-all approach to prevent CAN injection attacks and provide all aforementioned security properties. A holistic security concept consisting of multiple layers has to be developed to address CAN injection attacks. The following three-layer approach can be considered [194]:

- Access control to network
- Secure on-board communication
- Anomaly detection and defense

The first layer restricts non-authorized access to the in-vehicle network (IVN) by the deployment of firewalls. Given the IVN architecture from Fig. 1.5, access control can be implemented in the central gateway. This leads to *separation of domains*, i.e., if one bus is compromised, malicious messages cannot reach other — potentially more safety-critical — domains. This is a very simple countermeasure, especially considering that the infotainment bus (which is less safety-critical) with its wireless

interfaces has been compromised in previous attacks [127]. Safety-critical busses such as the powertrain CAN usually lack connectivity and the gateway would block any malicious CAN injection attempts from the infotainment CAN that could affect powertrain-related functions.

The second layer suggests ways for message authentication and ensuring data integrity. Since CAN is a broadcast protocol, CAN messages do not contain any information about their sender ECU. This is a serious problem, since any compromised ECU or attacker node that taps into the CAN bus can transmit messages with an existing CAN ID. As a result, receiver ECUs have no provisions about knowing if CAN messages were sent by a genuine or malicious transmitter. Besides CAN ID spoofing, the attacker can also spoof the payload as part of their CAN injection attack. The CAN protocol also does not suggest the use of Message Authentication Codes (MACs) to be included anywhere in a CAN message to check the integrity of the transmitted message. To sum up, *authenticity* cannot be guaranteed by CAN. Furthermore, a proper CAN injection attack needs to know the semantics of the communication matrix, i.e., the DBC file, to inject a well-formed CAN message. For instance, if the attacker wants to steer the vehicle into a ditch, it needs to know in which CAN ID (and what position within the payload) the steering wheel angle signal is located. Since CAN messages are exchanged in plaintext, an attacker can reverse engineer the desired signal manually by listening sufficiently to the bus. If the CAN payload was encrypted — providing *confidentiality* — it would be impossible for an attacker to reverse engineer the semantics.

Finally, the third layer consists of anomaly/intrusion detection and prevention systems (ADPS/IDPS). These solutions are constantly monitoring several properties of network traffic and reacting to deviations. Literature for ADPS/IDPS is growing and several survey papers have been recently published to provide a taxonomy of the different approaches taken to respond to anomalies or intrusions [115, 188, 193].

Broadly speaking, countermeasures include fingerprint-based, parameter monitoring-based, information theory-based and machine learning-based solutions. Each category extracts CAN features on a different layer of the OSI stack. For instance, fingerprint-based approaches monitor the physical bus level, whereas machine learning-based solutions are data-driven and operate on the CAN payload. IDPSes can be used to satisfy the last security property of *availability* on the CAN bus by detecting DoS attacks [133].

Defenses for V2X communication are manifold as well. Similar to IVN security, a holistic multi-layer approach is required to address different adversaries. For instance, BSMs from *external* attackers (e.g., roadside attackers with V2X radio) will be discarded immediately due to lack of valid credentials to join the BSM broadcast. In contrast, *internal* attackers (e.g., compromised ECUs) are "real" vehicles that are authenticated to exchange BSMs with their surrounding vehicles and other entities. Just like the different kinds of CAN injection attacks, V2X injection comprises false data injection/spoofing and DoS attacks. Another unique attack type for V2X are Sybil attacks [152].

1.5 Challenges

As laid out in the previous subsection, both CAN and V2X security comprise similar defenses against similar attack types. Despite the importance of automotive security and several proposed solutions from academia, their adoption in real-world commercial vehicles by OEMs is rare. This lack of adoption can be attributed to the following three challenges:

- **(C1) Cost:** This is usually the dominant driver behind the lack of adoption. OEMs operate within extremely tight cost constraints and aim to avoid any increase in their production cost. One dimension how added security can

contribute to overall cost is resource constraints. ECUs in current vehicles only require simple operations, and thus most ECUs are not built with high-performance hardware. For instance, current Engine Control Modules can have 80MHz clock frequency, 1.5MB Flash memory and 72kB of RAM (Bosch [16]). Adding cryptographic security operations for encryption and authentication on the CAN bus, or machine-learning-based techniques for ADPS/IDPS (both for CAN and V2X) would require significantly more performant hardware which would, in turn, increase the costs to acquire more powerful ECUs. Furthermore, OEMs rely heavily on the economy-of-scale of their supply chain. They source several ECUs directly from different suppliers and Tier-1s. By reusing legacy ECUs for many generations of their vehicles, OEMs can avoid development costs and keep their purchasing costs low. Satisfying new security requirements would lead to new ECU developments which would increase the OEMs' cost of manufacturing a vehicle [175].

- **(C2) Latency:** This functional parameter is extremely important in vehicles which come with stringent hard real-time requirements to satisfy automotive safety certifications such as ISO 26262 [12]. For instance, in the case of CAN security, ensuring confidentiality and authenticity requires encryption and adding MACs. The maximum permissible end-to-end (E2E) latency on CAN is in the sub-second range [71]. Both encryption and MAC calculation add a non-negligible delay, especially considering the low power and cost hardware. This can lead to deadline misses which can compromise driver safety and is thus unacceptable. In terms of V2X security, detection latency is an important metric since anomalies/intrusions need to be responded to as quick as possible. Numerous existing works ignore this requirement while proposing performant ADPS/IDPS solutions and also do not consider how slowly their heavy solutions would run on resource-constrained automotive hardware.

- **(C3) Mindset:** This challenge specifically applies to CAN security. OEMs never disclose DBC files to the general public. They keep them as proprietary secrets within their organization. DBCs are even only partially shared with Tier-1s. By doing so, OEMs believe that they can both prevent eavesdroppers from logging interpretable data on the CAN bus, as well as deter attackers from launching CAN injection attacks. In fact, keeping DBCs secret acts as a barrier to CAN injection since attackers need to tediously reverse-engineer the signal information they want to target first. Since DBCs differ between vehicle models, they need to repeat this step for each model they want to attack. As a result, OEMs believe that their *security-by-obscurity* mindset will help them improve the security of the CAN bus by deterring attackers.

1.6 Thesis Contributions

1.6.1 Thesis Statement

Following up on the challenges C1-C3 outlined in the previous subsection, I identify three major intellectual contributions (ICs) that will eventually lead to my thesis statement.

IC1: Semantics can be automatically reverse engineered, accelerating CAN injection attacks. OEMs believe that they can prevent attackers from launching CAN injection attacks through their *security-by-obscurity* mindset, as attackers will not have access to the necessary semantics. Previous attacks have already shown that this is not necessarily a deterrent [124, 125, 127], since semantics can be manually reverse engineered. Since this a tedious process that can take up days depending on the number of targeted signals, I demonstrate that the reverse-engineering process can be accelerated by an automated tool, practically eliminating the barrier to CAN injection attacks.

IC2: Solve CAN security problems by satisfying functional and cost constraints of OEMs. Given the constraints that OEMs impose on their vehicles, the main objective of this intellectual contribution is to develop security solutions that are *feasible* and *practical* to be deployed on the CAN bus. This stands in stark contrast to previous extensive work in this domain. As depicted in the previous subsection, *security* comes at the expense of *performance*. As a result, a trade-off has to be made to add security to vehicles (and thus prevent CAN injection attacks) without compromising its commercial adoption.

IC3: Solve V2V security problems by hybrid approach combining in-vehicle and off-vehicle anomaly detection. A connected vehicle (i.e., V2X-equipped) needs to detect rogue vehicles broadcasting malicious sensor data and prevent it from going to its actuators. Numerous anomaly detection systems have been proposed to solve this problem, but similarly to IC2, none of them focused on the actual deployability on real vehicles. Purely machine learning-based solutions are too heavy for the vast majority of contemporary cars, especially considering that a fast detection time (or short latency) is required, whereas purely statistical techniques might not cover the entire threat model of an attacker. A trade-off between good detection performance (i.e., security level) and reasonable resource consumption has to be found.

My thesis statement can be summarized as follows:

Thesis Statement: Demonstrate the omnipresence of CAN injection attacks and develop novel, feasible security solutions for CAN and V2X by analyzing the trade-off between security and performance.

1.6.2 Thesis Components

My thesis work was motivated by **LibreCAN** [CCS'19]. We demonstrated that the barrier for CAN injection attacks to cause vehicle malfunction can be significantly reduced by automated CAN bus reverse engineering. This chapter covers IC1. To mitigate this attack, I proposed **S2-CAN** [ACSAC'21] to bring *confidentiality*, *integrity* and *authenticity* to the CAN bus without the use of cryptography. It offers negligible overhead to latency and other computational resources, although it requires compromises in the security level. To further mitigate attacks on the *availability* of the CAN bus, I designed **MichiCAN** that makes use of novel automotive hardware features to detect and prevent Denial-of-Service (DoS) attacks in a backward-compatible and light-weight way, i.e., without adding significant overhead to CAN communications. These two works address IC2. Finally, I applied the security-performance trade-off to V2V-enabled connected vehicles by proposing **CARdea** which is a hybrid anomaly detection system consisting of in-vehicle and off-vehicle detection components. The final chapter solves IC3. Fig. 1.6 depicts these four components/chapters of my thesis, together with the order of appearance in this thesis, as well as the corresponding intellectual contribution.

1.6.3 LibreCAN [141]:

CAN messages are unencrypted and any adversary with access to the CAN bus can sniff the plaintext data. Nevertheless, each make and model encodes the data differently, typically requiring an attacker to interpret data using a translation table (called DBC). Carmakers keep DBCs private in the hopes of hiding them from attackers. This has failed to prevent attackers from conducting manual reverse engineering of the CAN bus, though this is a tedious and long process. I showed that by exploiting the security-by-obscurity principle of automotive carmakers in an automated fashion, an attacker can quickly and easily launch a CAN injection attack and cause

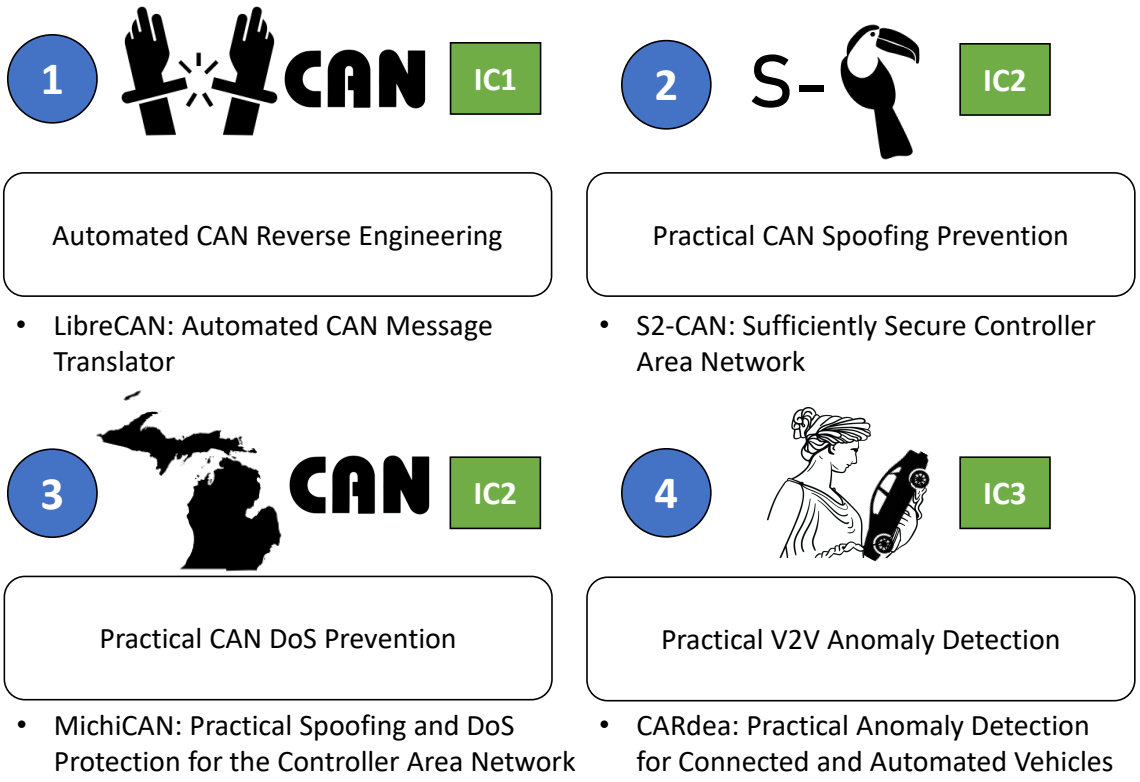


Figure 1.6: Components in this Thesis Proposal

the vehicle to malfunction. For this purpose, I designed an automated CAN message reverse engineering tool called LibreCAN that can reverse engineer most of the DBC in under two minutes with 82.6-95.1% accuracy, much higher than existing works.

1.6.4 S2-CAN [139]:

As my work demonstrated, automated CAN reverse engineering accelerates CAN injection attacks on unknown vehicles. Spoofing can be prevented by adding a Message Authentication Code using cryptographic means. However, this comes at the expense of latency and the need for more powerful ECUs. Since hard real-time deadlines and cost requirements make this form of authenticity and integrity protection infeasible, I developed a novel security solution called S2-CAN. It presents a trade-off between security and performance on the CAN bus. S2-CAN adds the security properties of confidentiality, authenticity, and freshness to CAN messages without using

cryptography. By modifying LibreCAN to attack S2-CAN, I showed that a secure CAN is possible with minimal overhead on ECU resources and latency, as long as its design parameters are correctly chosen.

1.6.5 MichiCAN

CAN is also susceptible to DoS attacks which have traditionally been a focus of CAN-based intrusion detection systems (IDSes), both in academia and industry [133, 168]. Licensing costs or the need for a dedicated ECU have held carmakers back from adopting it in their vehicles. Furthermore, IDSes do not prevent attacks and thus lack practicality. Hence, DoS attacks need to be detected *and* prevented with existing ECUs as fast as possible. Since DoS attacks cannot be prevented by using cryptography (or alternatives) on the application layer, the only viable option is to leverage the error handling mechanism of CAN and confine the attacking ECU into bus-off mode. Some recent work proposed solutions to bus-off the attacking ECU according to CAN protocol specifications with the drawback of severely increasing the bus load and bus-off time [67]. To mitigate these drawbacks, I proposed a new backward-compatible and real-time approach called MichiCAN to defend against DoS (and spoofing) attacks by using novel integrated/on-chip CAN controllers which are gaining traction in real-world ECUs. Integrated CAN controllers allow the ECU to bypass the CAN controller, enabling bit banging, which is used for real-time detection and prevention of DoS attacks without adding any significant overhead to the CAN bus.

1.6.6 CARdea

Vehicle-to-vehicle (V2V) communication as a complementary source to in-vehicle cameras, radars, and lidars can help connected vehicles improve traffic management, provide driver assistance and prevent possible crashes [23]. On the other hand, com-

promised vehicles can broadcast malicious information to trick vehicles into collisions or cause traffic congestion. Existing solutions to sanitize incoming V2V data either focus on certain attacks (e.g., GPS spoofing) or rely on computationally-heavy algorithms that are impractical on the restricted resources of existing ECUs. To address this, I designed and implemented a novel intrusion detection system for V2V which detects anomalous broadcasts from malicious or faulty vehicles. My system called CARdea uses a two-phase approach with a statistics-based, light-weight Phase 1 deployed on the vehicle and a machine learning-based, resource-heavy Phase 2 that can be executed on the vehicle, edge, or cloud. The first phase detects anomalous BSMs from vehicles with up to 98% sensitivity in only 0.04ms, whereas the second phase handles certain cases that the first phase cannot detect. The experimental evaluation consists of 132 hours of simulated BSM data in realistic traffic scenarios and multiple attack types. I showed that using a two-stage approach, practicability does not need to be compromised at the expense of detection performance.

1.7 Organization of Thesis Proposal

This dissertation is organized as follows. Chapter II presents the automated CAN bus reverse engineering tool `LibreCAN`. Chapters III and IV propose the defensive solutions `S2-CAN` and `MichiCAN`, respectively. Furthermore, practical security is extended to connected vehicles in Chapter V with `CARdea`. Finally, I conclude the dissertation in Chapter VI where I also discuss future research directions.

CHAPTER II

LibreCAN: Automated CAN Message Translator

2.1 Introduction

Nearly all functions inside a modern vehicle, even in more traditionally mechanical domains like the powertrain, are controlled electronically. Moreover, purely electronic systems have become more prevalent as the number of sensors present in a vehicle has increased, particularly given the rise of Advanced Driver Assistance (ADAS) systems. All of these systems are controlled by Electronic Control Units (ECUs), embedded microprocessors that interface between a given system and the rest of the vehicle. Over the last few years, the number of ECUs inside a vehicle has increased significantly. Compared to the early 1990s, when few ECUs were present in a given vehicle, a modern vehicle features more than 40 ECUs (as of 2015 in Europe) [123]. Meanwhile, premium cars can be equipped with up to approximately 100 ECUs. These ECUs need to communicate over a unified communications network that is sophisticated and robust enough to handle all network traffic inside a vehicle, particularly for time-critical information. To meet this need, Bosch introduced the Controller Area Network (CAN) technology in 1987, which has since become the *de facto* standard in-vehicle network.

According to Frost & Sullivan [142], data security and privacy are among the most critical drivers and inhibitors of next-generation mobility services. Automotive cyber-

security is a relatively young field, with the first major publications appearing in 2010 [58, 109]. In 2015, several attacks were reported, including three major wireless attacks: an attack on BMW Connected Drive [166], an attack on GM OnStar [55], and the Tesla Door Attack [134]. Although the first two attacks received some attention, it was not until Miller and Valasek's Jeep attack [127] that automotive cybersecurity was perceived as a mainstream research and engineering issue. This attack exploited vulnerabilities in the wireless Telematic Control Unit (TCU) and In-Vehicle Infotainment (IVI) system to allow for remote control of a vehicle. In the first-generation of automotive security research, attacks were mounted through vehicles' physical interfaces, e.g., through the OBD-II port or wired interfaces on the IVI. Meanwhile, remote or "wireless" attacks exploit wireless interfaces, such as the Bluetooth, Wi-Fi, or cellular connections of the TCU, as in the aforementioned Jeep attack.

A commonality between wired and wireless attacks is the need to eventually inject messages onto the CAN bus in order to make the vehicle act in an undesired or unexpected way. Even in the sophisticated Jeep attack, the researchers had to manually reverse-engineer portions of the CAN bus protocol in order to gain remote control over the vehicle, e.g., over its steering control. This is very tedious and unscalable. Additionally, these attacks can usually only target a specific model or make of vehicle since message semantics are OEM-proprietary and can even differ from model to model of the same vehicle make. Academic offensive automotive cybersecurity research suffers greatly from this lack of scalability. Although most defensive solutions, such as Intrusion Detection Systems (IDSs) [61, 84, 102, 183], do not require knowledge of the message semantics of a vehicle, a straightforward and automated mechanism to reverse-engineer CAN bus data could greatly accelerate vulnerability research and allow software patches to be distributed before malicious entities become aware of vulnerabilities.

The current *security through obscurity* paradigm pursued by OEMs attempts to

prevent wide-scale automotive attacks by keeping CAN message translation tables, called *DBC files*, secret (and therefore placing an additional barrier to vehicle hacking) is outdated and infeasible. Vehicles should be secure *by design* and not *by choice*, following *Kerckhoffs's principle* [105]. Therefore, automotive Electrics/Electronics (E/E) architectures and networks should be resilient against CAN injection attacks originating from external sources, e.g., by firewalling messages from the OBD-II port, and without making assumptions about the knowledge of an attacker.

In this thesis chapter, we propose `LibreCAN`, a tool to automatically translate most CAN messages with minimal effort. Unlike prior limited research on automated CAN reverse-engineering, `LibreCAN` not only focuses on powertrain-related data available through the public OBD-II protocol, but also leverages data from smartphone sensors, and furthermore reverse-engineers body-related CAN data. To the best of our knowledge, `LibreCAN` is the first system that can reverse-engineer a relatively complete CAN communication matrix for any given vehicle, as well as the full-scale experimental evaluation of such a system.

This chapter is organized as follows. Sec. 2.2 gives a primer on the CAN bus, its typical messages and signals, and the interpretability of CAN data, as well as in-vehicle network architecture. Sec. 2.3 details the design of `LibreCAN`, while Sec. 2.4 evaluates the accuracy, coverage, and required manual and computation time for reverse-engineering CAN messages. Sec. 2.5 discusses the limitations and potential other use-cases of `LibreCAN`, as well as possible countermeasures. Sec. 2.6 discusses related efforts in manual and automated CAN reverse-engineering, while Sec. 2.7 concludes the chapter.

2.2 Background

Please refer to Sec. 1.2 for a primer on the CAN bus, DBC files and in-vehicle network architectures.

2.2.1 Information Sent on the CAN Bus

In order to know which data to reverse-engineer, we must first determine the information commonly available in vehicles. This depends greatly upon the age and price of the vehicle, and can drastically differ even among comparable vehicles from different OEMs. As a result, we must first establish a basic knowledge of the most frequently deployed ECUs in vehicles and the signals that they transmit on the CAN bus.

It is difficult to arrive at a deterministic answer to this question since this information is only located in DBC files, which are proprietary to the OEMs. As a result, reverse-engineering *all* signals present in a vehicle is nearly impossible. Thus, our goal is to reverse-engineer the most common subset of vehicular signals that are of interest to both security researchers and third-party app developers. [62] provides an overview of the automotive electronic systems present in a typical vehicle. After analyzing multiple sources [124, 125, 127], we derived a list of ECUs (Table A.1 in Appendix A.1) typically present in a vehicle (each of which usually transmits data using one or more CAN message IDs), along with the signals present in their respective CAN messages.

Raw CAN data is not encoded in a human-readable format and does not reflect the actual sensor values. In order to obtain the actual sensor values, raw CAN data must first be decoded [65]. Letting r_s, m_s, t_s, and d_s be the raw value, scale, offset, and decoded value of sensor s, respectively, the actual value can be found with the following equation:

$$d_s = m_s \cdot r_s + t_s. \tag{2.1}$$

2.3 System Design

Fig. 2.1 provides an overview of `LibreCAN`'s system design, which consists of three phases discussed below. Our system relies upon the following three sets of signals as input:

- P: The set of IMU sensor data (called "motion sensors" in Android), i.e., 3-dimensional accelerometer and 3-dimensional gyroscope data collected from the smartphone (via the Torque Pro app) while recording OBD-II data (V).
- V: The set of OBD-II data. It consists of all OBD-II PIDs that the vehicle supports. The sampling rate depends on the used data collection dongle and vehicle. As a result, we resample the data to 1 Hz. A full list of OBD-II PIDs can be found in [9].
- R: The set of raw CAN data that we recorded with the OpenXC dongle. It includes the entire trace of driving data broadcasted on the CAN bus and is accessible through the OBD-II port.

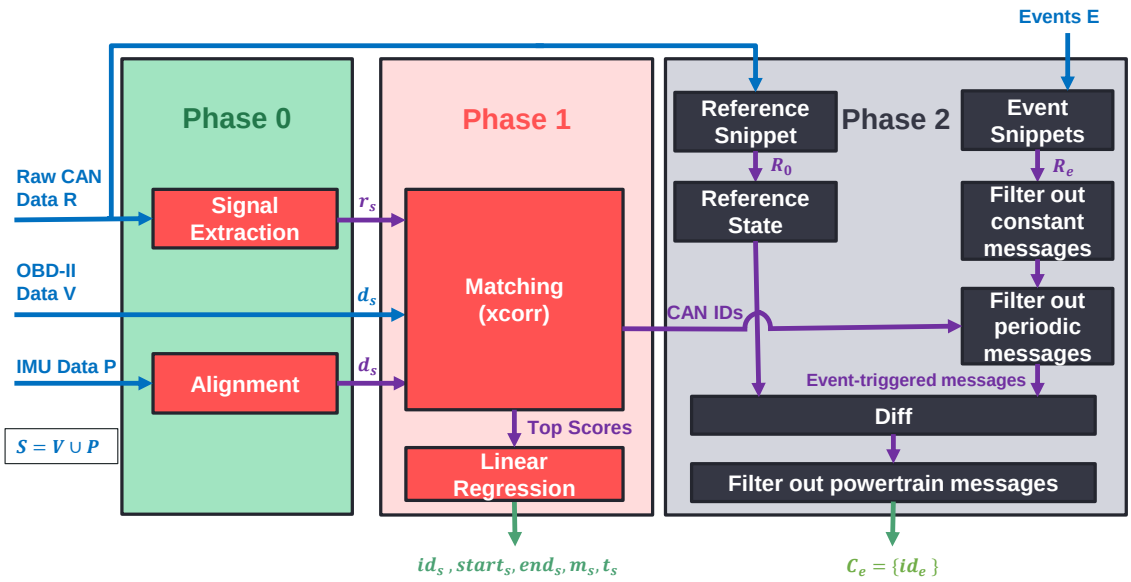


Figure 2.1: System design overview

Data from sets P and V are only used in Phase 1. As shown in Table A.2, we have 9 IMU sensors $\in P$ and 15 OBD-II PIDs $\in V$ that we are analyzing. As we will see later, OBD-II PIDs only cover less than 2% of the possible signals that can be reverse-engineered on each of our evaluation vehicles.

2.3.1 Phase 0: Signal Extraction

As described in Sec. 1.2.2, CAN messages can contain multiple signals, and hence we need to extract the signals associated with each CAN ID. We built the signal extraction mechanism in this phase on top of the `READ` algorithm in [120].

Using the rate at which the value of each bit changes, `READ` determines signal boundaries under the assumption that lower-order bits in a signal will more likely change *more frequently* than higher-order bits. READ then labels each extracted signal as either a counter, a cyclic redundancy check (CRC), or a physical value based upon other characteristics of the bit-change rate of the particular signal. Counters are characterized by a decreasing bit-flip rate, with the latter approximately doubling as the significance of the bit rises. Meanwhile, CRCs are characterized by a bit-change magnitude of approximately 0. Physical signals (`PHYS`) are those that do not fit into any of the above two categories.

We further defined three special types of physical signals: `UNUSED` (all bits set to 0), `CONST` (all bits constantly set to the same value across messages, but with at least one bit set to 1), and `MULTI` (the value of the signal is from a set of n possible values).

We also modified the mechanism the READ algorithm uses to determine signal boundaries. The original READ algorithm marks a signal boundary when the value of $\lceil \log_{10} \text{Bitflip} \rceil$ for a bit decreases as compared to the previous bit. However, our implementation of READ instead checks whether the bit-flip rate decreased by a specific percentage from the previous bit – this value was set via an input parameter to our algorithm, as discussed below. In this original implementation, pairs of consecu-

tive bits whose bit-flip rates change from (>.1 to <.1), (>.01 to <.01), or (>.001 to <.001) would indicate a signal boundary. However, with our modification, a change in bit-flip rate from 0.9 to 0.2 would only indicate a boundary with any percentage threshold less than 77%. We found that using a percentage decrease allowed us to extract more signals correctly than the original READ.

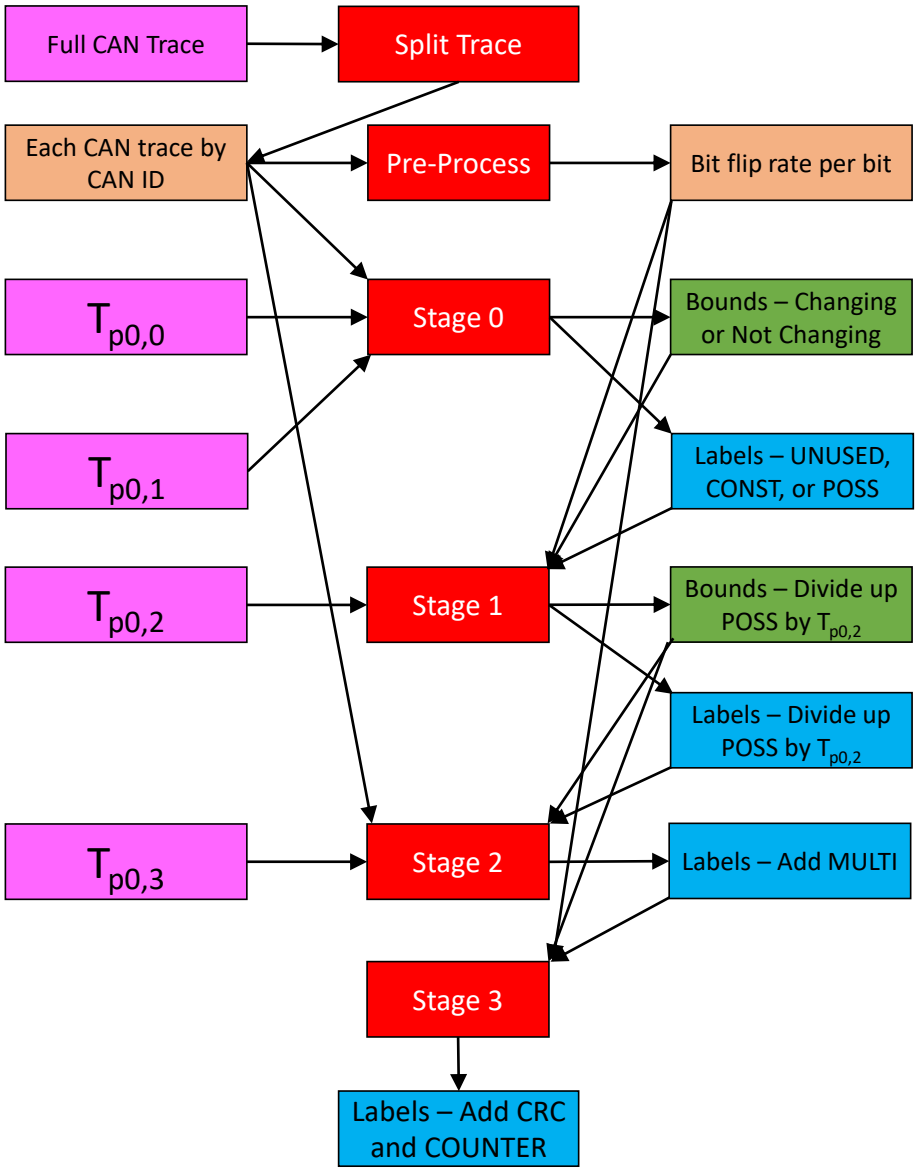


Figure 2.2: Flowchart of Phase 0 algorithm

A flowchart of the algorithm for this phase is provided in Fig. 2.2. The remainder of this subsection provides the details of the different stages of this algorithm. Stages

0 and 2 are our own enhancements to the `READ` algorithm [120].

Pre-Processing Stage: In this stage, we parse a CAN trace in order to obtain the bit-flip rate of each payload bit. To achieve this, we count the number of times the value of each bit changes in the payload of a given CAN ID and then divide this by the number of messages in the trace with this CAN ID.

Stage 0: This stage separates bits into three bins: `UNUSED`, `CONST`, and `POSS` (possibly a `COUNTER`, `MULTI`, `CRC`, or physical signal `PHYS`). This stage generates the preliminary signal boundaries and labels for each signal from the above three categories.

To achieve this, we first separate the bits from the previous stage into two sets: those that change and those that do not. These bits are then grouped together into signals with preliminary boundaries, assigning the boundaries based upon where regions of bits that change transition regions of bits that do not, and vice versa. The regions of bits that change are assigned the preliminary label of `POSS` and are left to be processed later. Meanwhile, the bits that do not change are processed using Alg. 1. We define two configurable parameters for the algorithm, namely $T_{p0,0}$ and $T_{p0,1}$. The former is the length that a signal must have to be considered an unused signal. If a signal is shorter than this length, we attempt to append it to the next signal. This is because we assume that, if there is a short unused field, it actually contains the MSBs of the adjacent signal for which we never observed a change in value. For example, if 8 bits are used to express the speed in MPH, the most significant bit would not change unless the trace included driving over 128 mph). We use $T_{p0,1}$ to determine how long the next signal must be in order to have bits appended to it in this manner. This is necessary since it does not make sense to always re-append unchanging bits as the MSBs of the next signal.

Stage 1: This stage is similar to `READ` and evaluates all possible signal boundaries and their bit-flip rates. We iterate from the LSB of a signal to the MSB of the next adjacent signal, searching for a decrease in bit-flip rate. However, unlike the `READ`

Algorithm 1 Stage 0

```
procedure stage0(trace_file, T_p0,0, T_p0,1)
    bits_that_dont_change_label ← []
    for l, r ∈ bits_that_dont_change do
        if True ∈ changes[l : r] then
            bits_that_dont_change_label.append(CONST)
            break
        else if r − l < T_p0,0 then
            reinserted ← false
            for l_c, r_c ∈ bits_that_change do
                if l_c == r + 1and r_c − l_c > T_p0,1 then
                    l_c ← l
                    reinserted ← false
                    delete l, c
                    break
            if reinserted == false then
                bits_that_dont_change_label.append(UNUSED)
```

algorithm, we are looking for a certain percentage decrease, denoted as $T_{p0,2}$. For example, if $T_{p0,2} = 10\%$, we would mark a signal boundary when the bit-flip rate decreases by greater than 10%. The output of this phase is an array of boundaries that contains all partitions within the boundaries of the previously marked `POSS` signals. This output contains the final signal boundaries that are used in the rest of our evaluations.

Stage 2: This stage evaluates all signal boundaries marked `POSS` and determines the number of unique values they contain throughout the trace. To achieve this, we parse through the trace to determine the number of unique values that each extracted signals from Stage 1 is set to — if this number is less than a pre-determined threshold ($T_{p0,3}$), the signal is not considered in future stages. Any remaining `POSS` signals at the end of this stage are marked as `MULTI` values. The output of this phase is a new signal labeling set, now additionally containing signals labeled as `MULTI`.

Stage 3: This stage is also similar to the `READ` algorithm and evaluates any values still labeled as `POSS` to determine if their bit-flip rates resemble a counter. If this is not the case, we label the signal as a `PHYS` value.

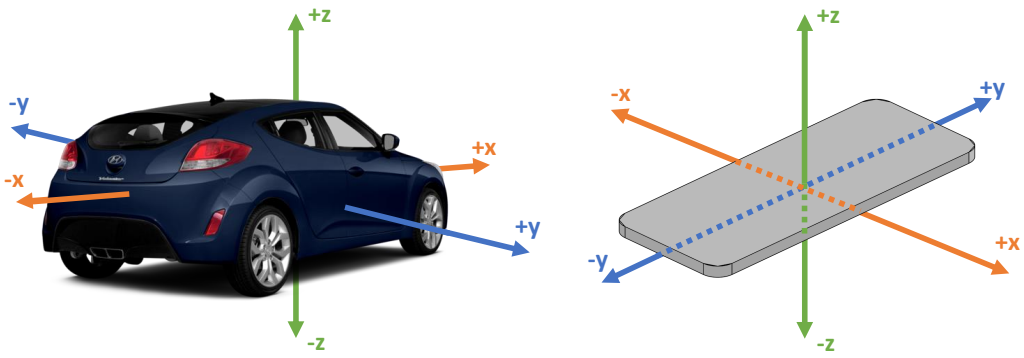


Figure 2.3: Alignment of phone's coordinate system (right) with vehicular coordinate system (left)

Alignment: Phase 0 also encompasses phone alignment. As Fig. 2.3 shows, the vehicular coordinate system is not necessarily consistent with the phone's coordinate system, particularly if the user moves their phone during the data-collection process. Therefore, it may be necessary to align these coordinate systems using rotation matrices, as discussed in [59]. In order to avoid this additional step, we suggest that users pre-align their phone with the vehicular coordinate system by mounting the phone inside their vehicle, e.g., in a phone/cup holder. Using the coordinate systems from Fig. 2.3, the phone should be located on the center console, with the short edge parallel to the direction of the vehicle's motion.

2.3.2 Phase 1: Kinematic-related Data

The goal of this phase is to match the extracted signals from Phase 0 to openly available OBD-II PIDs (V), as well as mobile sensor data (P). The latter data can easily be collected using a smartphone.

The OBD-II PIDs (V) and IMU sensors (P) that we consider from our data collection with Torque Pro — making up the set S (see Fig. 2.1) — are depicted in Table A.2. The commonality between these signals (i.e., V, P, and S) is that they are kinematic- or powertrain-related, i.e., they are captured while the vehicle is in motion. The OBD-II protocol was standardized for the purpose of capturing and diagnosing

emissions data, which is powertrain-related. The IMU sensors capture the movement of the smartphone in the vehicle and therefore the movement of the vehicle, if the phone is fixed within the vehicle and properly aligned. These signals are also present on the CAN bus since this data is generated by and exchanged between ECUs, with a copy mirrored out to the OBD-II connector.

As mentioned in Eq. (2.1), CAN signals usually do not encode an absolute value, but instead a value with a linear relationship to the latter. As a result, comparing the temporal sequence of a raw CAN signal from set R and a signal from set S should yield a high cross-correlation value. Hence, for each signal $d \in S$, we run *normalized cross-correlation* (`xcorr`) with all extracted signals $r \in R$, which yields a list of cross-correlation values. We then arrange them in a descending order with respect to the cross-correlation value. Since multiple CAN signals r can match a signal d (e.g., the four wheel speeds match the OBD speed), we need to define an intelligent cut-off point that keeps those relevant signals d with a high correlation value, but deletes those starting with a correlation score that deviates significantly from the last signal d that we wish to remain. For this purpose, we define a threshold T_{p1}. Alg. 2 describes how to set the cut-off point. We will experiment with T_{p1} in Sec. 2.4.2 to achieve the best precision and recall for Phase 1.

Algorithm 2 Defining the Cut-Off Point

function $Top\_X(corr\_result, T_{p1})$
  $running\_sum, running\_avg, cutoff \leftarrow corr\_result[0]$
  $count \leftarrow 1$
  for $val \in corr\_result[1:]$ **do**
    if $val < (1 - T_{p1}) \cdot running\_avg$ **then**
      break
    $cutoff.append(val)$
    $running\_sum \leftarrow running\_sum + val$
    $count \leftarrow count + 1$
    $running\_avg \leftarrow \frac{running\_sum}{count}$
  return $cutoff$

It is essential to re-sample the two input sets R and S before running `xcorr` so

that both signals are temporally aligned.

Some of these signals are highly correlated with each other so that they can be matched to the same CAN signal extracted in Phase 0. For instance, engine load is a scaled version of the engine output torque. As a result, while generating our ground truth for each vehicle, we need to consider these physical relationships and confirm that they indeed hold during the evaluation of Phase 1. The reason behind this lies in the `xcorr` function that we use in the aforementioned phase. It cannot distinguish between different physical signals as long as their temporal sequences are similar. This is a limitation of Phase 1 and is left as part of our future work. See Appendix A.1 for a complete summary of relationships between certain elements in set S.

The goal of Phase 1 (apart from finding the correct CAN signal positions) is to output the scale (m_s) and offset (t_s) of each sensor (s). We can use linear regression on the *matched* CAN signals R and signals from S to obtain these values. The latter can also be validated against the ground truth DBC file, but this is omitted from our evaluation.

To a greater extent, we are interested in comparing the *matched* signal positions from before against the ground truth in order to determine the accuracy of our algorithm in Phase 1. For this classification task, we define a confusion matrix as shown in Table 2.1.

Table 2.1: Confusion Matrix for Phases 1 and 2

		Ground Truth	
		Positive	Negative
Results from Phases 1 & 2	Positive	**TP** Phase 1: Signals that are correctly identified as part of the ground truth Phase 2: Candidate CAN IDs that were correctly identified as being related to an event	**FP** Phase 1: Signals that are incorrectly identified and are not part of the ground truth Phase 2: Candidate CAN IDs that were incorrectly identified as being related to an event
	Negative	**FN** Phase 1: Signals that are not identified, but are part of ground truth Phase 2: CAN IDs that were incorrectly rejected during the filtering process	**TN** Phase 1: Signals that are not identified, but are also not part of ground truth Phase 2: CAN IDs that were correctly identified as not being related to an event

2.3.3 Phase 2: Body-related Data

Phase 2 consists of a three-stage filtering process performed on snippets of CAN data recorded while performing body-related events. These events R_e, $e \in E$ are listed in Table A.3.

A reference snippet R_0 was recorded while the vehicle's engine/ignition was off, but with accessory power on. A reference state, used later in the filtering process, was generated using this snippet. In this section, we will describe how to generate the reference state from R_0.

In Eq. (2.2), we first count the number of bit-flips (BFC_j) in consecutive messages $m_{n,i,j} \in id_n$ for that particular CAN ID (id_n) in each of its 64 bit-positions $j \in [0, 63]$:

$$BFC_{n,j} = \sum_{i=0}^{|id_n|-1} 1, \forall j \in [0, 63] \text{ and if } m_{n,i,j} \neq m_{n,i-1,j}. \tag{2.2}$$

Then, we define the bit-flip array ($BFA_{n,j}$) for a particular CAN ID (id_n) in each of its bit positions:

$$BFA_{n,j} = \frac{BFC_{n,j}}{|id_n|}. \tag{2.3}$$

Finally, we define the bit-flip rate (BFR_n) of a CAN ID (id_n) as:

$$BFR_n = \frac{\sum_{j=0}^{63} BFA_{n,j}}{64}. \tag{2.4}$$

Note that the above bit-flip rate BFR_n is different from the one defined in Phase 0. The reference state contains a mapping of CAN IDs id_n to message payloads that have a bit-flip rate lower than, or equal to a threshold $T_{p2,0}$ ($BFR_n \leq T_{p2,0}$), since messages that change less frequently are more likely to be constant or alternating between a few constant states. Messages that change more frequently, as evidenced by $BFR_n > T_{p2,0}$, are less likely to be associated with a single body-related event, especially because the reference snippet R_0 was recorded without any human interaction in the

vehicle that could have triggered body events.

STAGE 1: Constant Messages
STAGE 2: Reference Messages
STAGE 3: Powertrain Messages

TRACE

TIME	ID	PAYLOAD	FILTERED IN
00.000	700	1111111100000000	STAGE 3
00.001	100	0000000000000000	CANDIDATE
00.002	300	000002000E20BE20	STAGE 1
00.004	900	FFFFFFFFFFFFFFFF	CANDIDATE
00.008	300	000002000E20BE20	STAGE 1
00.009	300	000002000E20BE20	STAGE 1
00.011	600	000000024CB016EA	STAGE 2
00.015	800	00000000075BCD15	CANDIDATE
00.016	500	0000000000000000	STAGE 3
00.018	400	056089000A00A000	STAGE 2
00.020	200	0000000000000000	CANDIDATE

REFERENCE

ID	PAYLOAD
100	0000A00A000BC300
200	0070070070070070
300	00000000075BCD15
400	056089000A00A000
500	0012300AE0030000
600	000000024CB016EA
700	1000000001100001
800	00000000000000FF
900	0F00B9900A0A0F0E

POWERTRAIN

ID	CORRELATION SCORE
100	0.7433
200	0.5192
300	0.7990
400	0.6648
500	0.9882
600	0.7102
700	0.8361
800	0.1034
900	0.2023

Figure 2.4: Phase 2 Filtering Example

Fig. 2.4 depicts an example of the filtering process in Phase 2. The event snippet is shown in the `TRACE` section and the generated reference state is shown in the `REFERENCE` section.

After generating the reference state, each event snippet R_e was filtered through three separate stages, each designed to independently identify potential candidate CAN IDs. The order of these filtering stages was set based upon extensive evaluation

to achieve the highest accuracy. Stages 1, 2, and 3 operate under the assumption that body-related events should trigger visible and immediate changes in the messages broadcast on the CAN bus.

Stage 1: Filtering messages with constant payloads. We assume that body-related events should trigger changes in message payloads for at least one CAN ID, so we removed all CAN IDs whose payloads did not change throughout the snippet. As an example, in Fig. 2.4, messages with a CAN ID of 300 were filtered out at this stage because all payloads sent in the event snippet were the same.

Stage 2: Filtering messages present in the reference state. We removed candidate messages if their CAN IDs and payloads matched a (CAN ID, payload) pair found in the reference state. If a candidate's payload from the event snippet was identical to the reference state, when no body-related events occurred, it is highly unlikely this message was sent due to a change in the state of the vehicle's body. This stage can be considered a *diff* between the reference state and each event R_e. In Fig. 2.4, messages with the (CAN ID, payload) pairs (400, 056089000A00A000) and (600, 000000024CB016EA) were filtered out because they were present in the reference state. Furthermore, we found better results obtained by rejecting candidates whose CAN IDs were not present in the reference state.

Stage 3: Filtering messages which were likely powertrain-related. To reduce the quantity of remaining candidates, we removed those CAN IDs that were identified as potential candidates for powertrain-related events in Phase 1. This was possible since there was little overlap between the events being identified in both phases. To minimize the removal of candidates that were mistakenly classified as powertrain-related in Phase 1, we only removed CAN IDs if their correlation scores from Phase 1 were higher than a threshold ($T_{p2,3}$). The correlation scores for each CAN ID in the example in Fig. 2.4 can be observed in the section `POWERTRAIN`. In such a situation, messages were filtered out at this stage if their correlation scores were greater than

0.80.

Finally, those messages that were not filtered out are considered the candidates for that particular event snippet. In Fig. 2.4, the (CAN ID, payload) pairs that were not filtered out are labeled `CANDIDATE` in the `TRACE` section. Eventually, we need to compare the results obtained from our intelligent filtering algorithm against the ground truth. As in Phase 1, a ground truth needs to be created from manual inspection of the DBC files for each test vehicle — a confusion matrix is defined for this classification task in Table 2.1.

2.4 Evaluation

2.4.1 Data Collection

Four vehicles are used for our evaluation, all from the same OEM: Vehicle A is a 2017 luxury mid-size sedan, Vehicle B is a 2018 compact crossover SUV, Vehicle C is a full-size crossover SUV while Vehicle D is a full-size pickup truck. We have acquired DBC files for all four vehicles and used them as the ground truths against which to compare the results of `LibreCAN`. Vehicles A, C and D have at least two HS-CAN buses, both of which are routed out to the OBD-II connector, whereas Vehicle B has at least one HS-CAN and one MS-CAN, with only the former being accessible via OBD-II.

We collected two types of data: Free driving data for an hour with each vehicle (for Phase 1) as well as event data for reverse-engineering body-related events (for Phase 2). For the former, data was collected through the OBD-II port with two devices: an ELM327 dongle and an OpenXC dongle. A Y-cable was used to allow both devices to connect to the port at the same time, allowing us to gather raw CAN data via the OpenXC dongle, while simultaneously gathering OBD-II data and smartphone data via the ELM327 dongle. The recorded CAN dump consists of raw JSON data with

CAN message metadata such as the CAN ID and timestamp, along with the payload data. We used the Torque Pro Android app to interface with the ELM327 dongle via Bluetooth. This produced a CSV file with around 22 signals $d \in S$, containing both OBD-II PIDs V as well as mobile sensor data P (see Table A.2). For Phase 2, we solely used the OpenXC dongle to record raw CAN data.

2.4.2 Accuracy and Coverage

In the previous subsection, we introduced several parameters for each phase x that are denoted as $T_{px,y}$, where y is an incremental number. Besides tuning these parameters to achieve the highest accuracy, another design goal is to find a set of parameters for each vehicle — henceforth called *parameter configuration* — that does not significantly differ from the configuration of other vehicles. In a real-world use-case of `LibreCAN`, DBC files are not available, and thus the parameters cannot be tuned to achieve optimal performance. So, we would like to show the existence of a universal configuration that can achieve good performance on any vehicle without any prior knowledge of its architecture or DBC structure.

Phase 0: Signal Bounds Accuracy and Reverse-Engineering Coverage. To evaluate how well our implementation and enhancements to the `READ` algorithm's extracted signal boundaries, we compared the boundaries produced by Phase 0 with the ground truth boundaries extracted from the DBC files for both vehicles. To find the *optimal* values of the four parameters defined in Section 2.3.1, we performed a brute-force search through all possible combinations as depicted in Table 2.3. For Phase 0, we defined *optimal* as the total number of correctly extracted signals (`CE`). We sorted all *parameter configurations* in a descending list by this metric. For the maximum number of `CE`, we manually inspected these configurations among all four vehicles for similarity and selected the configurations with the smallest distance to

each other. As shown in the first four columns of Table 2.3, the numbers of each 4-tuple configuration are very close to each other.

The results of the run with the optimal parameters for Phase 0 are summarized in Table 2.2. It shows the number of correctly extracted signals (`CE`) that we optimized our parameter configurations for, the number of total extracted signals (`TE`) and the total number of signals in the DBC files (`TDBC`). Note that Vehicle B has a lower number of `TDBC` since we can only reverse-engineer one CAN bus (the second one is not available through the OBD-II port). We define two ratios: `CE/TE` and `TE/TDBC`. The latter can be defined as reverse-engineering coverage. `LibreCAN` can always extract more than half of the available signals, with varying success for the number of correctly extracted signals. There are multiple reasons for these less than desirable numbers.

Table 2.2: Phase 0 Evaluation Metrics

Veh.	**Correctly Extracted (CE)**	**Total Extracted (TE)**	**Total in DBC (TDBC)**	**CE / TE**	**TE / TDBC**
Veh A	308	846	1640	36.4%	51.6%
Veh B	95	453	829	21.0%	54.6%
Veh C	208	698	1236	29.8%	56.5%
Veh D	251	828	1327	30.3%	62.4%

First, not all signals can be triggered in the recordings. Although we use both free driving and event data for signal extraction in Phase 0, it is impossible to capture everything, e.g., deployed airbags or emergency call signals. Since all our evaluation vehicles were newer with several features and also not the highest trim level for that particular model, the number of functionalities and thus signals is relatively higher than an older vehicle. This explains the `TE/TDBC` ratio. Second, it is not always possible to match the exact signal boundaries to the ground truth DBC file. For instance, the engine speed (RPM) range can go up to 8000 RPM in most vehicles.

Under normal driving conditions with an automatic transmission, the vehicle will shift to the next gear in the range of 2000–3000 RPM. As a result, we will miss the most significant bits of that particular signals. The same applies to another physical signals, such as vehicle speed or engine coolant temperature. This will intrinsically result in a low `CE/TE` ratio.

As a result, the aforementioned ratio in Table 2.2 should not be used to draw conclusions about the performance of `LibreCAN` since the signals inspected in Phases 1 and 2 yield high accuracy numbers.

Table 2.3: Optimal Parameters in `LibreCAN`

	$T_{p0,0}$	$T_{p0,1}$	$T_{p0,2}$	$T_{p0,3}$	T_{p1}	$T_{p2,0}$	$T_{p2,3}$
	[0,64]	[0,64]	[0,1]	[0,64]	[0,1]	[0,.1]	[.2,1]
Veh. A	0	3	0.02	2	0.05	0.03	0.70
Veh. B	2	3	0.01	2	0.07	0.03	0.70
Veh. C	0	4	0.01	2	0.05	0.03	0.55
Veh. D	2	3	0.01	2	0.06	0.02	0.60

Phase 1: Correlation Accuracy. We analyzed the accuracy of Phase 1 both independently from Phase 0 (using correct signal boundaries from the DBC files) in order to avoid possible error propagation, as well as with the extracted signal boundaries from Phase 0.

Using the terminology from the confusion matrix in Table 2.1, we defined the following metrics to assess for Phase 1:

- Accuracy $= \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}$
- Precision $= \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$
- Recall $= \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$

In Phase 1, we introduced one parameter that can be tuned to achieve the best performance. This parameter is the threshold T_{p1} to define the cut-off point, defined previously in Sec. 2.3.2. One mechanism to define the optimal value for T_{p1} is via the *Receiver Operating Characteristic* (ROC) curve. Since we have an unbalanced ground truth (e.g., the speed contains more CAN signals r than altitude), a *Precision-Recall* (PR) curve is a better option. Fig. 2.5 shows the PR curve for both vehicles. Each data point depicts a value of $T_{p1} \in [0, 1]$.

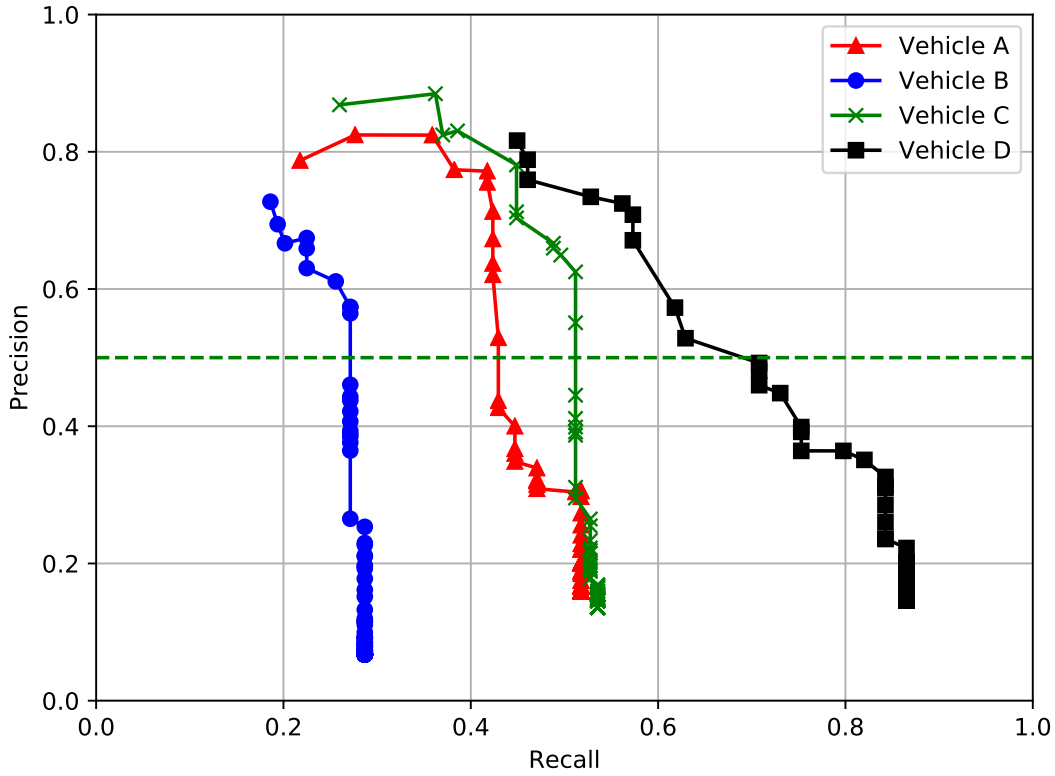


Figure 2.5: Precision-Recall Curve for Phase 1

The closest data point to the upper right corner delivers the optimal threshold T_{p1} for the best performance. The PR curve depicted in Fig. 2.5 does not have an ideal shape for Vehicles A, B and C because the recall value never exceeds 0.55. According to the above definition of recall, this means that the *True Positives* (TP) are always smaller than the number of *False Negatives* (FN), i.e., the ground truth

contains CAN signals that can never be found by our algorithm. Since the ground truth is a subjective interpretation which we generated by manual inspection of the DBC files, we assume that some CAN signals r are unrelated to the analyzed signal d. This is a limitation of our work since we could not receive the OEM's help in interpreting the DBC files. Some examples where we encountered this phenomenon are the z-component of accelerometer, altitude and bearing (all from phone). The former two can be explained by the fact that all our driving took place in a relatively flat area without many hills. The latter could be caused by GPS issues since bearing is collected from the phone's GPS module.

The first part of Table 2.4 sums up the precision and recall values using the optimal threshold T_{p1} (see Table 2.3) obtained from the PR curve analysis. entering The precision and recall values reflect the evaluation of Phase 1 with correct bounds

Table 2.4: Phases 1 and 2 Evaluation Metrics

	Phase 1		**Phase2**		
	Prec.	Recall	Acc.	Prec.	Recall
Vehicle A	82.6%/ 77.2%	44.1%/ 41.8%	88.0%	8.9%	58.2%
Vehicle B	66.7%/ 61.1%	26.4%/ 25.6%	90.1%	8.5%	46.2%
Vehicle C	74.4%/ 78.1%	45.7%/ 44.9%	91.5%	11.7%	51.6%
Vehicle D	79.7%/ 70.8%	61.8%/ 57.3%	95.1%	15.0%	47.2%

in the first line and with the signal bounds from Phase 0 in the second. The latter values are shown to be slightly lower for all vehicles, with the exception of Vehicle C. High precision values mean that most of the identified signals are part of the ground truth, whereas relatively low recall values mean that we cannot match the majority of signals defined in our subjective ground truth due to the high number of FNs, as mentioned previously.

The anomaly for Vehicle C can be explained as follows: With more signals available for the run with correct boundaries, Phase 1 over-identifies signals and causes a higher number of *false positives* for that specific vehicle. This is certainly possible.

Phase 2: Candidate Accuracy. The goal of this phase was to identify CAN IDs that were likely associated with a body-related event defined in Table A.3. To evaluate the results of our algorithm, we used metrics such as accuracy, precision, and recall. To evaluate these metrics, we need to revisit the terms from the confusion matrix in Table 2.1. Note that this is a coarser-grained analysis than Phase 1. We assessed how well Phase 2 identified the corresponding CAN IDs of events, not the signal position within a CAN message.

Our three-stage filtering process uses two input parameters that were defined in Sec. 2.3.3: 1. the bit-flip threshold ($T_{p2,0}$), used to generate the reference state and 2. the powertrain minimum correlation score ($T_{p2,3}$), used in the powertrain filtering stage.

We ran the collected event traces through Phase 2 for each *parameter configuration*, calculating the accuracy, precision, and recall metrics for each event. Since our goal was to facilitate the identification of potential candidate CAN IDs, we preferred those parameters that resulted in a high FP rate instead of a high FN rate — we wanted to avoid excluding a potential candidate from consideration. The optimal parameter values discovered for each vehicle are shown in the last two columns of Table 2.3.

The second part of Table 2.4 summarizes the mean values of our metrics for all 53 events while Fig. 2.6 shows the median number of CAN IDs remaining after each filtering stage (per event), as well as the total number of ground truth CAN IDs lost over all events at each filtering stage. As predicted, our *accuracy* is high since we filter out most unrelated CAN IDs for each event, whereas our *precision* is relatively low. The latter metric indicates the ratio of correct CAN IDs in the candidate set to

the total number of candidates. However, we do not consider low precision to be an issue. As Fig. 2.6 shows, we can reduce the number of CAN IDs after three filtering stages by more than 10x, despite losing some correct CAN IDs at each stage.

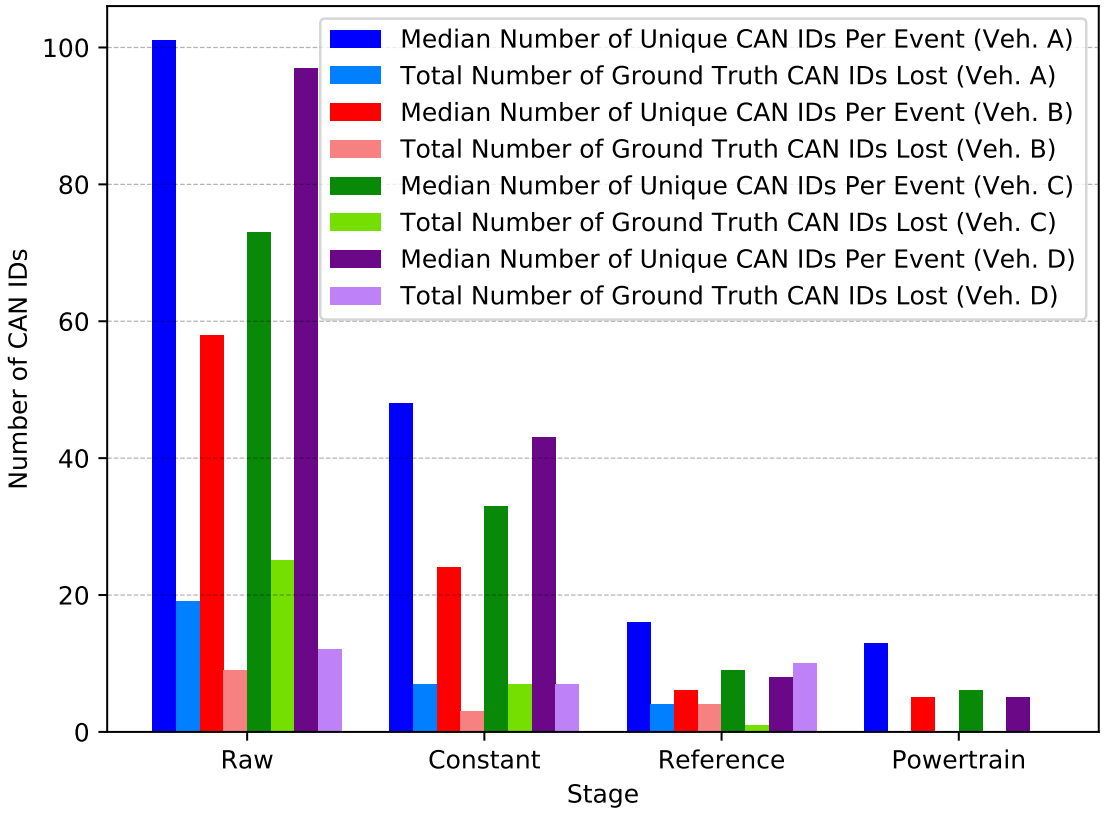


Figure 2.6: Filtering out CAN IDs in each stage

Additionally, some signals for body-related events were not available on the CAN buses we used for our evaluations. For instance, the signal for the horn was not available on the CAN bus of any vehicle we evaluated. We were unable to record data for 7 events for Vehicle A, 15 events for Vehicle B, 7 events for Vehicle C, and 10 events for Vehicle D. However, 10 of the events we were not able to record for Vehicle B were on the MS-CAN that was not accessible through the OBD-II port. We opted to not remove those events from our evaluation since it is likely that CAN data recorded on another vehicle would yield similar results.

2.4.3 Manual Effort

An important metric for demonstrating the feasibility of `LibreCAN` is the level of automation available, compared with the amount of manual effort required on the part of the user. Although all three phases in the system can run and generate results without human intervention, there is still manual effort required to collect input traces. The goal of `LibreCAN` is to enable every user to reverse-engineer the CAN message format of their vehicle with as little effort as possible. Hence, we want to assess how much data has to be collected for Phase 1 to yield a reasonable precision and how long it takes to record all 53 of the events used in Phase 2.

Phase 1. The recorded traces of all evaluation vehicles were around 60 minutes long. The precision reported in Sec. 2.4.2 reflects the entire re-sampled trace. We wanted to see how a shorter recording would affect this metric. We re-ran Phase 1 with signals obtained in Phase 0, with 25%, 50% and 75% of the trace length. In order to avoid a bias towards more city or highway driving, we calculated the precision for overlapping segments of this trace. For instance, to analyze recordings of only half the length of the original trace, we would use evaluate the following segments of the trace: 1. the first half of the trace, 2. the slice of the trace between the first and last quarters of its length, and 3. the last half The mean results of these evaluations are plotted in Fig. 2.7.

A reduction in trace length results in a slight precision drop for all vehicles except Vehicle B. The latter exhibits different behavior because a significantly higher number of signals were extracted with its 100% trace compared to the one in other vehicles — since a greater number of signals were extracted in Phase 0, a greater number signals were processed in Phase 1. Both the 75% and 100% traces for this vehicle yielded the same number of correct signals (our design goal in Phase 0), but the 100% trace resulted in more signals being processed (due to a higher number of total extracted signals), which increased the number of *false positives* and thus decreased

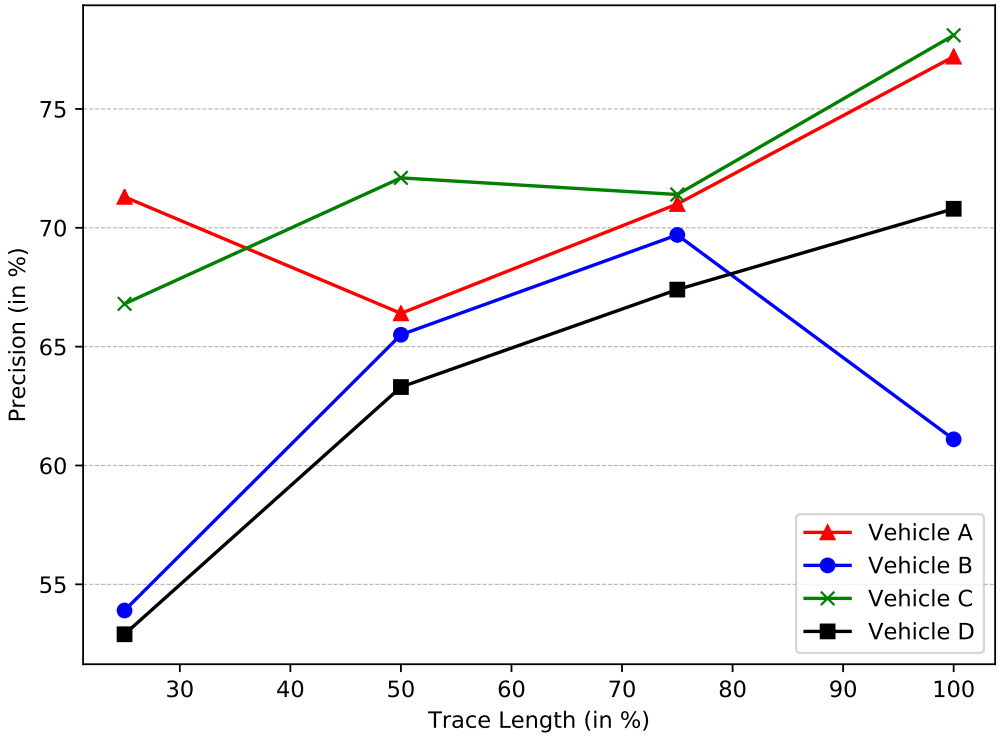
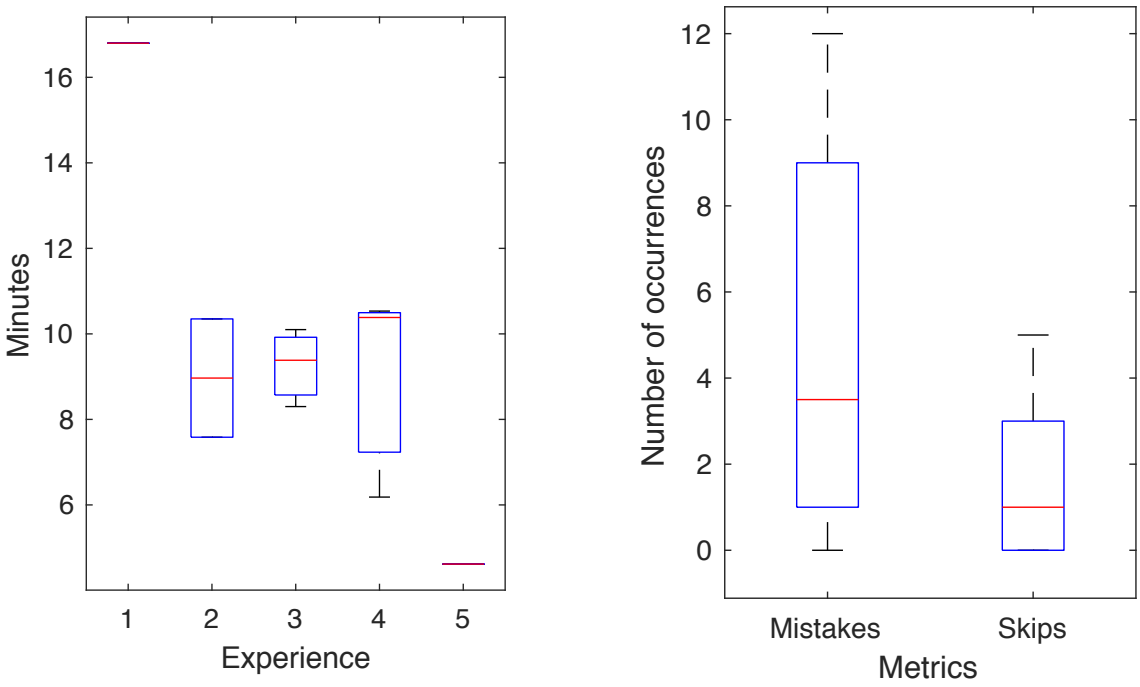


Figure 2.7: Precision of Phase 1 with varying trace lengths

the precision. In order to achieve at least 65% precision, we recommend using a trace covering 30 minutes or more.

Phase 2. In order to assess the time required to record all 53 events listed in Table A.3, we conducted a human-study experiment, for which we obtained an IRB approval (Registration No. IRB00000245). For this purpose, we developed an Android app that ran on top of CarLab [138]. The participant was required to interact with this app, which loops through all 53 events, displaying them one at a time on the screen. A timer begins with the start of recording for the first event and the participant, seated in the driver's seat, is instructed to perform each event and then click the *Next Event* button. The timer stops after the last event has been performed. During the experiment, a member of the study team sat in the passenger seat and evaluated participant's performance of the events, namely if one was performed incorrectly or skipped.



(a) Experience-time correlation

(b) Key metrics

Figure 2.8: Results in user-study experiment

A total of ten people participated in this experiment. They were instructed on how to operate the app and were not allowed to ask questions once the experiment began. After completing all events, the team member recorded how long the participants took and asked them how familiar they were with the test vehicle (Vehicle A) on a scale from 1 to 5, with 5 being the most familiar. Fig. 2.8 (a) summarizes the correlation between the level of experience with the time span. Note that the completion time was not affected much by the experience level, except for one totally inexperienced (1/5) and one very experienced (5/5) participant. Specifically, for users with experience levels ranging from 2 to 4, the median of their completion time varies between 9.0 to 10.4 minutes. Fig. 2.8 (b) shows the key behavioral metrics (i.e., number of mistakes and skips) of all participants. The median numbers of mistakes and skips are 3.5 and 1, respectively. As a result, drivers of different experience levels are capable of performing all 53 events with the median rates of error and skip at 6.6% (=3.5/53) and 1.9%, respectively.

In conclusion, we estimate that a 30 minute drive for Phase 1 and a 10 minute experiment session for Phase 2 are sufficient to produce good results. These numbers are feasible for an otherwise completely automated CAN reverse-engineering framework, especially given the time that manual reverse-engineering would likely take. The latter can take from days to weeks, given the detail and precision of the reverse-engineering needed. Although no explicit times are reported for manual reverse-engineering, tutorials [161] imply significant effort is required. However, researchers from the well-known Jeep hack [127] provide a reference in their paper: "(...) we spent an entire year figuring out which messages to send for the Ford and Toyota (...)". Although they very likely did not spend that entire time frame for reverse engineering of CAN messages, it shows that is not a trivial process and takes a lot of experimenting to find the correct payload for their CAN injection attack.

2.4.4 Computation Time

Having discussed the manual effort required to use `LibreCAN`, we analyze the computation time of all three phases individually.

All experiments were conducted using Python 3 on a computer running 64-bit Ubuntu 16.04. This computer featured 128 GB of registered ECC DDR4 RAM and two Intel Xeon E5-2683 V4 CPUs (2.1 GHz with 16 cores/32 threads each). Phase 0 utilizes all available computational resources (64 threads), whereas Phase 1 uses one thread per signal d plus one main thread (23 threads). Meanwhile, the computationally inexpensive Phase 2 runs in a single thread.

Table 2.5 reports the time required for all computation steps. Note that these values have been generated for a run with the optimal parameter configuration. The total runtimes include operations that finished in less than one second, which are listed as completing in 0 seconds in Table 2.5.

The entire three phase automated process takes 79 seconds for Vehicle A, 74

Table 2.5: Summary of computation time in each phase and stage (units are in seconds)

Phases	**Stages**	**Veh A**	**Veh B**	**Veh C**	**Veh D**
Phase 0	Parse Raw CAN File	11	12	9	9
	Split Trace	2	2	2	2
	Remove Unused Columns	0	0	0	0
	Extract Signals	4	9	5	5
	Move Small Files	0	0	0	0
	Total	17	23	16	16
Phase 1	Run Correlate	40	30	36	40
	Calculate Scale and Offset	17	18	16	13
	Total	57	48	52	53
Phase 2	Create Ref. State	0	0	0	0
	Filter Constant Messages	4	2	2	3
	Compare to Ref. State	0	0	0	0
	Filter Powertrain Related Messages	0	0	0	0
	Total	5	3	2	3
`Libre CAN`	Total	79	74	70	72

seconds for Vehicle B, 70 seconds for Vehicle C and 72 seconds for Vehicle D. All vehicles have a similar computation time, indicating that `LibreCAN`is highly efficient in reverse-engineering a vehicle's CAN bus (slightly more than 1 minute) with only a small amount of manual effort (around 40 minutes).

2.4.5 Testing on Generic Parameters

As mentioned before, `LibreCAN` was designed to achieve a good performance with a universal set of parameters in all three phases. In order to show that anyone can achieve a comparable performance as reported in the previous subsections without *a priori* knowledge of the parameters, we would like to introduce an accuracy analysis similar to the one in Sec. 2.4.2. Since one of our design goals was to select similar parameters among the four evaluation vehicles, we can now pick any configuration of these four vehicles for testing. We evaluated all four vehicles on parameters $T_{p0,0} = 2$, $T_{p0,1} = 3$, $T_{p0,2} = 0.01$, $T_{p0,3} = 2$, $T_{p1,0} = 0.05$, $T_{p2,0} = 0.03$, and $T_{p2,4} = 0.70$. The results are summarized in Table 2.6. A comparison with the optimal results for each vehicle in Table 2.4 shows that they are relatively similar. Through our design goals as well as exhaustive evaluation on four vehicles, we found a parameter configuration that can produce favorable results for any testing vehicle. This corroborate the *scalability* of `LibreCAN`.

Table 2.6: Phases 1 and 2 Evaluation Metrics for Generic Parameters

	Phase 1		**Phase2**		
	Prec.	Recall	Acc.	Prec.	Recall
Vehicle A	77.2%	41.8%	88.0%	8.9%	58.2%
Vehicle B	65.9%	22.5%	90.1%	8.5%	46.2%
Vehicle C	78.1%	44.9%	91.5%	11.7%	51.6%
Vehicle D	72.5%	56.2%	94.6%	13.7%	47.2%

2.5 Discussion

2.5.1 Limitations and Improvements

During the evaluation phase, we discovered some limitations of `LibreCAN`. First, not all possible values of a kinematic-related signal will be "exercised" with normal driving behavior. For instance, RPM values over 3000 are unlikely due to the nature of automatic transmissions, except in cases of aggressive acceleration. We tried to compensate for this in Phase 0 by classifying signals as correct even if we missed 20% of the *Most Significant Bits* (MSBs).

Second, for Phase 2, not all vehicles may have the 53 events defined in Table A.3. We conducted experiments on newer vehicles, but cannot guarantee that older vehicles will have the same functionalities. These events are present on the IVN, but cannot be accessed via the OBD-II port. A possible solution to this problem would be to physically tap into the CAN bus by opening compartments. However, this voids the vehicle's warranty, and hence is not feasible for average drivers.

Third, our accuracy evaluations are somewhat subjective (as discussed earlier) despite their reflection of inputs from multiple other researchers. The only way to address this subjectivity would be to involve the vehicle OEMs.

One can also make some improvements to `LibreCAN`. For instance, a fine-grained analysis could be performed in Phase 2 to identify the correct regions of the events within a CAN ID. Signal extraction in Phase 0 could also be enhanced by leveraging the *Data Length Code* (DLC) field in the CAN header (see Fig. 1.3). Finally, we could construct additional d signals that are not directly available on SAE J/1979 or mobile phones. For example, *steering wheel angle* (SWA) is a popular signal (especially in AVs) that we could reconstruct using the gyroscope readings from a phone [114].

2.5.2 Other Use-Cases of `LibreCAN`

The main use-case of `LibreCAN` is as a tool for security researchers or (white-hat) hackers. It can help them lower the car-hacking barrier and allow vulnerabilities to be exploited faster. Another potential use-case we envision for `LibreCAN`is as a utility to enable the development of apps for vehicles, both in industry and academia.

Big data generation and sharing will lead to the monetization of driving data and create an additional source of revenue for OEMs and services. According to PwC, by 2022 the connected car space could grow to $155.9 billion, up from an estimated $52.5 billion in 2017 [174]. OEM-independent, universal access to data by third-party service providers can make the latter a major player in automotive data monetization. Third-parties already offer OBD-II dongles that can access the in-vehicular network and obtain publicly available data (OBD-II PIDs [9]). In particular, usage-based insurance (UBI) companies [4, 31, 33, 46] are known to distribute dongles to track driving behavior, allowing them to adjust insurance premiums. As mentioned previously, CAN data contains richer information than OBD-II PIDs and can be leveraged to build more powerful third-party apps. This also encompasses academic research, which usually has limited knowledge about vehicular data collection.

Table 2.7: Comparison to Related Work

	LibreCAN			READ [120]			ACTT [179]		
	Phase 0	Phase 1	Phase 2	Phase 0	Phase 1	Phase 2	Phase 0	Phase 1	Phase 2
Precision (Phase 0 & 1) Accuracy (Phase 2)	36.4%	82.6%	95.1%	97.1%	-	-	16.8%	47.7%	-

2.5.3 Countermeasures

Our point of entry to vehicles was the OBD-II port. Although we only read data from this port (OBD-II and raw CAN data), it is possible to inject CAN data into the vehicle via this port, as shown by [109, 124, 127]. A very simple and intuitive, but also powerful, solution to this attack would be to implement access control into the vehicular gateway that the OBD-II port attaches to (see Fig. 1.5).

Recently, there have been efforts to secure IVNs from outside attacks. For instance, the Society of Automotive Engineers (SAE) is planning to harden the OBD-II port [6]. In the corresponding SAE standard [151], data access via OBD-II (SAE J/1979) and Unified Diagnostic Services (ISO 14229-1) is categorized as *intrusive* and *non-intrusive*, respectively. Nevertheless, this standard does not classify how intrusive the actions of reading data via OBD-II (Service 0x01 of J1979) or reading raw CAN data are.

In any case, these changes are only possible with an improved vehicular gateway. This topic has been discussed since 2015 [78], when coverage of car hacking by news outlets increased significantly [3]. [7] also suggests enhancing existing gateway designs by adding additional security measures, such as a firewall. The aforementioned SAE standard [151] even hints that some OEMs might want to continue without a gateway at all, primarily due to cost.

Finally, we want to point out existing academic work in this area. Automotive gateways have many advantages for vehicle cybersecurity as summarized in [118, 155]. In addition to traditional functions such as routing, gateways can also be used for secure CAN or Automotive Ethernet communications through the use of authenticated ECUs [89, 118] or via access control/firewalls [117, 140].

2.6 Related Work

2.6.1 Manual CAN Reverse Engineering

[66] extracted CAN messages using the OBD-II port, interpreted those messages by examining how different bytes changed over time given different actions being performed on/by the vehicle, and then replayed these messages to manipulate their corresponding functions. However, the experiment they performed is limited because it requires prior knowledge of the implementation details of the vehicle — the paper mentions in several places that it is important to have an understanding the specific car being hacked. They also discuss the proprietary nature of the CAN bus and in-vehicle E/E architecture, meaning that there could be differing numbers or locations of CAN buses across different vehicle models, and thus the functions of each bus could be split up differently. In order to gain knowledge about the car they evaluated, they purchased a subscription to an online data service that provided this information.

Other automotive attacks, such as [124, 127], require that the E/E architecture be analyzed and that the CAN message format be manually reverse-engineered before data can be injected. This is a tedious process that can require days to weeks to reverse-engineer a targeted portion of CAN data and is not scalable to other vehicles.

Additionally, several tools exist that can help manually reverse-engineer CAN data. For instance, [75] demonstrates how Wireshark can be leveraged to capture CAN traffic and visualize changing bits in real time when an event is executed, as in our Phase 2.

2.6.2 Automating CAN Reverse-Engineering

[122] built an anomaly detection system to split CAN messages into different fields/signals without prior knowledge of the message format. Their classifier identified the boundaries and types of the fields (Constant, Multi-Value, or Counter/Sensor).

READ [120] proposed an algorithm to split synthetic and recorded CAN messages into signals, comparable to Stages 1 and 3 of our Phase 0. They present methods to isolate counters and CRCs, with all other values marked as physical signals, the type of signal we seek to evaluate in Phase 1 of LibreCAN. Although they reported high precision values (see Table 2.7), it is important to note that their experiments were conducted on an older vehicle (confirmed by e-mail to the authors), with less signals available in its DBC. Along with LibreCAN, we report the best results of READ in the aforementioned table.

ACTT [179] proposes a simple algorithm to extract signals from CAN messages and label them using OBD-II PIDs. Their signal extraction only considers signals that do not consist of contiguous sets of constant bits. Furthermore, they do not distinguish between signal types as we did. The authors find that roughly 70% of the CAN traffic consists of constant bits (comparable to constant signals in LibreCAN), matching only 16.8% of the present bits to OBD-II PIDs. The paper also lacks an extensive evaluation, only showing some examples of matched signals. Furthermore, they evaluated their framework on an older vehicle from 2008 such as READ.

2.7 Conclusion

In this chapter, we propose LibreCAN, an automated CAN bus reverse engineering framework. To the best of our knowledge, this is the first complete tool to reverse-engineer both kinematic- and body-related data. LibreCAN has been tested extensively on four real vehicles, showing similarly good results on all of them. It consists of three phases: extracting signals from raw CAN recordings, finding kinematic signals, and reducing body events to a minimal candidate set by 10x. Besides the very high accuracy of the novel Phase 2, we demonstrated that Phase 1 can achieve better precision than prior related work.

In addition to achieving considerably good accuracy, LibreCAN reduces the tedious

manual effort required to reverse-engineer CAN bus messages to around 40 minutes on average. Since CAN reverse-engineering is a crucial step in numerous automotive attacks, we pride ourselves in overcoming the car hacking barrier and highlighting the importance of automotive security. The *security by obscurity* paradigm that automotive OEMs follow by keeping CAN translation tables proprietary needs to be overcome and replaced by more advanced security paradigms. Finally, we also proposed some countermeasures to mitigate attacks on vehicles if the aforementioned CAN translation tables are made public through frameworks such as `LibreCAN`.

CHAPTER III

S2-CAN: Sufficiently Secure Controller Area Network

3.1 Introduction

Since the advent of the first comprehensive automotive security analysis in 2010 [58, 109], this field has attracted significant attention. While the first generation of vehicle security (c. 2010-2015) focused on exploiting physical interfaces, such as the OBD-II port [126], or reverse-engineering Electronic Control Unit (ECU) firmware [124], the second generation (c. 2015-now) has been focusing on scaling attacks to multiple vehicles by analyzing remote attack surfaces [125]. The most prominent and comprehensive attack of this generation that led automotive cyber security to become a mainstream research and engineering subject was the Jeep Hack [127] that allowed the attacker to remotely compromise and steer the affected vehicles. With further scaling in each generation, the risk of automotive vulnerabilities towards driver/-passenger safety and privacy, as well as financial and operational damage potential increases [94]. All attacks in each generation have (CAN) injection/spoofing as the necessary (final) component of causing havoc in common. This enables the compromise of the vehicle which can, in the worst case, have a serious impact on driver safety, for instance, by electronically disabling the brakes or accelerating the vehicle.

Unfortunately, CAN injection is the easiest part of the aforementioned attacks. This can be explained by vulnerabilities in the CAN design which dates back to 1987. Despite allowing a fast, robust, and reliable communication, CAN was not designed with security in mind, and vehicles can no longer be regarded as closed systems due to an increased number of external interfaces with unpredictable input. CAN is a broadcast bus without encryption and authentication. Messages are sent in plain text and everyone who has access to the CAN bus can inject arbitrary messages or spoof existing ones. Encryption and authentication in a vehicle should usually go hand in hand. In order for spoofed messages to cause a visible impact on the compromised vehicle, the attacker needs to **(a)** know the syntax and semantics of the crafted CAN payload, and **(b)** be allowed to inject the targeted CAN message. In case of **(a)**, this is only possible by reverse-engineering unencrypted CAN data traces since OEMs keep the aforementioned semantics secret instead of disclosing them publicly (*security by obscurity*). Recently, automated CAN reverse-engineering is shown to be achievable in a few minutes [141], enforcing existing attack vectors and necessitating an encrypted CAN. Finally, for case **(b)**, authentication will prevent unauthorized entities to perform the CAN injection.

To defend against vehicular attacks, we need a holistic, multi-layer security approach. The authors of [194] propose 4 layers of countermeasures which build on one another: access control, secure on-board communication, data-usage policies and anomaly detection/prevention (see Sec. 3.3). Here we assume OEMs follow basic security practices such as access control and focus on the challenges of secure on-board communication. As we discuss in Sec. 3.4, many researchers have attempted to apply the security properties of confidentiality, authenticity, integrity, freshness, and availability on the CAN bus. Almost all of them provide authentication and replay protection — but no encryption — by deploying well-studied cryptographic algorithms. A comparison of existing approaches is provided in Table 3.1.

Table 3.1: Comparison with related approaches

	Protection	**Algorithm**	**HW/SW**	**Bus Load**	**Latency**	**MAC Length**	**Security Level**
CaCAN [111]	Authenticity + Freshness	SHA256-HMAC	HW+SW	+100%	+2.2-3.2μs	1 Byte	2^7
IA-CAN [90]	Authenticity	Randomized CAN ID + CMAC	SW	+0%	8bit: +72ms 32bit: +150μs	1-4 Bytes	2^7-2^{31}
vatiCAN [132]	Authenticity + Freshness	SHA3-HMAC	SW	+16.2%	+3.3ms	8 Bytes	2^{63}
TESLA [136]	Authenticity + Freshness	PRF+HMAC	SW	+0%	+500ms	10 Bytes	2^{79}
LeiA [145]	Authenticity + Freshness	MAC	SW	+100%	N/A	8 Bytes	2^{63}
CANAuth [177]	Authenticity + Freshness	HMAC	HW+SW	+0%	N/A	10 Bytes	2^{79}
`S2-CAN`	Confidentiality + Authenticity + Freshness	Circular Shift + Internal ID Match	SW	+0%	+75μs	N/A	$\sim 2^{49}$

Although mechanisms such as encryption and authentication are widely used and accepted in traditional computer communication networks, their adoption in the automotive domain comes with three major problems related to performance that currently limit their deployment in commercial vehicles:

(1) Cost: For cost reasons, ECUs in an in-vehicle network (IVN) are resource-constrained. Since most safety-critical functionalities require simple computations and do not need high-performance hardware, these legacy ECUs are very simple and highly optimized for repetitive control operations. For instance, current Engine Control Modules can have 80MHz clock frequency, 1.5MB Flash memory and 72kB of RAM (Bosch [16]). Using cryptographic algorithms for encryption and/or authentication would require more performant hardware which drive up the cost for OEMs. Besides unit costs, adding security protocols to certain legacy ECUs (especially in the powertrain domain) that have been in use in cars for multiple years or even decades due to lack of necessary software improvements would increase the development cost [175].

(2) Latency: In order to guarantee functional safety in a vehicle, there are stringent hard real-time requirements for certain safety-critical control data. The maximum permitted end-to-end (E2E) latency for cyclic control data transmitted on the CAN bus can range from a few milliseconds to a second [71]. Since secure encryption and authentication algorithms add a non-negligible delay (see Sec. 3.7), as well as block CAN messages to be sent until fully encrypted (due to block size), message deadlines can be missed which can endanger driver safety (imagine the car braking too late!).

(3) Bus Load: CAN messages contain only 8 bytes of payload. Message Authentication Codes (MACs) to protect data integrity have to be appended to the data, but due to lack of space, several existing solutions [111, 132, 145, 180] send the MAC in a separate CAN message. This increases the bus load which is an indicator for the utilization of the network. A high bus load can lead to certain CAN messages missing

their (hard) deadlines, harming safety. To avoid this, the average bus load must be kept under 80% at all times [15].

For the above reasons, encryption and authentication on the CAN bus have not yet been adopted in commercial vehicles. Traditional cryptography-based solutions — we will summarize these under the generic term ***S**ecure **CAN*** (`S-CAN`) — offer a medium to high level of security (see the number of combinations to brute-force MAC, labeled as *Security Level*, in Table 3.1) at the expense of performance (i.e., CPU, latency, bus load). As the authors of [132] have shown, brute-forcing a MAC would take too long for in-vehicle ECUs, especially if keys are dynamically refreshed. As a result, we would like to break away from traditional cryptography-based solutions to address the aforementioned three problems while providing reasonable, albeit relaxed security guarantees. We propose `S2-CAN` (**S**ufficiently **S**ecure **CAN**) to enable a **tradeoff** between *performance* and *security* to offer a feasible and secure real-world solution for the automotive industry.

`S2-CAN` tries to protect the confidentiality, authenticity and freshness of CAN data during operation of the vehicle without using cryptography. In particular, `S2-CAN` consists of two phases in its core: a handshake and operation phase. In the former, it establishes unique sessions of specific length and distributes necessary *session parameters* to all participating ECUs. This phase resembles the key management phase in traditional `S-CAN` approaches where session keys are shared among the ECUs to both encrypt and authenticate CAN messages in their respective operation phase. Since `S2-CAN` avoids using cryptography in its operation phase, it uses the *session parameters* from the handshake to **(a)** first include a randomly generated *internal ID* and counter for authenticity and freshness into the CAN payload before **(b)** each byte of the payload is shifted cyclically by a random integer (*encoding parameter*) in fixed time intervals. These two steps can be compared to **(a)** appending a MAC to provide authenticity and **(b)** encrypting the plain-text CAN message to provide

confidentiality in S-CAN. Compared to breaking traditional CAN authentication solutions that only require brute-forcing the MAC, the cyclic shift encoding further masks the plain-text by making it more difficult to decode and thus provides confidentiality protection as well. Due to the encoding, CAN reverse-engineering — which is the first essential step of a CAN injection attack — has to be performed in real time for the current *encoding parameter* and cannot be computed *a priori* to be used for the lifetime of the vehicle. Despite intentional weaker security of S2-CAN, a frequent update of sessions with new *encoding parameters* will render reverse-engineering very tedious, if not impossible. Hence, session cycle is *the* crucial parameter to provide security in S2-CAN. Furthermore, even after guessing the encoding correctly, an attacker would still need to calculate the *internal ID* and counter to bypass authentication. All in all, brute-forcing S2-CAN would require $\sim 2^{49}$ combinations for an ECU (see Sec. 3.8 and 3.9) while it does not increase the bus load in the operation phase, outperforms the E2E latency of the best comparable S-CAN approach by 44x, as well as incurs less than 0.1% CPU overhead as evaluated with our experimental setup (see Sec. 3.7). Finally, we conduct a security evaluation in Sec. 3.8 to demonstrate that even an intelligent attacker who leverages protocol-specific knowledge to circumvent brute-forcing can be thwarted to show that S2-CAN can be sufficiently secure.

3.2 Background

Please refer to Sec. 1.2 for a primer on the CAN bus, DBC files and in-vehicle network architectures.

3.3 Threat Model

As briefly mentioned in Sec. 3.1, the common and final part of every automotive attack — which is the main threat to protect against — is to gain access to the

CAN bus for a *CAN injection* attack which can lead to various forms of vehicle misbehavior, including (safety-critical) sudden acceleration. In general, there are two ways an attacker can achieve CAN bus access: **(a)** by connecting a *physical* CAN device/ECU to the IVN, e.g., an OBD-II dongle or by tapping into the CAN bus, or **(b)** compromising an existing ECU *remotely*. The former is relatively easy to accomplish as long as the attacker has physical access to the target vehicle, while the latter is more complicated and multi-layered (and thus less likely) as the attacker has to usually leverage vulnerabilities in wireless interfaces of an ECU to gain access to the device. We refer to the attacker in case of **(a)** as an *external* attacker, whereas an *internal* attacker is capable of **(b)**. Furthermore, the aforementioned separation of domains by a central gateway complicates a compromised ECU — which is usually on a less safety-critical bus (e.g., infotainment) — to affect more safety-critical domains such as powertrain which has no remote attack surfaces. Finally, even if a proper `S-CAN` approach is implemented, an *internal* compromise of an ECU (as in case **(b)**) will lead to exposure of secret keys which the attacker can use to forge the desired message's Message Authentication Code (MAC) and/or encrypt the CAN payload.

Although remote attacks on vehicles have skyrocketed over the last decade [48], a breakdown of attack vectors shows that most of these *remote* attacks are targeting key fobs, OEM servers and mobile companion apps. Remote attacks to compromise an ECU usually exploit the In-Vehicle Infotainment (IVI) and require significant effort (usually multiple months) as shown in the Jeep Cherokee hack [127] to achieve CAN bus access and cannot be thwarted even by a properly secured CAN bus (`S-CAN`). In contrast, OBD-II attacks are the fourth most common attack vector and account up to over 10% of all attacks. Nevertheless, recent research [185] has shown that remote attacks can also be launched by an *external* adversary by exploiting vulnerabilities in wireless OBD-II dongles. Many commercial OBD-II dongles feature Wi-Fi or Bluetooth capabilities which open a new over-the-air attack surface. The researchers'

findings show that CAN injection can also be performed by remote, external attackers. As a result, *external* attackers in scenario **(a)** form the most crucial threat. In what follows, we will focus on protection from this type of adversaries and describe their attack capabilities.

Once CAN bus access has been achieved, the attacker will continue a *CAN injection* attack. The authors of [61] introduce three possible CAN injection attacks as discussed next. *Fabrication attacks* allow the adversary to fabricate and inject messages with a forged CAN header and payload at a higher frequency to override cyclic CAN messages sent by legitimate ECUs that can render safety-critical receiver ECUs inoperable [109]. *Suspension attacks* on the compromised ECU prevent its broadcast of legitimate, potentially safety-critical CAN messages to the intended recipient(s). Finally, *Masquerade attacks* combine both of the above attacks by *suspending* the CAN broadcast of one ECU and deploying another ECU to *fabricate* malicious CAN messages. Only *fabrication attacks* can be mounted by our adversary from scenario **(a)**, since the others require an *internally* compromised ECU. We would like to emphasize that fabrication attacks can not only be mounted by attackers having physical access to the car, but also by remote attackers [185] which makes *external* attacks from scenario **(a)** an highy likely and scalable threat.

As a result, we assume the (external) adversary to only be able to perform *fabrication attacks* in our threat model. Even then, the attacker can cause havoc for both vehicle and driver, as shown in the Toyota Prius hack [124]. To prevent fabrication attacks, a solution for secure CAN must have the following two security properties:

Authenticity. As outlined before, any CAN node can join the IVN. There is no provision of verifying the authenticity of an added malicious device to the CAN bus by default. So, device authentication is important, i.e., only pre-authorized ECUs will be allowed to communicate. Furthermore, an attacker should not be able to spoof legitimate CAN messages during a fabrication attack. This can be prevented

by adding a MAC to each message to ensure *integrity*. The latter also includes protection against replay attacks by adding a counter to each message. The major drawback of protecting against fabrication or replay attacks is the required additional space for MACs and freshness values. This is challenging because CAN only has an 8-byte payload field, with most of the space already occupied by control data (see Sec. 3.5.2).

Confidentiality. CAN message data is not encrypted, and therefore, messages between ECUs can be eavesdropped and analyzed by anyone accessing the IVN. To prevent this type of attack, mechanisms to guarantee *confidentiality* are required. As mentioned before, plaintext data can be recorded and used for reverse-engineering the proprietary CAN message format (i.e., signal location, scale and offset) which can be ultimately used to craft well-formed CAN messages in a fabrication attack to cause visible damage. Encryption with symmetric session keys between participating ECUs is a solution, although it will incur additional latency overhead.

In this thesis chapter, we want to protect against fabrication attacks by leveraging a combination of confidentiality and authenticity protection. Since we focus on the tension between security and performance as previously discussed, `S2-CAN` uses a non-traditional approach instead of cryptographic encryption and authentication in order to optimize performance.

3.4 Related Work

3.4.1 Authenticity and Integrity

Most existing work on Secure CAN (see Table 3.1) focuses on the authentication of sender ECUs, protecting the integrity of the payload, as well as against replay attacks.

vatiCAN [132] offers backward-compatible sender and message authentication, as

well as protection against replay attacks for safety-critical CAN messages via HMACs computed from preinstalled keys. The HMAC is sent in a separate message with a different CAN ID. vatiCAN adds 3.3ms latency per CAN message, a 16.2% increase in bus utilization and 400 bytes of memory overhead.

IA-CAN [90] provides sender authentication via randomization of CAN IDs on a per frame basis and payload data authentication using two different session keys. The receiver only accepts a message if the MAC is correct and the CAN frame has the expected CAN ID that changes with each frame using a function. The receiver's filter is updated accordingly when the next frame is accepted.

CaCAN [111] uses a hardware-modified central monitoring node to perform the entire authentication on the CAN bus. As with the general case of centralized authorities, if the monitor node is compromised or removed, the entire network is compromised. Furthermore, no encryption is used and the bus load is doubled.

TESLA [136] protocol is a lightweight authentication protocol, relying on delayed key disclosure to guarantee message authenticity. It provides authenticated broadcast capabilities, albeit with additional latency during authentication.

CANAuth [177] uses out-of-band transmission of integrity and freshness values to avoid bus load overhead. Its major drawback is the lack of backward compatibility with regular CAN controllers.

LeiA [145] is a counter-based authentication protocol that uses extended (29-bit) CAN IDs to include freshness values and a generic MAC algorithm for authentication. The MAC is 8 bytes long and transmitted in a separate CAN message, doubling the bus load. No latency numbers are reported.

3.4.2 Confidentiality

The space-limited payload field of 8 bytes in CAN messages is a major problem for encryption algorithms such as AES-128 that depend on a 16-byte block size. As a

result, multiple messages have to be sent, increasing the bus load. Latency is another issue due to the limited computational power on ECUs if implemented in software to guarantee backward compatibility. [53] surveyed different encryption methods for the CAN bus in terms of bus load, latency and security. Existing approaches use AES-128 [73], AES-256 [160], XOR [81, 92], Tiny Encryption Algorithm (TEA) [97] and Triple DES (3DES) [91].

3.4.3 Key Management

Secret keys are necessary to generate and verify MACs, and to encrypt and decrypt data. Instead of using a single long-term key for the entire lifespan of a car — which is 12 years on average [57] — session keys can be generated periodically that are only valid for a certain period to limit their exposure.

In Secure CAN (S-CAN) solutions, there are two general approaches to in-vehicle key management. The first approach is to deploy an OEM backend and request new keys periodically via Over-the-Air (OTA) using the *authenticated key exchange protocol 2* (AKEP2) [189]. Keys can be stored in the central gateway (acting as the in-vehicle key master) in a *Trusted Platform Module* (TPM) or *Hardware Security Module* (HSM). The second approach tries to do the key management completely on-board without the need for an OEM-provided backend which can reduce complexity, bandwidth and cost [170]. The key distribution *inside* the vehicle can be done in two ways. First, the key master generates and distributes new session keys based on the *Secure Hardware Extensions* (SHE) Key Update Protocol. Second, the key master triggers the ECUs to derive session keys from a nonce and long-term keys installed at manufacturing time. The first approach is superior if security is the most important and waiting on startup time is acceptable. The second approach can be used when speed is the most important and no wait time for key distribution is acceptable.

3.5 System Design

We now present the system design of `S2-CAN`, which consists of three phases: **Key Management**, **Handshake**, and **Operation**. Although no cryptography will be used in the operation phase (Sec. 3.5.3), establishing a session S_i during the handshake (Sec. 3.5.2) needs the distribution of keys which will be briefly discussed in Sec. 3.5.1. In our prototype, we use $N = 2$ slave ECUs and one master ECU which is the central gateway. The master ECU will be responsible for establishing new sessions during the handshake phase. There is no real value of expanding the testbed to more than 2 slave ECUs since the benchmark in Sec. 3.7 shows that `S2-CAN` does not add any communication overhead and is thus independent of traffic/bus load during the operation phase, i.e., when operation-related CAN messages are exchanged between ECUs. `S2-CAN` is applied to each CAN sub-bus independently. As a result, the OEM can choose which CAN buses to protect. We will use the syntax $m = (CAN\_ID, Payload)$ for a CAN message m exchanged on the bus. Furthermore, we require a logical ordering of the slave ECUs for error handling and timeout purposes during the handshake (Sec. 3.5.2), i.e., that ECU_A transmits before ECU_B. The ordering can be assigned randomly (as in our case) or according to criticality/relevance of the ECU, with the more safety-critical slave ECU being assigned as ECU_A. This knowledge of ordering can be stored as an additional one-byte unsigned integer in each ECU's non-volatile memory.

3.5.1 Phase 0: Key Management

`S2-CAN` refrains from using Message Authentication Codes (MACs) and encryption based on cryptographic keys during the vehicle's operation mode (Sec. 3.5.3). During the handshake phase (Sec. 3.5.2), we will distribute `S2-CAN`-specific *session parameters* from the master ECU (gateway ECU_{GW}) to the two slaves ECU_A and ECU_B on a safety-critical CAN domain named CAN_1. These session parameters establish a new

S2-CAN session S_i that is valid for a *Session Cycle* T. To distribute these parameters securely in each session, we CANNOT avoid cryptography in the handshake phase and need to ensure that the CAN payload is both authenticated and encrypted to defend against spoofing and eavesdropping attacks on the handshake. This requires the existence of pre-shared secret keys that are provided by the key management system in a vehicle. Since a detailed discussion of key management is not in the scope of this chapter, we use pre-installed symmetric keys on each ECU and refer to the aforementioned best practices of in-vehicle key management (see Sec. 3.4.3). Note that it is transparent to the design of S2-CAN of *how* these symmetric keys are obtained, i.e., if a backend periodically provides them via OTA or they are derived from a long-term key installed at manufacturing time. Nevertheless, the use of short-lived session keys is recommended to limit exposure of the long-term key which would allow eavesdropping attacks on the handshake and thus fully compromise S2-CAN.

3.5.2 Phase 1: Handshake

Overview: Upon initialization, ECU_{GW}, ECU_A and ECU_B on CAN_1 will perform a 3-way handshake in order to exchange the information about the aforementioned session parameters and agree on "talking" in S2-CAN syntax. The session parameters consist of a global **(a)** *encoding parameter* f, **(b)** a slave ECU-specific *integrity parameter* int_ID_j, **(c)** a slave ECU-specific *integrity parameter* $pos_{int,j}$, and **(d)** a slave ECU-specific *counter value* cnt_j, with j denoting the respective slave ECU. Parameter **(a)** will be distributed in Stage 1, whereas the other three parameters **(b)-(d)** will be exchanged between ECUs in Stage 2. The handshake comprises three stages and repeats for each new session S_i in periodic fixed-intervals T which represents the *session cycle*. In what follows, we will describe the handshake process for an arbitrary session S_i. The communication diagram for Phase 1 is depicted in Fig. 3.1 and separated into the three stages. The CAN IDs used for messages during

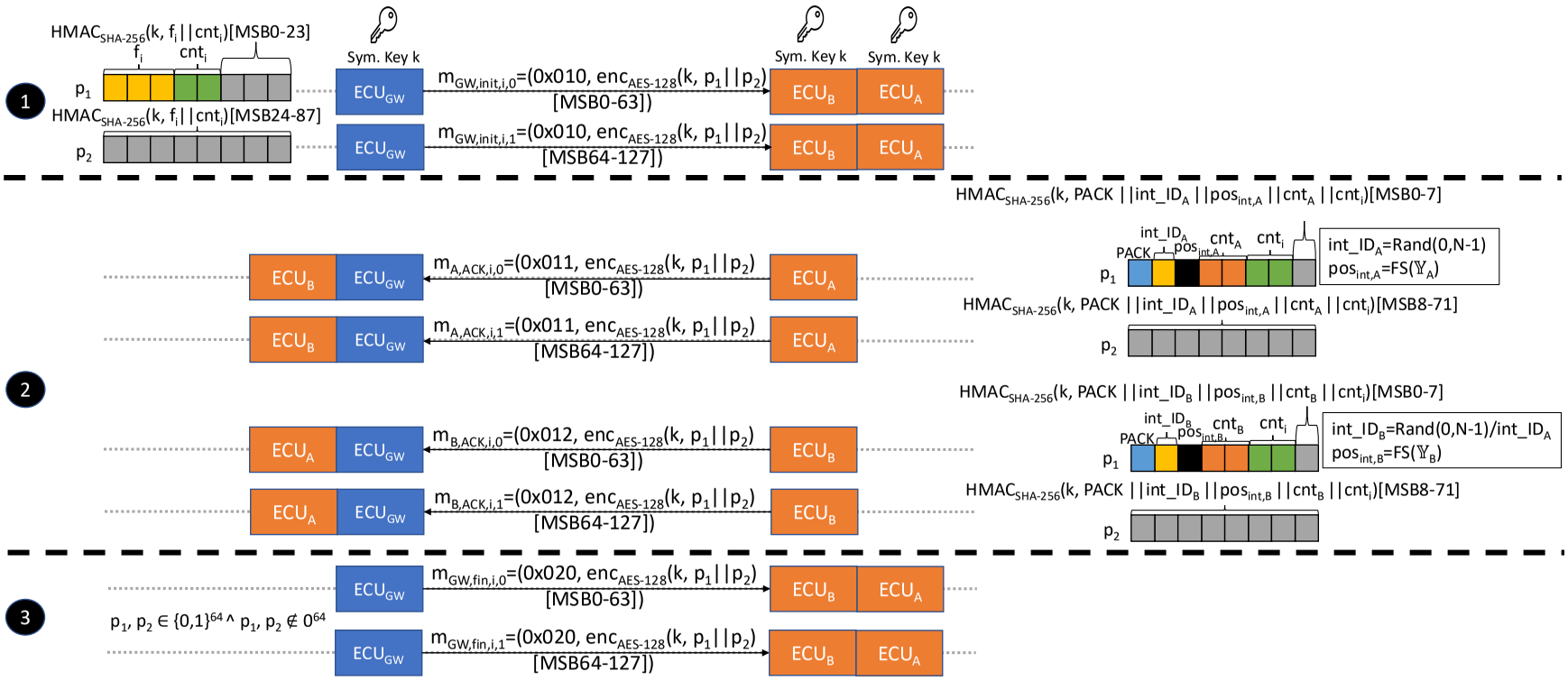


Figure 3.1: Handshake communication diagram

the handshake are merely examples, but should have a low ID or high priority.

Stage 1 (Initialization): The master ECU (ECU_{GW}) indicates that it wants to start a new session S_i. It randomly generates an 8-byte *encoding parameter* $f_0 = (r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_7), r_l \in [0, 7]$. r_l corresponds to the bit rotation number for the l^{th} byte in the 8-byte CAN payload. Each r_l can be expressed with 3 bits for a total of 3 bytes to include in the payload p of the gateway initialization message $m_{GW,init}$ = (0x010, p). As discussed before, due to the sensitivity of handshake messages, each CAN message during the handshake has to be both authenticated and encrypted to prevent spoofing and eavesdropping, but also replay attacks. To achieve the latter, we first add a 2-byte counter cnt_0 (not to be confused with the ECU-specific session parameter cnt_X) to defend against replay attacks. In order to prevent spoofing attacks on this message, we calculate the SHA256-HMAC of the previous 5 bytes (i.e., f_i and cnt_i) to obtain a 32-byte output with the symmetric key k from Phase 0. Since the payload of $m_{GW,init,i}$ only has another 3 bytes of free space to fit the MAC which would be too small to defend against brute-force attacks, we have to truncate the HMAC (taking the MSBs per definition). The truncation can be done safely since the increased advantage of the attacker would be offset by the limited availability of a CAN message due to the cyclic message nature of CAN and the invalidation through the counter value cnt_i. Nevertheless, we believe that 3 bytes for a truncated HMAC is too small. As a result, we split $m_{GW,init,i}$ into two consecutive CAN messages $m_{GW,init,i,0}$ and $m_{GW,init,i,1}$ with respective payloads p_1 and p_2 to **(a)** utilize another 8 bytes for the truncated HMAC, resulting to a total of 11 bytes, and **(b)** allow encryption with a secure block cipher such as AES-128 which has a block size of 16 bytes.

In summary, two CAN messages with the following syntax are broadcast sequen-

tially on CAN_1:

$$m_{GW,init,i,0} = (0x010, enc_{AES128}(k, p1||p2)[MSB0 - 63])$$

$$m_{GW,init,i,1} = (0x010, enc_{AES128}(k, p1||p2)[MSB64 - 127])$$

Stage 2 (Acknowledgment): Upon receiving both initialization messages from ECU_{GW}, ECU_A and ECU_B first decrypt the ciphertexts $p_1^\star$ and $p_2^\star$ using the symmetric key k and extract the *encoding parameter* f_i into local memory. Eeach slave ECU will then broadcast an acknowledgment (ACK) message $m_{j,ACK,i}$ (which will be split into two messages again due to AES-128 encryption), where $j \in [0, \ldots, N-1]$, consisting of a 1-byte positive acknowledgment code (PACK) and the three slave ECU-specific parameters **(b)-(d)** in the CAN payload. Parameter **(b)** is a randomly generated unique internal ID $int\_ID_j \in [0, N-1]$ representing ECU_j on CAN_1 during the current session S_i. This parameter can be encoded with 1 byte since a CAN domain (or even vehicle in general) never has more than 256 ECUs.

Next, parameter **(c)** specifies the random position $pos_{int,j}$ of where the internal ID (parameter **(a)**) will be located within the CAN payload. Since space within the payload is limited and specific positions are occupied by CAN signal data that cannot be overwritten, the internal ID has to be included in available free space. The set of available free spaces for a CAN ID in a given vehicle is defined as $\mathcal{Y}_j$. Sec. 3.6 discusses the distribution of free spaces among CAN IDs by analyzing the DBCs of 4 different vehicles. For instance, $\mathcal{Y}_j = 12, 13, 14, 25, 26, 54, 55, 63$ states that the CAN ID belonging to ECU_j possesses only 8 bits of free space over 4 non-consecutive "regions". This set of bits is then used by the *Free Space* (FS) function to randomly determine the first bit $pos_{int,j}$ where int_ID_j will be placed:

$$pos_{int,j} = FS(\mathcal{Y}_j) \tag{3.1}$$

In our example, if $pos_{int,j} = 54$, the MSB of the one-byte internal ID will be stored at bit position 54 and the LSB at bit position 26.

The last parameter **(d)** is the initial value of an ECU-specific counter cnt_j for replay protection and is also randomly generated. This parameter consists of 2 bytes and is also included in available free space together with int_ID_j by Eq. 3.1.

Besides including these functional handshake parameters, the ACK messages will also include a 2-byte handshake counter cnt_i and truncated HMAC for integrity and freshness protection, just like in Stage 1. We obtain 2 consecutive CAN messages broadcast by ECU_j that are both authenticated and encrypted with the following syntax:

$$m_{A,ACK,i,0} = (ID_j, enc_{AES128}(k, p1||p2)[MSB0 - 63])$$

$$m_{A,ACK,i,1} = (ID_j, enc_{AES128}(k, p1||p2)[MSB64 - 127])$$

Due to the aforementioned pre-determined order for all slave ECUs, ECU_A will first transmit with CAN ID 0x011 and ECU_B needs to wait until it has received both $m_{A,ACK,i,0}$ and $m_{A,ACK,i,1}$ from ECU_A before it can broadcast $m_{B,ACK,i,0}$ and $m_{B,ACK,i,1}$. For the latter two messages, the CAN ID can simply be incremented by one as depicted in Fig. 3.1, as each ECU will use a distinct CAN ID. Once ECU_B receives the aforementioned ACK message, it first extracts the received *integrity parameters* into its memory and then repeats the ACK process for itself. To avoid collisions in internal ID assignment, it needs to exclude int_ID_A during the random ID generation.

Stage 3 (Finalization): ECU_{GW} finalizes the handshake after receiving ACKs from all slave ECUs. It sends $m_{GW,fin,i}$ with a random non-zero payload to signal that it has received well-formed ACK messages from all slave ECUs and monitored a successful handshake. The finalization message is again split into two CAN messages and broadcast with CAN ID 0x020.

Security and Reliability Analysis: Due to authentication, an adversary cannot spoof the contents of a handshake message. An attacker cannot replay handshake messages due to the freshness counter, and eavesdropping attacks can be mitigated by encryption.

If any ACK message takes too long due to bus or ECU errors, the handshake times out and ECU_{GW} restarts the handshake with Stage 1. If the handshake is still unsuccessful even after repeating it r times, all ECUs on CAN_1 can revert to regular CAN communication until the next start of the vehicle. Although this countermeasure has been designed for non-adversarial reliability issues, an adversary still cannot exploit it. An attacker could launch a Denial-of-Service (DoS) attack through the OBD-II device by injecting high-priority CAN IDs (e.g., 0x0) with the goal to circumvent successful handshakes and downgrade to regular CAN communication. Since vehicles have a holistic security concept in place (as discussed in Sec. 3.1), the gateway (which is directly connected to the OBD-II port) can defend against this availability attack by discarding injected CAN messages under a certain CAN ID threshold, i.e., the lowest handshake CAN ID.

3.5.3 Phase 2: Operation

After the handshake for a session S_i has been completed, slave ECUs can start the *Operation Mode* exchanging regular data on CAN_1. To save space in the CAN payload field, we perform the following operation on the 1-byte int_ID_j and 2-byte cnt_j that ECU_j stored during the handshake to calculate the 2-byte parameter q_j:

$$q_j = \text{LEFTZEROPAD}(int\_ID_j, 8) \oplus cnt_j. \tag{3.2}$$

First, the payload of a CAN message is being logically ORed with q_j which includes the integrity parameters into the free space of a CAN message. Second, a *Circular*

Shift (CS) operation is performed on the new payload using the stored encoding parameter f_i which does a byte-wise bit rotation to the l^{th} byte according to the value of the l^{th} element of f_i. Finally, the message is broadcast on CAN_1. For the next CAN message sent by ECU_j, its local counter will be incremented.

On the receiver side, the respective slave ECU(s) need(s) to execute the above process reversely, i.e., rotate each byte of the encrypted payload in the opposite direction according to r_l, extract the position information from $pos_{int,j}$, determine the internal ID and finally the counter/freshness value by XORing it with int_ID_j of the sender.

Based on these extracted values, the receiver can then perform an integrity and freshness check: 1. The extracted counter cnt_j is compared with the expected counter for the respective sender. If the two values match, the local counter for sender ECU_j on the receiver is incremented, and 2. the internal ID of the sender int_ID_j is compared with the stored internal ID for the respective sender on the receiver ECU. Only if these two checks do not fail, the receiver can assume that the message came from a legitimate sender ECU_j and start processing the data in the payload. Otherwise, it may either suspect a replay attack or a message with fabricated information from a malicious ECU and drop the CAN message.

The operation mode with the respective encoding and integrity parameters ends once a new handshake has been completed. A new session S_{i+1} begins. The operation mode does not get interrupted by the start of a new handshake to guarantee functionality and safety.

Finally, we discuss what happens in the case of packet drops that can happen naturally on the CAN bus. Since each CAN message has a counter to prevent replay attacks and the receiver expects the next message with an incremented counter value, a packet drop can lead to inconsistencies with the local state counter on the receiver side. In order to account for packet drops, the receiver ECU will still accept CAN

messages with counter values higher than the previous message within a specific threshold. The latter depends on the packet loss rate on the CAN bus which is usually very robust. The authors of [196] suggested to setting this threshold to 1.

3.6 Finding Free Space

To gain a better understanding of how many signals are used in a CAN ID and thus how much of free space (FS) is available to include our integrity parameters int_ID_j and cnt_j, we analyzed the DBC files of four passenger vehicles from a North American OEM under NDA (see Sec. 3.8.1). Since we include a 2-byte parameter q_j into the CAN payload, only a maximum of 6 bytes may be used for data. Among all CAN IDs in each DBC, we identified certain low-priority *non-operation-related CAN IDs* that do not occur during regular operation of the vehicle. Hence, we manually removed these irrelevant CAN IDs for our purposes and analyzed the remaining *operation-related CAN IDs* for available unused space.

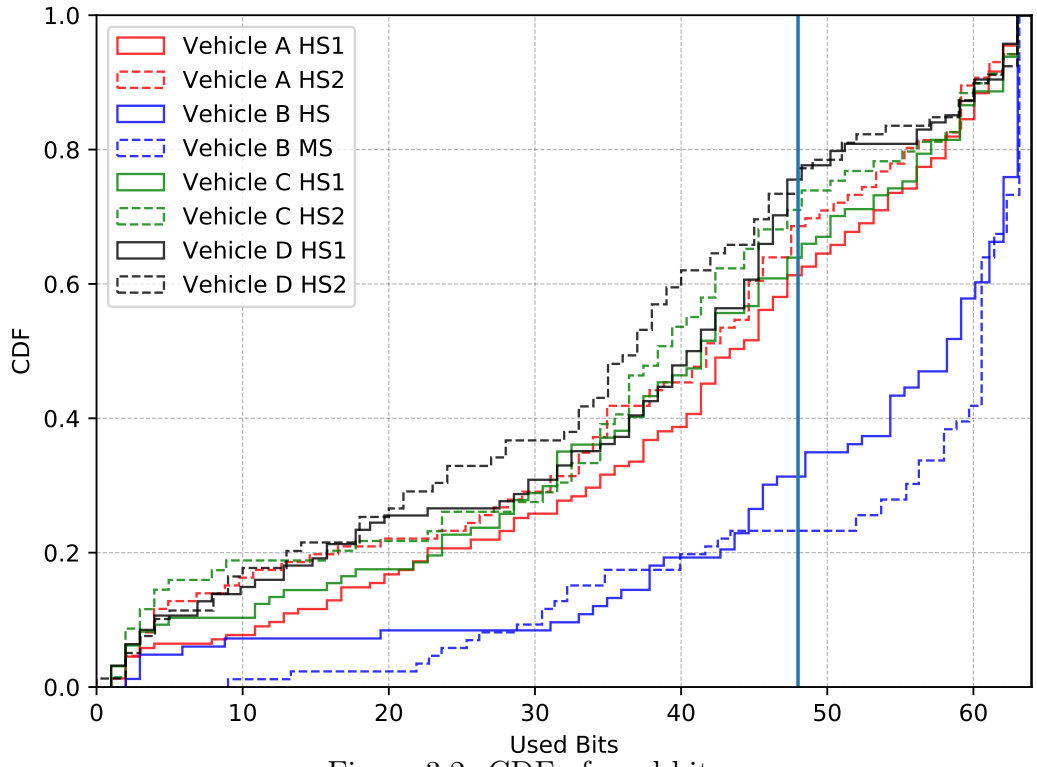


Figure 3.2: CDF of used bits

A Cumulative Distribution Function (CDF) for each vehicle is plotted in Fig. 3.2. The vertical marker indicates that all vehicles — with the exception of Vehicle B — contain between 60% and 80% CAN IDs that have at least 16 bits of free space. As a result, we can apply `S2-CAN` for the majority of CAN IDs, but would like to analyze how to further improve this metric to maximize the number of usable CAN IDs. Note that we are referring to the free space in the CAN payload/data field and not the CAN ID field (see Fig. 1.3).

Table 3.2: Free space in DBCs

Veh.	**Bus**	**#IDs**	**#Rebalancable IDs**	**#IDs with FS**	**Usable CAN IDs (%)**
Veh.A	HS1	102	31	63	92.2
	HS2	53	2	35	69.8
Veh.B	HS	81	5	26	38.3
	MS	62	3	16	30.6
Veh.C	HS1	57	7	38	78.9
	HS2	42	1	26	64.3
Veh.D	HS1	58	7	43	86.2
	HS2	51	4	38	82.4

OEMs could re-balance the disparity of available space in a CAN message with a more careful design of the CAN communication matrix while still considering functional requirements. In what follows, we present a possible re-balancing approach. CAN messages are differentiated by four types: fixed-periodic, event-periodic, event-on-change and network management. First, we grouped CAN IDs based on the sender ECU. As mentioned before, a sender can transmit multiple CAN IDs with different cycle times if the CAN ID is fixed-periodic or event-periodic. The latter message type is similar to fixed-periodic except a CAN message is not necessarily transmitted at every cycle time. Both message types cannot be grouped together.

As an example, Fig. 3.3 depicts the number of used bits of fixed-periodic CAN messages with exactly the same cycle time that a sender ECU transmits on HS1-CAN

(high-speed CAN 1) of Vehicle A. Points above the red threshold line of 48 bits depict CAN IDs that do not have sufficient free space for `S2-CAN`. Since all vertical dots are grouped by sender ECU and cycle time, they can be re-balanced by packing signals of their mean value per CAN ID (depicted with marker `x`). For Veh. A HS1, there are a total of 101 fixed-periodic CAN IDs. A mean value below 48 bits indicates that the CAN IDs in the group can be re-balanced. 27 CAN IDs can be re-balanced this way, besides those already under this threshold. We repeated this experiment for all other vehicles and buses for both fixed-periodic and event-periodic messages and summarized the number of re-balancable and existing CAN IDs with free space in Table 3.2. The sum of these two yields the number of usable CAN IDs for `S2-CAN`. With the exception of Veh. B, around 79–92% of all CAN IDs can be used with `S2-CAN` for the more safety-critical HS1-CAN. The remaining non-periodic CAN IDs can be re-balanced further by OEMs based on functionality — something that we cannot interpret.

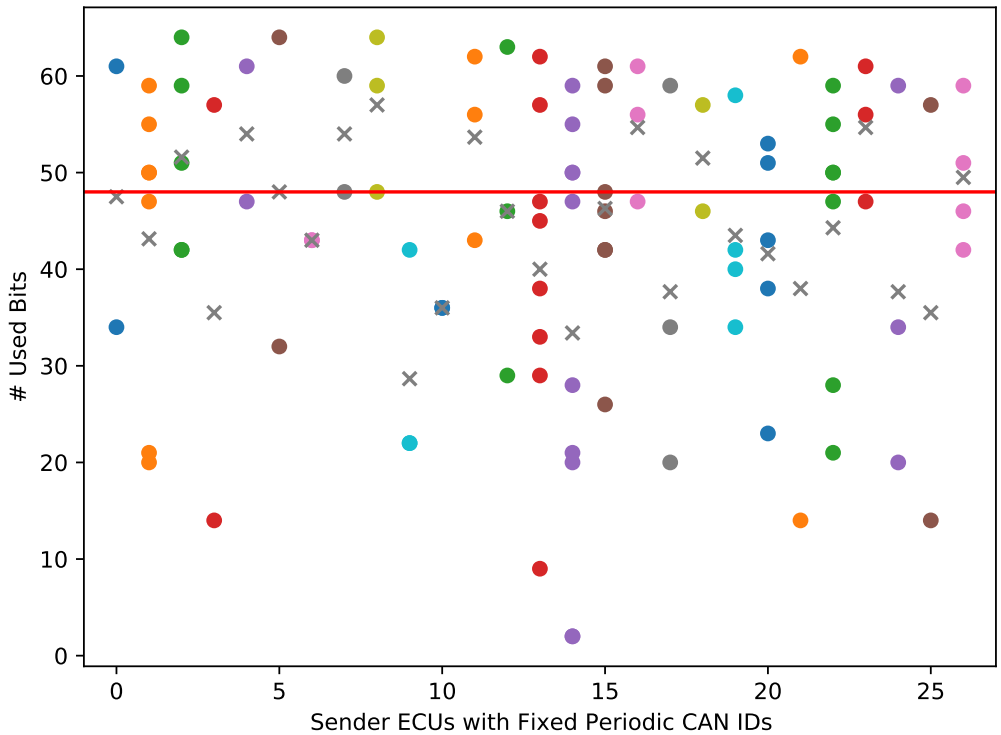


Figure 3.3: Re-balancing Vehicle A HS1

Finally, no relationship between message priority and free space can be derived. This analysis is depicted in Fig. 3.4.

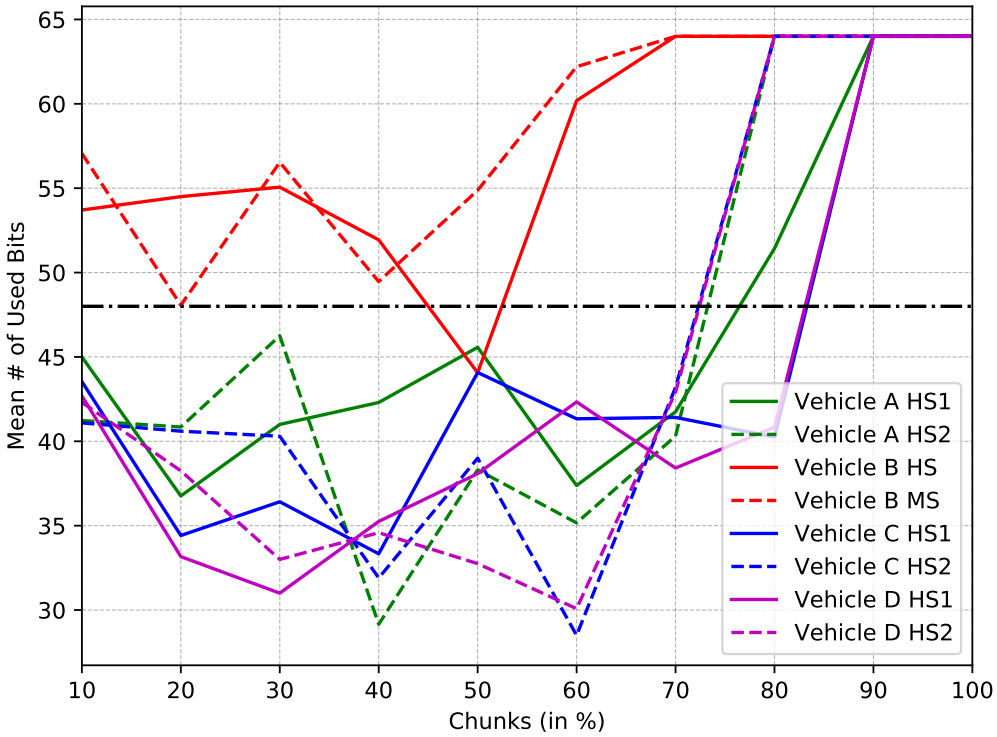


Figure 3.4: Relationship between Free Space and Message Priority

3.7 Evaluation

3.7.1 Experimental Setup

We have built a prototype with three CAN nodes, each of which consists of an Arduino Mega 2560 board and a SeeedStudio CAN shield [197]. This prototype was set up to operate at a 500 kBit/s baud rate as in a typical high-speed safety-critical CAN bus. Note that the entire evaluation is based on a simple scenario with the sender ECU transmitting only one CAN message. In reality, multiple CAN messages will be broadcast on the CAN bus in a relatively short time and CAN scheduling will pick the highest-priority CAN message to be broadcast first. This will inherently lead to blocking time t_b for lower-priority messages which depends on the number

of higher-priority messages that have to be transmitted first. Nevertheless, using a simpler setup does not affect our evaluation metrics except the operation latency which is discussed in Sec. 3.7.3.

Since we want to compare the performance of `S2-CAN` with prior work, we implemented existing CAN bus encryption methods from Sec. 3.4.2 with vatiCAN [132] for authentication. We chose vatiCAN among various existing SW-only CAN authentication approaches due to its decent performance for both latency and bus load, as well as existing and well-documented Arduino implementation.

3.7.2 Handshake Latency

We measured the time it took to complete a handshake while varying the number of slave ECUs in a CAN domain. As outlined in Sec. 3.5.2, the handshake process is repeated every T. The old session still continues with the existing parameters until the handshake is completed. As a result, no critical message exchange during the operation mode of the previous session is interrupted. The handshake of the new session will be executed in parallel with the operation of the previous session. The only critical time when the handshake latency can affect operations of the car is during the initial start-up of the car since a session S_0 of `S2-CAN` cannot start until the initial handshake has been completed. We simulated a varying number of slave ECUs by having our two prototype ECUs take turns to send ACK of the handshake, in a ping-pong manner. We surveyed the DBCs of four vehicles (see Sec. 3.8.1) to find that each CAN bus has 9–23 different ECUs. So, we consider a maximum of 25 slave ECUs in our simulation. For two slave ECUs, the average total handshake time stands at 303ms, for five at 529ms, for ten at 907ms and for the maximum number of 25 slave ECUs, we achieve around 2 seconds of handshake latency t_{hs}, i.e., the car starts talking `S2-CAN` after 2s when it is powered on. Our calculations also show that each additional slave ECU on the bus will add an average of 75.5ms towards the

latency. Furthermore, the handshake process will be started at $P \cdot T - t_{hs} - Q \cdot t_b$ *before* the current session expires to provide a smooth transition to the next session. P denotes the session number and Q the average number of higher-priority CAN messages that can be expected to cause the blocking of handshake messages.

3.7.3 Operation Latency

CAN messages have stringent deadlines, i.e., when they must arrive at the receiver. Although the authors of [71] suggest deadlines of cyclic safety-critical CAN messages standing at 2.5–10ms, this is outdated. Modern HS-CAN buses have minimum cycle times (and thus deadlines) of 10ms, as our manual inspection of the four DBCs also confirmed. Latency measurements are averaged from a sample of 1000 messages sent over 100 seconds, or one message every 100ms. We were interested in calculating the E2E latency t_{E2E} for

1. Regular CAN with vatiCAN authentication (`"NONE"`),
2. 3DES, TEA, XOR, AES-128 and AES-256 encrypted CAN with vatiCAN authentication,
3. and finally `S2-CAN`.

In the first case, E2E latency consists of processing delays of the sender and receiver, the time to calculate the MAC on the sender and check the MAC on the receiver, as well as the CAN bus network latency. In the second case, encryption/decryption latencies are added on the respective sides. `S2-CAN` uses the latter calculation methodology as well, while the MAC and encryption/decryption latencies are replaced by the delay to calculate/check the internal ID and counter, and encode/decode through Circular Shift (CS).

Fig. 3.5 depicts the breakdown of the E2E latency for all three aforementioned cases. Furthermore, the dotted horizontal line indicates the aforementioned deadline

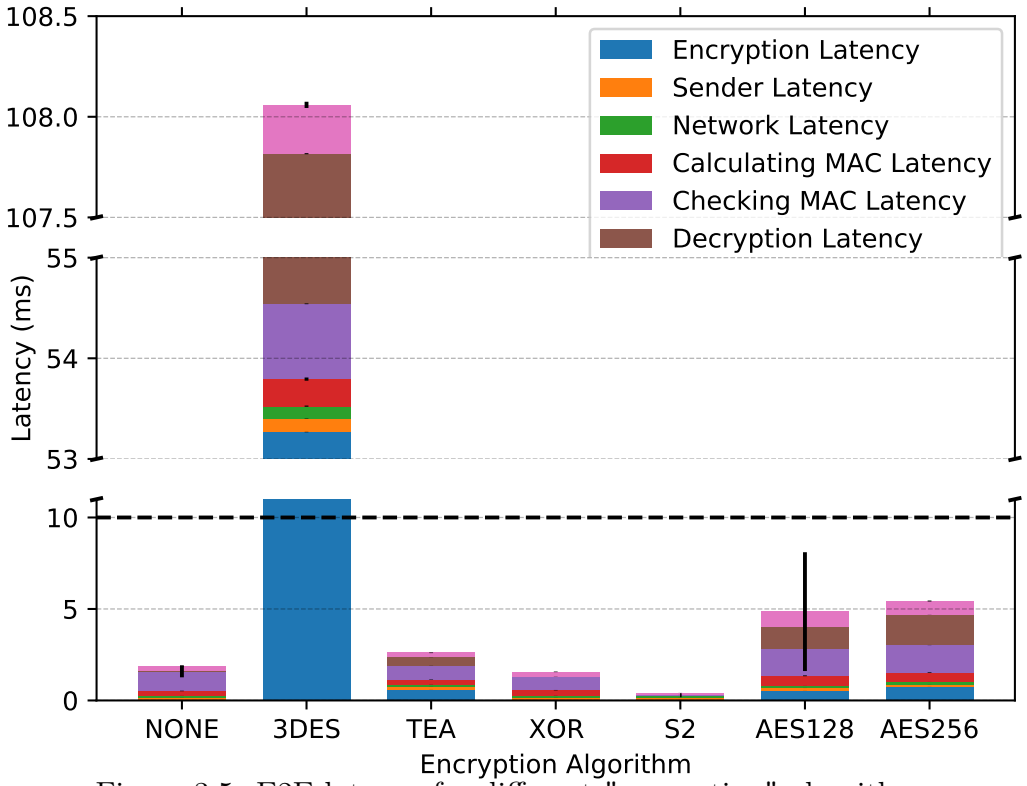


Figure 3.5: E2E latency for different "encryption" algorithms

of 10ms. It can be easily seen that the encryption/decryption of 3DES takes much longer on Arduinos than other encryption algorithms that can still satisfy the 10ms deadline. Tiny Encryption Algorithm (TEA) and XOR seem to satisfy it although they are not considered secure [18, 104] and are thus not recommended to be used in production. Furthermore, in all experiments, we did not include any additional traffic, so that the reported E2E latencies assume no blocking time due to higher-priority CAN messages and can be considered a lower bound. Hence, even AES-128 and AES-256 are likely to miss the 10ms deadline if they lose the CAN arbitration to a message with lower ID. `S2-CAN` with $t_{E2E} = 414\mu s$ satisfies both deadlines and only adds an overhead of $75\mu s$ to the E2E latency of a regular CAN message (i.e., no encryption or authentication).

Latency numbers for MAC operations by vatiCAN are lower in Fig. 3.5 than the reported 3.3ms from Table 3.1. We used a sponge capacity of $c = 8$ instead of

the original, more secure $c = 128$ to provide a lower bound for vatiCAN's latency overhead.

3.7.4 Other Metrics

Table 3.3: Benchmark of other metrics

Encr.	**Auth.**	**BL (%)**	**CPUo (%) S / R**	**RAM(kB) S / R**	**Flash(kB) S / R**
None	None	0.25	0/0	1.24/1.29	10.1/11.96
	VatiCAN	0.5	86.7/82.3	1.57/1.66	17.25/17.07
AES128	None	0.5	0.8/2.0	1.25/1.30	10.30/12.02
	VatiCAN	1	87.0/82.8	1.60/1.67	17.35/17.13
AES256	None	0.5	1.0/2.5	1.27/1.31	10.31/12.04
	VatiCAN	1	87.0/82.9	1.61/1.69	17.37/17.15
3DES	None	0.25	52.8/53.5	1.26/1.31	12.27/14.22
	VatiCAN	0.5	93.8/90.8	1.60/1.69	19.38/19.33
TEA	None	0.25	0.5/0.5	1.27/1.32	10.55/12.50
	VatiCAN	0.5	86.8/82.4	1.60/1.69	17.78/17.61
XOR	None	0.25	0.01/0.01	1.25/1.30	10.16/12.05
	VatiCAN	0.5	86.7/82.3	1.57/1.67	17.31/17.17
S2	S2 Auth	0.25	0.04/0.03	1.25/1.30	10.24/12.10

Besides the E2E latency, we measured bus load, CPU overhead, and memory usage of each encryption method with and without vatiCAN authentication. The results are summarized in Table 3.3. The metrics are calculated for messages exchanged during *Operation Mode*, unless noted otherwise.

Bus Load. The bus load (BL) b is calculated as follows [14]:

$$b = \frac{s_{frame}}{f_{baud}} \sum_{m \in M} \frac{1}{p_m}, \tag{3.3}$$

where we used $f_{baud} = 500\ kBit/s$ as baud rate on the CAN bus, and p_m is the period/cycle time of message m, and assuming each CAN frame uses 125 bits, $s_{frame} = 125$. With regular CAN (no encryption and authentication), we send one message

every p_m. AES has a block size of 16 bytes and the maximum size of the payload is 8 bytes. Thus, we send two consecutive messages, each with a period of p_m. With vatiCAN authentication, an additional MAC is sent after each message, effectively doubling the bus load. Table 3.3 shows that only `S2-CAN` does not add any overhead to the bus load of regular CAN during operation mode, but provides protections against both confidentiality and integrity. Note that the bus load does increase during each handshake due to additional $2(N + 2)$ exchanged messages. Nevertheless, the handshake adds an overhead of merely 2.5% to the bus load.

CPU Usage. CPU overhead (CPUo) c_y of ECU_y is calculated by measuring how many idle cycles pass per message. We establish regular CAN to be the baseline, then calculate overhead c_y for $y \in \{\mathbf{S}ender, \mathbf{R}eceiver\}$ as follows:

$$c_y = 1 - \frac{cycles_{idle}}{cycles_{baseline}}. \tag{3.4}$$

We see in Table 3.3 that vatiCAN authentication accounts for the largest CPU overhead. (with the exception of 3DES). The CPU utilization on each ECU almost doubles. With `S2-CAN`, we have a negligible CPU overhead that demonstrates the lightness of our approach on computational resources.

Memory Consumption. Finally, Flash and RAM usage are reported when our code compiles to the Arduinos. No dynamic memory is used. All approaches except `S2-CAN` add up to 30% more RAM and 70–90% of Flash usage compared to the memory consumption for regular CAN. The memory consumption (both RAM and Flash) for `S2-CAN` is minimal.

3.8 Security Analysis

To measure the security level of `S2-CAN`, we need to determine the time an attacker requires to correctly spoof a specific CAN message. To be more concrete, we

assume the adversary will try to accelerate the vehicle by CAN injection through the OBD-II port. Furthermore, we assume that the gateway blocks CAN messages with IDs under a certain threshold to secure the handshake (see Sec. 3.5.2) and no intrusion detection system is installed in the target vehicle. Given the current state of commercial passenger vehicle security, this is a very likely scenario. In order to affect the acceleration behavior by CAN message injection, the adversary needs to know the message format (i.e., CAN ID, signal position, scale and offset) of the signal they want to spoof. For regular CAN, this is possible by existing automated CAN bus reverse-engineering tools such as `LibreCAN` [141]. In the following security analysis, we will deploy Phases 0 and 1 of `LibreCAN` with some modifications to adapt to `S2-CAN` and try to measure the time an attacker would need to determine the correct payload to inject into the CAN bus. The modified attack tool is called `LibreCAN+`, consisting of three stages that are discussed below.

3.8.1 Experimental Setup

All experiments were conducted using Python 3 on a computer running 64-bit Ubuntu 18.04.4 LTS with 128 GB of registered ECC DDR4 RAM and two Intel Xeon E5-2683 V4 CPUs (2.1 GHz with 16 cores/32 threads each). We evaluate the security of `S2-CAN` by using one-hour real-world traces collected from four recent (2016-2019) vehicles: Veh. A is a luxury mid-size sedan, Veh. B a compact crossover SUV, Veh. C a full-size crossover SUV and Veh. D a full-size pickup truck. Veh. A, C and D have at least two HS-CAN buses, both of which are routed out to the OBD-II connector, whereas Veh. B has at least one HS-CAN and one MS-CAN, with only the former being accessible via OBD-II. All raw CAN data was collected with the OpenXC VI [2].

3.8.2 Stage 0: Generating S2-CAN Traces

The recorded traces from our four evaluation vehicles are in regular CAN-syntax. To enable S2-CAN-compliant communication, we have to process the one-hour traces according to simulated handshake parameters and convert them into S2-CAN-syntax. First, we analyze the DBC file of the vehicle to determine the ECU nodes that are present in the network, free space of each CAN ID payload, and group CAN IDs based on the node that emits them since the handshake assigns the parameters on a per-node basis. Then, we randomly assign each node a unique internal ID $\in [0, N_{ECU} - 1]$. The counter of each node is also initialized to a random number in range $[0, 2^{16} - 1]$. Third, we assign incrementing counter values for each CAN message. After specifying values for the internal ID and counter of each CAN message, we XOR the two values to obtain q_j, assign it to a free space in each CAN message (if possible) and finally OR it with the original payload. In order to be compliant with S2-CAN, the payload needs to have at least 2 bytes of free space, but these do not have to be contiguous. We removed CAN IDs from the trace that do not have the necessary free space. Finally, we perform the byte-wise circular shift (CS) on each remaining message according to the randomly generated encoding parameter f.

3.8.3 Stage 1: Cracking the Encoding

First, the adversary can assume that the targeted CAN signal is two bytes or less in size since this applies to most powertrain-related signals. In all four vehicles the target signal is 13 bits long. Next, the attacker can brute-force the CAN trace with each possible encoding for each of the 7 pairs of contiguous bytes in the CAN message. Our encoding scheme has 8 possibilities for each byte, so without accounting for duplicates, there are $8 \cdot 8 \cdot 7 = 448$ combinations an attacker must try. However, because encodings for unconsidered bytes are set to zero, we can reduce this to 400 combinations by eliminating duplicates: One combination of all zeros, $7 \cdot 8 = 56$

combinations where all but one byte are zero, and $7 \cdot 7 \cdot 7 = 343$ combinations where all but two contiguous bytes are zero. For each potential encoding, the attacker decodes the trace and runs it through Phases 0 and 1 of the original `LibreCAN`, resulting in a list of three-tuples (candidate CAN ID, encoding, normalized cross-correlation score). The pairs with the highest X correlation scores (X is a design parameter in Sec. 3.8.5) can then be used in Stage 2. Note that we used multi-threading in this stage to calculate up to 50 combinations simultaneously.

3.8.4 Stage 2: Authenticating Correctly

For the adversary to successfully spoof a message, they must be able to increment the message counter to the correct value. This requires the knowledge of the position of the counter bits within the message, the value of the counter, and the internal ID. After determining the top X CAN IDs by correlation score from Stage 1, the adversary can extract a subtrace consisting of only the messages for that candidate CAN ID. With the subtrace in hand, the adversary calculates the frequency of bit flips for each bit in the subtrace's messages, and matches these flip frequencies to what frequency the bits of a counter should be. This is done using Algorithm 3. Note that only the lowest $\lfloor \log_2(\text{trace length}) \rfloor$ bits of the counter can be determined, since these are the only bits that are guaranteed to flip at least once.

Algorithm 3 Determine Counter Position

procedure MATCH-FREQUENCY($flip\_freqs, trace\_len$)
  $counter\_length \leftarrow \min(16, \lfloor \log_2 trace\_len \rfloor)$
  $counter\_positions \leftarrow []$
  for $i \leftarrow counter\_length$ **to** 1 **do**
    $match \leftarrow \text{argmin}(\{|f - 2^{-(i-1)}| : f \in flip\_freqs\})$
    APPEND($counter\_positions, match$)
  return $counter\_positions$

After determining the position of the counter bits, the internal ID can be extracted. To do this, the adversary compares consecutive messages in the subtrace, and sees if one of the counter bits flips in the second message. If this occurs, the adversary

knows the next lowest bit of the counter must have been a 1 in the first message. Then, to extract the internal ID, the adversary XORs the counter bit with 1. This is repeated until all bits of the internal ID are known. This procedure is summarized in Algorithm 4.

Algorithm 4 Determine Internal ID

```
procedure CALCULATE-INT-ID(counter_pos, subtrace)
    c_length ← LENGTH(counter_pos)
    id_length ← min(8, c_length − 1)
    int_id ← []
    offset ← c_length − id_length
    c_pos ← counter_pos[offset : c_length]
    prev_m ← GET(subtrace, 0)
    for i ← 0 to id_length − 1 do
        for m ∈ subtrace do
            if m[c_pos[i]] ≠ prev_m[c_pos[i]] then
                int_id[i] ← prev_m[c_pos[i + 1]] ⊕ 1
                BREAK
    return BITS-TO-INTEGER(int_id)
```

Now, after obtaining the position of the counter and the internal ID, the attacker can spoof a message. First, they use the encoding determined in Stage 1 to decode the latest message from the desired CAN ID. Next, the attacker replaces the value of the signal they are spoofing with their own fabricated value in that message. Before re-encoding the message with f, the attacker extracts the counter value from the latest real-time message on the CAN bus, increments it by 1, and inserts it into their new message. This spoofed message will then be injected through the adversary's rogue node into the CAN bus and accepted by the respective receiver ECUs.

3.8.5 Difficulty of Successful Cracking

The recorded traces of all evaluation vehicles were around 60 minutes long. We integrated the above procedure into `LibreCAN` — creating a new version of `LibreCAN`, named `LibreCAN+` — and evaluated its success on those four traces using the ground truth DBC files of each vehicle. The outcome is shown in the last column of Table 3.4.

The cracking success is dependent on finding the correct CAN ID and encoding in Stage 1 (abbreviated at ST1 in the table) by picking the top candidate in the sorted correlation list, as well as determining the correct internal ID (ID) and counter (cnt). For Vehicles A, B and C, cracking `S2-CAN` with `LibreCAN+` works. Vehicle D already failed in Stage 1 to determine the correct CAN ID for spoofing the desired signal.

Furthermore, we wanted to analyze how a shorter recording would affect this metric. We re-ran all three stages with 5%, 10%, 25%, 50% and 75% of full trace length. To avoid bias towards more city or highway driving, we calculated the precision for all non-overlapping segments of this trace. As can be seen in Table 3.4, traces of 5% and 10% length fail in most cases. We color-coded the table to indicate the number of split traces cracked correctly. If all split traces can be cracked, we highlighted them in green color. Otherwise, if under 2/3 of split traces are unsuccessful, we highlighted these in red, with the remaining portion colored in orange.

Table 3.4: Cracking Success based on Trace Length (in %)

Trace Length		**5**	**10**	**25**	**50**	**75**	**100**
Veh. A	ST1	11/20	6/10	4/4	3/3	2/2	1/1
	ID	10/20	6/10	4/4	3/3	2/2	1/1
	cnt	11/20	6/10	4/4	3/3	2/2	1/1
Veh. B	ST1	12/20	4/10	3/4	2/3	1/2	1/1
	ID	11/20	3/10	3/4	1/3	1/2	1/1
	cnt	12/20	4/10	3/4	2/3	1/2	1/1
Veh. C	ST1	8/20	5/10	3/4	3/3	2/2	1/1
	ID	8/20	5/10	3/4	3/3	2/2	1/1
	cnt	8/20	5/10	3/4	3/3	2/2	1/1
Veh. D	ST1	6/20	3/10	0/4	0/3	0/2	0/1
	ID	6/20	3/10	0/4	0/3	0/2	0/1
	cnt	6/20	3/10	0/4	0/3	0/2	0/1

Table 3.4 only considers those candidates in Stage 1 with the highest correlation score ($X = 1$) that match the correct encoding and CAN ID as successful. In many cases, we observed that the second-best candidate was ideal. As a result, we also wanted to see if considering the top $X = \{2, 3, 5, 10\}$ candidates from Stage 1 would

lead to success in cracking S2-CAN . If any of the candidates in the top X were correct, we would mark ST1 for the respective vehicle and split trace as correct. Similar tables for the aforementioned values of X are presented in Appendix B.1. Based on these, we summarize the cracking performance for varying X in Table 3.5. The values are reported as average numbers over all four vehicles. Note that the color coding is different from Table 3.4. Green cells indicate that the adjacent X value to its right is identical and thus does not provide a performance improvement. We suggest using at least a trace of 25% length (15 minutes) and consider the Top 3 candidates for optimal brute-forcing success.

Table 3.5: Brute-Forcing Success for Top X Candidates

TL (%)		**Top 1**	**Top 2**	**Top 3**	**Top 5**	**Top 10**
5	ST1	46%	58%	58%	61%	65%
	ID	44%	54%	54%	58%	61%
	cnt	46%	58%	58%	61%	65%
10	ST1	45%	68%	68%	73%	78%
	ID	43%	58%	58%	63%	68%
	cnt	45%	68%	68%	73%	78%
25	ST1	63%	81%	88%	88%	88%
	ID	63%	81%	88%	88%	88%
	cnt	63%	81%	88%	88%	88%
50	ST1	67%	92%	92%	92%	92%
	ID	58%	83%	83%	83%	83%
	cnt	67%	92%	92%	92%	92%
75	ST1	63%	88%	88%	88%	88%
	ID	63%	88%	88%	88%	88%
	cnt	63%	88%	88%	88%	88%
100	ST1	75%	100%	100%	100%	100%
	ID	75%	100%	100%	100%	100%
	cnt	75%	100%	100%	100%	100%

3.8.6 Determining Session Cycle T

So far, we observed that brute-forcing S2-CAN successfully is possible. The total time t_a required by an attacker to crack S2-CAN is the sum of the passive recording time t_r, time t_{st1} to crack the encoding in Stage 1, time t_{st2} to determine the integrity

parameters in Stage 2 and time t_i to inject a well-formed CAN message on the CAN bus:

$$t_a = t_r + t_{st1} + t_{st2} + t_i \approx t_r + t_{st1}. \tag{3.5}$$

Our timing analysis shows that the time to determine the two integrity parameters int_ID and cnt on the full trace (60 minutes) takes less than one second. The time to inject the correct CAN message can also occur instantly with minimal network delay from the workstation to the adversary's CAN node (e.g., an Arduino). Hence, t_{st2} and t_i are negligible and the main contributing factors are t_r and t_{st1}.

Table 3.6: Timing analysis for full traces (minutes:seconds)

	CAN (`LibreCAN`)	**S2-CAN (`LibreCAN+`)**
Veh. A	0:27	10:33
Veh. B	0:36	18:32
Veh. C	0:26	10:42
Veh. D	0:26	10:52

As shown in Table 3.6, the total time stands at around $t_a = 70$ min for full traces (i.e., $t_r = 60$ min). Since our threat model stipulates that the attacker can also physically tap into one specific CAN bus (and thus only has access to one bus), we run `LibreCAN+` with messages from Bus 1 only. Unfortunately, due to architecture specifics of Vehicle B, all messages are logged on Bus 1, which makes the trace longer and thus affects cracking time. The attacker can only perform a CAN injection attack on a bus equipped with `S2-CAN` if the session cycle T is larger than t_a since with each new handshake, new parameters will be generated and the attacker has to re-do the entire attack. As a result, we can deem `S2-CAN` secure if the following condition is met:

$$t_a \approx t_r + t_{st1} > T. \tag{3.6}$$

In Sec. 3.8.5 an attacker was shown to succeed cracking `S2-CAN` with less passive recording time t_r. Since less messages have to be processed, t_{st1} will also be pro-

portionally smaller. With the minimum recording time $t_{r,min}$ to have a successful outcome, we can now set the maximum session cycle T_{max}. We already determined that a trace length of $t_r = 15$ minutes is sufficient to succeed. The Top X consideration does not affect the timing since Stage 2's contribution is negligible. If the attacker doesn't achieve the desired outcome (i.e., vehicle malfunction), they can repeat the process with the second and third candidates immediately. For Vehicles A, C and D, t_{st1} stands at less than 3 minutes and for Vehicle B at less than 5 minutes. Hence, the maximum session cycle T_{max} will stand at 18-20 minutes.

3.9 Discussion and Conclusion

Based on the results from the previous section, we can guarantee that `S2-CAN` is secure if the cycle time T does not exceed 18-20 minutes. The experiments were conducted on a machine with relatively good specs (see Sec. 3.8.1). Nevertheless, a determined attacker can use an even more powerful setup to brute-force `S2-CAN` faster. The feasibility of such an attack depends on the attacker's incentive, i.e., tradeoff between monetary cost and dedication towards the outcome.

To be flexible, an attacker could rent computational resources online. Both Amazon and Google provide cloud computing resources called AWS EC2 [11] and Google Cloud [35]. The main bottleneck of brute-forcing is the time required in Stage 1. Due to multi-threading the combinations, these can be linearly scaled with multiple instances. We obtained the cost of running a comparable instance to our experimental setup on AWS. Their pricing calculator [13] suggested an on-demand hourly cost of US$1.088 for an EC2 instance with 32 vCPUs and 64 GB RAM. In our experiments from Sec. 3.8, the peak RAM usage stood at 16 GB, but with the configured number of cores, EC2 did not provide any smaller instance. To brute-force `S2-CAN` with a passive recording time $t_r = 15$ minutes in less than 20 seconds, 10 EC2 instances have to be rented. This sums up to a monthly cost of $7,972.40 for the attacker. Given

that the attacker only spends $t_a \approx 15$ minutes per attempt (if $T > t_a$), they could conduct 2880 attempts per month at an average cost of \$2.77 and still fail, if T is set smaller than the minimum recording time $t_{r,min}$. Although the actual cracking (i.e., t_{st1}) can be sped up, $t_{r,min}$ acts as a lower bound to the total attack time t_a and thus the attacker will have no chance of cracking `S2-CAN`.

Finally, we would like to briefly compare `S2-CAN`'s security with `S-CAN` approaches. For instance, vatiCAN [132] discusses how long it would take to forge the SHA3-HMAC which depends on the length of the MAC tag. On average, it requires $2^{MAC\_Length-1}$ combinations to brute-force the MAC which is depicted in the last column of Table 3.1. The authors mentioned that it would still take a day to brute-force all combinations on a powerful in-vehicle ECU, but due to their nonce update interval of 50ms (comparable to our session cycle T), it would be impossible for the attacker to calculate a correct HMAC. Although the same calculation cannot be directly applied to `S2-CAN` due to lack of MAC and changing position for each CAN message, an online attacker (i.e., on an in-vehicle ECU) would require $\binom{64}{16} \approx 2^{49}$ combinations to spoof the valid 2-byte integrity parameters which allows a fair comparison with the other numbers in Table 3.1. Given modern GPUs' capabilities [68] (also considering advances since this paper's publication), an attacker with similar cost assumptions from above could brute-force `S2-CAN` in multiple hours due to its 49-bit entropy. Such an attacker would still fail if $T_{max} \approx 15$ minutes.

In this thesis chapter, we have developed `S2-CAN` by making a trade-off between performance and security, and verified its performance on Arduinos mimicking real ECUs on a CAN bus. with regards to multiple metrics. It performs better for all metrics than each surveyed `S-CAN` approach, especially reducing E2E latency. Then, we have tried to brute-force `S2-CAN` by using a modified version of the existing CAN reverse-engineering tool `LibreCAN`. Although the total attack time can be minimized to roughly 15 minutes, by setting the session cycle properly, our approach is deemed

secure. Due to both favorable performance and practically acceptable security guarantees, we envision `S2-CAN` to finally be a compelling and practical security solution for OEMs to be deployed in their vehicles in the near future.

CHAPTER IV

MichiCAN: Practical Spoofing and DoS Protection for the Controller Area Network

4.1 Introduction

The Controller Area Network (CAN) has been the *de facto* in-vehicle network (IVN) protocol for over three decades. It connects various in-car computers — called *Electronic Control Units* (ECUs) — and allows them to exchange information with each other. CAN was designed in the 1980s without security in mind and its vulnerabilities have started to get exploited in the past decade [58, 86, 109]. Researchers soon started to develop countermeasures to provide the security properties of *confidentiality*, *authenticity*, *integrity* and *availability*. All traffic in the standard CAN is in plain text, there are no provisions of sender or message authentication, and CAN is highly susceptible to availability attacks, such as Denial-of-Service (DoS).

Most CAN security research has focused on authentication. There are a myriad of publications on the prevention of CAN spoofing attacks [88, 111, 132, 177]. They all use Message Authentication Codes (MACs) to provide message integrity. Furthermore, Bozdal *et al.* [53] show how an encrypted CAN payload can prevent sniffing attacks. All these schemes use cryptography which imposes very heavy computation loads on resource-constrained ECUs and incurs a significant computation delay. Fur-

thermore, some standardization for secure on-board communication is provided by the AUTomotive Open System ARchitecture (AUTOSAR) consortium [30], although it mainly deals with integrity, authenticity and confidentiality protection.

The easiest and thus riskiest CAN availability attack is Denial-of-Service (DoS). DoS attacks target the CAN message identifier (CAN ID) by injecting CAN messages with low CAN IDs. Since CAN is a multi-master broadcast protocol and lower CAN IDs indicate higher priority, they will always win arbitration and be allowed to transmit before higher ID messages on the CAN bus. By "continuously" sending CAN messages with a low ID, higher ID messages will always lose arbitration and thus become *unavailable* on the CAN bus. Attackers can choose to make the transmission of all ECUs unavailable (*traditional* DoS) or selectively choose which ECUs to silence (*targeted* DoS). In the most powerful DoS attack — the traditional DoS — an attacker continuously injects CAN messages with ID 0x0 and blocks other ECUs' communications on CAN. The major impact of a DoS attack could be safety-critical, especially when the vehicle can no longer perform certain powertrain control functions. However, vehicles implement a *limp mode* which still allows certain safety-critical ECUs to work with limited functionality in the event of losing CAN communication [121]. Another impact of a DoS attack could be *ransomware* which is financially motivated. A targeted DoS attack can shut down ECUs with a high CAN ID (and thus lower priority). This would mostly affect convenience features such as remote keyless entry (RKE) or advanced driver assistance systems (ADAS). In modern vehicles, the loss of RKE could also prevent anti-theft systems from being disengaged and the car from being started. The victim would either need to pay ransom to the attacker or take their car to the dealership to reset/reflash the affected ECU(s), which will cost time and money.

As mentioned earlier, there have been a very few DoS countermeasures proposed thus far to detect and possibly prevent DoS attacks (see Table 4.1). But all of them

Table 4.1: Comparison of countermeasures against CAN DoS

	IDS [133], [95, 99]	**Parrot+ [67]**	**CANSentry [96]**	`MichiCAN`
Backward Compatibility	✓	✓	✗	✓
Real-Time Capability	✗	✗	✗	✓
Overhead on Network	None	Very High	Negligible	Negligible
Prevention Capability	✗	✓	✓	✓

come with their own limitations, such as lack of backward compatibility and real-time capability, as well as heavy CAN traffic overhead.

No backward compatibility. Ideally, the countermeasure against DoS attacks should be based on software. Any additional hardware or modifications of existing hardware are not backward compatible to existing cars. OEMs and the entire supply chain would have to produce or modify their products to add this countermeasure which is not viable for *cost* reasons. For instance, CANSentry [96] is a hardware-based message firewall that can defend against various spoofing and DoS attacks. However, CANSentry introduces a stand-alone device deployed between a high-risk ECU (i.e., ECU with the highest risk to be compromised) and the CAN bus which limits its backward compatibility.

No real-time capability. Since DoS attacks may impact people's safety, they have to be detected and prevented as quickly as possible, preferably in real time. Our surveyed Intrusion Detection System (IDS) mechanisms [95, 99, 133] are incapable of real-time detection of attacks. CANSentry [96] incurs an additional propagation time because the intermediate hardware must both decode and re-encode the message before passing it to the main bus. Parrot [67] is a distributed spoofing detection and prevention framework where each ECU is responsible for monitoring the bus to detect frames with its own CAN ID. If a CAN message with the same CAN ID that is not

sent by the ECU itself is detected on the CAN bus, Parrot will launch a counterattack to shut down the attacking ECU in a CAN protocol-compliant way, called *bus-off*. However, it incurs an additional delay in launching a counterattack (i.e., bus-off time) because it only starts destroying an attacker message after its second instance, with the first instance required for detection. Note that Parrot was not designed as a DoS prevention tool, but can effectively be used as such. In this case, ECUs will not only mark their own CAN ID as malicious, but also CAN IDs which are lower than their own.

Traffic overhead on network. The bus load of CAN represents how busy the bus is at any given time. To avoid the difficulty/problem of scheduling messages (with safety-critical implications), the bus load must be kept as low as possible, with 30% being a recommended upper bound [15]. When Parrot launches the counterattack, it needs to start at the exact same time as the second instance of the attacker's CAN message. As a result, Parrot *floods* the CAN bus with counterattack messages to collide with the attacker's CAN messages in a brute-force fashion. The bus load can reach up to 100% during those times, limiting/prohibiting the adoption of Parrot in real production vehicles.

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

To overcome/remedy all these limitations, we propose `MichiCAN`, a distributed software solution that can run on every modern ECU. It does not only detect DoS attacks, but can also be used for spoofing prevention, such as the original Parrot [67]. Each ECU equipped with `MichiCAN` stores a list of legitimate CAN IDs from the set

$\mathbb{E}$ of all participating ECUs in the IVN. A CAN node $ECU_i \in \mathbb{E}$ can detect a spoofing attack if their own CAN ID is transmitted by another node. ECU_i can also mark a message with a CAN ID lower than its own as a DoS attack on ECUs generating lower-priority messages than itself, i.e., other lower legitimate CAN IDs originating from other ECUs stored in the list are not affected. After marking the incoming CAN message as malicious, ECU_i will start a counterattack. Using weaknesses in the CAN error handling mechanism, it will eventually force the attacking ECU into *bus-off*, thus stopping the ECU from transmitting or receiving any CAN messages. Forcing ECUs into bus-off state is not new as previous work [60] has shown. In fact, there is a growing literature on bus-off attacks to silence legitimate ECUs [128, 135] which leverage the aforementioned vulnerabilities in CAN's error handling. The main challenge in busing off an attacker is the timing of counterattack. Since the application software can only send and receive complete CAN frames, the protecting ECU needs to know *precisely when* to start the counterattack so its message can exactly overlap with the attacker's CAN message. This was a major limitation of Parrot [67], as well as others, such as the one proposed by Cho *et al.* [60]. In contrast, `MichiCAN` is leveraging the integrated CAN controller of modern ECUs which allows the application software to gain direct read/write access to each individual bit of a CAN frame. This technique is called *bit banging*. As a result, `MichiCAN` will always be correctly synchronized to the bus and can start the counterattack at any given time, without having to flood the bus to acquire the correct start time. As a result, we will not have to flood the bus like Parrot without increasing the bus load. Furthermore, by sampling the CAN ID bit-by-bit, we will also be able to detect a spoofing or DoS attack before the end of the 11-bit CAN ID field in most cases. This will allow us to launch the counterattack much faster, thus reducing the required bus-off time for the attacking ECU. By reducing the overhead on the CAN bus in regards to the aforementioned metrics, `MichiCAN` is the first DoS prevention solution that is practical and usable by OEMs. Unfortunately,

the added software logic comes at the expense of additional CPU cycles on the ECUs. We will show through our extensive evaluations how this overhead can be minimized and why this trade-off is still favorable to OEMs.

4.2 Background

Please refer to Sec. 1.2 for a primer on the CAN bus.

4.2.1 CAN Error Handling

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

Each ECU on the CAN bus has a *transmit error counter* (TEC) and a *receive error counter* (REC). In this thesis chapter, we focus mainly on transmission errors. A transmission error occurs when a transmitting ECU observes an error frame sent by a different ECU during its transmission of a CAN message on the bus. In such a case, a CAN-compliant node will do one of two things depending on the current value of its TEC. Each ECU starts in the *error-active state*. When the counter is between 0 and 127 (in the error-active state), the node that detects the error will transmit an *active error flag* consisting of 6 dominant (logical 0) bits followed by 8 recessive (logical 1) bits as an indication to all other nodes that the transmitted

frame had an error and should be ignored. If the node's TEC is in the *error-passive state* (when the TEC exceeds 127), it will transmit a *passive error flag* consisting of 14 recessive bits. Note that the passive error does not destroy other bus traffic, and hence the other nodes will not hear "complaint" about bus errors. In both cases, the node will increment its TEC by 8 and then retransmit the message. The minimum separation between the original transmission and retransmission are 11 recessive bits (8 bits from error flag + 3 bits from IFS) in the error-active state and 25 recessive bits (14 bits from error flag + 3 bits from IFS + 8 bits from additional transmission suspension) in the error-passive state. When the TEC reaches 256, the node enters *bus-off mode* and will no longer participate in CAN traffic activities. According to the CAN protocol, a device in bus-off mode is allowed to recover into the error-active state after observing at least 128 instances of 11 recessive bits on the bus. The state diagram is depicted in Fig. 4.1.

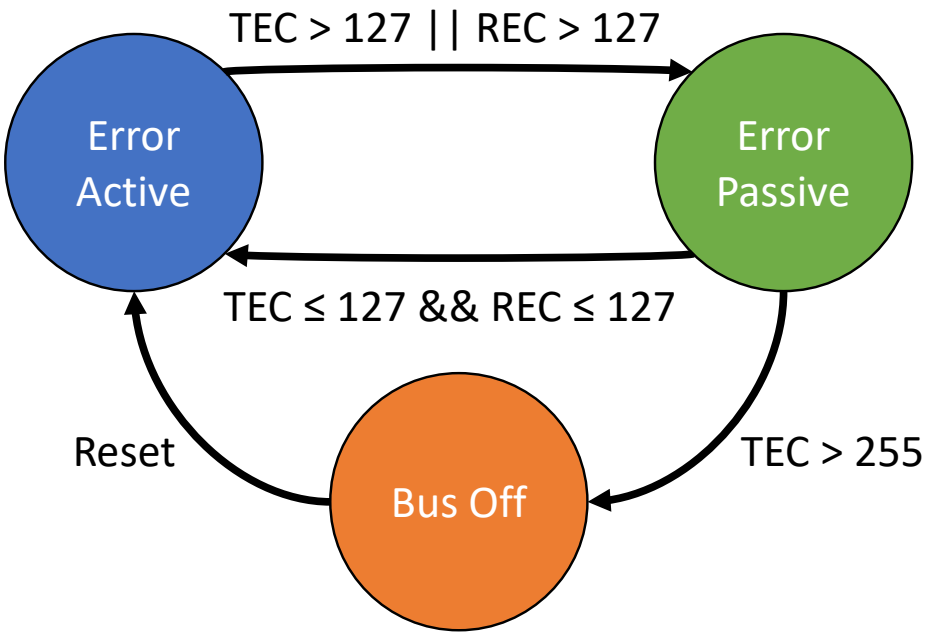


Figure 4.1: State diagram for CAN error handling

4.2.2 CAN Hardware

We will henceforth refer to ECUs connected to CAN simply as *CAN nodes*. A CAN node is usually composed of three main components: Microcontroller Unit (MCU), CAN controller, and CAN transceiver. MCU executes the application whereas the latter two are integral components of CAN bus communication. However, the controller and the transceiver have different responsibilities as they are located in different layers of the OSI stack.

CAN Controllers operate on the *data link layer* and take certain information about a CAN message (i.e., CAN ID, DLC, Data) from the application in the MCU and build a complete CAN frame (effectively a digital bitstream) as described in Sec. 1.2.1. Each CAN controller has two interfaces to the lower *physical layer*, namely CAN_TX and CAN_RX. Outbound data that will be sent on the CAN bus will be written to CAN_TX and inbound data that is read from the CAN bus will be on CAN_RX. Furthermore, the CAN controller implements the core logic of the CAN protocol such as error handling. It is also responsible for adding and removing stuff bits.

CAN Transceivers, also known as CAN PHYs, operate on the *physical layer*. They are responsible for translating digital bitstreams from CAN_TX to an analog voltage (in the 0-5V range) and generating a bitstream from the analog voltage for CAN_RX. CAN uses differential voltage signaling using two levels CAN_H and CAN_L. For instance, a high-speed CAN transceiver (which we use in this chapter) interprets a differential voltage (i.e., $|CAN\_H - CAN\_L|$) of up to 0.5V as a recessive bit, while a differential voltage that exceeds 0.9V is considered as a dominant bit.

In the last decade, the internal design of CAN nodes had been gradually changing (apart from better specifications, such as CPU and memory). An overview of this evolution is depicted in Fig. 4.2. In early CAN nodes (A), the MCU, CAN controller and transceiver were separate chips, such as Microchip's MCP2515 [36] and

MCP2551 [37]. The MCU/application would send and receive CAN frames from the controller via SPI. Such a CAN node could be recreated by an Arduino Uno with a CAN bus shield [197] that includes both the transceiver and the controller. In recent years (CAN Node B), the CAN controller and the transceiver are found combined in a single chip. One example for this is the popular MCP25625 [38]. The main driver behind this integration is the reduction of cost and physical space. The MCU interacts with the combined/integrated chip via SPI, just like in CAN Node A. The main novelty lies in CAN Node C which represents novel ECUs. It consists of an MCU with an integrated/on-chip CAN controller. The latter are embedded in MCUs and allow memory-mapped access to CAN bus functions. In many MCUs, this involves access to interpreted CAN data, configuration of filters, and access to interrupts on arrival of new messages. Further, MCUs tend to allow the user to multiplex the pins within the hardware at run-time, e.g., allowing the application software to directly read and write each bit of the CAN_RX and CAN_TX lines. In Nodes A and B, the application could only pass certain CAN message fields such as CAN ID, DLC and data to the CAN controller which would be responsible for generating a complete CAN frame. The application could also only process the data from an incoming CAN message after successful receipt of the entire frame. `MichiCAN` exploits the MCUs with integrated CAN controllers to detect and prevent DoS attacks as fast as possible (see Sec. 4.4). On-chip CAN controllers are already widely used by major ECU manufacturers such as NXP, ST or Renesas [17, 19, 41]. One example of an MCU with integrated CAN controller is the Renesas V850ES/FJ3 MCU [41] which was even used in the infamous Jeep hack [86] in 2015.

4.3 Threat Model

Due to the CAN's lack of security support, its attack landscape is wide and extensive [53]. As briefly discussed in the Introduction, an adversary can have different

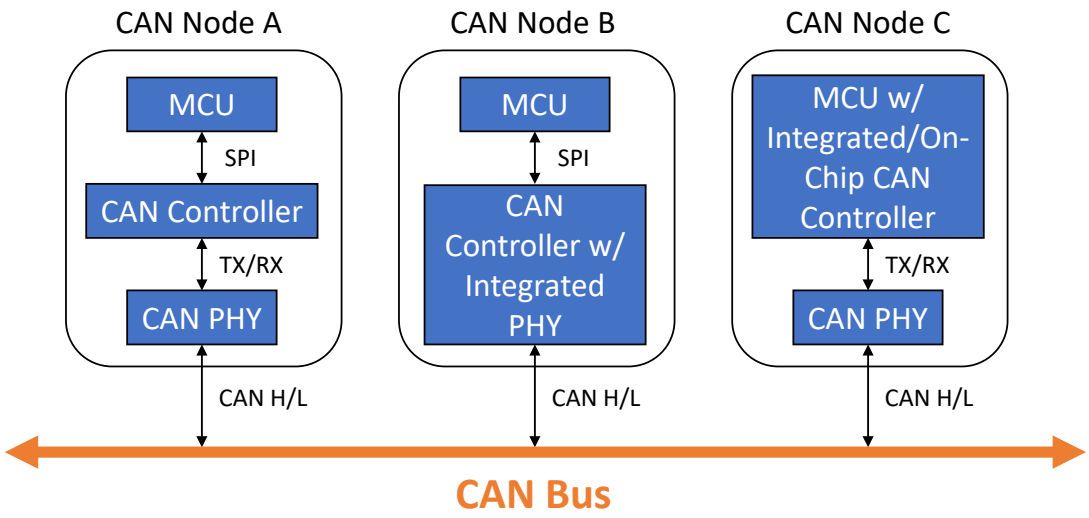


Figure 4.2: Evolution of CAN hardware in ECUs

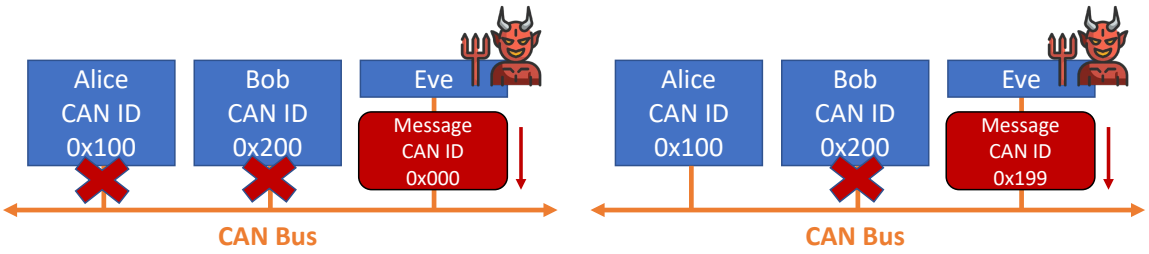
(e.g., safety-critical or monetary) incentives for attacking the CAN bus of a vehicle. These incentives can be fulfilled by a CAN injection attack which has always been the final step of any attack seen/reported during the last decade, irrespective of its sophistication level. The attacker can either have physical access to the CAN bus [58, 109] (through the OBD-II port inside the vehicle) or remotely compromise an ECU [86] (e.g., the infotainment ECU which has wireless connections to the outside). Although physical access to the vehicle might sound like an infeasible attack vector, recent research [185] has shown that remote attacks can also be caused by exploiting vulnerabilities in wireless OBD-II dongles. Many commercial OBD-II dongles feature Wi-Fi or cellular capabilities which open a new over-the-air attack surface. A one-time physical access to the vehicle would be sufficient to gain remote connection to the CAN bus. For instance, passenger vehicles are left unattended for valet parking. A malicious valet can install such a wireless dongle in the vehicle which will very likely go undetected by the victim or the vehicle owner due to its small size and difficult-to-find location, usually under the steering wheel column. In any case, the attacker needs to inject a well-formed CAN message to the in-vehicle network to achieve a visible/perceivable outcome, e.g., accelerating the car without any legitimate driver's

input. Cho *et al.* [61] described the following three possible CAN injection attacks: *fabrication, suspension, and masquerading.*

Fabrication attacks allow the adversary to fabricate and inject CAN messages with a legitimate CAN ID, but with arbitrary data. Since there is not any means of authentication on CAN, other ECUs on the CAN bus will not know if the source of this message is legitimate or malicious. This is the weakest form of CAN injection attack and can be considered a basic *spoofing attack*. In case the attacker attaches their attacking node to the CAN bus (instead of having remotely compromised a legitimate ECU), the legitimate ECUs will keep transmitting, and the adversary needs to transmit fabricated CAN messages at a higher frequency to override the data of CAN messages from legitimate ECUs. We refer to this adversary as an *external* attacker. In case an ECU has been compromised remotely (such as in the Jeep hack), the attacker has full control over that ECU and can transmit a CAN message with the original/-genuine CAN ID, but with malicious payload. This adversary is called an *internal* attacker. Although shown to work, remote ECU compromises require significant effort to achieve CAN bus access and cannot be thwarted by current state-of-the-art defenses [139]. Technically, there is no possible way to distinguish a compromised ECU from a legitimate one by purely monitoring the CAN bus. Good security practices, such as ECU hardening and network segmentation on the gateway [140], should be followed to make remote compromises more difficult.

Suspension attacks on the victim ECU prevent its transmission of legitimate CAN messages by "silencing" it and are effectively *DoS attacks*. According to [133], there are three types of DoS attacks on CAN bus: *traditional*, *random*, and *targeted*. Traditional attacks use the lowest possible priority CAN IDs (0x0) to always win arbitration and silence *all* ECUs on the CAN bus. Random attacks send messages with a random CAN ID ranging from 0x0 to the highest legitimate CAN ID in the IVN. Targeted DoS attacks only send messages with a constant, targeted CAN ID.

Random and targeted DoS attacks are very similar by silencing only a subset of ECUs since CAN messages with lower message IDs will still be able to win arbitration. In what follows, we will focus on targeted and traditional DoS attacks. Fig. 4.3 illustrates these two types of DoS attacks. In contrast to fabrication attacks, DoS attacks can be detected on the network even if the adversary is an internal attacker due to its use of different CAN IDs.



(a) Traditional DoS (b) Targeted DoS

Figure 4.3: Different types of DoS attacks [133]

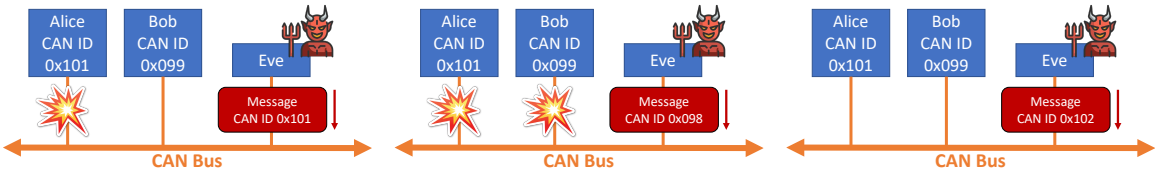
Masquerade attacks combine both of the above attacks by first suspending a legitimate ECU's CAN broadcast and then fabricating its data field. It shows why preventing DoS attacks is of utmost importance for a secure CAN bus.

As discussed in the Introduction, `MichiCAN` can both detect and prevent spoofing (in the case of an external attacker) as well as DoS attacks (for both internal and external attackers). As a result, `MichiCAN` provides protection against all types of CAN injection attacks that are feasible at the time of this writing. We assume that the adversary is operating within the CAN protocol which is a necessary pre-requisite for *any* countermeasure to work. For instance, an attacker that does not adhere to the CAN error-handling mechanism can never be confined to the bus-off state.

4.4 System Design

`MichiCAN` operates in five phases: *Initial Configuration* is done offline and only once by the OEM at the time of vehicle manufacturing, *Synchronization* and *Detec-*

tion are performed for each received CAN message, whereas *Pin Multiplexing* and *Prevention* phases get engaged only if an incoming CAN message is malicious, i.e., a spoofing or DoS attack. Below we detail these five phases.



(a) Spoofing (b) Denial-of-Service (c) Miscellaneous

Figure 4.4: Attack Variants

4.4.1 Initial Configuration

As briefly discussed in the Introduction, `MichiCAN` is a distributed solution and needs to be implemented on every ECU on the IVN. Suppose there are N ECUs on the IVN, all of which are equipped with integrated CAN controllers as described in Sec. 4.2. We define an ordered list of all ECUs as $\mathbb{E} = \{ECU_1, \ldots, ECU_N\}$. Without loss of generality, assuming that each ECU transmits one unique CAN ID, $ECU_i \in \mathbb{E}$ is equal to its CAN ID. As restrictive as this may sound, `MichiCAN` also generalizes to ECUs transmitting more than one CAN ID, as long as the same CAN IDs are not transmitted by any other node — which has been followed in all production vehicles. For an easier understanding, we will henceforth stick to the above assumption and nomenclature. In this ordered list of ECUs, ECU_1 would have the lowest CAN ID and thus the highest priority, whereas ECU_N has the highest CAN ID and the lowest priority.

Similar to Parrot, $ECU_i \in \mathbb{E}$ detects a spoofing attack (see Fig. 4.4a) if it observes a CAN message with CAN ID ECU_A (injected by the adversary) that is equal to ECU_i's:

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

ECU_i detects a DoS attack (see Fig. 4.4b) if it observes a message with a lower CAN ID ECU_A than its own ID that does not originate from any other legitimate ECU:

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

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

Last but not least, an attacker can inject a message with CAN ID ECU_A higher than ECU_N which is equal to the highest CAN ID in the IVN (see Fig. 4.4c). This is called a *miscellaneous attack*:

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

If the attacker injects this message at the same time as another CAN message, it will lose arbitration. If the message is injected during *bus idle*, i.e., when there are no other CAN messages transmitted on the bus, the attacker will naturally win arbitration and broadcast its message. Since no other ECUs know (or listen to) this CAN ID, there will be no perceivable impact on the vehicle's operation. The only drawback is that a higher-priority CAN message (with a lower CAN ID) will need to wait until the attacker's message has completed transmission. Given that an average CAN frame consists of 125bits, the blocking time at a 500kBit/s bus speed is 250μs. The higher-priority message which has been buffered by the legitimate ECU will then start transmitting its message after 11 recessive bits on the bus. Even if the

attacker repeats its attack and finds a suitable bus-idle time, the maximum blocking delay for the legitimate ECU is much smaller than the deadline for safety-critical CAN messages which stands around 10ms [71]. As a result, miscellaneous attacks can never shut down legitimate CAN communications and thus do not pose a serious threat to the CAN bus. Thus, we will focus on spoofing and DoS attacks from the previous definitions. Each `MichiCAN`-equipped $ECU_i \in \mathbb{E}$ needs to store the *detection ranges* $\mathbb{D}$ of CAN IDs that it needs to mark as malicious:

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

Since integrated CAN controllers allow direct read access to every bit of the incoming CAN frame C during its transmission, the detection ranges $\mathbb{D}$ can be encoded as a *finite state machine* (FSM). In effect, the FSM is a binary tree since each bit transition can be either 0 or 1. The root of the tree is the start-of-frame (SOF) bit since the 11-bit CAN ID $C = c_0||...||c_{10}$ will immediately follow that bit. The FSM is run for each bit individually and needs to traverse all 11 bits only in the worst case. If a decision can be made after 11-th bit or earlier, it will terminate since $C \in \mathbb{D}$ and set the *malicious* flag to true. Alternatively, if $C \notin \mathbb{D}$, the FSM will set the aforementioned flag to false.

This initial configuration phase is done offline only once by the OEM or Tier-1 suppliers during the development of the vehicle. The FSMs are generated in four stages as detailed below. Each generated FSM is unique for one particular ECU_i and will then be added to the corresponding ECU's source code. To better illustrate and explain the respective stages, let us consider an example with $\mathbb{E} = \{0x100, 0x101, 0x110, 0x150\}$. Table 4.2 provides the CAN IDs' binary and decimal representations as well. Below we will use all $ECU_i \in \mathbb{E}$ as a binary 11-bit string. We further define $ECU_{i,j}$ with $i \in [1, N]$ and $j \in [0, 10]$ to address each respective bit of the CAN ID, while $j = 0$ indicates the most significant bit (MSB).

Table 4.2: Example IVN Configuration

ECU_i	Hex	Binary	Decimal
ECU_1	0x100	001 0000 0000	256
ECU_2	0x101	001 0000 0001	257
ECU_3	0x110	001 0001 0000	272
ECU_4	0x150	001 0101 0000	336

Stage 1: Find Globally Malicious Bits. Theoretically, up to 2,048 CAN IDs can be encoded with an 11-bit message identifier. A typical size of $\mathbb{E}$ — i.e., used CAN IDs in an IVN — usually does not exceed 200 (see Sec. 4.5). Over 1,800 CAN IDs — or 90% of $\mathbb{E}$ — remain unassigned. As a result, certain bits j in CAN IDs $ECU_{i,j} \in \mathbb{E}$ *may* stay constant. We call this bit j *globally malicious* (GM) since the opposite value at that specific bit position of an incoming CAN ID C (i.e., c_j) would immediately indicate that $C \in \mathbb{D}$. For each bit j, we define a variable GM_j and set it to true only if the bit j is globally malicious:

$$GM_j = \neg \oplus_{i=1}^{N} ECU_{i,j}. \tag{4.1}$$

Looking at the example from Table 4.2, $GM_0 = GM_1 = GM_2 = GM_3 = GM_5 = GM_7 = GM_8 = GM_9 = 1$. Fig. 4.5 shows the binary tree for our example (globally malicious bits labeled as GM). The depth of the tree for 11-bit CAN IDs is 11. Due to the small size of this example, there are several GM bits. In reality, with a growing $|\mathbb{E}|$, the number of GM bits will drop, often to zero. Nevertheless, this stage is very efficient in terminating the FSM as early as possible in the presence of GM bits. This will reduce detection latency and CPU overhead on the ECU since it need not run the FSM for the less significant bits. In Fig. 4.5, all branches below a GM bit get pruned. The algorithm for marking GM bits is described in Algorithm 7 (see Appendix C.1).

Stage 2: Identify Malicious Outliers. All 8 unpruned leaf nodes $L \in \mathbb{L}$ are *potentially non-malicious* CAN IDs. Since our example only has 4 legitimate CAN IDs

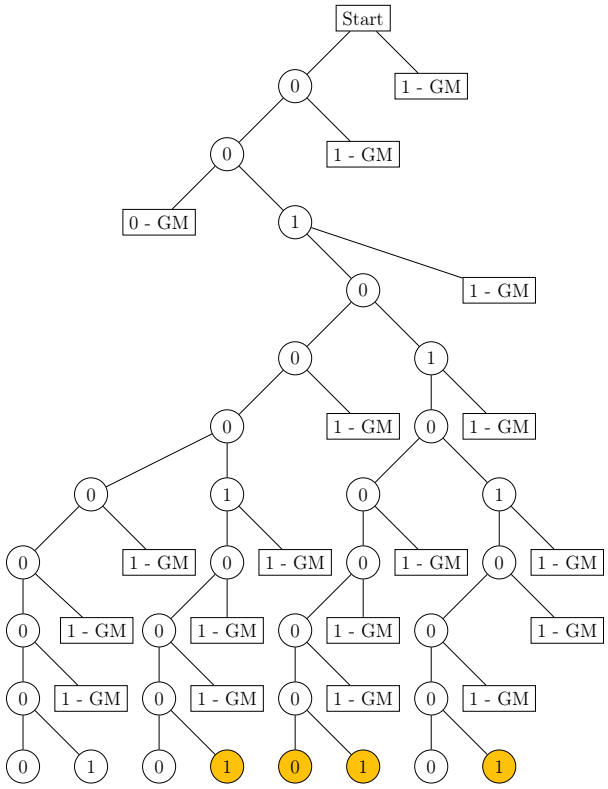


Figure 4.5: Example binary tree

$ECU_i \in \mathbb{E} \subset \mathbb{L}$, the other 4 *outliers* $O \in \mathbb{L} \backslash \mathbb{E}$ need to be removed in the next stage. In the example of Fig. 4.5, these outliers are highlighted in yellow. As mentioned before, four leaf nodes, namely 0x111, 0x140, 0x141 and 0x151, are outliers. In the next stage, we will generate local prefixes to remove these outliers. The algorithm for identifying malicious outliers is described in Algorithm 8 (see Appendix C.1).

Stage 3: Generate Local Prefixes. A straightforward and intuitive way to remove outliers is to parse all 11 bits of the CAN ID and set the malicious flag to true for them. In the above example, we only have 4 outliers which can be much larger in an IVN with more ECUs. Hence, the FSM would become very large and computationally heavy. It would also increase the detection latency since the entire CAN ID has to be parsed. If we can guarantee an outlier after p bits, we can terminate the FSM at that bit position and set the malicious flag to true. The bit sequence of the outlier's CAN ID up to that bit position p is called *local prefix*, denoted as

$O^p = o_0||...||o_p$. It is the minimum bit sequence that needs to be parsed to distinguish the outlier O from a truly non-malicious CAN ID $ECU_i \in \mathbb{E}$. The bit position p for an outlier O can be calculated by logically XORing the bit sequence of O with ECU_1. Bits that match will generate a 0 as the result of the XOR operation and the first bit that is different will yield a 1. At the first occurrence of 1, we can finally distinguish O from ECU_1. To account for the worst case, we need to repeat with every $ECU_i \in \mathbb{E}$ and choose the latest occurrence as the minimum bit position p:

$$p = 11 - \log_2(min(o_j \oplus ECU_{i,j})), \forall\ ECU_i \in \mathbb{E}. \tag{4.2}$$

In our example, 0x111, 0x140, 0x141 and 0x151 were the outliers. Logically XORing the first outlier, 0x111 with every $ECU_i \in \mathbb{E}$ shows that the largest bit sequence until a logic 1 appears is with CAN ID 0x110. As a result, the local prefix for outlier 0x111 will be $O^{11} = 00100010001$ and all $p = 11$ bits need to be checked for setting the malicious flag to true. The same applies to outlier 0x151. For the other two outliers, both local prefixes evaluate to $O^7 = 0010100$ and a decision can be made early at $p = 7$. The procedure for generating local prefixes is described in Algorithm 9 (in Appendix C.1).

Stage 4: Generate FSM Code. In the last step, we take the GM bits and local prefixes from previous stages and generate the FSM. The C++ code for the example from Table 4.2 is provided in Listing C.1 (in Appendix C.1). The code is specifically generated to be uploaded to ECU_4 with CAN ID 0x150 from our example. As discussed before, miscellaneous attacks will be ignored (lines 7-9) and spoofing attacks detected (lines 10-13). Next, the logic from the previous stages is used to generate the if-statements for DoS attack detection. The first comparisons are made for GM bits (lines 14-45) since they can cover a wide range of CAN IDs in the detection range. The last three if-statements cover local prefixes (lines 46-57). The generated

FSM code for another $ECU_i \in \mathbb{E}$ only differs between lines 7-13. The state value needs to be changed to the decimal representation of each respective ECU_i. The procedure for generating the FSM code is described in Algorithm 10 (see Appendix C.1).

Alternative FSM Generation. The generated FSM gets larger with the number of GM bits or local prefixes as more if-statements will be added to its code and thus increase its complexity. We analyzed the complexity of FSMs with different IVN sizes $|\mathbb{E}|$ in Sec. 4.5.2. Since the complexity of the FSM will have an effect on the required CPU cycles, we want to minimize the number of if-statements in the generated code. Currently, each $ECU_i \in \mathbb{E}$ will detect both spoofing and DoS attacks. This enhances reliability and robustness since each ECU_i will detect a malicious transmission simultaneously. This is very beneficial in case legitimate ECUs fail. Even if $|\mathbb{E}| - 1$ ECUs fail (which is highly unlikely), one ECU can still detect the attack.

Alternatively, if the IVN is composed of a large number of ECUs, we can split $\mathbb{E}$ equally into two subsets $\mathbb{E}_1$ and $\mathbb{E}_2$ of size $\frac{|\mathbb{E}|}{2}$ each, with the former subset containing the lower half of CAN IDs and the latter the upper half. $\mathbb{E}_2$ will run the above-described procedure. In contrast, $\mathbb{E}_1$ will only detect spoofing attacks (on their own respective CAN IDs). The FSM code can be truncated to lines 1-13 in this case. The advantage of this approach is that the lower half of CAN IDs (which are higher-priority and usually more safety-critical) will execute the FSM very fast which incurs lower computational overhead to those ECUs. Nevertheless, the network will still be protected from DoS attacks since all ECUs in $\mathbb{E}_2$ will still run the DoS protection routine. We define this as a *light scenario*, whereas every ECU running the original FSM (including spoofing and DoS protection) is called *full scenario*.

4.4.2 Pin Multiplexing

As mentioned before, modern ECUs/MCUs are equipped with an integrated CAN controller. The CAN transceiver (also called **CAN PHY**) is a standalone chip that

converts the analog CAN_H and CAN_L differential voltage to a digital bitstream (CAN_TX/CAN_RX), and vice versa. MCUs interface outside/peripheral components using their *peripheral I/O* (PIO) controller. Broadly speaking, there are two categories of PIO pins: *System I/O* (SIO) and *general-purpose I/O* (GPIO). For instance, an ECU features SIO pins to connect to the CAN PHY. By default, these pins are usually only read by the CAN controller (a system component of the MCU package) since the application software does not need access to this low-level bitstream. The application can interact with peripheral I/O using its GPIO pins. Nevertheless, the PIO controllers of modern MCUs have multiplexing capabilities which allow a GPIO pin to be multiplexed to a SIO pin. As a result, the PIO controller can be configured such that the ECU's application software has direct access to the CAN_TX/CAN_RX lines, which, in turn, allows the ECU to directly read and write every single bit on the CAN bus. Pin multiplexing is depicted in Fig. 4.6.

Pin multiplexing can be configured dynamically in software, i.e., can be done once at boot time or anytime while the MCU is running. `MichiCAN` requires read access to the CAN_RX line once booted up, but write access to CAN_TX only when it starts a counterattack. After the counterattack has been completed, `MichiCAN` will deactivate the multiplexing. Pulling the CAN bus low after the counterattack would destroy all traffic on the CAN bus and pulling it high would cause issues with non-malicious CAN messages, as each CAN controller has to acknowledge the receipt of a well-formed CAN message by writing a dominant bit to the ACK bit in the CAN trailer. If the CAN_TX pin multiplexing is still active, the MCU would have to continue sampling until the trailer (at the end of the CAN frame!) and correctly insert a dominant bit at the correct bit time, unnecessarily increasing the computational cost.

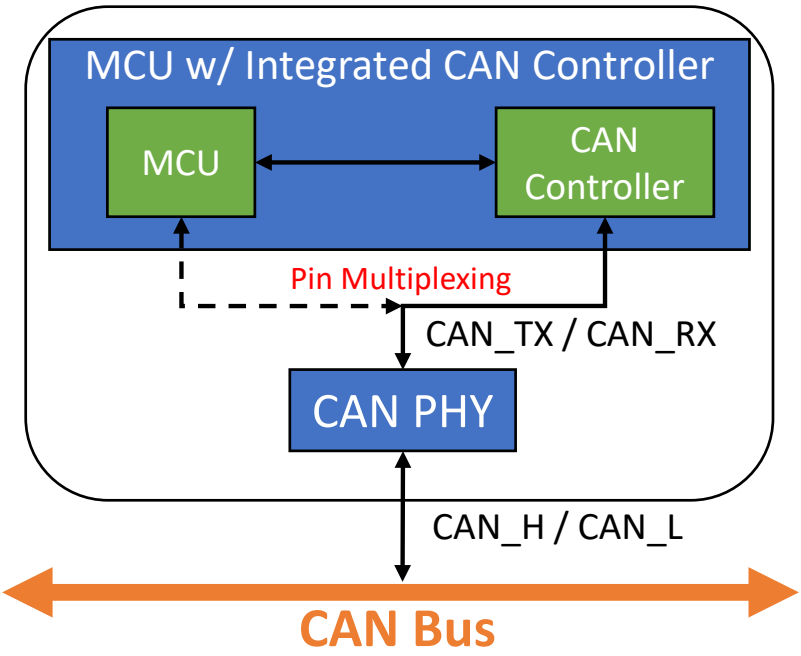


Figure 4.6: Pin multiplexing. Straight lines depict connections to SIO pins, but can be multiplexed to GPIO pins (dashed lines).

4.4.3 Synchronization

All ECUs on the CAN bus have their own clock and need to be synchronized to sample the bus reliably and correctly. This is especially important during the arbitration phase when each ECU competes for transmission. A discrepancy between ECUs' clocks would result in errors which need to be avoided if possible. Since all ECUs operate on the same bus speed (e.g., 500 kBit/s), their *nominal bit time* is fixed (e.g., 2μs). During that bit time, either a logical 0 or 1 will be observed by all ECUs on their CAN_RX pin. Due to bit transitions (e.g., from 1 to 0) and hardware imperfections, sampling the bit right at the beginning of the bit time might result in sampling a wrong logical value. To avoid this problem, CAN controllers usually sample the bit at 70% within the nominal bit time. Fig. 4.7 depicts how a nominal bit time is split into 10 time quantas (TQ) and indicated the desired sampling point. CAN controllers continuously re-synchronize due to oscillator/clock drifts. A hard synchronization is done at each start-of-frame (SOF) bit, i.e., when a transition from

1 to 0 occurs after at least 11 recessive bits during the idle bus.

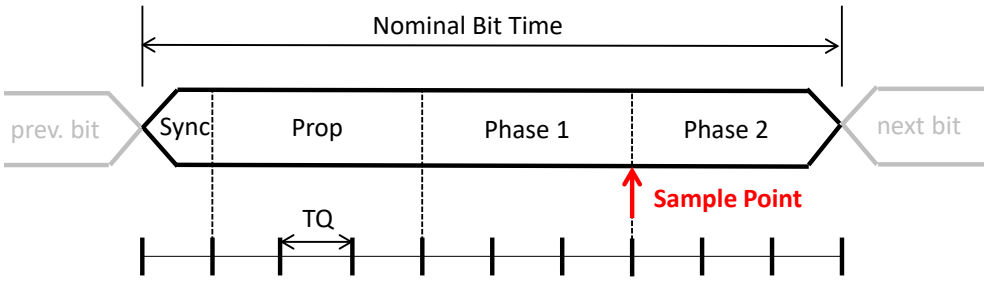


Figure 4.7: CAN bit timing

`MichiCAN` has to replicate the synchronization process in software since we are circumventing the CAN controller. One simple way is to trigger timer interrupts every bit time (e.g., 2μs) and then read in the value from CAN_RX. However, there are two issues with this straightforward approach: (i) we cannot guarantee *where* each bit is sampled and (ii) due to oscillator drift of the clock that the timer interrupts use, the interrupts will not be triggered at the same location within each bit time. To overcome this, we introduce an additional *external* interrupt that will be triggered at each SOF, indicated by a bit transition from 1 to 0 after at least 11 recessive bits. At this point, we can restart the main timer interrupt to trigger at 70% of the nominal bit time. For a 500kBit/s CAN bus, the timer interrupt would first activate after 1.4μs. Since we also reset the FSM and some other counter variables at the beginning of each CAN frame (which takes a constant number of clock cycles), we need to account for this when we restart the timer interrupts. As a result, we will first trigger the interrupt at a constant delta (called *fudge factor*) less than 1.4μs. This can be determined empirically since the required clock cycles (and thus execution time) for the fudge factor will always be constant. Since we already know that the current bit is the SOF, we can just skip this bit and restart the timer interrupts for the first bit of the CAN ID. When we execute the main interrupt handler for the first time (i.e., during the first bit of the CAN ID), we will disable the interrupt timer and restart it to trigger every 2μs since there will be no additional operations (such as resetting

the FSM).

4.4.4 Detection

Since the CAN_RX can be read directly and we are properly synchronized to the CAN bus, `MichiCAN` can start with the detection routine. The latter is described in the first half of Algorithm 5. The main interrupt handler will trigger for the first time at the first bit of the CAN ID. At the very beginning of the interrupt handler, we read the bit from CAN_RX. Since we are using a PIO controller for pin multiplexing, we can directly read the value of CAN_RX from the MCU's registers (line 2). This avoids using an external read function from the MCU's libraries which would add unnecessary computational overhead. Then, we increment a counter to keep track at which bit position within a CAN frame `MichiCAN` is located. Since the interrupt is triggered every bit time, each execution of the interrupt handler will correspond to a new bit in the frame. As mentioned in Sec. 4.2, CAN_RX will contain *stuff bits* which are automatically inserted by the CAN controller if there are more than 5 bits of the same polarity. As a result, we need to detect and identify these stuff bits (lines 6-18). While we are reading the 11-bit CAN ID, `MichiCAN` needs to remove those before appending them to a $frame$ array. For each bit (that is not a stuff bit), `MichiCAN` runs the FSM that is outlined in Sec. 4.4.1. Once the FSM determines that the CAN ID indicates a spoofing or DoS attack, the malicious flag *start_counterattack* will be set to true. To reduce computational overhead, `MichiCAN` will then stop running the FSM for the remaining bits of the CAN ID (lines 11-12 and 16-17) and just continue monitoring stuff bits.

4.4.5 Prevention

Once `MichiCAN` sees that the *start_counterattack* flag has been raised, it will execute its prevention routine. The basic idea behind attack prevention is to launch

Algorithm 5 Main interrupt handler

```
1: function INTERRUPT_HANDLER()
2:     value ← Read CAN_RX register with PIO controller
3:     if sof == True then
4:         cnt ← cnt + 1
5:         if cnt < 25 then
6:             if frame[cnt-2] != value && stuff==5 then
7:                 stuff ← 0
8:                 cnt ← cnt - 1
9:             if frame[cnt-2]==value && stuff < 5 then
10:                frame[cnt-1]← value
11:                if !start_counterattack then
12:                    state_machine_run(value)
13:                stuff ← stuff + 1
14:            if frame[cnt-2] != value && stuff < 5 then
15:                frame[cnt − 1] ← value
16:                if !start_counterattack then
17:                    state_machine_run(value)
18:                stuff ← 0
19:        if cnt == 20 then
20:            Disable CAN_TX Multiplexing
21:            sof ← False
22:            cnt ← 0
23:        else if cnt == 13 then
24:            start_counterattack ← False
25:            Enable CAN_TX Multiplexing
26:            Pull CAN_TX Low
27:    else
28:        if value == 1 then
29:            cnt_sof ← cnt_sof + 1
30:        else if value == 0 && cnt_sof < 11 then
31:            cnt_sof ← 0
32:        if value == 0 && cnt_sof ≤ 11 then
33:            sof ← true
34:            cnt_sof, frame[0] ← 0
35:            stuff, cnt ← 1
36:            reset_state_machine()
```

a counterattack and bus off the attacker node according to CAN error-handling rules (see Sec. 4.2.1). This can be accomplished by causing an error in the adversary's transmission. The two error types we are exploiting are *bit* and *stuff* errors which can be achieved by transmitting a sequence of dominant bus levels. Note that we are not sending a complete CAN message, but merely pulling the bus low for a period of time. Since dominant bus levels always overwrite recessive ones, a recessive bit transmission by the adversary will result in a bit error. Alternatively, if the attacker ECU transmitted 5 consecutive dominant bits, it will be followed by a recessive stuff bit. Since we are pulling the bus low by transmitting dominant bus levels, the stuff bit will be overwritten by another dominant bit which results in a stuff error. Fig. 4.8 depicts the prevention routine. Two major questions to address are (i) when to start injecting a dominant sequence of bits, and (ii) how many dominant bits to inject.

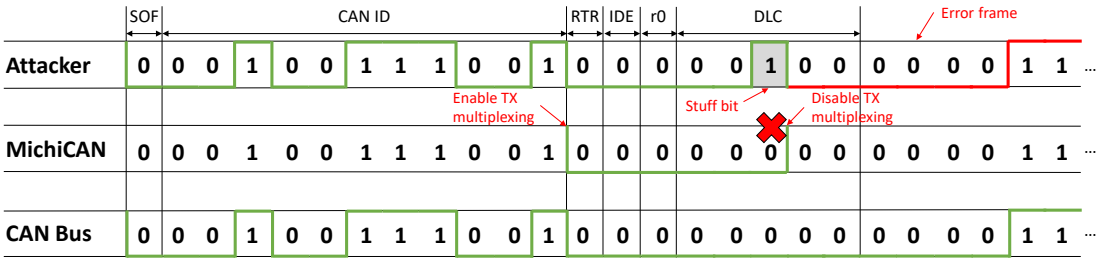


Figure 4.8: `MichiCAN` prevention routine

Although the detection routine of `MichiCAN` can terminate before the end of the CAN ID field, we cannot start injecting dominant bits during arbitration. This would lead to the attacker losing arbitration, but not generating an error frame. Since our goal is to bus off the attacker, `MichiCAN` needs to inject the first dominant bit right after the CAN ID field, i.e., during the RTR bit (see Fig. 1.3). Since this bit (and the following IDE and r0 bits) is already dominant, no bit error will be generated. The following DLC field is usually encoded as "1000" (since CAN messages mostly consist of 8 data bytes), so the earliest bit error can be caused at the fourth bit. Hence, the minimum duration that `MichiCAN` needs to pull the bus low to cause a bit error is four

bits. However, if the least-significant bits (LSB) of the CAN ID consist of consecutive dominant levels, a stuff error can be caused as early as in the RTR bit. For this to happen, the five LSBs of the CAN ID need to be dominant. It will be sufficient for `MichiCAN` to just transmit one dominant bit during the RTR slot to raise an error frame. In the worst case, if the CAN data field consists of only one byte (indicated by DLC bit string "0001"), causing a stuff error will require to inject 6 dominant bits if the LSB of the CAN ID is recessive. To sum up, an error frame can be caused by `MichiCAN` injecting 1–6 dominant bits. Since this will depend on several factors (such as the DLC length) that are unpredictable during sampling the CAN frame, `MichiCAN` needs to make sure to inject 6 dominant bits. The worst-case scenario is depicted in Fig. 4.8. As described in Sec. 4.4.2, the CAN_TX pin is disabled by default. At frame position 13 (1 SOF + 11 CAN ID + 1 RTR), `MichiCAN` will enable CAN_TX multiplexing, pull the pin low and set the *start_counterattack* flag to false (lines 23-26 in Algorithm 5). At frame position 20, it will then disable CAN_TX multiplexing which will automatically stop pulling the bus low, set the start-of-frame flag to false and the frame counter to 0 since `MichiCAN` is done processing the frame (lines 19-22).

The attacker (which has to comply with the CAN protocol) will immediately raise an *active* error frame consisting of six dominant bits followed by eight recessive bits. Even if `MichiCAN` would have succeeded in only transmitting one dominant bit (in the best-case scenario as outlined above), five additional dominant bits will not do any harm due to the six-bit dominant error flag. The transmission error counter (TEC) of the attacking ECU will be increased by 8 and it will attempt a retransmission after a total of 11 recessive bits (lines 27-36). At the SOF bit, several variables, as well as the FSM will be reset. `MichiCAN` will repeat the detection procedure and start another counterattack. After a total of 15 retranmissions, the attacking ECU will transition into its error-passive region and start transmitting *passive* error frames. After another

16 retransmissions (summing up to a total of 32 attempts), the attacking ECU will be confined into bus-off state.

So far, we assumed that the retransmissions will not be disturbed by any other CAN messages on the bus. Since any ECU can transmit after an idle period of 11 recessive bits (which is the minimum between retransmissions as well), a CAN message with lower CAN ID would win arbitration and thus interrupt the repeated transmissions. This will come at the expense of increasing the *bus-off time*, an important metric we will evaluate in Sec. 4.5.5. However, since `MichiCAN` compares the CAN ID of a frame even for a retransmission, it will still work as expected. Even in the presence of multiple attackers, `MichiCAN` is capable of busing off all attackers according to CAN specifications.

4.5 Evaluation

4.5.1 Experimental Setup

To evaluate `MichiCAN`, we can choose from different evaluation boards that come with integrated CAN controllers and either meet/resemble the specifications of automotive ECUs or are specifically built for it. One prominent and affordable platform is the Arduino Due which features an Atmel SAM3X8E ARM Cortex-M3 CPU [28] clocked at 84MHz, 512kB of Flash memory, 96kB of SRAM and most importantly, an on-chip CAN controller. Since the Arduino only provides CAN_TX and CAN_RX lines and comes without a CAN transceiver, we can use separate CAN PHY breakout boards [43] in conjunction with that to build a CAN bus.

Our experimental setup is depicted in Fig. 4.9 and uses two Arduino Dues, one running `MichiCAN` and the other acting as the attacker ECU. Sec. 4.6 will discuss how `MichiCAN` operates around multiple attackers and how this affects our evaluation metrics. To evaluate certain metrics, we need to measure the execution time of

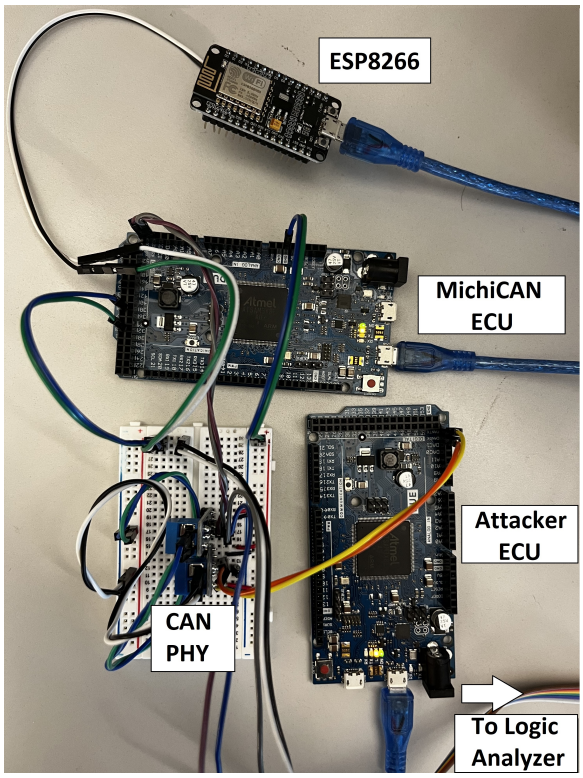


Figure 4.9: Experimental setup with two ECUs

`MichiCAN`. To minimize the overhead on the Arduino Due and report accurate numbers, we use an external timer. For this purpose, we chose the ESP8266 [32] which can sample pins at up to 160MHz. Furthermore, we have also connected a logic analyzer to the breadboard so that we can monitor the CAN traffic and obtain other time measurements for other evaluation metrics.

We evaluate `MichiCAN` using CAN messages from real production vehicles of the same OEM manufactured between 2016 and 2019:[1] Vehicle A is a luxury mid-size sedan, Vehicle B a compact crossover SUV, Vehicle C a full-size crossover SUV and Vehicle D a fullsize pickup truck. All vehicles have two CAN buses and each bus is analyzed separately. Vehicle A has 208 and 127 CAN IDs, respectively. These numbers stand at 161 and 153 for Vehicle B, 123 and 103 for Vehicle C, and 122 for both buses on Vehicle D.

[1]Due to NDA, we cannot disclose the name of the automotive OEM.

The next three subsections evaluate the detection rate, complexity and latency using a computer running 64-bit Ubuntu 18.04.4 LTS with 128GB of registered ECC DDR4 RAM and two Intel Xeon E5-2683 V4 CPUs (2.1 GHz with 16 cores/32 threads each). The remaining subsections elaborate on *online metrics* such as bus-off time, CPU utilization, bus load and memory which are all evaluated on the CAN bus prototype depicted in Fig. 4.9. In our setup, each CAN controller can be configured to transmit messages at speeds up to 1 Mbit/s, although all ECUs on a CAN bus need to share the same *bus speed*. This is fixed by the OEM at production time and cannot be altered afterwards. The most common bus speeds are 125, 250 and 500kbit/s. Our online evaluation will be based on 50 and 125kbit/s bus speeds. Although `MichiCAN` can run at 250kbit/s, we observed several issues with both detection and prevention routines. This can be explained by the interrupt handler which has less time to execute before the next bit arrives. Although we acknowledge this drawback of `MichiCAN`, it can be explained by the overhead in the detection routine. The two aforementioned bus speeds guarantee reliable functionality of `MichiCAN`. Nevertheless, we show in Sec. 4.6.2 that `MichiCAN` can indeed run at higher bus speeds on different hardware.

Furthermore, we make direct comparisons of two crucial metrics, namely bus-off time and bus load, of `MichiCAN` with the closest related work Parrot [67].

4.5.2 Detection Rate

Sec. 4.4.1 illustrates the generation of finite state machines (FSMs) for spoofing and DoS detection with an example CAN bus consisting of 4 CAN IDs, i.e., $|\mathbb{E}| = 4$. Real vehicles contain 100–200 CAN IDs per CAN bus as depicted in Sec. 4.5.1. However, the maximum number of CAN IDs is $2^{11} = 2,048$. With a growing number of CAN IDs $|\mathbb{E}|$, the FSM is expected to contain more if-statements (see Listing C.1). As a result, the execution time of the interrupt handler and thus the CPU usage will

increase. In what follows, we will analyze the number of if-statements in the FSM for varying $|\mathbb{E}|$.

Let $\mathbb{E}^{|\mathbb{E}|}$ denote an IVN consisting of $|\mathbb{E}|$ CAN IDs. To empirically verify that `MichiCAN` works for all combinations, we can exhaustively generate FSMs using the algorithms from Sec. 4.4.1 for each ECU_i in $\mathbb{E}^j$ with $i \leq j$ and $j \in \{1, 2047\}$. This would lead to $\sum_{k=1}^{2047} \binom{2047}{k} \approx 1.6 \times 10^{616}$ different FSMs. Generating and testing each FSM on all possible CAN IDs (from 0x000 to 0x7FF) takes roughly 1.5 seconds. Given the extremely large number of combinations, it is impossible to run exhaustive testing on any platform. To overcome this difficulty, we randomly sampled 160,000 combinations. The tests passed on all random combinations, yielding a 100% detection rate.

4.5.3 Detection Complexity

Fig. 4.10 depicts the number of if-statements in each of the 160,000 generated FSMs. Each dot in the figure represents one combination. Since an incoming CAN ID does not have to run through all if-statements, the figure shows the worst-case scenario. Even with 160,000 random combinations, a specific pattern emerges. As expected, the number of if-statements increases with a larger $|\mathbb{E}|$ since more and more local prefixes are added. After roughly 500 CAN IDs, the number of if-statements starts to decrease because at that point, there are enough outliers that do not require new local prefixes, but can eventually start sharing them. Furthermore, as $|\mathbb{E}|$ continues to grow, there are fewer malicious CAN IDs and thus less outliers and local prefixes. It is worth noting that the testing time for varying $|\mathbb{E}|$ follows a similar, but more dampened pattern, as Fig. 4.10. The mean testing time stands at 0.5–0.6 second on our desktop setup, while the 99-th percentile stands at 0.7 second. Fig. C.1 in Appendix C.1 depicts this relationship.

In addition to testing over 160,000 random combinations, we also used the eight

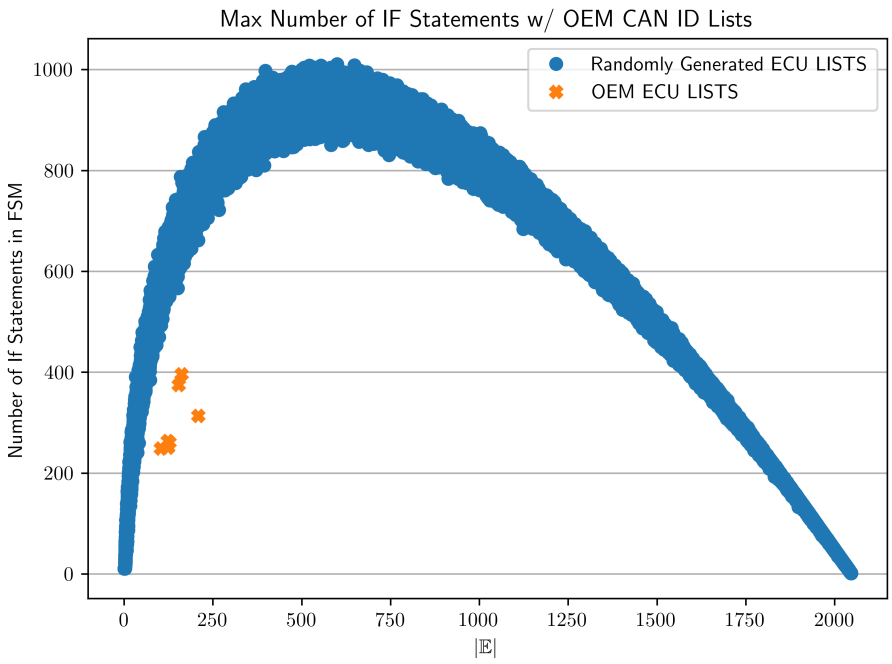


Figure 4.10: Maximum number of if-statements in each FSM

CAN buses $|\mathbb{E}|$ used on four real vehicles from Sec. 4.5.1. We found that the FSM will require less if-statements than our randomly-generated networks (see orange dots in Fig. 4.10), since the CAN IDs are assigned with more structure. The main reason behind that is that many CAN IDs are close in range, with big gaps between clusters.

The algorithms to generate these FSMs need to be run only once offline during the vehicle's production phase. This means that, while their complexity is $\mathcal{O}(n^2)$, they can run offline in as much time as needed. However, for the IVNs of a real production vehicle, roughly 400 FSMs need to be generated which take less than 10 minutes on a similar setup like ours. The FSMs generated from this offline algorithm are then a finite number of if-statements which never exceed 1000.

4.5.4 Detection Latency

Another important metric for our detection routine is the detection latency, i.e., at what bit position within the CAN ID `MichiCAN` will know to stop executing its

FSM and set the counterattack flag to true. We will henceforth refer to this as the *detection bit position*. Although the earliest the counterattack can start is *after* the arbitration field (as discussed in Sec. 4.4.5), stopping the FSM early can be beneficial to avoid using additional CPU cycles. CPU utilization is an important online metric and will be evaluated in Sec. 4.5.6.

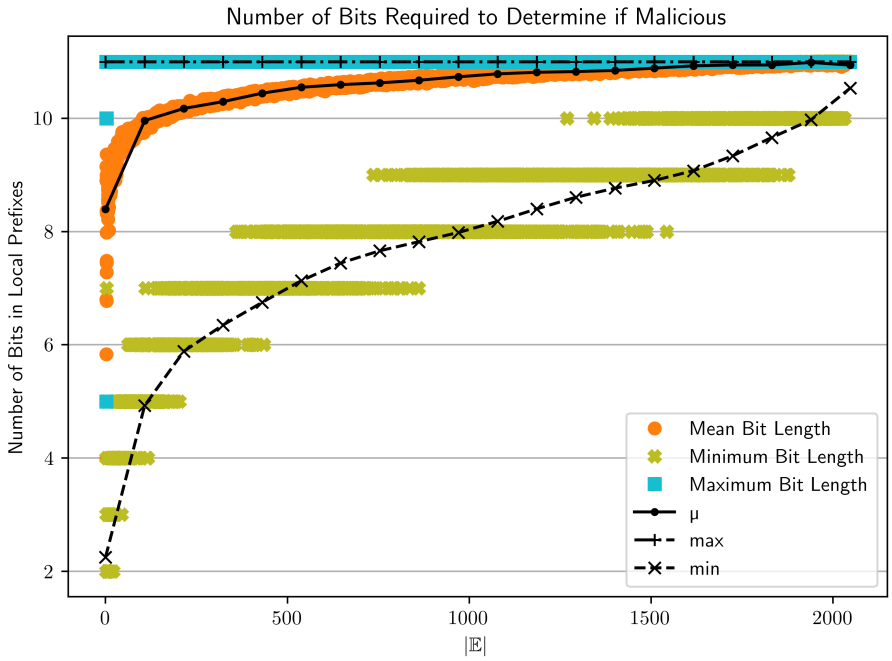
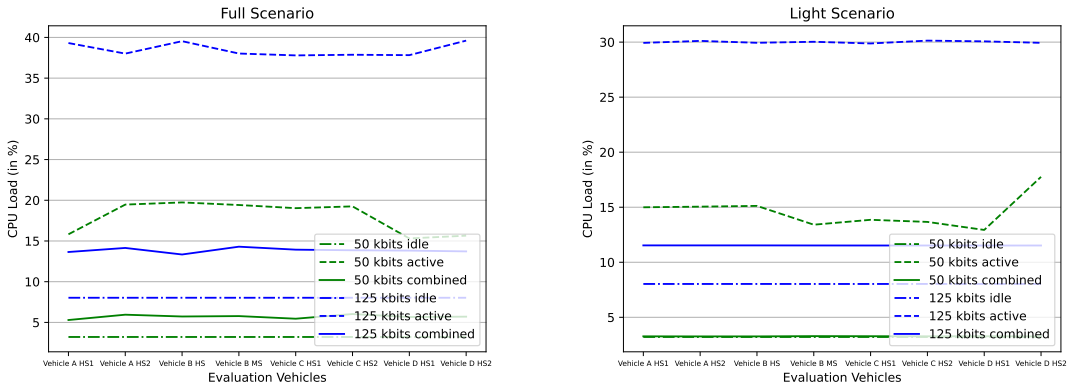


Figure 4.11: Bit position at which CAN ID is malicious

For our 160,000 random combinations, Fig. 4.11 depicts the best-case, worst-case, and average-case scenarios of the detection bit position. As the size of IVN $\mathbb{E}$ grows, the detection bit position rises for the best-case and average-case scenarios as expected. It is interesting to note that for the average case, the detection bit position already reaches 10 bits at around 200 ECUs/CAN IDs which is a common size for real production vehicles. The CAN communication matrices of vehicles are optimized so that the number of if-statements is less, but the detection bit position has a relatively insignificant change for $|\mathbb{E}| = 200$ (standing at 9 bits on average). Vehicle manufacturers could select CAN IDs in an engineered manner in order to



(a) CPU full scenario (b) CPU light scenario

Figure 4.12: CPU usage for full and light scenarios

better optimize this detection bit position. Finally, the detection latency is calculated as the detection bit position multiplied by the nominal bit time.

4.5.5 Bus-off Time

`MichiCAN` starts to bus off the attacker by generating error frames as soon as possible, i.e., right after the CAN ID field. In total, 31 retransmissions are required after the initial transmission of a malicious CAN ID. Note that no complete CAN frames are sent since the attacker will retransmit its CAN message after the 14-bit error frame and 3-bit inter-frame space (IFS) in its error-active region and an additional 8-bit suspend period in its error-passive region. The total time from the first bit of a malicious CAN message to the last bit of the passive error frame in the 31st retransmission is called *bus-off time*. It depends on the attacker's CAN ID since in the best case, `MichiCAN` has to inject one dominant bit, whereas in the worst case, it has to inject 6 dominant bits to trigger an error frame (see Sec. 4.4.5).

Best-Case Scenario `MichiCAN` injects the dominant bit during the RTR bit. As a result, the error frame starts at the 14th bit position within the CAN frame (1 SOF + 11 CAN ID + 1 RTR). The error flag itself consists of 14 bits, in addition to the 3

bit IFS, so the (re-)transmission of an error-active attacker takes 30 bits. Note that this calculation excludes stuff bits which depends on the most-significant bits of the CAN ID. For the error-passive attacker, this number stands at 38 bits including the additional suspend period.

Worst-Case Scenario `MichiCAN` injects six dominant bits (depicted in Fig. 4.8). The error frame starts at the 19th bit within the CAN frame. The bus-off time stands at 35 bits and 43 bits for the error-active and error-passive attacker, respectively.

Table 4.3 shows the error-active and error-passive transmission times, as well as the entire bus-off time for the best-case (BC) and worst-case (WC) scenarios, respectively. As with the detection latency, the number of bits has to be multiplied by the nominal bit time which is the inverse of the bus speed. Our evaluation results show a WC bus-off time of 10ms for a 125kbit/s bus. Using a higher-speed bus running at 500kbit/s, the WC time will decrease to 2.5ms. After analyzing the communication matrices of production vehicles (see Sec. 4.5.1), we found that minimum deadlines of periodic CAN messages stand at 10ms. The added overhead of less than 2.5ms is thus feasible and will not affect bus communications.

Table 4.3: Bus-off time for one attacker

Bus Speed (in kbit/s)	Scenario	Error-Active Time (in μs)	Error-Passive Time (in μs)	Bus-off Time (in ms)
50	B.C.	600	760	21.8
	W.C.	700	860	25.0
125	B.C.	240	304	8.7
	W.C.	280	344	10.0

In case of a busy CAN bus, the CAN ID of the attacker's retransmissions can get interrupted by higher-priority CAN messages. If the message is not malicious, the bus-off time will be extended only by one CAN frame (125 bits including stuff bits). However, if the higher-priority message is malicious, it will prolong the bus-

off time by an entire bus-off attempt (see Table 4.3). Furthermore, bused-off ECUs will wake up after observing 128 instances of 11 recessive bits and might have an effect on future message transmissions (if they are high-priority). To avoid this, we suggest implementing a persistent bus-off as described by Serag *et al.* [156]. However, this feature is not in the scope of `MichiCAN` and hence not implemented, although `MichiCAN` is fully transparent to this addition.

As mentioned in the Introduction, Parrot [67] will launch a counterattack to bus off the attacking ECU in case of a spoofing attack. Parrot uses MCUs with external CAN controllers and can thus only inject entire CAN frames to the bus. In contrast, `MichiCAN` uses integrated controller which can read and write each individual bit on the CAN bus, so it knows exactly when to drive the CAN_TX low to generate an error frame in the attacker. Parrot's defensive message contains the same CAN ID and DLC as the attacker's message, but consists of a payload of only dominant bits which would overwrite any recessive bit in the attacking ECU's data to cause a bit error. Since Parrot can only send complete CAN frames, it cannot start the counterattack until the second occurrence of the malicious CAN message. During the first transmission, Parrot detects by observing its own CAN ID that an attacker is present. Since it needs to inject a complete CAN frame, it cannot launch a counterattack on that first instance. This is particularly problematic as even one spoofed CAN message can have safety-critical impact on the victim ECU, i.e., disabling the brakes [86]. Since the bus-off time is defined from the first bit of a malicious CAN message, there will be a delay of one entire CAN message (i.e., 125 bits). Furthermore, due to Parrot's limitation of only being able to send complete CAN frames, each defensive message needs to be injected at the exact same time as the attacker. To force a collision, Parrot needs to send defensive messages at a very high speed to overlap with the attacking message. Since this is not deterministic and also depends on the separation of the attacker's message, we will assume that both the malicious and defensive messages

start immediately after an IFS of 3 bits. This is considered the best-case scenario for the bus-off time.

The earliest first collision will occur after 19 bits, i.e., during the first bit of the data if the attacker's message contains a recessive bit. This *bit error* will lead to the attacker generating an active error flag which, in turn, causes a *stuff error* in the victim ECU after 6 bits. Both ECUs will increase their transmission error counters (TECs) by 8 and re-transmit after the error flag and IFS. In total, one (re-transmission) attempt in the error-active region accumulates to 41 bits. After 16 occurrences, both attacker and victim ECU will enter the error-passive region. The attacker will detect another collision after 19 bits, but raise a passive error flag this time which consists of recessive bits and does not cause a stuff error in the victim. As a result, the victim can finish their re-transmission and start decreasing their TEC. However, the attacker will continue increasing their TEC by 8 for each collision. After 16 instances of successful re-transmissions by the victim (each attempt consisting of 125 CAN frame + 3 IFS + 8 suspend transmission bits), the attacker will finally be confined into the bus-off state. In total, it takes a minimum of 2,960 bits to bus off the attacker. This equals the best-case bus-off time of 23.7ms for a 125kbit/s bus which is more than twice as long as the worst-case bus-off time using `MichiCAN`.

Besides the long bus-off time, another shortcoming of Parrot is its inability to cancel messages that have a data field with all zeros. To cause a bit error in the spoofed message, Parrot injects a CAN message with a data payload consisting of zeros. If the payload of the attacker's message contains only zeros, Parrot will not be able to create any CAN errors since its external CAN controller will inject recessive stuff bits. Since `MichiCAN` is only sending a pulse of 6 dominant bits instead of (any part of) a CAN frame and does not use the CAN controller, it will not face this problem. Last but not least, by generating error frames in the attacker, Parrot will not only raise the attacker's TEC, but also its own. The reason behind this is that

Parrot itself sends a complete CAN frame with the same CAN ID, but different data. Although the authors propose a way to reduce the defending ECU's TEC again, any non-malicious bus error during that time might bus off the legitimate ECU as well.

4.5.6 CPU Utilization

Since `MichiCAN` incurs computational overhead to the MCU, it is important to measure its impact on CPU utilization. We envision `MichiCAN` as a software patch to existing application software running on MCUs, and hence a minimal overhead is desired. Since there is no advanced operating system on our evaluation board, the Arduino Due, the CPU usage cannot be directly acquired from a system monitor application. `MichiCAN` uses interrupts that are triggered every nominal bit-time. We can measure the overhead induced by `MichiCAN` by measuring the execution time of the interrupt handler and dividing it by the nominal bit-time. Although there is an external interrupt for re-synchronization at the start of the frame, the CPU cycles consumed by this step are negligible compared to the main interrupt.

The execution time of `MichiCAN`'s main interrupt handler is recorded via an external timer, namely the ESP8266, as mentioned in Sec. 4.5.1. With a clock frequency of 160MHz, ESP8266 has a time resolution of 6.25ns which is sufficiently fine-grained for our purposes. At the very beginning of the interrupt handler, we toggle digital pins on the Arduino which are captured by the ESP's external interrupts. The ESP then proceeds to start and increment a counter for each clock cycle. At the end of `MichiCAN`'s interrupt handler, we toggle the pins again and the ESP stops its counters, yielding the total number of clock cycles in between these two external interrupts. The clock cycles multiplied by the aforementioned resolution gives us the execution time of `MichiCAN`.

CPU utilization is evaluated for both *full* and *light* scenarios of the FSM that have been introduced in Sec. 4.4.1. For either scenario, there are two distinct periods

while reading CAN_RX in which the CPU utilization significantly varies. The first period is the bus-idle state, i.e., when no CAN messages are on the bus. The interrupt handler in Alg. 5 will only execute lines 28-29. This results in a very low execution time which is constant during this entire period. The CPU utilization during this period is called *idle load*. The second distinct period is whenever a CAN frame is processed (lines 3-26). Since the FSM is only run during the arbitration field, the execution time during this time will be higher than during the rest of the frame. We analyze the CPU usage while `MichiCAN` processes it and refer to it as *active load*. Finally, the *combined load* covers both periods and thus describes the average CPU utilization overhead on the Arduino.

Fig. 4.12 depicts the mean idle, active and combined CPU loads for both the full and light scenario, respectively. The evaluation was conducted using the eight CAN buses $\mathbb{E}$ of the four production vehicles from Sec. 4.5.1. For each $\mathbb{E}$, we deployed the FSM for ECU_N on the Arduino for maximum testing coverage (and worst-case scenario) and then calculated `MichiCAN`'s CPU utilization overhead by measuring the execution time of the interrupt handler. We make the following observations from Fig. 4.12.

CPU load depends on bus speed Since the external time measurement is divided by the nominal bit-time, the higher bus speed the CAN bus operates on, the higher the CPU utilization will be. By looking at the active load of `MichiCAN` on a 125kbit/s bus, the mean is shown to hover around 40%. The next-fast CAN bus of 250kbit/s would accordingly use 80% of the CPU which does not include jitter. This explains why `MichiCAN` does not always reliably work on higher bus speeds than 125kbit/s.

CPU load depends on FSM complexity As expected, a larger (and thus more complex) FSM requires the MCU to spend more clock cycles on its if-statements. While the active load for 125kbit/s in the full scenario hovers around 40%, the light

scenario that only consists of spoofing detection consumes less computation power at 30% utilization. This amounts to a 25% reduction in CPU cycles for the light scenario.

4.5.7 Bus Load

The bus load (BL) b is calculated as [14]:

$$b = \frac{s_{frame}}{f_{baud}} \sum_{m \in M} \frac{1}{p_m}, \tag{4.3}$$

where f_{baud} is the bus speed, and p_m is the period/cycle time of a CAN message m. Each CAN frame consists of 125 bits on average (including stuff bits), i.e., s_{frame}=125.

Since we do not use real CAN traffic on our experimental setup, there is no point of measuring the absolute bus load. Our sender ECUs can transmit periodic messages at intervals we can configure, so we can arbitrarily adjust the absolute bus load. More interesting is the bus load overhead, i.e., how `MichiCAN`'s prevention routine affects the overall bus load. One CAN message at 125kbit/s is transmitted within 1ms. Table 4.3 shows that if this message is counterattacked by `MichiCAN`, it will be on the bus for 10ms in the worst case including retransmissions and in the absence of higher-priority CAN messages. Theoretically, we increase the bus load by a factor of 10. However, using a persistent bus-off attack [156], the attacker will be bused off once and the remaining CAN communications will continue normally. As a result, there will only be a short spike in the bus load during the counterattack for around 10ms. As discussed in Sec. 4.5.5, the bus-off time during which the bus load will peak is much smaller than typical message deadlines. Despite the non-zero bus load overhead, the scheduling of higher-priority messages will not be affected due to CAN arbitration. Lower-priority messages that have even longer deadlines will experience a negligible blocking delay during the bus-off attempt. For instance, the bus-off time

for a CAN bus operating at 500kbit/s stands at 2.5ms. Low-priority messages have deadlines standing at 50 or 100ms as observed in the eight vehicles from Sec. 4.5.1. As a result, the bus-off attempt will incur a bus load overhead of 2.5–5% for low-priority ECUs. Given that the bus load will never exceed 80% [15] and a real observed bus load of 40% in real vehicles [67], the overhead can be considered negligible.

As mentioned before, Parrot [67] transmits defensive messages at a very high frequency to cause collision with the start of the attacker's message. Our experimental results show that the gap between two defensive messages need to be 3 bits which is also equal to the IFS. As a result, the bus load overhead during the time until a collision is forced stands at $\frac{125}{128} \approx 97.7\%$. Note that `MichiCAN` does not incur this overhead at all. During the bus-off attempt, Parrot's bus load overhead will be at least 2x higher than `MichiCAN`'s, according to the bus-off time reported in Sec.4.5.5.

4.5.8 Memory

Finally, Flash and RAM usage are reported when our code compiles into the Arduinos. Memory usage is evaluated (i) without `MichiCAN` (regular CAN communication), (ii) using a light scenario, and (iii) using a full scenario of `MichiCAN`. Table 4.4 reports all numbers in bytes. As expected, the majority of program storage in Flash comes from the interrupt handler, effectively doubling the amount of required Flash memory. However, static RAM allocation barely changed due to the low number of variables introduced by `MichiCAN`. All in all, only a small fraction of the 512kB of Flash and 96kB of RAM are used.

Table 4.4: Memory Usage of `MichiCAN`

	Without `MichiCAN`	`MichiCAN` Light	`MichiCAN` Full
Flash	12776	25744	28032
RAM	5036	5140	5140

4.6 Discussion

4.6.1 Prevalence of integrated CAN controllers

MCUs with on-chip CAN controllers have been used in production vehicles for several years. For instance, the Renesas V850ES/FJ3 infotainment MCU used in the 2015 Jeep hack [86] had an integrated CAN controller. To the best of our knowledge, `MichiCAN` is the first to use this integrated MCU–CAN controller for attack detection and prevention. However, researchers have already leveraged the idea of bypassing the CAN controller for launching *bus-off attacks* as early as 2017. For instance, Palanca *et al.* [135] use an Arduino Uno with a MCP2551 CAN PHY to read and write bits directly. However, this setup lacks a CAN controller completely (since the Arduino Uno does not come with a CAN controller in contrast to the Arduino Due), so they had to re-implement the majority of the CAN protocol on the Arduino (without any guarantee of correct implementation) whereas `MichiCAN` still uses the built-in CAN controller to correctly receive CAN messages. Another work by Murvay *et al.* [128] builds on the previous work with a more sophisticated attacker model, but still uses an MCU (NXP S12XD512) without an integrated CAN controller.

As described in the previous subsection, an ECU enters bus-off mode when either its TEC or REC reaches 256. Although this has been originally designed as a fault-confinement mechanism, researchers found that this feature can be exploited to disconnect benign ECUs by an attacker [52, 60]. At the moment, it is unclear how to effectively defend against bus-off attacks effectively since the only possible way is to detect this attack and try to bus off the attacker first (if the CAN specification is not modified) [165, 173]. Novel research even shows that the bus-off attack can be made persistent [156], i.e., the ECU will not recover after bus-off. Since bus-off attacks generate error frames, a defense against them could track errors based on their frequency and CAN ID to determine a pattern. Longari *et al.* [116] attempts to find

the source of the bus-off attack, but it (i) cannot determine if the error frames were caused maliciously or by legitimate bus faults, and (ii) has no prevention capabilities since it is an intrusion detection system in its core. However, a straightforward, but expensive approach to mitigate bus-off attacks using integrated CAN controllers can be achieved by secure boot [153]. The goal of secure boot is to only execute authenticated software when the ECU is booted up. For the aforementioned bus-off attack, the adversary needs to change the firmware. Using secure boot, the ECU will discard the changes and boot from the authentic firmware. The secure boot usually requires a *Hardware Security Module* (HSM) which is expensive, but already used by Tier-1 suppliers such as NXP [83] or Renesas [112].

The most sophisticated work that actually mentions integrated CAN controllers and uses them to launch a stealthy bus-off attack was proposed by Kulandaivel *et al.* [110]. Their attack framework, called `CANnon`, can inject single bits and force the victim to generate error frames until it is bused off. Instead of pin-multiplexing which `MichiCAN` is using, they deploy a technique, called *peripheral clock gating*, to arbitrarily pause and resume the clock of the CAN controller. Although they have shown that their method is even harder to detect by existing techniques, they manipulate the functioning of the CAN controller. `MichiCAN` does not modify any part of the CAN controller, but adds redundancy to re-implement certain functions of the CAN protocol/controller in application software which guarantees full backwards-compatibility.

4.6.2 Replicability on other MCUs

Our evaluation is done on the Arduino Due which uses an AT91SAM3X8EA 32-bit MCU. This MCU features an integrated CAN controller and allows pin multiplexing. We wanted to see if we could replicate `MichiCAN` on other MCUs as well. Kulandaivel *et al.* [110] present an overview of MCUs with integrated CAN controllers which

are used in the automotive domain, including the Microchip SAM V71 Xplained Ultra board, which uses an ATSAMV71Q21 32-bit MCU operating at 150MHz and the STMicro SPC58EC Discovery board, which uses an SPC58EC80E5 32-bit MCU operating at 180MHz.

Although we expect `MichiCAN` to work without further modifications on any of these, we implemented part of `MichiCAN` (only spoofing detection) on the S32K144 [42] which is a low-cost evaluation and development board for general-purpose industrial and automotive applications. It uses a 32-bit ARM Cortex-M4F S32K14 MCU operating at 112 MHz and also features integrated CAN controllers. We were able to confirm that `MichiCAN` works as intended on this MCU as well, and even exceeds the Arduino by fully working on a 500kbit/s CAN bus. This can be explained by only implementing spoofing detection on one hand, and a more optimized interrupt handling on the other hand.

4.6.3 Limitations and Future Work

The main limitation of `MichiCAN` is the added complexity to read and write each bit directly. The FSMs for DoS detection also consume a considerable amount of CPU resource as evaluated in Sec. 4.5.6. As mentioned before, the interrupt handler's execution time needs to stay below the inverse of the interrupt frequency which is tied to the bus speed. Our experimentation on the Arduino Due achieved correct performance on bus speeds up to 125kbit/s, with even 250kbit/s working for most CAN messages. Due to the high overhead (i.e., additional CPU cycles) to enter and exit the interrupt handler on the Arduino Due compared to other comparable MCUs [119], a more optimized code together with a more powerful MCU will run `MichiCAN` on bus speeds up to 1Mbit/s. In fact, our partial `MichiCAN` implementation on the S32K144 demonstrated that `MichiCAN`'s main limitation can be addressed.

Although we provide an extensive evaluation using multiple metrics in Sec. 4.5, we

used a minimal working example of a bench prototype consisting of Arduinos. One way to stress-test `MichiCAN` is by adding real CAN traffic to the testbed. This can be done by recording CAN traffic from a production vehicle and replaying it to the testbed which includes the `MichiCAN`-equipped MCU (i.e., Arduino Due). This technique is called *rest-bus simulation* [26] (since the remaining parts of $\mathbb{E}$ are simulated) and can be performed by USB-CAN interfaces such as the popular PCAN [39].

4.7 Conclusion

In this thesis chapter, we have developed `MichiCAN` which is a practical spoofing and Denial-of-Service detection and prevention framework for the CAN bus. By using integrated CAN controllers deployed in many modern ECUs, we can solve various issues of prior work, such as the lack of real-time detection and prevention, as well as significant network overhead. `MichiCAN` also guarantees backward compatibility without requiring any additional hardware or modifying the CAN protocol. Our extensive evaluation demonstrated the efficacy of `MichiCAN` using multiple metrics. Since DoS attacks are an overlooked but important threat vector in automotive security standards such as AUTOSAR SecOC [30], we envision `MichiCAN` to be a compelling and practical security enhancement for OEMs that can be easily implemented in their production vehicles.

CHAPTER V

CARdea: Practical Anomaly Detection for Connected and Automated Vehicles

5.1 Introduction

The future of intelligent transportation systems (ITS) will be spearheaded by vehicle-to-everything (V2X) communication. V2X is one of the complementary technologies to enhance and support Advanced Driver-Assistance Systems (ADAS) and autonomous vehicles (AVs). Primarily, the V2X communication range is greater than the sensing ranges of current ADAS and AV sensors, such as radar, LiDAR and cameras. At the same time, V2X allows non-line-of-sight (NLOS) detection and communication which is not possible with ADAS/AV sensors. Among others, V2X allows connected vehicles to talk to other vehicles (V2V), smart infrastructure (V2I) and pedestrians (V2P). V2X can also be used with cars that have a lower level of automation [29], such as traditional cars, and help them avoid traffic congestion and prevent collisions. For these purposes, vehicles exchange *Basic Safety Messages* (BSMs) in the US which are defined in the SAE J2735 standard [149]. BSMs contain state information about a vehicle, such as its location, speed, acceleration, heading and yaw rate. Vehicles listen to BSM broadcasts and can plan their future actions accordingly, e.g., by slowing down or speeding up. This can enhance road safety as long as

there is no malicious interference. Note that CAVs do not exclusively rely on BSMs, but received sensory data is considered *advisory* information, as CAVs use different sources of data to make informed decisions. Unfortunately, with the deployment of

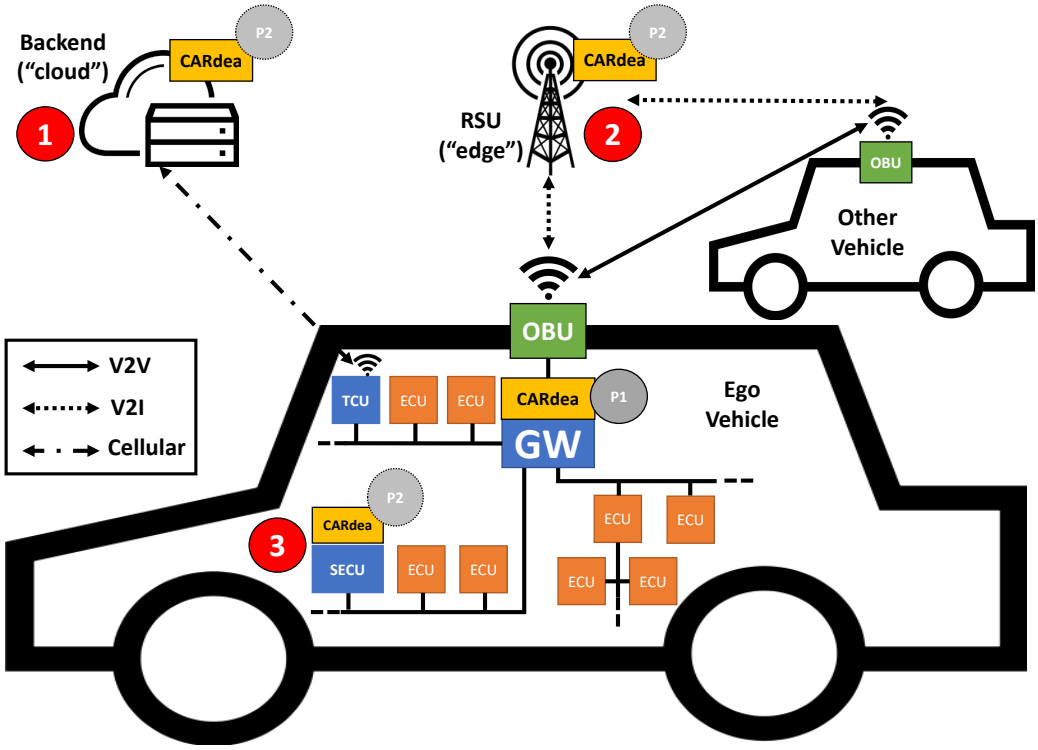


Figure 5.1: `CARdea` deployment options in V2X infrastructure

complex connected computing platforms, the risk of cyber attacks grows. As a result, the field of V2X security has received increasing scrutiny over the last decade [93], with various standardization bodies in the US and Europe working to add security to the respective V2X protocols [150, 157]. The field of V2X security is broad with several challenges in responding to different types of threats as described in Sec. 5.2. Depending on the attackers' capabilities and attack types, a holistic multi-layered security concept is required. For instance, BSMs from *external* attackers (e.g., road-side attackers with V2X radio) will be discarded immediately due to lack of valid credentials to join the BSM broadcast. In contrast, *internal* attackers (e.g., compromised ECUs) are "real" vehicles that are authenticated to exchange BSMs with their surrounding vehicles and other entities. They can launch a variety of attacks, such

as Denial-of-Service (DoS), Sybil, replay, or false data injection [152]. Compared to these three attack types which are all of adversarial nature, false data broadcast can also be caused by faulty sensors in non-malicious vehicles [148]. This can be achieved through a compromised in-vehicle network [58, 124, 127] or a malicious On-Board Unit (OBU) — the vehicle's external interface responsible for V2X communication. Nevertheless, false data injection is considered to be a serious security threat due to its low deployment complexity [113].

Although there are several distinct existing defenses against false data injection and Sybil attacks as depicted in Table 5.1 (see Sec. 5.3 for more details), most of them are *a*) requiring sensor fusion with other data (e.g., Angle of Arrival (AoA), Radar) that might not be available on any car, *b*) having a large detection latency, and/or *c*) using machine learning (ML) models that are computationally heavy for the vehicle's constrained computing resources. Ideally, all attacks need to be detected instantaneously at the vehicle for immediate mitigation of the potentially evolving safety risk, as well as offer high detection rate (TPR) and low false alarm rate (FPR). Specifically, prior work paid little attention to the detection latency due to a vehicle's limited computation power for cost reasons. Since most safety-critical functionalities require simple computations and do not need high-performance hardware (i.e., CPU and memory), these legacy ECUs are very simple and highly optimized for repetitive control operations. Despite the lack of publicly available ECU specifications, existing literature demonstrates that even the highest-end ECUs in current in-vehicle architectures are not as fast as the slowest workstations that prior work tested on. For instance, a 32-bit ARM 1GHz Infotainment ECU is considered high-end [132] and a tear-down of the technologically-advanced Tesla Model S [45] revealed that the most powerful ECUs (e.g., NVIDIA Tegra) can only be comparable to smartphone CPUs from over five years ago.

Table 5.1: Comparison with related work

Work	So19 [162]	Yao17 [190]	Yav17 [192]	Bi12 [51]	Su17 [172]	Va19 [178] [1]	CARdea
BSM Data	p	N/A	p, v, a	p, h, v	p, a	p, v, a	p, v, a
Other Data	RSSI	RSSI	Map	Radar	AoA, DS	None	None
Anomalies	Constant (+Offset) Random (+Offset) Eventual Stop	Sybil	Random Error	Sybil	Constant Offset	Instant Constant Gradual Drift Bias	Constant (+Offset) Random (+Offset) Sybil Replay
Approach	Physical Layer Consistency Check	Dynamic Time Warping (DTW)	Vehicle Physics (VP)	Particle Filter (PF)	Extended Kalman Filter (EKF)	CNN+KF	VP+(RF,SVM,DNN)
Performance (%)	TPR: 83.7	TPR: > 90 FPR: < 10	TPR: 90 FPR: 2.7	N/A	TPR: 100 FPR: 5	TPR: 51.5-99.2	Phase 1 TPR: 98, FPR: 0.02 Phase 2 TPR: 99, FPR: 0.16
Detection Time (ms)	N/A	630	N/A	5-25	3.2	KF: 5.1	Phase 1: 0.09-0.11 Phase 2: 5.1-15.5

CARdea attempts to bridge this disconnect between feasible in-vehicle resource usage and good performance of anomaly detection geared towards attacks and anomalies in BSMs. We assume that any ML-based anomaly detection would increase the detection latency significantly if implemented on existing in-vehicle ECUs. Despite this disadvantage, existing work (see Table 5.1) demonstrates that detection performance is generally favorable. On the other hand, light-weight statistical approaches can be deployed inside an existing in-vehicle ECU (i.e., on-board) without incurring too much overhead on latency and computational resources. To achieve the best of both worlds, CARdea uses a *hybrid* of statistical and ML-based schemes which are incorporated in two *sequential* phases.

Phase 1 runs a data-centric plausibility check inside the vehicle, leveraging the correlation of three sensors included in BSMs (GPS, speed, acceleration) according to a vehicle's physical dynamics model. Its goal is to detect an anomaly in a BSM (i.e., sensor fault or adversarial attack). This phase acts as the first layer of defense and needs to detect broadcasts from anomalous vehicles as fast as possible. To minimize both false negatives and false alarms, Phase 1 will only come to a decision if the current data can be marked as anomalous or non-anomalous with a high likelihood. If Phase 1 is uncertain about the condition of BSMs, it will delegate these to Phase 2, which will determine if that piece of data was anomalous or not.

While Phase 1 runs locally inside the vehicle (e.g., on a dedicated ECU), Phase 2 will be executed on a node with more computational resources, such as the edge, OEM's backend or a "super-ECU" in future in-vehicle zonal architectures [108] that are equipped with high computational resources and network bandwidth. Although the latter may initially seem counter-intuitive, some vehicles with ADAS or self-driving capabilities might already have the necessary hardware, whereas many others are unlikely to have these capabilities in the years to come. We envision CARdea to work on any V2V-equipped car, so that the entity that will run Phase 2 is a de-

sign choice for OEMs. For instance, lower-end vehicles will likely run Phase 2 in the cloud, whereas more advanced AVs can run it locally. The latter will save network bandwidth, but will require more expensive hardware. Furthermore, even if the computational resources are available in AVs, they are tailored towards the specific computer vision applications. Any additional software module that runs along the aforementioned ADAS apps will require additional processing resources (especially memory) since OEMs tailor even their advanced hardware to use the minimal amount of resources. The primary driver behind this trend is cost. An overview of Phase 2 deployment options is depicted in Fig. 5.1.

This thesis chapter makes the following main contributions:

- Development of a novel and hybrid two-phase approach, `CARdea`, that combines statistical- and ML-based anomaly detection to protect against anomalous or malicious BSMs;
- Use of only sensor information in BSMs to build a robust anomaly detection scheme without relying on other (wireless) measurements;
- Low detection latency and memory usage on in-vehicle components which enhances the practical/low-cost deployability;
- Support for the detection of various types of anomaly;
- Extensive evaluation with 108 hours of simulated data in Veins [164].

5.2 Background and Threat Model

5.2.1 Primer on V2X

Vehicular sensor data is generated by in-vehicle Electronic Control Units (ECUs) which are distributed embedded systems usually specialized for simple repetitive au-

[1]Re-implemented on same evaluation setup as `CARdea` for comparison.

tomotive control tasks. ECUs are interconnected via an in-vehicle network (IVN) such as the CAN bus.

The central gateway (see Fig. 5.1) is connected to the On-Board Unit (OBU) which is the radio for V2X communication. A V2X frame can contain the payloads of several application layer protocols which vary with geographic region [64]. In the US, SAE standardized *Basic Safety Message* (BSM) [149] as the application protocol for road safety, whereas in Europe the pendant is called *Cooperative Awareness Message* (CAM) by ETSI [80]. The data format of both protocols is very similar for our purposes, i.e., both BSM and CAM contain information about the vehicle's position, speed and acceleration. Despite this chapter's focus on BSM, `CARdea` can also be used with CAM. Note that both are periodically broadcasting messages at 10Hz.

5.2.2 Threat Model

Security threats against V2X are diverse and have been briefly introduced in Sec. 5.1. In order to establish a threat model, we need to further classify threats. Based on the adversary's incentives to attack V2X systems [169] (e.g., physical damage, financial, etc.), it is possible to derive the following taxonomy of attack variants [93]: An *active* attacker presumes that it can interact with the system, whereas a *passive* attacker comprises eavesdropping on wireless data. An *internal* attacker has system-level access and acts according to the underlying protocol, whereas an *external* attacker does not have valid credentials for system access. Most attacks in V2X systems assume an active, internal attacker and comprise Denial-of-Service (DoS), Sybil, replay and false data injection [152]. DoS attacks attempt to exhaust the available resources in the system to shut down potentially safety-critical communication. In Sybil attacks, a malicious vehicle pretends to have multiple identities and thus introduces *ghost vehicles* on the road. Attackers can resend old, stale BSMs in replay attacks. In false data-injection attacks, a rogue vehicle generates false data,

e.g., spoofed sensor information in BSMs, and broadcasts the data to surrounding vehicles.

`CARdea` will focus on the detection of Sybil, replay and false data-injection attacks. DoS attacks are not part of our threat model since `CARdea`'s design cannot detect them. To address them, we refer to best practices [146, 176] that should complement `CARdea`. False data can be caused by faulty sensors, or maliciously spoofed by an attacker using *a*) a compromised OBU, or *b*) a compromised IVN. For instance, faulty sensors can broadcast the sensor value with a constant offset or not update at all. [124] showed that the CAN bus is susceptible to both physical [58] and remote [127] tampering. As a result, anomalies caused by faulty sensors or deliberate attacks must be detected before relevant sensor data is packaged into BSMs and broadcast, or when the receiving vehicle passes the BSM data on towards its actuator ECUs for control tasks (e.g., braking). The former can be achieved with local (CAN) intrusion detection systems that have been extensively covered in literature [188] in case the IVN has been compromised. If the data generated by ECUs on the IVN is correct, but the OBU tampers with the data, these solutions will not work. Hence, it is most effective to detect false data-injection attacks *at the time of their entry* into the receiving vehicle. The vehicle's OBU will extract the data and pass it on to the central gateway which eventually distributes it to the relevant ECUs. Since we would like to focus on adversarial attacks in contrast to faulty or noisy sensors, we assume that a vehicle has been compromised and is broadcasting contiguous segments of spoofed sensor data to our ego vehicle. We also assume the attacker may control all values in a BSM. On the other hand, noisy sensors might cause discrepancies in just a few BSM frames, but usually not during the entire broadcast. As a result, we propose `CARdea` to detect the aforementioned attacks and prevent them from causing havoc in the vehicle's actuators, to run as a module *within* the central gateway. If anomalies within the received BSM are detected, the extracted sensor data can be discarded,

as well as the entire communication with the potentially malicious vehicle suspended if anomalies are detected continuously over a certain period of time. In summary, `CARdea` can be regarded as data-centric, plausibility-based Vehicle-to-Vehicle (V2V) anomaly detection.

5.3 Related Work

5.3.1 Statistical Approaches

Several approaches have been proposed to leverage inherent vehicle physics, road dynamics and other statistical/heuristic techniques. Yavvari *et al.* [192] propose anomaly detection using both road geometry and vehicle dynamics (VD) for V2V communication. Yao *et al.* [191] propose a Sybil attack detection framework based on RSSI that is fully distributed and does not rely on any centralized infrastructure. Müter *et al.* [129] and Schäfer *et al.* [154] propose an entropy-based anomaly detection method for in-vehicle networks and a motion verification system using Doppler shift measurements, respectively. Bissmeyer *et al.* [51] propose a framework for checking the plausibility of vehicle location data utilizing particle filters (PFs). Leveraging multi-modal sensor fusion, Sun *et al.* [172] use side-channel measurements, such as Angle-of-Arrival (AoA) and Doppler Shift (DS), as an input to an Extended Kalman Filter (EKF) for detection of false data. Kim *et al.* [106] propose a BSM plausibility check based on leveraging low-power beaconing messages in a communication-based checking scheme to verify the contents of received messages. One major problem with the previous data-centric approaches is that plausibility checks conducted at the application layer sometimes fail to detect certain GPS spoofing attacks that mimic the movements of a real vehicle. To overcome this limitation, So *et al.* [162] proposed a physical-layer-based plausibility check using the received signal strength indicator (RSSI) which can complement data-centric approaches.

5.3.2 ML-Based Approaches

Van *et al.* [178] proposed an anomaly detection and identification approach combining Convolutional Neural Networks (CNN) and a Kalman Filter (KF). The KF with a χ^2-detector for further anomaly detection, and fusion of non-anomalous readings. Similarly, Wang *et al.* [182] proposed an EKF that feeds a One Class Support Vector Machine (OCSVM) the measured discrepancy of predicted and actual sensor measurements (coined the *innovation*) based on vehicle trajectory and onboard sensors. [181] used an unsupervised deep autoencoder (DAE) with vehicle locations and RSSI to detect anomalies in self-reported vehicle locations. Other studies, such as Fenzl *et al.* [82], explored deep learning use-cases for in-vehicle anomaly detection using continuous fields classification to compute the alignment of CAN bus payloads to detect intrusions.

5.3.3 Differences of `CARdea` from Previous Work

Despite the promising results of the prior work mentioned above, few of them dealt with the feasibility and practicality of actual deployment in the vehicles. As shown in color-coded Table 5.1, they suffer from multiple drawbacks which can be summarized as follows.

Data Required: Prior work requires side-channel data, e.g., wireless measurements (RSSI, AoA) and/or radars to perform sensor fusion. This inherently increases cost for car-makers. For instance, AoA requires multiple antennas [22], and thus more expensive OBUs. Not all vehicles are currently equipped with radars, although this will change with the rise of AVs in the future. First-generation connected vehicles would still require a mandatory radar for some of the existing solutions, driving up cost. Nevertheless, some of the more sophisticated attacks introduced recently [47, 159] can only be detected with external measurement data, such as AoA.

Approaches Employed: As summarized in Sec. 5.3, the two main, common approaches taken for plausibility verification in V2X systems are either statistical or ML-based. Although there is no correlation between performance and the choice of approach, ML-based approaches require more computational resources (such as CPU, memory and network bandwidth) and incur longer detection latencies. Since in-vehicle ECUs are computationally "light" for cost reasons (as mentioned in Sec. 5.1), testing an ML model will take longer than statistical approaches, as shown in Table 5.1.

Detection Latency: Besides the advantage of statistical approaches in detection latency, prior work employed workstations for experimental evaluation. Although detection latency is crucial to assess their feasibility in a real-world scenario (e.g., inside the car), it has not been properly evaluated. In fact, the assessment of this metric is either completely ignored [72, 181, 192] or coarsely estimated [51]. Since all latency measurements in Table 5.1 have been evaluated on different platforms, a direct comparison of latency with `CARdea` is not possible. Thus, we selected the most comparable approach [178] and implemented its source code (partially) on the same evaluation setup as `CARdea`. We could only obtain the source code for the Kalman Filter (KF)-based anomaly detector which is used in combination with a Convolutional Neural Network (CNN). Our latency benchmark on the Raspberry Pi (compared to Phase 1 of `CARdea`) demonstrates a detection time of 5.1ms per analyzed data point which is still at least 46x greater than `CARdea`. Note that the CNN detector (which we could not evaluate due to the unavailability of source code) will constitute the dominant part of the overall latency compared to the KF alone and greatly increase their detection time.

Anomaly Detectabilities: The type of attacks that can be detected by the proposed solution is crucial to assess the overall capability of anomaly detection. Most of

the surveyed solutions only provide limited evaluation of certain types of anomalies. Despite the lack of comprehensive anomaly datasets for V2X, generating multiple distinct anomaly categories and evaluating each proposed scheme is necessary to avoid a specially tailored and limited evaluation.

5.4 System Design

5.4.1 Overview

Considering the limitations of related work discussed in Sec. 5.3, we propose `CARdea` to provide a real-time, performant, and light-weight detection of BSM anomalies. It uses a novel, hybrid of statistical (Phase 1, Sec. 5.5) and ML-based (Phase 2, Sec. 5.6) anomaly detection. Phase 1 runs locally *inside* the vehicle to provide very low detection latency (i.e., in real time) that only statistical approaches can provide. Another benefit of `CARdea` is the low memory consumption which is important, given most automotive ECUs feature only a few KBs to MBs of RAM for cost reasons [16, 41]. In contrast, Phase 2 can run *outside* the vehicle, e.g., on the car-maker's backend or the roadside edge or even cloud, and can rely on more computing resources. Splitting the anomaly detection into these two phases can be reasoned about as follows. Phase 1 performs real-time detection of both *highly-likely anomalous frames* and *highly-likely non-anomalous frames*. We define a *frame* f as a sequence of contiguous sensor readings/samples from a transmitter vehicle. A frame is classified as *highly-likely anomalous* if the mean of the predicted conditions for each (anomalous or non-anomalous) sample in the frame is above a certain per-frame threshold $\theta_{f,a}$. Similarly, a frame is said to be *highly-likely non-anomalous* if the mean of the predicted conditions for each sample in the frame is smaller than $\theta_{f,na}$. Frame predictions between these two thresholds will be sent to Phase 2 for fine-grained classification. This is depicted in Fig. 5.2. Phase 1 will temporarily deem such frames *potentially*

anomalous if the frame prediction is greater than 0.5, and *potentially non-anomalous* otherwise, until a classification result from Phase 2 becomes available. Thus, the goal in Phase 1 is to maximize the True Positive Rate (TPR) since this metric is very important for safety, as well as minimize the False Positive Rate (FPR). However, Phase 1 is optimized for frames that consist of mainly anomalous samples (i.e., from an attacker vehicle) or mainly non-anomalous samples (i.e., from a benign vehicle). Frames that consist of a mixture of anomalous and non-anomalous samples (i.e., a 50/50 split) are sent to Phase 2 for *per-sample detection*, as this does not occur often to reduce network bandwidth. Our ego vehicle can temporarily suspend its communication with vehicles that are marked anomalous as a result of anomalies found in several consecutive BSMs.

The rationale behind introducing Phase 2 is not to detect anomalies in misclassified frames from Phase 1, but rather, to leverage a computationally-powerful, tested ML approach for per-sample detection on frames that Phase 1 cannot make a highly-likely decision on. This differs from traditional two-phase systems. For example, in the program analysis domain two-phase approaches involve a first stage focusing on soundness (i.e., maximizing TPR at the expense of FPR) for a second phase to verify these results. However, in V2V communication, a first phase with high FPR favors safety at the expense of V2X usability. `CARdea`'s two-phase design allows for V2X usability by avoiding constant temporary suspension of communication with other vehicles, while still keeping safety a priority. Thus, with Phase 2, our ego vehicle can reinstate its communication with the vehicles that had been previously flagged as potentially anomalous or non-anomalous and leverage the advantages of BSMs. These improvements overcome limitations (2) and (3).

Furthermore, both phases leverage the correlation between these three sensors – while Phase 1 describes the correlation through vehicle dynamics (VD) equations and Phase 2 trains ML models to capture the correlation through extracted sensor

information. CARdea does not use any other measurement data than the existing sensor samples in BSMs or RSSI. This eliminates the challenges of sensor fusion, as well as renders CARdea viable for the first-generation connected vehicles without requiring installation of any additional sensors. This overcomes limitation (1). In the next subsection, the detectable anomalies in CARdea are introduced, which also mitigate limitation (4).

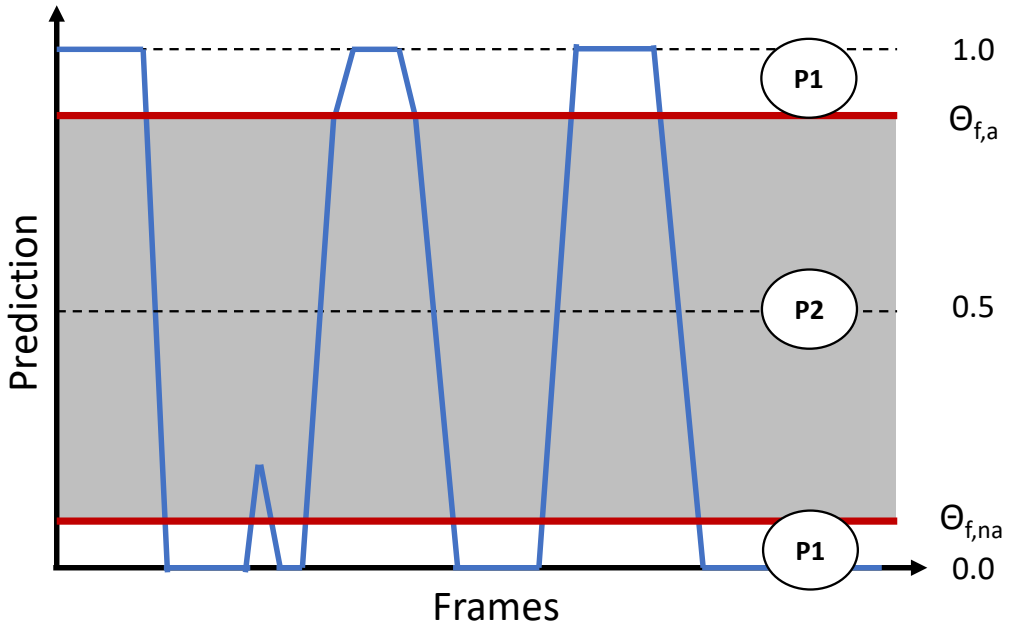


Figure 5.2: CARdea interactions of Phases 1 and 2

5.4.2 Anomalies under Consideration

It is challenging to evaluate anomaly detection schemes for CAVs due to the lack of anomaly datasets for BSMs. Thus, we evaluate CARdea using the Vehicular in Network Simulation (Veins) which is based on the OMNET++ Discrete Event Simulator and Simulation of Urban Mobility (SUMO) [164]. By using a simulation instead of generating anomalies on top of a static dataset, we ensure that false data may propagate to the control system over time and affect the behavior of the ego vehicle and neighboring vehicles. A simulation can thus account for vehicle dynamics.

We execute and evaluate `CARdea` on 7 distinct attack types. The first 4 attack types (AT1–AT4) are falsifications of latitudinal and longitudinal position, speed and longitudinal acceleration sensors, extended from [100, 101] and consistent with literature [50, 158]. In addition to these attacks, which may occur as a result of either faulty sensors or malicious attacks, we also evaluate `CARdea` on real-world, stealthier attacks. Specifically, we conduct Sybil (AT5), Data Replay (AT6), and Stealthy attacks (AT7). Sybil and Data Replay were also extended from [100], but our Stealthy attack implementations were motivated by stealthy physical attacks against robotic vehicles [69, 70, 144]. When a framework or system for vehicle security is proposed, it is crucial to evaluate its performance against a sophisticated adversary, so as to validate its robustness and resilience in the worst case. Thus, we devised a false data-injection attack with the intent to maximize "damage" while not raising any alerts. In other words, an adversary successfully spoofs sensor data while attempting to remain largely undetected. We expect these attack types to cover a range of attacker behavior consistent with the threat model outlined in Sec. 5.2. Note that the superscript i denotes the sensor in the following attacks.

1. **Constant (AT1):** Each attacker vehicle broadcasts a unique constant measurement x_c at time t:

$$x_t^i = x_c. \tag{5.1}$$

2. **Constant Offset (AT2):** Each attacker vehicle broadcasts a unique constant measurement offset to the vehicle's true measurement Δx_c at time t:

$$x_t^i = x_t^i + \Delta x_c. \tag{5.2}$$

3. **Random (AT3):** Each attacker vehicle broadcasts a uniformly random measurement at time t. The min. and max. values x_{min} and x_{max} are determined

by the size of the simulation space:

$$x_t^i = \mathcal{U}([x_{min}, x_{max}]). \tag{5.3}$$

4. **Random Offset (AT4):** Each attacker vehicle broadcasts a unique, uniformly random measurement offset to the vehicle's true measurement at each time t:

$$x_t^i = x_t^i + \mathcal{U}([-x_c, x_c]). \tag{5.4}$$

5. **Sybil (AT5):** Each attacker vehicle generates a set of ghost vehicles within a defined grid in the simulation map, relative to the true data of the attack vehicle at time t. Each square grid is defined by length S_x. x_l denotes the local relative measurement and x_r denotes a random local measurement:

$$x_t^i = S_x \cdot (x_l^i + x_r^i). \tag{5.5}$$

6. **Data Replay (AT6):** Each attacker vehicle chooses a target BSM and replays its data with a certain delay at time t:

$$x_t^i = x_t^i \in_R [x_0^i, x_{t-1}^i]. \tag{5.6}$$

7. **Stealthy (AT7):** Each attacker vehicle injects the maximum relative amount of false sensor data using the optimal per-sample threshold θ_s^i and parameter b_i such that $b_i > 0$ at time t:

$$x_t^i = x_t^i + (1 - \theta_s^i \cdot b_i). \tag{5.7}$$

Furthermore, any particular scenario we simulate may spawn V_n normal vehicles,

and V_a attacker vehicles with probability α, referred to as the *anomaly rate* (AR), which may transmit malicious messages based on the attacker type. Each of these attacker vehicles may broadcast sequences of anomalous and/or non-anomalous messages of varying length, with probability β, referred to as the *in-vehicle anomaly rate* (IVAR). After conducting various simulations, with specific parameters and scenarios detailed in 5.7.2, we process the simulation output to generate simulated datasets for `CARdea`.

5.5 Phase 1: Local Anomaly Detection

5.5.1 Overview

Phase 1 is concerned with real-time detection of anomalies inside the vehicle by leveraging statistical models of inherent sensor correlations. Among the sensor data included in BSMs [149], we select three fundamental sensors for expressing vehicle dynamics: (GPS) Position $\vec{p}$, velocity $\vec{v}$, and longitudinal acceleration a. BSMs are broadcast by vehicles every 100ms, denoted by dt. As a result, our ego vehicle will receive BSMs from M surrounding vehicles (which vary with traffic and driving behavior of the other cars) every 100ms. The objective of Phase 1 is to perform the plausibility verification of each BSM upon its reception by the On-Board Unit (OBU) and determine if the analyzed BSM is anomalous.

Let t be the absolute timestamp, then the correlated sensor groups can be expressed by the following fundamental physics equations:

$$\vec{p}_{t+dt} = f(p_t, \vec{v}_t) = \vec{p}_t + \int_t^{t+dt} \vec{v}_t dt, \tag{5.8}$$

$$v_t = f(\vec{p}_t, \vec{p}_{t+dt}) = \frac{|\vec{p}_{t+dt} - \vec{p}_t|}{dt}, \tag{5.9}$$

$$a_t = f(v_t, v_{t+dt}) = \frac{v_{t+dt} - v_t}{dt}, \tag{5.10}$$

As shown in Fig. 5.3, Phase 1 is decomposed into threshold calibration and validation modules. The *correlation groups* G1–G3 characterize the correlation between the three sensors as expressed by Eqs. (5.8)–(5.10), respectively. Namely, G1 describes the relationship between position $\vec{p}$ and velocity vector $\vec{v}$; G2 the relationship between speed v and position $\vec{p}$; and G3 the relationship between longitudinal acceleration a and speed v. Note that speed v is defined as the magnitude of two-dimensional velocity $\vec{v} = (v_x, v_y)^T$. Below we will elaborate on the threshold calibration and validation modules of Phase 1.

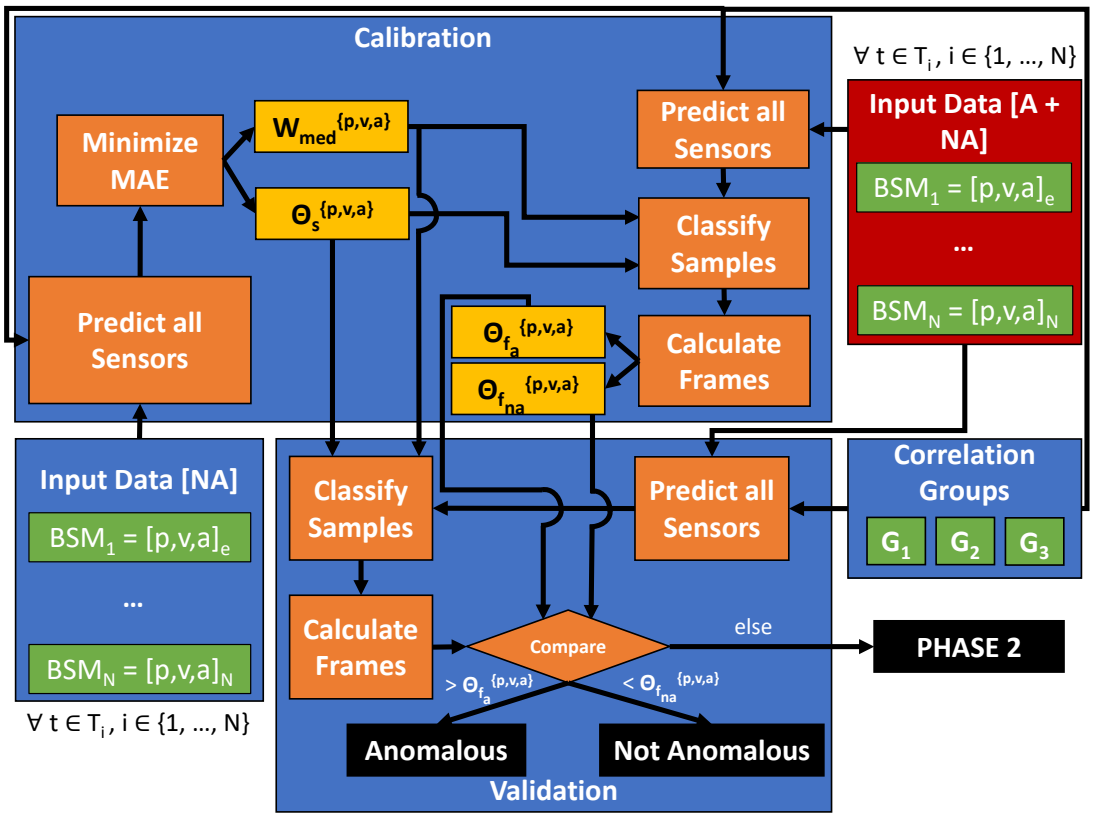


Figure 5.3: Phase 1 system design (A: Anomalous, NA: Non-anomalous)

5.5.2 Calibration

The calibration module is configured to operate offline, presumably at the time of vehicle manufacturing. The objective of calibration is to compute four thresholds for each sensor $i \in \{p, v, a\}$: the median window size W^i_{med}, per-sample threshold θ^i_s, per-frame anomaly threshold $\theta^i_{f,a}$, and per-frame non-anomaly threshold $\theta^i_{f,na}$.

The first main idea behind Phase 1 is that a single anomalous sensor reading in a BSM can be detected by leveraging the aforementioned correlation groups. For instance, if only the vehicle's speed v is anomalous (due to intentional spoofing or faults), Eq. (5.9) can be used to calculate the correct speed from the GPS readings. If v is not anomalous, G2 should yield a similar value to the speed in the received BSM. Since GPS readings can be noisy/erroneous (like any other sensor readings), the difference between *calculated* and *received* values will have a small error. The mean of all these errors over the entire non-anomalous datasets is characterized by the per-sample threshold θ^i_s for each correlation group. This threshold is determined by computing the *Mean Absolute Error* (MAE) for each sample t between the received signal x_t and estimated signal $\hat{x}_t$:

$$MAE_t = |x_t - \hat{x}_t|. \tag{5.11}$$

Ideally, the estimated and received signals should almost be identical for non-anomalous data. Nevertheless, one problem with the estimation process via Eqs. (5.8)–(5.10) is its proneness to noise. Using a rolling mean filter of length W, the raw estimation can be smoothed and the error between received and estimated signal minimized. The window length W is a design parameter which is used to find the window W^i_{med} for a sensor i to minimize the MAE. This process is repeated for all trips in the non-anomalous dataset and the minimized MAEs are then averaged, yielding θ^i_s. The filter uses the previous W^i_{med} samples up to the current, guaranteeing real-time

detection.

These two calibrated parameters W^i_{med} and θ^i_s may be sufficient for detecting the anomaly in a BSM. Nevertheless, we introduce further statistical techniques and design choices to reduce the false alarm rate of Phase 1. Similar to a cumulative sum, we compute an average $\theta^i_{f,a}$ for all anomalous frames, for each sensor group. A frame prediction is the mean of predicted conditions (1 for anomalous or 0 for non-anomalous) based on θ^i_s for samples in a frame f. This same calculation is computed for non-anomalous frames, $\theta^i_{f,na}$. The per-frame threshold computation can be generalized in Eq. (5.12), where N is the number of samples and $|f|$ the frame size:

$$\theta^i_{f,*} = \frac{1}{N}\sum_{j=1}^{N}\frac{1}{|f|}\sum_{k=j\cdot|f|}^{j\cdot|f|+|f|} x^i_k. \tag{5.12}$$

The introduction of the above thresholds helps ensure truly anomalous or truly non-anomalous frames are detected immediately by Phase 1. Therefore, Phase 1 considers frames, and the computed frame value is compared to the respective $\theta^i_{f,*}$ for anomalous and non-anomalous vehicles to make a decision. Note that all trained thresholds at manufacturing time can be updated by the OEM since modern vehicles have over-the-air update capabilities.

5.5.3 Validation

Anomaly detection of Phase 1 operates online and extracts the relevant signal data from each BSM. Using Eqs. (5.8)–(5.10) and the four calibration parameters W^i_{med}, θ^i_s, $\theta^i_{f,a}$, and $\theta^i_{f,na}$, it first calculates the MAE for all three correlation groups and compares it with the threshold θ^i_s computed during the training. If the computed MAE is larger than the threshold, the predicted condition for the respective sample is marked as *anomalous*. Likewise, if the computed MAE is smaller than the threshold, the predicted condition for the sample is marked as *non-anomalous*. After collecting W^i_{med} samples, the samples are smoothed by a rolling mean filter of window

size W^i_{med}, and the MAE is re-calculated and compared again with the threshold to update the appropriate markings. Although this might seem like a disconnect to the real-time nature of Phase 1, the prediction is only *updated* to achieve better detection performance. For small window sizes, this is, in fact, a favorable enhancement. Subsequently, we compute the mean of all predicted conditions for samples in a frame, such that if this computed frame prediction is greater than, or equal to $\theta^i_{f,a}$, we mark this frame (and all samples within the frame) as *highly-likely anomalous*, and if this computed frame prediction is less than, or equal to $\theta^i_{f,na}$, we mark this frame as *highly-likely non-anomalous*. If the computed frame prediction lies between these two thresholds, the affected frames are sent to Phase 2 for further investigation.

5.6 Phase 2: Remote Anomaly Detection

5.6.1 Overview

Phase 2 is concerned with ML-based anomaly detection to be used for the most opportunistic deployment (see Figure 5.1). However, a key distinction lies with the fact that Phase 2 is done *remotely*, i.e., neither in real time (due to unpredictable network latency and lack of wide-scale edge deployment) nor necessarily aboard a vehicle. The goal of Phase 2 is to detect anomalies in BSMs in frames sent from Phase 1, marked as *potentially anomalous/non-anomalous* (i.e., between $\theta^i_{f,a}$ and $\theta^i_{f,na}$). Thus, Phase 2 receives a given frame of sensor readings $f = [f_1, f_2, \ldots, f_N]$ of length N, where f_t and f_{t+1} are vectors of features corresponding to samples with consecutive timestamps. This allows Phase 2 to accurately detect samples that Phase 1 fails to make a definite decision. As shown in Fig. 5.4, each sensor group has an associated model which is executed in parallel (followed by a logical-OR similar to Phase 1). To demonstrate anomaly detection performance in Phase 2, we employ three models: Random Forest (RF), Support Vector Machine (SVM), and Deep Neural

Network (DNN).

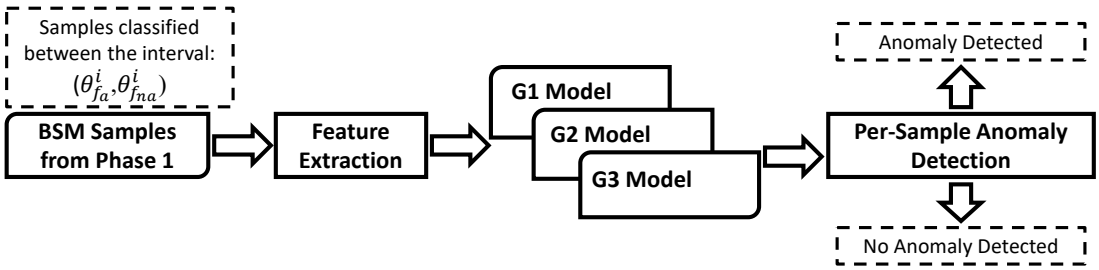


Figure 5.4: Phase 2 system overview

5.6.2 Feature Extraction

To provide reasonable performance benchmarks for Phase 2 evaluation, we carefully extract features for the candidate models. Despite the potential increase of computation time, feature extraction generally improves ML model performance [87]. Motivated by [163], for each correlation group, we extract the rolling mean, rolling mean displacement, e stimated displacement and plausibility check, respectively. A detailed account of these features is provided in Sec. D.1.

5.6.3 Training and Validation

Random Forest For RF models, we use K-fold cross-validation for training and validation of our models, where $K = 5$. In particular, the RF models were tuned to have 100 estimators in its ensemble [54]. Furthermore, prior to training and evaluation, for each correlation group, we extract the relevant features detailed in Sec. 5.6.2.

Support Vector Machine For SVM models, we perform similar training and validation as RFs. To prevent skewed decision boundaries, we standardize all features. Performance generally improves as C (penalty for misclassification) increases, although the training time may also increase. As a result, we select $C = 10$ and the Radial Basis Kernel Function as our key hyperparameters [131].

Deep Neural Network For DNN models, we use a 60/20/20 training, validation, and testing split each for 25–50 epochs (early stopping) with a batch size of 32. Similarly, DNNs use the same feature extraction as RFs and SVMs. Furthermore, we adopt an architecture consisting of 6 hidden layers with $32, 64, 128, 64, 32$ hidden nodes, respectively, each with Rectified Linear Unit (ReLU) activation function. To reduce overfitting, our layers are followed by a dropout of 0.2 [167]. Likewise, the output layer is a Sigmoid activation function of its input.

5.7 Evaluation

5.7.1 Experimental Setup

For the experimental evaluation of `CARdea`, we use `Veins` based on OMNET++ and SUMO to conduct simulations with various attack types. More precisely, we use the Luxembourg SUMO Traffic Scenario (LuST) which contains a topology resembling European cities with real traffic demands and mobility patterns [63]. For our dataset, we use a subsection of the LuST network with a size of $6.63km^2$. We ran 108 hours of simulations featuring upwards of 49K vehicles where $\alpha = 0.05$, the probability an attacker vehicle was spawned. Furthermore, $\beta = 0.01$, the IVAR, equivalently the probability an attacker vehicle may broadcast a sequence of anomalous messages with sequence length varying from 30 to 70 (i.e., 3 seconds to 7 seconds) [171]. From this, we collect the BSM data under these simulated conditions for vehicles equipped with V2X communication capabilities.

All experiments in both phases were conducted using Python 3. In order to mimic an in-vehicle ECU in Phase 1, we used a Raspberry Pi 3 Model B clocked at 1.2GHz with 1GB of RAM. For Phase 2, a computer running 64-bit Ubuntu 18.04 LTS with 128GB of registered RAM and 64 Intel Xeon E5-2683 V4 CPUs was deployed.

5.7.2 Anomaly Generation

As discussed in Sec. 5.4.2, we simulate various scenarios containing normal and attacker vehicles. Each attacker vehicle executes one of the seven assumed attack types by broadcasting anomalous or non-anomalous messages to all other vehicles within range. From each scenario, we generate several simulated datasets for an ego vehicle containing at least one malicious message received from an attacker vehicle in range throughout the entire simulation. We execute all attack types on position, speed, and longitudinal acceleration as described in Eq. (5.1)–(5.7). More precisely, specific values that are assigned or added to each variable (based on the attack type) are provided in Table 5.2 for AT1–AT4. Sybil (AT5), Data Replay (AT6), and Stealthy (AT7) attacks do not rely on similar parametrization, but rather the nature of the attack. AT7 is conducted by configuring an attacker parameter b_i and *a priori* knowledge of each correlation group's computed θ_s^i.

Table 5.2: AT1–AT4 attack type parameters

AT*	AT1	AT2	AT3	AT4
p (m)	[0,3900]	[31.5,108.5]	[0,3900]	[-70,70]
v (m/s)	[0,40]	[3.15,10.85]	[0,40]	[-0.7,0.7]
a (m/s^2)	[0,2]	[0.9,3.1]	[0,2]	[-0.4,0.4]

5.7.3 Data Preparation

We first run simulations in Veins as detailed in Sec. 5.4.2 with parameters specified in Sec. 5.7.2. The simulator outputs JSON traces of received BSMs for each ego vehicle. Upon completion of a simulation run, we convert each JSON trace to an appropriate CSV file for numerous ego vehicles. Then, we label each sample, a single measurement at a point in time, as *anomalous* (1) or *non-anomalous* (0) and set aside specific datasets for training and testing to prevent temporal inversion. Note that our simulation includes several attacker vehicles, and messages transmitted by the attacker vehicles may consist of a probabilistic mix of anomalous *and*

non-anomalous sequences of messages (based on IVAR). Again, we define the three correlation groups as G1–G3, and assume one ML model per group for Phase 2. We perform feature extraction for each ego vehicle by grouping BSMs by transmitting vehicles and extracting relevant features pertaining to the data of that vehicle, as detailed in Sec. 5.6.2.

5.7.4 Evaluation Metrics

Performance For the performance of both phases, we evaluate the True Positive Rate (TPR), False Positive Rate (FPR), and F-1 score.

Computation Time In Phase 1, we refer to the computation time as *detection latency*, since it is the crucial real-time metric to first detect a potential anomaly. It is defined as the time it takes to compute all three correlation groups defined in Eqs. (5.8)–(5.10) consecutively.

In Phase 2, we discuss the more generic term *computation time* since the initial detection occurs in Phase 1. This metric is defined as the elapsed time during the prediction of all samples in a frame on an ML model. Note that the network latency is not considered in `CARdea` since it depends on factors that are outside our scope.

Memory Consumption For both phases, we evaluate memory consumption in terms of RAM which was measured using Unix `process status` feature. We focus on RAM instead of non-volatile memory because Phase 1 only has to save Eqs. (5.8)–(5.10) and some training parameters, and Phase 2 will have abundant resources to store the trained ML models. Assessing the RAM consumption is particularly important for the in-vehicle Phase 1 since resources there are limited. Measuring RAM usage for Phase 2 on the Raspberry Pi also indicates the infeasibility of its deployment inside a vehicle.

5.7.5 Phase 1

5.7.5.1 Performance

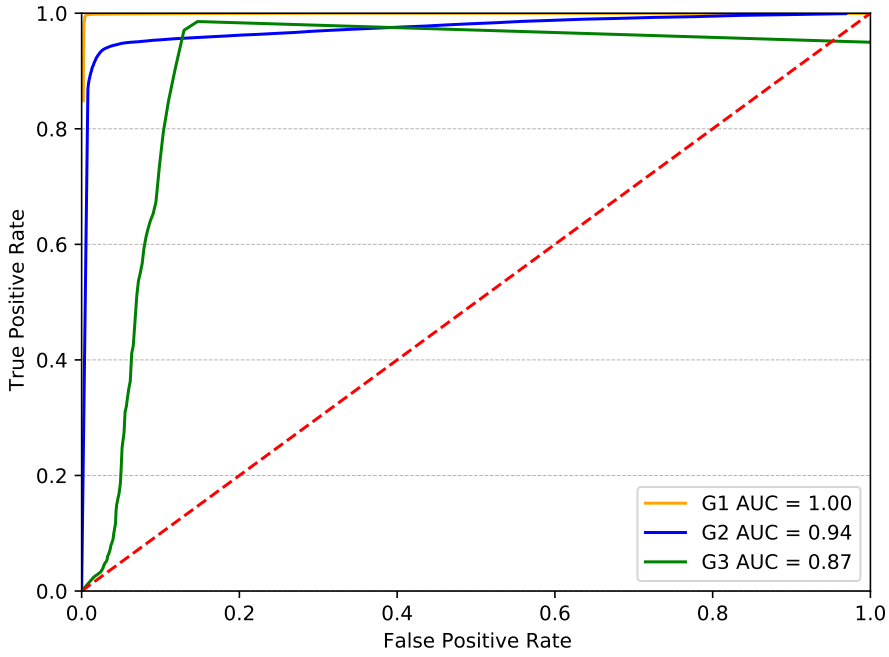
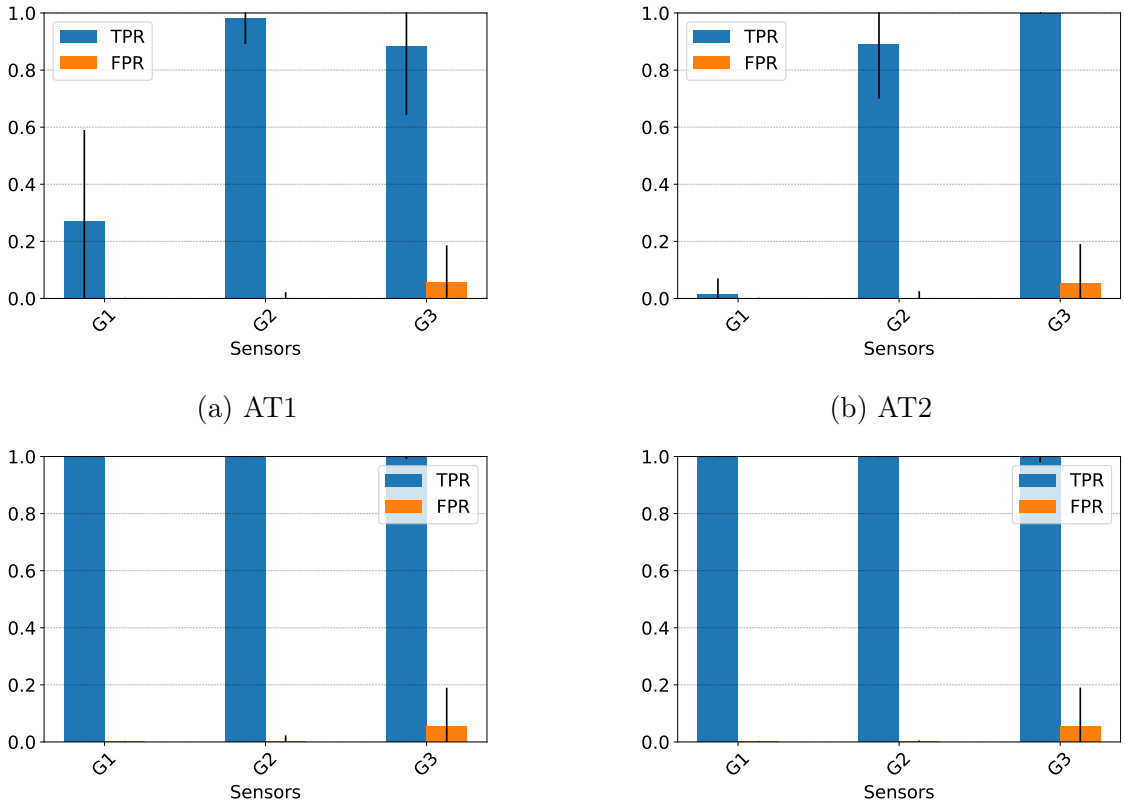


Figure 5.5: Phase 1 combined ROC curves for θ_s^i

We first calculated the Receiver Operating Characteristic (ROC) curve (see Fig. 5.5) using our training data for each correlation group to determine a satisfactory θ_s^i. To select an optimal θ_s^i, we elected the first threshold with at least 90% TPR, in accordance with the ROC curve. Then, we used θ_s^i to determine $\theta_{f,*}^i$ values for each correlation group. We calculated the means over all different configurations of the anomalous datasets (see Sec. 5.7.2). For G1, $\theta_{f,a}^1$ and $\theta_{f,na}^1$ are 0.95 and 0.015, respectively. For G2, the parameters are set at 0.83 and 0.021, and for G3, 0.90 and 0.04, respectively. Additionally, when computing the $\theta_{f,a}^i$ and $\theta_{f,na}^i$ values, we evaluated them on frames of size $3, 5, 7$, and 10. As the frame size increases, the performance generally improves, as shown in Fig. D.1 see Appendix D.1). G3 FPR, for example, decreases from $\approx 9\%$ for a frame size of 3 to $\approx 5.5\%$ for a frame size of 7, and then plateaus. As a result, we selected a frame size of 10 for detection performance, latency

and resource metrics.



(a) AT1

(b) AT2

(c) AT3

(d) AT4

Figure 5.6: Phase 1 AT1 – AT4 per-frame performance with frame size 10

Note that we want to detect anomalous frames with high TPR, and non-anomalous frames with low FPR, both in real time. Our performance evaluation in Fig. 5.6 reflects this goal for Phase 1. As shown in the figures, with high confidence, if Phase 1 classifies a frame as anomalous or non-anomalous, any mitigating actions can be taken without engaging Phase 2. However, G1 AT1 and AT2 performances are not comparable to the other attack types. This is a limitation of Phase 1, and these position spoofing attacks can be detected with signal strength (i.e., RSSI) as pointed out in [162]. Upon further analysis of estimated and received traces for G1 AT1 and AT2, we find that in the transition from non-anomalous to anomalous segments (or vice versa) in the data, the MAEs exhibit a sharp jump. As a result,

because G1 is the only correlation group that uses its previous value (which may have been spoofed) to calculate the estimated value, the MAEs are not likely to fall below the determined threshold θ_s^i consistently. Thus, complementing Phase 1 with an RSSI-based plausibility check (and taking a logical-OR after Phase 1 groups are run in parallel) can increase TPR for these attack types by up to 40% [162]. The authors utilize the difference between the sender and receiver's position and the RSSI to construct confidence intervals for normal RSSI behavior which can be stored at manufacturing time as well. For the more sophisticated attacks, namely AT5 and AT6, Phase 1 reports upwards of 99% TPR and 0.005% FPR. Stealthy attacks, however, were designed to evade Phase 1 detection. As a result, `CARdea` is particularly vulnerable to this group of attacks which need to be detected in Phase 1 to avoid sending all frames to Phase 2 and voiding the benefits of our hybrid system design. We provide discussion on detecting Stealthy attacks in Phase 1 in Sec. 5.8, but do not implement or evaluate them at this point.

5.7.5.2 Detection Latency

The mean latency has been calculated by averaging all data points in a particular anomalous dataset. The experiments were conducted on the less powerful Raspberry Pi 3 Model B that has similar specifications to the most powerful in-vehicle ECUs. As the top row of Table 5.3 shows, the mean latency for processing one frame by all 4 attack types stands at around 0.09–0.11ms with a standard deviation of 0.03ms, which is generally consistent among all 3 correlation groups. These numbers also hold up for AT5–7. Given that a BSM arrives every 100 ms, the latency overhead by our detection module running on the vehicular gateway is almost negligible. The estimated network latency stands at an additional 0.5ms if the transmission of BSM data to the anomaly detection module is prioritized. Considering an AV platooning scenario, Phase 1 detection is at least ≈2000x smaller than the total delay required to

return to full autonomy in a compromised platoon for platoon sizes of 2 (total delay = 247 ms) to 8 (total delay = 1457 ms) [79].

5.7.5.3 Memory Consumption

The bottom row of Table 5.3 shows that the average RAM usage among all four attack types stands around 39MB. Given the low RAM found on in-vehicle ECUs [16, 41] (few MBs), the assessed numbers might look too high. Nevertheless, note that the anomaly detection module is written in Python with multiple libraries (such as `pandas`) to process the data. An implementation on a more embedded system-friendly language such as C would drastically lower the required RAM usage, especially considering the few parameters and equations that have to be stored.

Table 5.3: Detection latency and RAM usage for Phase 1

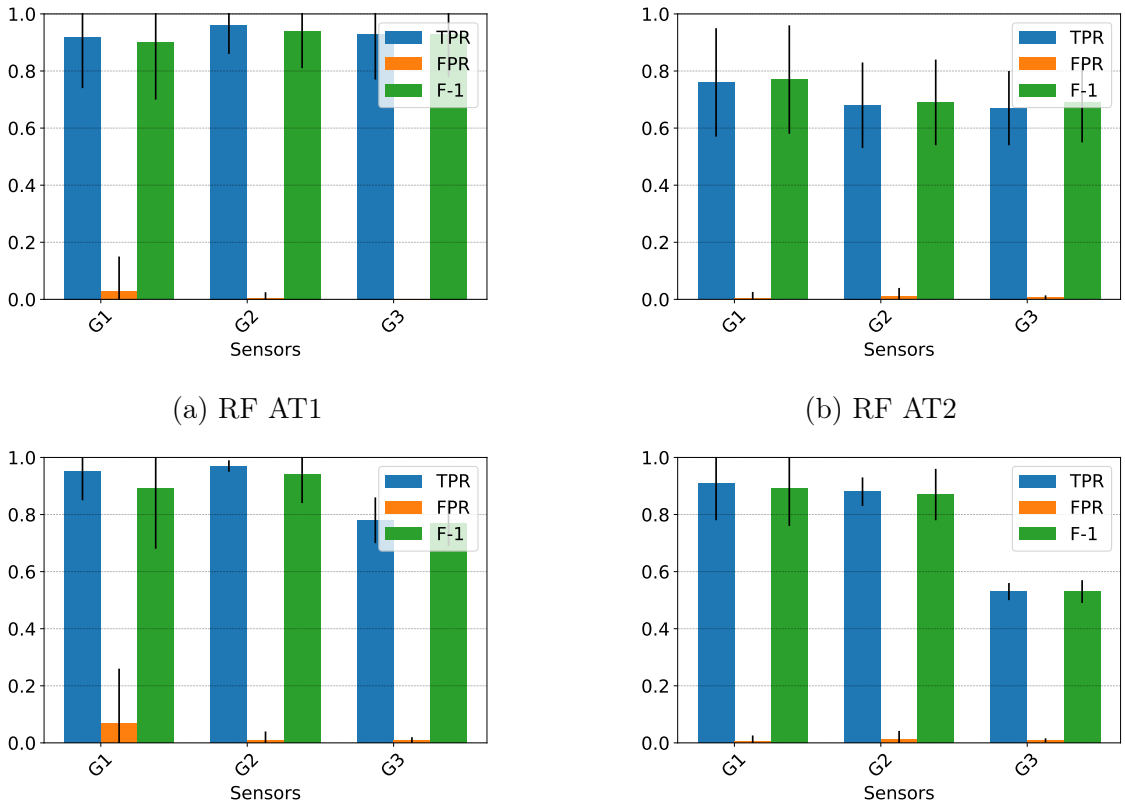
Attack Type	**AT1**	**AT2**	**AT3**	**AT4**
Latency (ms)	0.09 (0.03)	0.09 (0.03)	0.11 (0.03)	0.1 (0.03)
RAM Usage (MB)	38.8	38.9	38.8	38.9

5.7.6 Phase 2

We evaluate the TPR, FPR, and F-1 score of Phase 2, both independently and jointly with Phase 1. We first evaluate Phase 2 in a scenario where all frames from Phase 1 are sent to it.

5.7.6.1 Performance

Fig. 5.7 shows performance for RF models for AT1 – AT4. Compared to Phase 1's *per-frame* performance evaluation, Phase 2 focuses on *per-sample* classification which explains certain lower TPR values, despite its generally favorable performance. The authors of [178] also evaluate their system on a constant attacker. However, they inject anomalies on a static dataset, as opposed to conducting a simulation which



(a) RF AT1

(b) RF AT2

(c) RF AT3

(d) RF AT4

Figure 5.7: Phase 2 AT1 – AT4 per-sample performance

makes their evaluation less realistic, yet report comparable performance to our models. Additionally, prior work such as [172] implemented the constant offset attacker, yet relied on AoA and Doppler shift to achieve comparable results. One limitation of Phase 1 (as discussed in Sec. 5.7.5.1) was the need to incorporate RSSI to detect such attacks, which might also be required for Phase 2, as AT2 performance suggests. However, AT1 performance is significantly better than Phase 1, even without considering RSSI.

Fig. D.2 (in Appendix D.1) shows the performance for all three ML models for Sybil (AT5) and Data Replay (AT6) attacks. The general trend is 90%+ average TPR and $< 1-2\%$ FPR across all models and attack types. RFs specifically report less than 1% FPR and 95%+ TPR. The performance evaluation of the Stealthy attack (AT7) is

depicted in Fig. D.4 (see Appendix D.1). Note that these Stealthy attacks are a subtle extension of the Constant Offset attacker. As a result, Phase 2 is better at detecting some stealthy attacks, while others may go undetected. For example, some groups (G2) may achieve comparable performance to the previous attack types, upwards of 80–90% TPR. However, other groups (G3) may suffer, although there was no observed correlation in detection performance and the attacker's b_i selection. Overall, RFs consistently achieve the best performance of SVMs and DNNs (see Fig. D.3 in Appendix D.1).

5.7.6.2 Computation Time

Table D.4 (in Appendix D.1) provides the average testing time for RFs, SVMs, and DNNs. Consistent with Phase 1's evaluation, the computation time was benchmarked for frames of size 3, 5, 7, and 10 on the same RPi used in Phase 1 latency evaluation, as well as an Ubuntu server. As shown in Table D.4, the computation time of our models on the RPi significantly exceeds the computation time on the Ubuntu server. For example, RFs require only 16.0ms on the Ubuntu server for a frame size of 10, whereas requires upwards of 101.9ms on the RPi (longer than the BSM period of 100ms!). In comparison to 0.09–0.11ms (Phase 1 detection latency on the RPi), the heavier computation power required for ML models is clearly not ideal under the tighter runtime constraints for in-vehicle ECUs.

5.7.6.3 Memory Consumption

Table D.4 also presents the average RAM consumption of our ML models benchmarked on the Ubuntu server and Raspberry Pi. For in-vehicle ECUs, a high RAM consumption, caused by storing the trained ML model in dynamic memory, exceeds the vehicle's already constrained memory resource, and is thus not preferred due to cost. As observed, RFs and SVMs still require more than 100 MB, while deep-learning

based models may require as much as 3x more RAM. Despite the different target architectures in Phases 1 and 2, we observe the average RAM usage to be around 5x higher for ML-based techniques than statistical methods in Phase 1 (see Table 5.3). In contrast, memory usage levels are of less concern on the manufacturer's backend where Phase 2 can be deployed.

5.7.7 Interactions between Phase 1 and 2

In what follows, we perform a combined evaluation of Phases 1 and 2 jointly to evaluate Phase 2 on the frames sent from Phase 1. We can express this real-time interaction between the two phases in Algorithm 6, where the input X are Phase 1's estimated sensor readings of frame size N. We selected three different scenarios pertaining to three ego vehicles for AT1, AT3, and AT4 for Acceleration, Speed, and Position Spoofing, respectively. Phase 2 implements an RF model, and we select a frame size of 10, consistent with our preceding evaluation. For completeness, all combinations are reported in Appendix D.1. In all three selected scenarios, Phase 2 reports around 99% TPR and less than 1% FPR as shown in Table 5.4, in line with the results in Fig. 5.7.

Table 5.4: Phase 2 results on frames from Phase 1

G* AT*	**G1 AT4**	**G2 AT3**	**G3 AT1**
TPR (%)	99.51 (4.5)	99.06 (6.2)	98.64 (9.4)
FPR (%)	0.0 (0.0)	0.16 (2.2)	0.09 (0.99)

5.7.8 Bandwidth

Finally, we would like to discuss how many of the *potentially anomalous*-flagged frames from Phase 1 have to be sent to Phase 2. This will have an impact on bandwidth which is another important metric for an OEM since it drives up cost. The car will very likely use the built-in cellular connection. Each BSM has a default size of

Algorithm 6 Phase 1 and 2 Real-time Interaction

procedure PREDICT(X, N)
    $pc, frame_{pc} = \emptyset$
    for $x_t^i \in X$ **do**
        if $x_t^i \geq \theta_s^i$ **then** $pc[t] = 1$
        else$pc[t] = 0$
    $frame_{pc} = \overline{(pc)}$
    if $frame_{pc} \geq \theta_{f,a}^i$ **then** Mark as anomalous
    else if $frame_{pc} \leq \theta_{f,na}^i$ **then** Mark as non-anomalous
    else ▷ Offload frame to Phase 2
        if $f(X) = 1$ **then** Mark as anomalous
        else Mark as non-anomalous

254 bytes [49] and will be sent using the Transmission Control Protocol (TCP). This is necessary since we cannot assume that the network connection during a trip will always be reliable (e.g., driving through tunnels). Together with the TCP/IP header overhead, each packet including the BSM will have a total size of 318 bytes. Table D.5 (in Appendix D.1) summarizes the mean percentage of frames in each experiment to be sent to Phase 2 (a mean number of frames to be sent to Phase 2 d_{p2} over total number of frames d_{total} for given vehicle), as well as the required bandwidth b. The latter is calculated (in bytes/s) as:

$$b = \frac{d_{p2} \cdot 318bytes}{d_{total} \cdot 0.1s} \tag{5.13}$$

The required bandwidth is always 0.56 kB/s or lower only if every flagged frame is transmitted. On average, a maximum of 1.8 MB would be transmitted every hour which is comparable to less than 1 minute of streaming music through the popular Spotify app in very high quality (320kbps) [5]. In Table D.5, we considered the frame size f to be 10. For smaller frame sizes, the required bandwidth is lower since less frames are sent to Phase 2. As a result, the reported numbers represent an upper bound for the bandwidth consumption. Better performance with larger frame sizes (see Fig. D.1) comes at the expense of more bandwidth consumption. Choosing an

optimal frame size can thus be making a trade-off between performance and cost. Table 5.5 presents a condensed version of Table D.5.

Table 5.5: Bandwidth averages across attack types per G

Sensor Group G_i	**G1**	**G2**	**G3**
Mean Percentage (%)	6.2	3.8	18.3
Bandwidth (kB/s)	0.20	0.12	0.50

5.8 Discussion and Conclusion

One drawback of `CARdea`'s current system design is that it only comprises *single*-sensor spoofing in AT1–4, where one consistent sensor is anomalous per BSM. Another case is *multi*-sensor spoofing, where more than one sensor may be anomalous. We can enumerate multi-sensor spoofing scenarios as follows (note the same subsequent logic dcapplies when more than three sensors are utilized):

1. p is anomalous, v is anomalous, a is not anomalous
2. p is anomalous, v is not anomalous, a is anomalous
3. p is not anomalous, v is anomalous, a is anomalous
4. p is anomalous, v is anomalous, a is anomalous.

In the first three scenarios, Phase 1 should detect an inconsistency in at least one sensor because it leverages the correlation between these sensor groups. In the last scenario, all sensor values, albeit being spoofed, may be consistent with the vehicle dynamics. This is an inherent limitation of physics-based anomaly detection which is also pointed out in [162]. By complementing Phase 1 with RSSI (and optionally AoA) this risk may be mediated.

Thus far, we have shown and evaluated how splitting anomaly detection into two phases to reflect and satisfy the OEMs' constraints is the key advantage of `CARdea`. As discussed in Sec. 5.1, Phase 2 of `CARdea` can be deployed in the cloud, edge or

even on the vehicle itself, given it has sufficient computational resources. OEMs can choose where to deploy Phase 2 based on their priorities. Using "super-ECUs" inside the vehicle will reduce bandwidth and the cost associated with it, but also require more expensive ECUs, and vice versa. It is also possible to run Phase 2 selectively on whatever resource is the most optimal at a given time. For example, while driving through an area without good cellular coverage, existing ECUs that are normally used for ADAS tasks can be leveraged for a brief period of time.

Furthermore, we showed how a sophisticated adversary can conduct stealthy attacks that may evade `CARdea`'s detection. In future, we would like to improve `CARdea`'s resilience against such attacks. It is possible to extend `CARdea` to detect stealthy patterns in received data. For example, if Phase 1 continuously monitors the MAE, it can determine whether they follow a Gaussian distribution which is likely due to measurement noise for truly non-anomalous vehicles. On the other hand, non-Gaussian patterns may be a result of a stealthy attacker attempting to evade detection by staying a constant delta below the learned threshold, and `CARdea` can flag such a transmitting vehicle as anomalous without invoking Phase 2. Similarly, Phase 2 may incorporate further contexts, such as nonlinear physics equations (5.8)– (5.10) for physics-informed neural networks [147], weather, geolocation, etc., which is also part of our future inquiry.

We have proposed a practical real-time V2V anomaly detection scheme, called `CARdea`, to fill the disconnect between the detection performance focused by prior work and the practical deployment on resource-constrained vehicles. `CARdea` has shown that the trade-off between important metrics such as detection latency *and* favorable detection performance does not have to be a compromise if a carefully engineered two-phase system like `CARdea` is leveraged. Our experimental evaluation has shown that combining the advantages of a light-weight, in-vehicle, statistical-based anomaly detection technique with a powerful ML-based approach can fill the

aforementioned disconnect and offer a performant, yet practical solution for V2V anomaly detection.

CHAPTER VI

Conclusion and Future Directions

6.1 Conclusion

This dissertation has demonstrated that bringing security to ground vehicles does not have to come at the expense of practicality. OEMs impose several constraints on adding security solutions to their products as discussed in Sec. 1.5, with cost being the major driver behind their lack of adoption. The four main chapters of this thesis met these challenges with three intellectual contributions (ICs) outlined in Sec. 1.6. Summarized below is how each chapter covers the ICs.

6.1.1 IC1: Semantics can be automatically reverse-engineered, accelerating CAN injection attacks

LibreCAN. Chapter II demonstrated the relevance of CAN injection attacks as a final component of most automotive attacks so far. The reverse-engineering framework `LibreCAN` can greatly reduce the time spent on preparing a CAN injection attack from multiple days (spent for manual reverse engineering) to less than an hour. Besides covering most of a vehicle's CAN signals, `LibreCAN` is also accurate as demonstrated by our evaluation results.

6.1.2 IC2: Solve CAN security problems by satisfying the functional and cost constraints of OEMs

CAN injection attacks can be conducted as a result of several missing security properties on the CAN bus, such as *confidentiality*, *integrity*, *authenticity* and *availability*. Despite the existence of certain standards [30], they neither cover all of these properties nor meet the functional and cost constraints imposed by OEMs.

S2-CAN. Chapter III adds confidentiality, integrity and authenticity to the CAN bus by proposing `S2-CAN`. Taking a rather unconventional approach by avoiding use of cryptography, it significantly outperforms other relevant approaches with respect to latency, bus load, CPU utilization and memory consumption. Despite its weaker security level than existing solutions, `S2-CAN` is sufficiently secure under real-world assumptions. `LibreCAN` from IC1 was used further to assess `S2-CAN`'s security level which can both thwart CAN injection attacks that result from the lack of aforementioned security properties, as well as satisfy all the functional and cost constraints of OEMs.

MichiCAN. Chapter IV adds availability to the CAN bus by proposing `MichiCAN`. Denial-of-Service (DoS) attacks can be detected by monitoring the CAN ID of messages on the bus. Compared to existing work, `MichiCAN` can both detect the attack in real time during its first appearance on the bus, as well as prevent it immediately by confining the attacker ECU into its bus-off state. `MichiCAN` makes use of new MCUs with integrated CAN controllers that allow bit-level read and write access to the CAN bus. Furthermore, it is deployable by OEMs since it is fully backward-compatible and incurs minimal overhead to the network traffic — the limitations of prior work.

Finally, both `S2-CAN` and `MichiCAN` can be used together for a secure and practical CAN bus as they protect different security properties and thus thwart different types

of CAN injection attacks.

6.1.3 IC3: Solve V2V security problems by hybrid approach combining in-vehicle and off-vehicle anomaly detection

Chapter V demonstrates that practical security is needed not only in the in-vehicle network, but also in vehicle-to-vehicle (V2V) communications. The anomaly detection system `CARdea` is composed of two phases which analyze the incoming data from other vehicles before making a decision to maintain safety. Compared to existing work which deploys resource-heavy ML-based techniques for anomaly detection, most malicious or faulty broadcasts can be detected by the first phase of `CARdea` locally inside the vehicle without the use of machine learning. `CARdea` can pass on the sanitized input to the vehicle with minimal detection latency and computational overhead which satisfy OEM-imposed constraints. To enhance accuracy, the second phase that uses machine learning will only be applied to the data which the first phase marks as ambiguous. As a result, `CARdea` provides good detection performance without sacrificing practical deployment in future connected vehicles.

6.2 Future Directions

Threats to vehicles and its ecosystem will continue to pose unique challenges in the future, since the new technologies in this area are still in their infancy. In future, I plan to shift my focus from the well-researched area of defensive CAN security to connected and autonomous vehicles (CAVs).

6.2.1 Connected Vehicle Ecosystem

New wireless interfaces are emerging in contemporary vehicle ecosystems. One example is novel infotainment operating systems, such as Android Automotive (a car-specific version of the popular mobile operating system Android) [21]. It can

collect sensor data directly from the vehicle and share it with the carmaker and interested third parties. I have already conducted the first high-level security analysis of Android Automotive in 2020 [137]. Due to the integration of Android Automotive with the in-vehicle network, it must have a secure system architecture to prevent any potential attacks that might compromise the security and privacy of vehicles and drivers. In particular, malicious third-party apps could remotely compromise a vehicle's functionalities to impact vehicle safety, e.g., by launching a CAN injection attack *remotely*. This vulnerable interface will open the door to a new generation of increasingly scalable cyber-attacks against vehicles, eliminating the need to be physically near the target vehicle.

Another promising research area will be privacy. Privacy trackers are included by Google Automotive Services (GAS) and/or pre-installed third-party apps on production builds. Since Google, OEMs and third-party entities are interested in monetizing user data from vehicles, it will be necessary to analyze what data is shared for what purpose. Once more production vehicles are shipped with Android Automotive and the number of third-party apps that can access various vehicle data increases, I plan to conduct an in-depth analysis of Play Store apps that violate drivers' privacy by collecting sensitive user data.

Another example of novel interfaces for the connected vehicle ecosystem is widely available mobile companion apps (e.g., BMW ConnectedDrive [34]) that can remotely start or even *park* the vehicle [25], further increasing the interconnection of carmakers' infrastructure with their cars. Vulnerabilities in this ecosystem can seriously impact safety. So far, research has been very sparse in this field. I would like to study carmakers' connected infrastructure to understand this new threat. This will help me analyze existing commercial solutions and design countermeasures against attacks in the future.

6.2.2 Adversarial Attacks on Autonomous Vehicles

In recent years, many deep learning models have been adopted in autonomous perception systems. Security vulnerabilities are an ever-present concern to drivers' safety and manufacturers' liability. Recently, it has been shown that the underlying perception systems exhibit severe vulnerabilities when exposed to adversarial conditions [143]. Among others, attacks can be classified by the vehicular sensor they target (i.e., cameras, LiDARs) to the type of adversarial knowledge (i.e., white-box vs. black-box). While some attacks have been demonstrated in research settings, the extent to which deployed autonomous vehicles (AVs) are susceptible to physical world attacks is still not fully understood. To address this challenge, I plan to investigate the vulnerability of deployed AV perception systems under new physical world attacks and propose defenses for these trending and pressing security issues.

APPENDICES

APPENDIX A

LibreCAN: Automated CAN Message Translator

A.1 LibreCAN: Vehicular Signals

Table A.1 depicts an overview of frequently installed ECUs in newer vehicles. It also includes physical signals that each ECU might generate.

In the following, we present a full list of physical relationships between certain elements in set S:

- Torque (τ) and engine speed (rpm) share a linear relationship for engine speeds lower than 2000-3000 RPM, as can be extracted from torque curves [40]. Since the engine speed is lower than the aforementioned threshold during almost the entire drive, we can assume that τ and rpm are proportional to each other:

$$\tau \approx rpm. \tag{A.1}$$

- Engine load ($load_{engine}$) can be calculated as the fraction of actual engine output torque (τ) to the maximum engine output torque ($\tau_{engine,max}$):

$$load_{engine} \propto \tau. \tag{A.2}$$

Table A.1: Overview of common ECUs with respective signals

ECU	**Signals**
Powertrain Control Module (PCM) — usually combination of Engine Control Module (ECM) and Transmission Control Module (TCM)	Pedal Position Throttle Position Engine Oil Temperature Fuel Level Oil Pressure Wheel Speeds Engine Speed Torque Coolant Temperature Engine Load
Body Control Module (BCM)	HVAC Turn Signals Lights Wipers Trunk Doors Windows Mirrors Remote Keyless Entry
Telematic Control Unit (TCU)	Radio GPS
Advanced Driver Assistant Systems (ADAS)	Cameras (e.g. rear-view) Radar LiDAR
Instrument Cluster (IC)	Vehicle Speed Engine Speed Current Gear MIL Light TPMS Light Odometer Fuel Level Engine Temperature Turn Signals
Supplemental Restraint Systems (SRS)	Airbag Status Seatbelt Status
Electronic Power Steering (EPS)	Steering Wheel Torque Steering Wheel Position Wheel Speed

- For engine speed values up to approximately 2000 RPM, torque (τ) and pressure boost (p_{boost}) are linearly related [184]. Furthermore, for boosted engines, such as in vehicles with turbochargers (all of our evaluation vehicles except Vehicle C), the intake manifold pressure (p_{map}) is proportional to p_{boost}:

$$\tau \propto p_{boost} \propto p_{map}. \tag{A.3}$$

- The electrical circuitry in the Accelerator Pedal Position (APP) and Throttle Position (TPS) sensors is identical [103]. Both sensors are fixed to the throttle body and convert the position of the throttle pedal to a voltage reading. As a result, accelerator pedal position (ACC_PED) and throttle position (THR_POS) are highly related:

$$ACC\_PED \propto THR\_POS. \tag{A.4}$$

- The centripetal acceleration (a_y) is proportional to the product of yaw rate and vehicle speed:

$$a_y \propto \omega_z v. \tag{A.5}$$

- The barometric pressure reading (p) obtained from phone sensors does not only change with the weather, but is also a function of the altitude (h) [8]. Via the *barometric formula*:

$$p \propto e^{-k \cdot h \cdot M}. \tag{A.6}$$

  In this equation, k is a constant and M the molar mass of dry air. Despite having an exponential curve, for small altitude changes, the relationship between p and h is approximately constant. Furthermore, considering the fact that weather does not change significantly during data collection, changes in p can be directly

linked to h.

A.2 LibreCAN: Phase 1

Table A.2 depicts a complete list of all signals in set S that we are considering for correlation in Phase 1. Table A.3 shows all 53 events that were analyzed for Phase 2.

Table A.2: Complete List of 24 Signals in Set S (Italic Signals are from Set $P \subset S$)

- Intake Manifold Pressure
- Ambient Air Temperature
- Speed
- Voltage (Control Module)
- Turbo Boost & Vacuum Gauge
- Fuel Rail Pressure
- Engine Coolant Temperature
- Torque
- Accelerator Pedal Position D
- Accelerator Pedal Position E
- Engine RPM
- Intake Air Temperature
- Engine Load (Absolute)
- Absolute Throttle Position B
- Fuel Flow Rate
- *Acceleration Sensor(X axis)*
- *Acceleration Sensor(Y axis)*
- *Acceleration Sensor(Z axis)*
- *G(x)*
- *G(y)*
- *G(z)*
- *Barometric Pressure*
- *Altitude*
- *Bearing*

A.3 LibreCAN: Phase 2

Fig. A.1 depicts which CAN IDs have been filtered out at what stage for each of the 53 events for Vehicle A. Fig. A.2, Fig. A.3, and Fig. A.4 are similar, but for Vehicles B, C, and D, respectively.

Table A.3: Complete List of 53 Events

- Lock driver's side
- Lock passenger's side
- Unlock driver's side
- Unlock passenger's side
- Open trunk
- Close trunk
- Open driver's door
- Close driver's door
- Open passenger's door
- Close passenger's door
- Open door left back
- Close door left back
- Open door right back
- Close door right back
- Open driver's window
- Close driver's window
- Open passenger's window
- Close passenger's window
- Open window left back
- Close window left back
- Open window right back
- Close window right back
- Turn on heating
- Incremental fan speed increase
- Increase temperature incrementally 65-75F
- Decrease temperature incrementally 75-65F
- Incremental fan speed decrease
- Air circulation button on
- Air circulation button off
- Honking horn
- Headlights off-on
- Headlights on-off
- Hazard lights on
- Hazard lights off
- Windshield wipers once
- Windshield wipers speed 1
- Windshield wipers speed 2
- Windshield wipers speed 3
- Interior lights all on
- Interior lights all off
- Windshield wiper fluid
- Left turn signal on
- Left turn signal off
- Right turn signal on
- Right turn signal off
- Activate parking brake
- Release parking brake
- Open hood
- Close hood
- Drivers side mirror left right up down
- Passengers side mirror left right up down
- Buckle driver
- Unbuckle driver

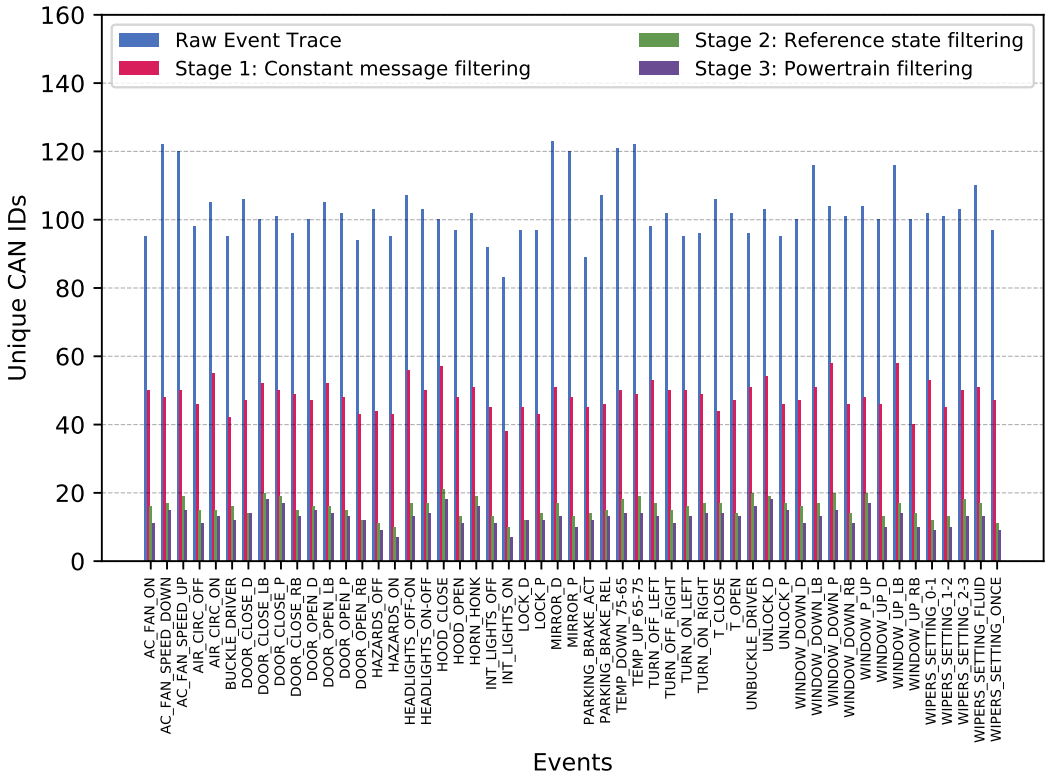


Figure A.1: Number of Unique CAN IDs Remaining After Each Stage for all 53 Events for Vehicle A

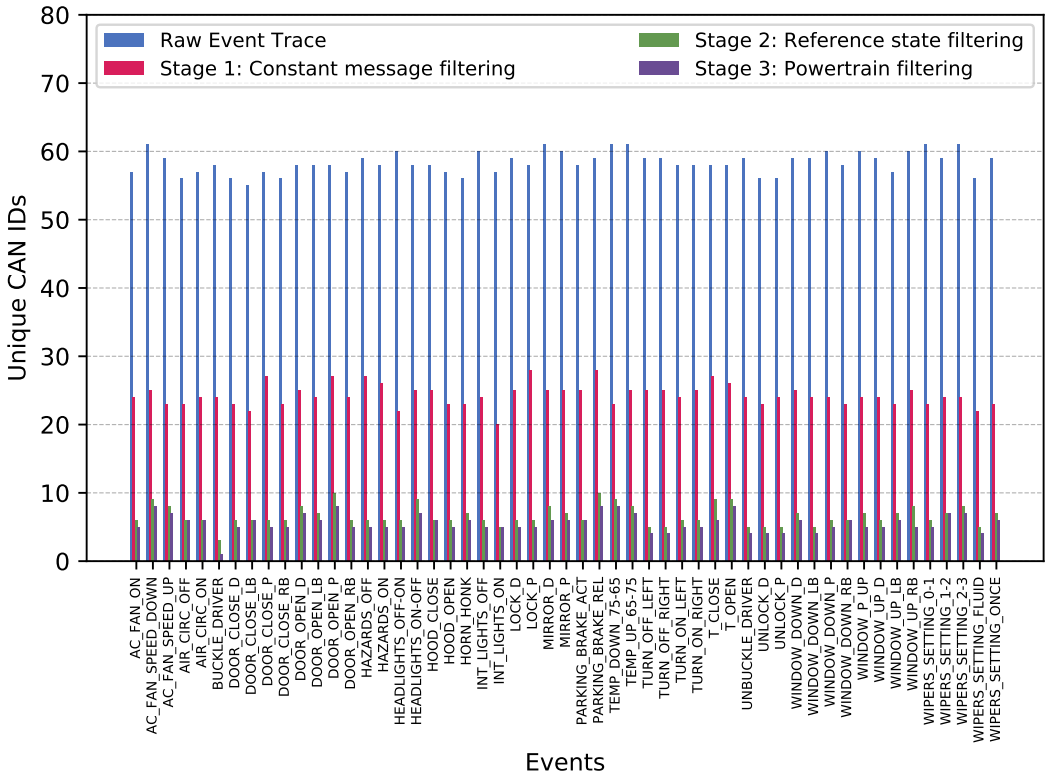


Figure A.2: Number of Unique CAN IDs Remaining After Each Stage for all 53 Events for Vehicle B

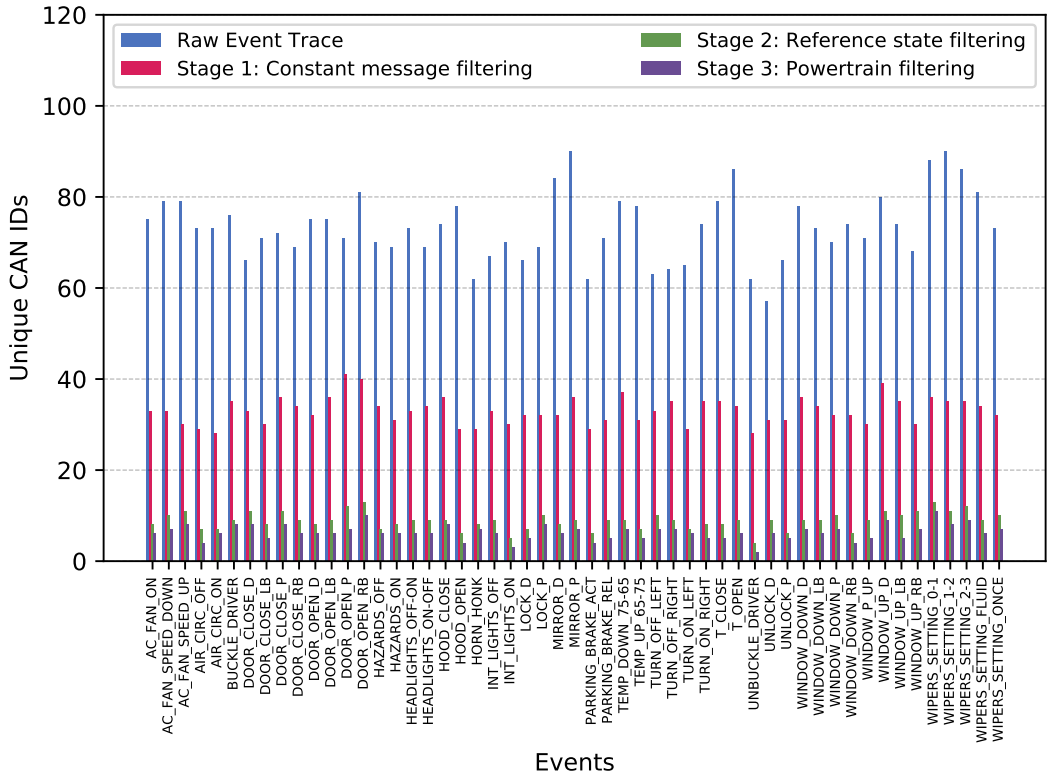


Figure A.3: Number of Unique CAN IDs Remaining After Each Stage for all 53 Events for Vehicle C

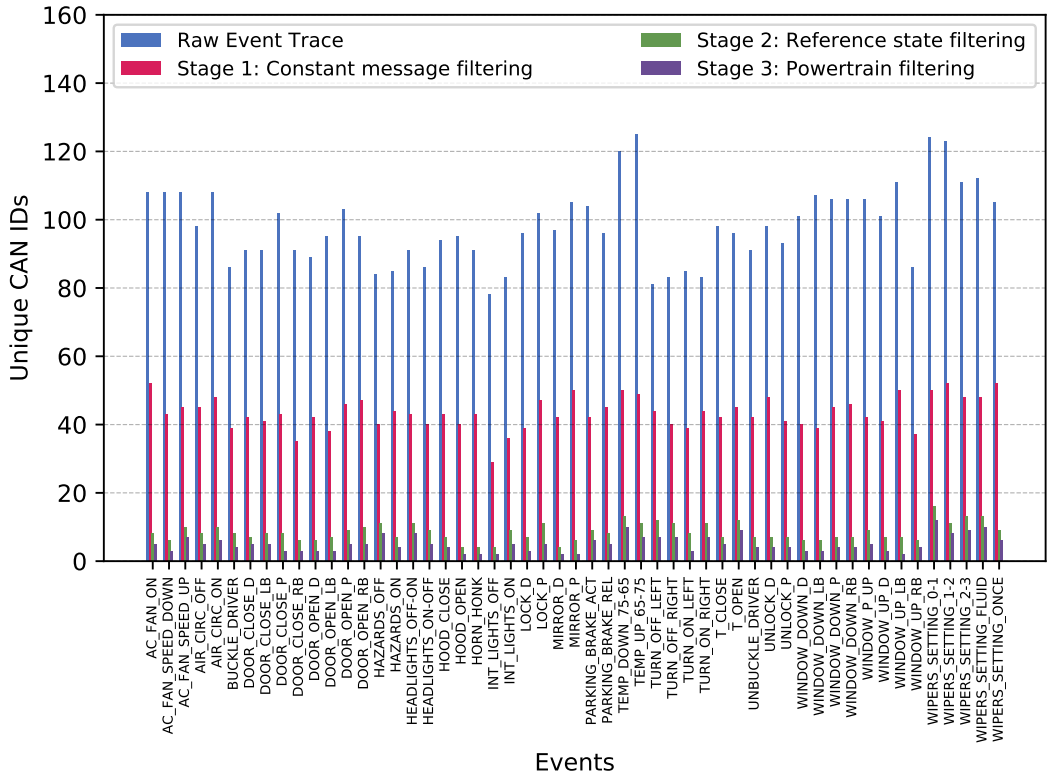


Figure A.4: Number of Unique CAN IDs Remaining After Each Stage for all 53 Events for Vehicle D

APPENDIX B

S2-CAN: Sufficiently Secure Controller Area Network

B.1 S2-CAN

Table B.1: Top 2 Cracking Success based on Trace Length (in %)

Trace Length		**5**	**10**	**25**	**50**	**75**	**100**
Veh. A	ST1	14/20	9/10	4/4	3/3	2/2	1/1
	ID	13/20	8/10	4/4	3/3	2/2	1/1
	cnt	14/20	9/10	4/4	3/3	2/2	1/1
Veh. B	ST1	12/20	4/10	3/4	2/3	1/2	1/1
	ID	11/20	3/10	3/4	1/3	1/2	1/1
	cnt	12/20	4/10	3/4	2/3	1/2	1/1
Veh. C	ST1	8/20	7/10	3/4	3/3	2/2	1/1
	ID	8/20	7/10	3/4	3/3	2/2	1/1
	cnt	8/20	7/10	3/4	3/3	2/2	1/1
Veh. D	f	12/20	7/10	3/4	3/3	2/2	1/1
	ID	11/20	5/10	3/4	3/3	2/2	1/1
	cnt	12/20	7/10	3/4	3/3	2/2	1/1

Table B.2: Top 3 Cracking Success based on Trace Length (in %)

Trace Length		5	10	25	50	75	100
Veh. A	ST1	14/20	9/10	4/4	3/3	2/2	1/1
	ID	13/20	8/10	4/4	3/3	2/2	1/1
	cnt	14/20	9/10	4/4	3/3	2/2	1/1
Veh. B	ST1	12/20	4/10	4/4	2/3	1/2	1/1
	ID	11/20	3/10	4/4	1/3	1/2	1/1
	cnt	12/20	4/10	4/4	2/3	1/2	1/1
Veh. C	ST1	8/20	7/10	3/4	3/3	2/2	1/1
	ID	8/20	7/10	3/4	3/3	2/2	1/1
	cnt	8/20	7/10	3/4	3/3	2/2	1/1
Veh. D	ST1	12/20	7/10	3/4	3/3	2/2	1/1
	ID	11/20	5/10	3/4	3/3	2/2	1/1
	cnt	12/20	7/10	3/4	3/3	2/2	1/1

Table B.3: Top 5 Cracking Success based on Trace Length (in %)

Trace Length		5	10	25	50	75	100
Veh. A	ST1	14/20	9/10	4/4	3/3	2/2	1/1
	ID	13/20	8/10	4/4	3/3	2/2	1/1
	cnt	14/20	9/10	4/4	3/3	2/2	1/1
Veh. B	ST1	13/20	5/10	4/4	2/3	1/2	1/1
	ID	12/20	4/10	4/4	1/3	1/2	1/1
	cnt	13/20	5/10	4/4	2/3	1/2	1/1
Veh. C	ST1	8/20	7/10	3/4	3/3	2/2	1/1
	ID	8/20	7/10	3/4	3/3	2/2	1/1
	cnt	8/20	7/10	3/4	3/3	2/2	1/1
Veh. D	ST1	14/20	8/10	3/4	3/3	2/2	1/1
	ID	13/20	6/10	3/4	3/3	2/2	1/1
	cnt	14/20	8/10	3/4	3/3	2/2	1/1

Table B.4: Top 10 Cracking Success based on Trace Length (in %)

Trace Length		**5**	**10**	**25**	**50**	**75**	**100**
Veh. A	ST1	15/20	10/10	4/4	3/3	2/2	1/1
	ID	14/20	9/10	4/4	3/3	2/2	1/1
	cnt	15/20	10/10	4/4	3/3	2/2	1/1
Veh. B	ST1	13/20	5/10	4/4	2/3	1/2	1/1
	ID	12/20	4/10	4/4	1/3	1/2	1/1
	cnt	13/20	5/10	4/4	2/3	1/2	1/1
Veh. C	ST1	8/20	7/10	3/4	3/3	2/2	1/1
	ID	8/20	7/10	3/4	3/3	2/2	1/1
	cnt	8/20	7/10	3/4	3/3	2/2	1/1
Veh. D	ST1	16/20	9/10	3/4	3/3	2/2	1/1
	ID	15/20	7/10	3/4	3/3	2/2	1/1
	cnt	16/20	9/10	3/4	3/3	2/2	1/1

APPENDIX C

MichiCAN: Practical Spoofing and DoS Protection for the Controller Area Network

C.1 MichiCAN

Algorithm 7 Find Globally Malicious Bits

Require: $\mathbb{E}$ ▷ List of CAN IDs used

```
1: function GLOBALLY_MALICIOUS(𝔼)
2:     GM_bits ← {}
3:     is_0 ← []
4:     is_1 ← []
5:     for i ← 0 to 11 do
6:         for j ← 0 to (|𝔼|) do
7:             ecu ← 𝔼[j]
8:             if ecu[i] == "0" then
9:                 is_0[i] ← 1
10:            else
11:                is_1[i] ← 1
12:        if is_0[i] == 0 then
13:            GM_bits[i] ← "0"
14:        if is_1[i] == 0 then
15:            GM_bits[i] ← "1"
16:    return GM_bits
```

Algorithm 8 Identify Malicious Outliers

```
1: function HAS_GM_BITS(n)
2:     for j ← 0 to 11 do
3:         if j in GM_bits then
4:             if GM_bits[j] == n[j] then
5:                 return True
6:     return False
7: OUTLIERS ← []
8: for i ← 0 to 2^11 − 1 do
9:     if i not in 𝔼 then
10:        if is_malicious(i) == False then
11:            AddItem(OUTLIERS, i)
```

Algorithm 9 Generate local prefixes

```
1: function GENERATE_LOCAL_PREFIX(outliers)
2:     diffs ← []
3:     for i ← 0 to |outliers|) do
4:         outlier ← outliers[i]
5:         smallest_diff ← 2048
6:         for j ← 0 to |𝔼| do
7:             ecu ← 𝔼 [j]
8:             diff ← outlier XOR ecu
9:             if diff < smallest_diff then
10:                smallest_diff ← diff
11:        diff_len ← 11−(log2(smallest_diff))
12:        AddItem(diffs, outliers[i][0:diff_len])
13:    differences ← diffs sorted in ascending length
14:    return remove_duplicates(differences)
```

Algorithm 10 Generate FSM Code for ECU_i

```
len ← 0
state ← 0
start_counterattack ← False
function FSM(value)
    if value == 1 then
        state |= (1 << (10 − len))
    len ← len + 1
    if len == 11 and state == ECU_i then
        start_counterattack ← True
        return True
    for key, val in GM_bits do
        if len == key + 1 and value == val then
            start_counterattack ← True
            return True
    for i ← 0 to |local_prefixes)| do
        p ← local_prefixes[i]
        pad ← p + "0" ∗ (11 − length(p))
        decimal_p ← pad converted to a decimal number
        if len == length(p) and state == decimal_p then
            start_counterattack ← True
            return True
    return False
```

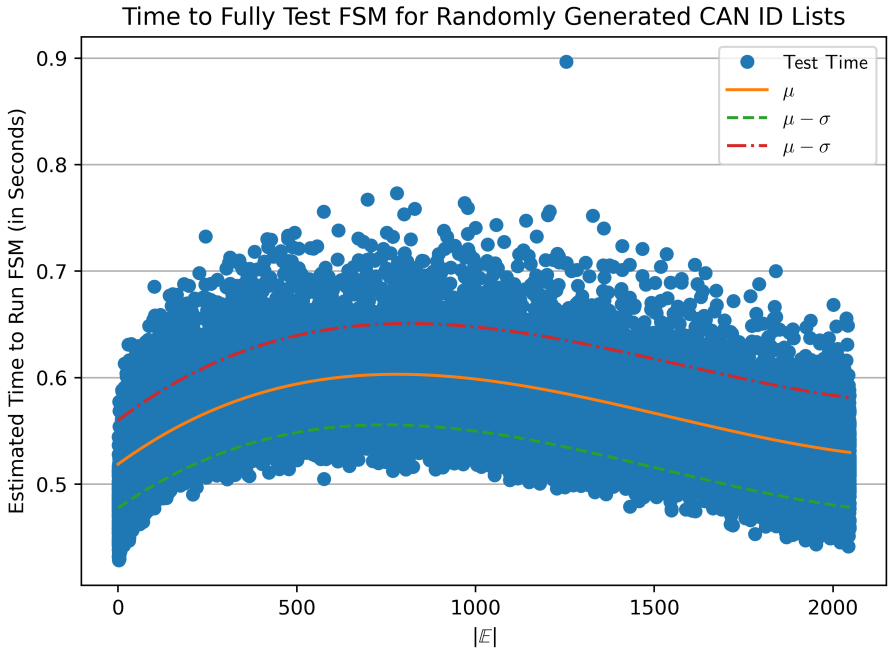


Figure C.1: Testing Time for Varying $|\mathbb{E}|$

```
1  // Generated for ECU_4 with CAN ID: 0x150
2  void state_machine_run(uint8_t value) {
3
4    bitWrite(state, 10 - len, value);
5    len++;
6
7    if (state > 336) {
8      return;
9    }
10   if (len == 11 && state == 336) {
11     start_counterattack = true;
12     return;
13   }
14   if (len == 1 && value == 1) {
15     start_counterattack = true;
16     return;
17   }
18   if (len == 2 && value == 1) {
19     start_counterattack = true;
20     return;
21   }
22   if (len == 3 && value == 0) {
23     start_counterattack = true;
24     return;
25   }
26   if (len == 4 && value == 1) {
27     start_counterattack = true;
28     return;
29   }
30   if (len == 6 && value == 1) {
31     start_counterattack = true;
32     return;
```

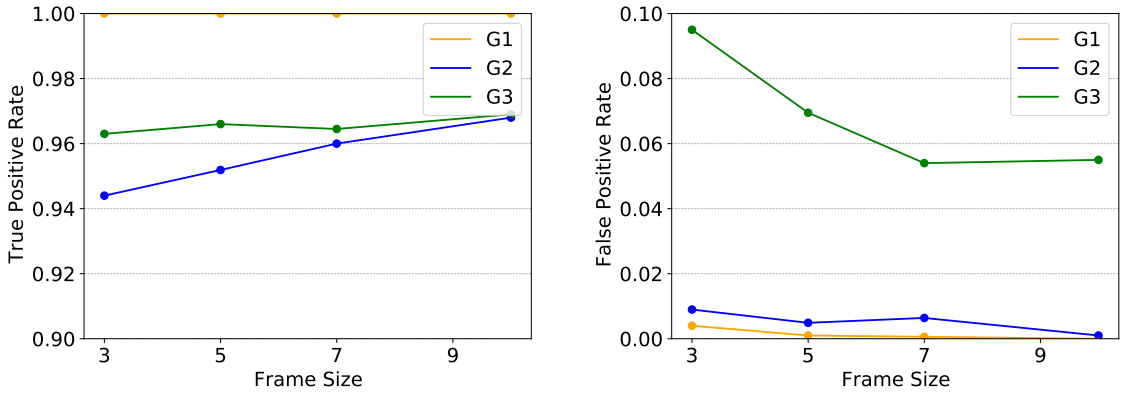
```
33    }
34    if (len == 8 && value == 1) {
35      start_counterattack = true;
36      return;
37    }
38    if (len == 9 && value == 1) {
39      start_counterattack = true;
40      return;
41    }
42    if (len == 10 && value == 1) {
43      start_counterattack = true;
44      return;
45    }
46    if (len == 7 && state == 320) {
47      start_counterattack = true;
48      return;
49    }
50    if (len == 11 && state == 273) {
51      start_counterattack = true;
52      return;
53    }
54    if (len == 11 && state == 337) {
55      start_counterattack = true;
56      return;
57    }
58    return;
59  }
60  // End generated code
```

Listing C.1: FSM code generated for ECU_4

APPENDIX D

CARdea: Practical Anomaly Detection for Connected and Automated Vehicles

D.1 CARdea



(a) TPR (b) FPR

Figure D.1: Phase 1 per-frame performance based on frame size

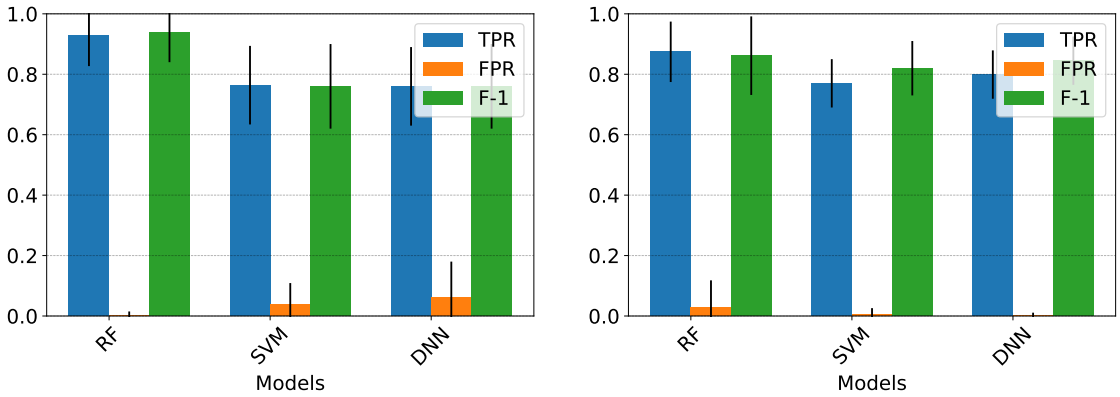
The extracted features from Sec. 5.6.2 are listed as follows:

Rolling Mean $\overline{X_t^i}$ is extracted with a window size of 5 for the measurement of interest;

Rolling Mean Displacement Difference between extracted $\overline{X_t^i}$ and the received measurement of interest at X_t^i;

Estimated Displacement Difference between estimated measurement of interest $\tilde{X}_t^i$ using the equations mentioned in Sec. 5.5 and the received measurement of interest;

Plausibility Check Difference between the current message's measurement X_t^i and the respective measurement of a previous message received from that vehicle X_{t-w}^i, for some constant window size w.



(a) AT5 (b) AT6

Figure D.2: Phase 2 AT5 and AT6 per-sample performance

Table D.1: G1 Phase 2 results on frames from Phase 1

G1 AT*	**G1 AT1**	**G2 AT2**	**G1 AT3**	**G1 AT4**
TPR (%)	100.0 (0.0)	99.75 (3.5)	99.67 (3.5)	99.51 (4.5)
FPR (%)	0.0 (0.0)	0.015 (0.4)	0.0 (0.0)	0.0 (0.0)

Table D.2: G2 Phase 2 results on frames from Phase 1

G2 AT*	**G2 AT1**	**G2 AT2**	**G2 AT3**	**G2 AT4**
TPR (%)	100.0 (0.0)	85.8 (23.3)	99.06 (6.2)	76.5 (26.8)
FPR (%)	0.0 (0.0)	1.3 (5.4)	0.16 (2.2)	1.9 (5.1)

Table D.3: G3 Phase 2 results on frames from Phase 1

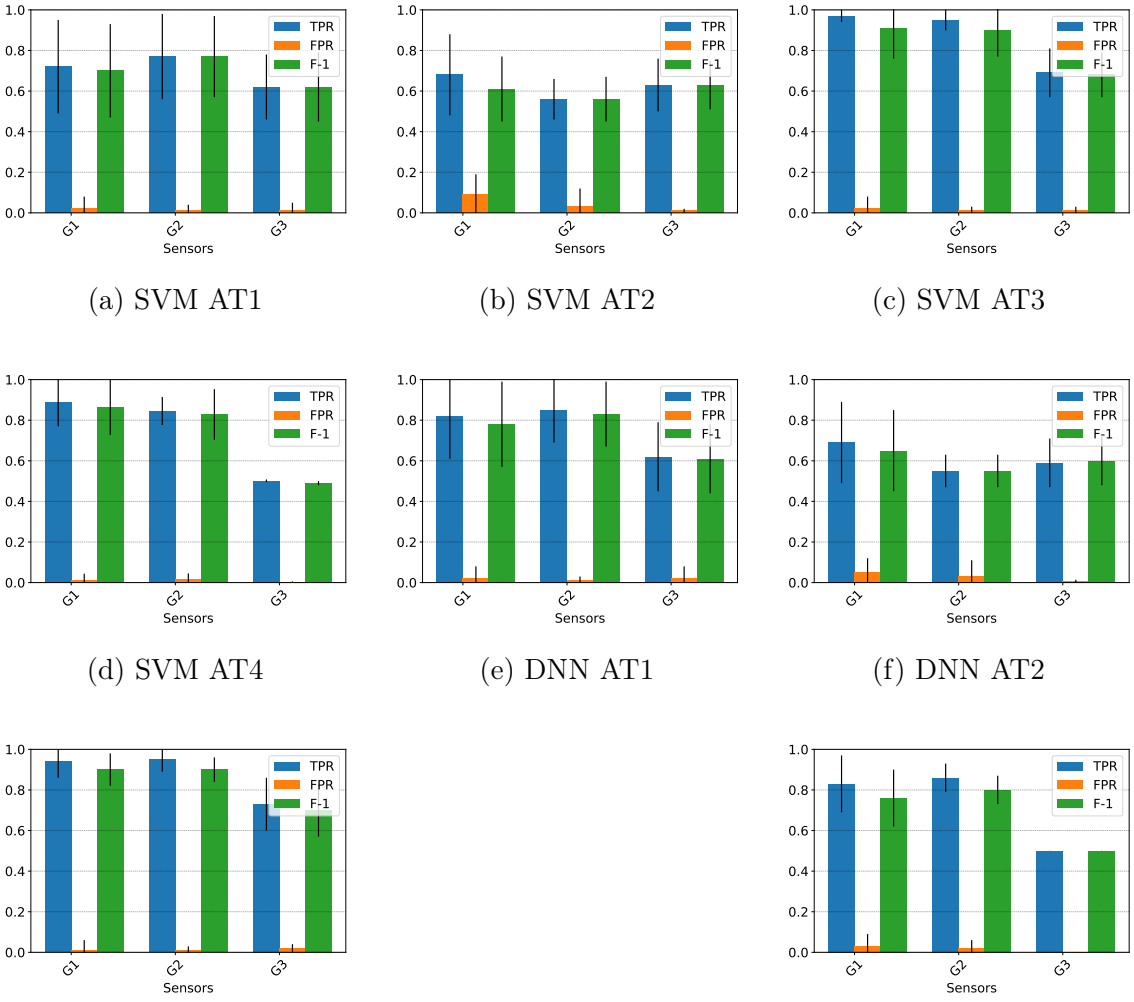
G3 AT*	**G3 AT1**	**G3 AT2**	**G3 AT3**	**G3 AT4**
TPR (%)	98.64 (9.4)	98.37 (9.0)	95.9 (14.9)	91.0 (21.3)
FPR (%)	0.09 (0.99)	0.3 (2.2)	1.06 (5.2)	1.8 (6.1)

Table D.4: Phase 2 latency (in ms) and RAM usage (in MB)

Model	**Frame Size**	**Ubuntu**		**RPi**	
		Latency	**RAM**	**Latency**	**RAM**
RF	3	15.1 (3.1)	153	98.9 (27.3)	106
	5	15.2 (1.7)	153	95.1 (21.4)	106
	7	15.6 (1.6)	153	99.4 (24.7)	106
	10	16.0 (2.2)	153	101.9 (25.0)	106
SVM	3	3.8 (0.4)	147	30.1 (9.2)	102
	5	4.2 (0.5)	147	31.9 (10.4)	102
	7	4.6 (0.5)	147	33.5 (11.6)	102
	10	4.9 (0.7)	147	38.8 (13.5)	102
DNN	3	41.5 (6.3)	485	570 (204)	357
	5	39.5 (5.8)	485	532 (99.6)	357
	7	39.9 (9.3)	485	528 (67.3)	357
	10	40.4 (8.8)	485	538 (119)	357

Table D.5: Bandwidth considerations

Sensor Group G_i	**Attack Type** AT^*	**Mean Perc. (%)**	**Bandwidth (kB/s)**
G_1	AT_1	5.3	0.17
	AT_2	5.2	0.17
	AT_3	5.2	0.17
	AT_4	5.3	0.17
	AT_5	8.7	0.28
	AT_6	7.6	0.24
G_2	AT_1	3.9	0.12
	AT_2	4.2	0.13
	AT_3	4.5	0.14
	AT_4	4.7	0.15
	AT_5	3.0	0.09
	AT_6	2.7	0.08
G_3	AT_1	14.5	0.46
	AT_2	14.0	0.45
	AT_3	15.0	0.48
	AT_4	15.6	0.50
	AT_5	17.9	0.56
	AT_6	17.8	0.56

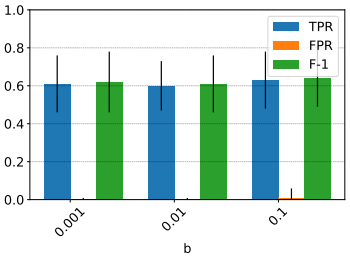


(a) SVM AT1 (b) SVM AT2 (c) SVM AT3

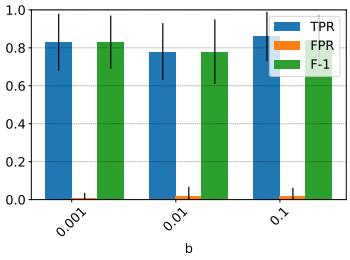
(d) SVM AT4 (e) DNN AT1 (f) DNN AT2

(g) DNN AT3 (h) DNN AT4

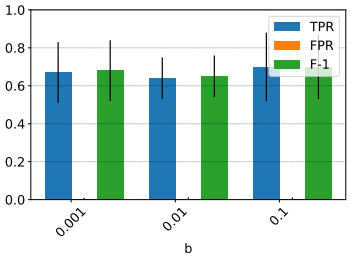
Figure D.3: SVM and DNN AT1 – AT4 per-sample performance



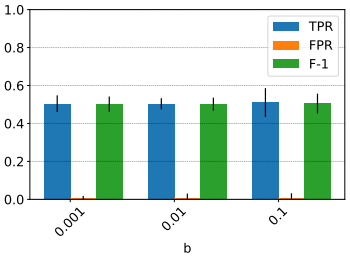
(a) RF G1



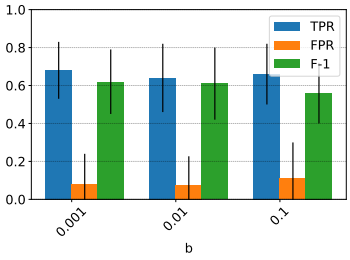
(b) RF G2



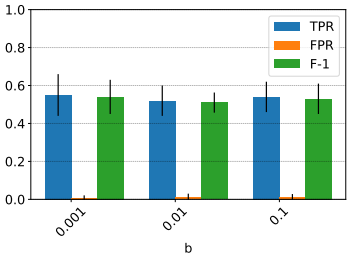
(c) RF G3



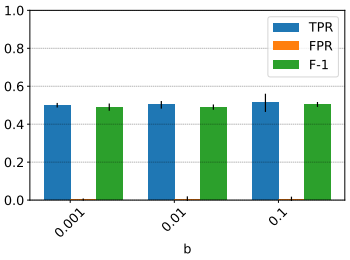
(d) SVM G1



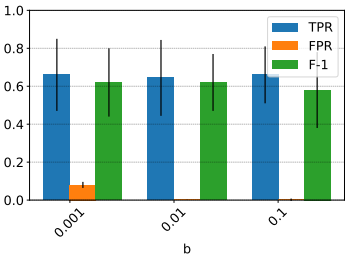
(e) SVM G2



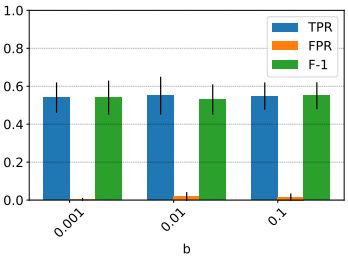
(f) SVM G3



(g) DNN G1



(h) DNN G2



(i) DNN G3

Figure D.4: Phase 2 AT7 per-sample performance

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Autosar xml schema. https://automotive.wiki/index.php/AUTOSAR_XML_Schema, 2014.

[2] The openxc platform. http://openxcplatform.com, 2017.

[3] A brief history of car hacking 2010 to the present. https://smart.gi-de.com/2017/08/brief-history-car-hacking-2010-present/, Nov 2018.

[4] Drive safe & save™ – state farm®. https://www.statefarm.com/insurance/auto/discounts/drive-safe-save, Aug 2018.

[5] How much data does spotify use? - probably less than you think. https://www.androidauthority.com/spotify-data-usage-918265/, Nov 2018.

[6] Sharpening the focus on obd-ii security. https://www.sae.org/news/2017/02/sharpening-the-focus-on-obd-ii-security, Jan 2018.

[7] Steps carmakers need to make to secure connected car data. https://internetofthingsagenda.techtarget.com/blog/IoT-Agenda/Steps-carmakers-need-to-make-to-secure-connected-car-data, Nov 2018.

[8] Barometric formula. https://www.math24.net/barometric-formula/, 2019.

[9] Obd-ii pids. https://en.wikipedia.org/wiki/OBD-II_PIDs, Feb 2019.

[10] On-board diagnostics. https://en.wikipedia.org/wiki/On-board_diagnostics#OBD-II_diagnostic_connector, Mar 2019.

[11] Amazon ec2, 2020. https://aws.amazon.com/ec2/.

[12] Automotive safety integrity level. https://en.wikipedia.org/wiki/Automotive_Safety_Integrity_Level, Dec 2020.

[13] Aws pricing, 2020. https://calculator.aws/.

[14] Can bus load calculation, 2020. https://kb.vector.com/entry/1519/.

[15] Can bus load calculator, 2020. http://www.canbusacademy.com/resources/can-bus-load-calculator/.

[16] Electronic engine control unit for commercial vehicles, 2020. https://www.bosch-mobility-solutions.com/en/products-and-services/commercial-vehicles/powertrain-systems/natural-gas/electronic-engine-control-unit/.

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

[18] One-time pad. https://en.wikipedia.org/wiki/One-time_pad, Nov 2020.

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

[20] Un regulations on cybersecurity and software updates to pave the way for mass roll out of connected vehicles. https://unece.org/sustainable-development/press/un-regulations-cybersecurity-and-software-updates-pave-way-mass-roll, Jun 2020.

[21] Android automotive. https://en.wikipedia.org/wiki/Android_Automotive#Vehicles_with_Android_Automotive, Oct 2021.

[22] Angle-of-arrival, 2021. https://www.sciencedirect.com/topics/engineering/angle-of-arrival, journal=Angle-of-Arrival - an overview | ScienceDirect Topics.

[23] Automated vehicles for safety. https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety, Nov 2021.

[24] Automotive electronics. https://en.wikipedia.org/wiki/Automotive_electronics, Aug 2021.

[25] Bmw group is showing automated valet parking for the first time at the iaa mobility 2021. https://www.press.bmwgroup.com/global/article/detail/T0342473EN/bmw-group-is-showing-automated-valet-parking-for-the-first-time-at-the-iaa-mobility-2021?language=en, 2021.

[26] The fundamentals of restbus simulation. https://www.ni.com/en-us/innovations/white-papers/12/the-fundamentals-of-restbus-simulation.html, Aug 2021.

[27] Android for cars overview. https://developer.android.com/training/cars, journal=Android Developers, 2022.

[28] Atmel sam3x / sam3a series. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf, 2022.

[29] Automated vehicles for safety, 2022. https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety.

[30] Autosar specification of module secure onboard communication, 2022. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_SWS_SecureOnboardCommunication.pdf.

[31] Drivewise - allstate. https://www.allstate.com/drive-wise/drivewise-device.aspx, 2022.

[32] Esp8266. https://www.espressif.com/en/products/socs/esp8266, 2022.

[33] Esurance insurance company. https://www.esurance.com/drivesense, 2022.

[34] Explore bmw connected drive technology. https://www.bmwusa.com/explore/connecteddrive.html, 2022.

[35] Google cloud, 2022. https://cloud.google.com/.

[36] Mcp2515. https://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf, 2022.

[37] Mcp2551. https://ww1.microchip.com/downloads/en/DeviceDoc/21667E.pdf, 2022.

[38] Mcp25625. https://ww1.microchip.com/downloads/en/DeviceDoc/MCP25625-CAN-Controller-Data-Sheet-20005282C.pdf, 2022.

[39] Pcan-usb. https://www.peak-system.com/PCAN-USB.199.0.html?&L=1, 2022.

[40] Power vs. torque. https://x-engineer.org/automotive-engineering/internal-combustion-engines/performance/power-vs-torque/, 2022.

[41] Rh850/e2m, 2022. https://www.renesas.com/us/en/products/microcontrollers-microprocessors/rh850/rh850e2x/rh850e2m.html.

[42] S32k144-q100 general purpose evaluation board. https://www.nxp.com/design/development-boards/automotive-development-platforms/s32k-mcu-platforms/s32k144-q100-general-purpose-evaluation-board:S32K144EVB, 2022.

[43] Sn65hvd230 can board. https://www.waveshare.com/sn65hvd230-can-board.htm, 2022.

[44] Tesla diagnostic port index. https://teslamotorsclub.com/tmc/threads/diagnostic-port-index.98663/, 2022.

[45] Tesla model s ecus. https://teslatap.com/undocumented/model-s-processors-count/, 2022.

[46] What is snapshot and how you can save. https://www.progressive.com/auto/discounts/snapshot/, 2022.

[47] Ahmed Abdo, Sakib Md Bin Malek, Zhiyun Qian, Qi Zhu, Matthew Barth, and Nael Abu-Ghazaleh. Application level attacks on connected vehicle protocols. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pages 459–471, 2019.

[48] Emad Aliwa, Omer Rana, Charith Perera, and Peter Burnap. Cyberattacks and countermeasures for in-vehicle networks. *arXiv preprint arXiv:2004.10781*, 2020.

[49] Sam Banani, Steven Gordon, Surapa Thiemjarus, and Somsak Kittipiyakul. Verifying safety messages using relative-time and zone priority in vehicular ad hoc networks. *Sensors*, 18(4):1195, 2018.

[50] Monowar H Bhuyan, Dhruba Kumar Bhattacharyya, and Jugal K Kalita. Network anomaly detection: methods, systems and tools. *Ieee communications surveys & tutorials*, 16(1):303–336, 2013.

[51] Norbert Bißmeyer, Sebastian Mauthofer, Kpatcha M Bayarou, and Frank Kargl. Assessment of node trustworthiness in vanets using data plausibility checks with particle filters. In *2012 IEEE Vehicular Networking Conference (VNC)*, pages 78–85. IEEE, 2012.

[52] Gedare Bloom. Weepingcan: A stealthy can bus-off attack. In *Workshop on Automotive and Autonomous Vehicle Security (AutoSec)*, volume 2021, page 25, 2021.

[53] Mehmet Bozdal, Mohammad Samie, Sohaib Aslam, and Ian Jennions. Evaluation of can bus security challenges. *Sensors*, 20(8):2364, 2020.

[54] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[55] Gabriel Brindusescu. Darpa hacked a chevy impala through its onstar system. https://www.autoevolution.com/news/darpa-hacked-a-chevy-impala-through-its-onstar-system-video-92194.html, February 2015.

[56] Richard R Brooks, Sam Sander, Juan Deng, and Joachim Taiber. Automobile security concerns. *IEEE Vehicular Technology Magazine*, 4(2):52–64, 2009.

[57] Ken Budd. How long do cars last? a guide to your car's longevity. https://www.aarp.org/auto/trends-lifestyle/info-2018/how-long-do-cars-last.html, Nov 2018.

[58] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security '11)*, pages 77–92. USENIX, August 2011.

[59] Dongyao Chen, Kyong-Tak Cho, Sihui Han, Zhizhuo Jin, and Kang G Shin. Invisible sensing of vehicle steering with smartphones. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 1–13. ACM, 2015.

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

[61] Kyong-Tak Cho and Kang G Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 911–927, 2016.

[62] Clemson Vehicular Electronic Laboratory. Clemson vehicular electronics laboratory: Automotive electronic systems, 2022.

[63] Lara Codeca, Raphaël Frank, and Thomas Engel. Luxembourg sumo traffic (lust) scenario: 24 hours of mobility for vehicular networking research. In *2015 IEEE Vehicular Networking Conference (VNC)*, pages 1–8. IEEE, 2015.

[64] A Costandoiu and M Leba. Convergence of v2x communication systems and next generation networks. In *IOP Conference Series: Materials Science and Engineering*, volume 477, page 012052. IOP Publishing, 2019.

[65] CSS Electronics. Can bus explained - a simple intro. `https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/language/en`, 2019.

[66] R Currie. Hacking the can bus: basic manipulation of a modern automobile through can bus reverse engineering. *SANS Institute*, 2017.

[67] Tsvika Dagan and Avishai Wool. Parrot, a software-only anti-spoofing defense system for the can bus. *ESCAR EUROPE*, page 34, 2016.

[68] Tomoiagă Radu Daniel and Stratulat Mircea. Aes algorithm adapted on gpu using cuda for small data and large data volume encryption. *International journal of applied mathematics and informatics*, 5(2):71–81, 2011.

[69] Pritam Dash, Mehdi Karimibiuki, and Karthik Pattabiraman. Out of control: stealthy attacks against robotic vehicles protected by control-based techniques. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 660–672, 2019.

[70] Pritam Dash, Guanpeng Li, Zitao Chen, Mehdi Karimibiuki, and Karthik Pattabiraman. Pid-piper: Recovering robotic vehicles from physical attacks. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38. IEEE, 2021.

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

[72] Stefan Dietzel, Rens van der Heijden, Hendrik Decke, and Frank Kargl. A flexible, subjective logic-based framework for misbehavior detection in v2v networks. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6. IEEE, 2014.

[73] Tri P Doan and Subramaniam Ganesan. Can crypto fpga chip to secure data transmitted through can fd bus using aes-128 and sha-1 algorithms with a symmetric key. Technical report, SAE Technical Paper, 2017.

[74] Ebroecker. ebroecker/canmatrix. `https://github.com/ebroecker/canmatrix/wiki/signal-Byteorder`, 2022.

[75] CSS Electronics. Can bus sniffer - reverse engineering vehicle data (wireshark). `https://www.csselectronics.com/screen/page/reverse-engineering-can-bus-messages-with-wireshark/language/en`, 2022.

[76] CSS Electronics. Can dbc file - convert data in real time (wireshark, j1939). `https://www.csselectronics.com/screen/page/dbc-database-can-bus-conversion-wireshark-j1939-example/language/en`, 2022.

[77] Elm Electronics, Inc. Obd. `https://www.elmelectronics.com/products/ics/obd/`, 2022.

[78] Equipment and Tool Institute. The case for a vehicle gateway. URL: `http://www.eti-home.org/TT-2015/Presos/ETI-ToolTech_2015_Gateway.pdf`, 2015.

[79] Jeremy Erickson, Shibo Chen, Melisa Savich, Shengtuo Hu, and Z Morley Mao. Commpact: Evaluating the feasibility of autonomous vehicle contracts. In *2018 IEEE Vehicular Networking Conference (VNC)*, pages 1–8. IEEE, 2018.

[80] TCITS ETSI. Intelligent transport systems (its); vehicular communications; basic set of applications; part 2: Specification of cooperative awareness basic service. *Draft ETSI TS*, 20(2011):448–51, 2011.

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

[82] Florian Fenzl, Roland Rieke, Yannick Chevalier, Andreas Dominik, and Igor Kotenko. Continuous fields: enhanced in-vehicle anomaly detection using machine learning models. *Simulation Modelling Practice and Theory*, 105:102143, 2020.

[83] Jürgen Frank. `https://www.nxp.com/docs/en/supporting-information/DWF13_AMF_AUT_T0112_Detroit.pdf`, Sep 2013.

[84] Arun Ganesan, Jayanthi Rao, and Kang Shin. Exploiting consistency among heterogeneous sensors for vehicle anomaly detection. Technical report, SAE Technical Paper, 2017.

[85] Andy Greenberg. Chrysler and harman hit with a class action complaint after jeep hack. `https://www.wired.com/2015/08/chrysler-harman-hit-class-action-complaint-jeep-hack/`, Aug 2015.

[86] Andy Greenberg. Hackers remotely kill a jeep on the highway—with me in it. `https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway`, 2015.

[87] Isabelle Guyon, Steve Gunn, Masoud Nikravesh, and Lofti A Zadeh. *Feature extraction: foundations and applications*, volume 207. Springer, 2008.

[88] Kyu Suk Han, Swapna Divya Potluri, and Kang G Shin. Real-time frame authentication using id anonymization in automotive networks, 2016. US Patent 9,288,048.

[89] Kyusuk Han, André Weimerskirch, and Kang G Shin. Automotive cybersecurity for in-vehicle communication. In *IQT QUARTERLY*, volume 6, pages 22–25, 2014.

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

[91] Adam Hanacek and Martin Sysel. Design and implementation of an integrated system with secure encrypted data transmission. In *Computer Science On-line Conference*, pages 217–224. Springer, 2016.

[92] Assaf Harel and Amir Hezberg. Optimizing can bus security with in-place cryptography. Technical report, SAE Technical Paper, 2019.

[93] M. Hasan, S. Mohan, T. Shimizu, and H. Lu. Securing vehicle-to-everything (v2x) communication platforms. *IEEE Transactions on Intelligent Vehicles*, 5(4):693–713, 2020.

[94] Olaf Henniger, Ludovic Apvrille, Andreas Fuchs, Yves Roudier, Alastair Ruddle, and Benjamin Weyl. Security requirements for automotive on-board networks. In *2009 9th International Conference on Intelligent Transport Systems Telecommunications,(ITST)*, pages 641–646. IEEE, 2009.

[95] Md Delwar Hossain, Hiroyuki Inoue, Hideya Ochiai, Doudou Fall, and Youki Kadobayashi. Lstm-based intrusion detection system for in-vehicle can bus communications. *IEEE Access*, 8:185489–185502, 2020.

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

[97] M Jukl and J Čupera. Using of tiny encryption algorithm in can-bus communication. *Research in Agricultural Engineering*, 62(2):50–55, 2016.

[98] Julietkilo. julietkilo/kcd. `https://github.com/julietkilo/kcd`, Jul 2017.

[99] Pallavi Kalyanasundaram, Venkatesh Kareti, Meghana Sambranikar, Narendra Kumar SS, and Priti Ranadive. Practical approaches for detecting dos attacks on can network. Technical report, SAE Technical Paper, 2018.

[100] Joseph Kamel, Mohammad Raashid Ansari, Jonathan Petit, Arnaud Kaiser, Ines Ben Jemaa, and Pascal Urien. Simulation framework for misbehavior detection in vehicular networks. *IEEE transactions on vehicular technology*, 69(6):6631–6643, 2020.

[101] Joseph Kamel, Michael Wolf, Rens Wouter van der Heijden, Arnaud Kaiser, Pascal Urien, and Frank Kargl. VeReMi Extension: A Dataset for Comparable Evaluation of Misbehavior Detection in VANETs. In *IEEE International Conference on Communications (ICC)*, Dublin (virtual), Ireland, June 2020. Virtual conference.

[102] Min-Joo Kang and Je-Won Kang. Intrusion detection system using deep neural network for in-vehicle network security. *PloS one*, 11(6):e0155781, 2016.

[103] Kalwinder Kaur. Accelerator pedal position sensors vs. throttle position sensors. `https://www.azosensors.com/article.aspx?ArticleID=51`, Mar 2019.

[104] John Kelsey, Bruce Schneier, and David Wagner. Related-key cryptanalysis of 3-way, biham-des, cast, des-x, newdes, rc2, and tea. In *International Conference on Information and Communications Security*, pages 233–246. Springer, 1997.

[105] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, 9:5–38, 1883.

[106] Hyogon Kim and Taeho Kim. Vehicle-to-vehicle (v2v) message content plausibility check for platoons through low-power beaconing. *Sensors*, 19(24):5493, 2019.

[107] Inhwan Kim, Hyunmi Yoo, Eom Young Hyun, Sungguk Cho, and Byungkook Jeon. An integrated communication message framework of inter-vehicles for connected vehicles using mobile virtual fence(mvf). *International Journal of Engineering and Technology(UAE)*, 7:102–105, 08 2018.

[108] Jochen Klaus-Wagenbrenner. Zonal ee architecture: Towards a fully automotive ethernet-based vehicle infrastructure. https://standards.ieee.org/content/dam/ieee-standards/standards/web/documents/other/eipatd-presentations/2019/D1-04_KLAUS-Zonal_EE_Architecture.pdf, Sep 2019.

[109] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447–462. IEEE, 2010.

[110] Sekar Kulandaivel, Shalabh Jain, Jorge Guajardo, and Vyas Sekar. Cannon: Reliable and stealthy remote shutdown attacks via unaltered automotive microcontrollers. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 195–210. IEEE, 2021.

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

[112] Philip Lapczynski. Achieving a root of trust with secure boot in automotive rh850 and r-car devices – part 1. https://www.renesas.com/us/en/blogs/introduction-about-secure-boot-automotive-mcu-rh850-and-soc-r-car-achieve-root-trust-1, 2021.

[113] Tim Leinmüller, Robert K Schmidt, and Albert Held. Cooperative position verification-defending against roadside attackers 2.0. In *Proceedings of 17th ITS World Congress*, pages 1–8. Citeseer, 2010.

[114] Xinhua Liu, Huafeng Mei, Huachang Lu, Hailan Kuang, and Xiaolin Ma. A vehicle steering recognition system based on low-cost smartphone sensors. *Sensors*, 17(3):633, 2017.

[115] Siti-Farhana Lokman, Abu Talib Othman, and Muhammad-Husaini Abu-Bakar. Intrusion detection system for automotive controller area network (can) bus system: a review. *EURASIP Journal on Wireless Communications and Networking*, 2019(1):1–17, 2019.

[116] Stefano Longari, Matteo Penco, Michele Carminati, and Stefano Zanero. Copycan: An error-handling protocol based intrusion detection system for controller area network. In *Proceedings of the ACM Workshop on Cyber-Physical Systems Security & Privacy*, pages 39–50, 2019.

[117] Feng Luo and Shuo Hou. Security mechanisms design of automotive gateway firewall. In *WCX SAE World Congress Experience*. SAE International, apr 2019.

[118] Feng Luo and Qiang Hu. Security mechanisms design for in-vehicle network gateway. In *WCX World Congress Experience*. SAE International, apr 2018.

[119] manitou48. Duezoo/isrperf.txt at master - manitou48/duezoo. `https://github.com/manitou48/DUEZoo/blob/master/isrperf.txt`, 2022.

[120] Mirco Marchetti and Dario Stabili. Read: Reverse engineering of automotive data frames. *IEEE Transactions on Information Forensics and Security*, 14(4):1083–1097, April 2019.

[121] Mirco Marchetti, Dario Stabili, and Michele Colajanni. Vehicle safe-mode, concept to practice limp-mode in the service of cybersecurity. *SAE International Journal of Transportation Cybersecurity and Privacy*, 3(1):19–39, feb 2020.

[122] Moti Markovitz and Avishai Wool. Field classification, modeling and anomaly detection in unknown can bus networks. *Vehicular Communications*, 9:43–52, 2017.

[123] Kirsten Matheus and Thomas Königseder. *Automotive ethernet*. Cambridge University Press, 2017.

[124] Charlie Miller and Chris Valasek. Adventures in automotive networks and control units. *Def Con*, 21:260–264, 2013.

[125] Charlie Miller and Chris Valasek. A survey of remote automotive attack surfaces. *black hat USA*, 2014:94, 2014.

[126] Charlie Miller and Chris Valasek. Car hacking: for poories. Technical report, Tech. rep., IOActive Report, 2015.

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

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

[129] Michael Müter and Naim Asaj. Entropy-based anomaly detection for in-vehicle networks. In *2011 IEEE Intelligent Vehicles Symposium (IV)*, pages 1110–1115. IEEE, 2011.

[130] D Nilsson and Ulf Larson. A roadmap for securing vehicles against cyber attacks. In *NITRD National Workshop on High-Confidence Automotive Cyber-Physical Systems*, 2008.

[131] William Noble. What is a support vector machine? *The journal of machine learning research*, 24:1565–1567, 2006.

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

[133] Shuji Ohira, Araya Kibrom Desta, Ismail Arai, Hiroyuki Inoue, and Kazutoshi Fujikawa. Normal and malicious sliding windows similarity analysis method for fast and accurate ids against dos attacks on in-vehicle networks. *IEEE Access*, 8:42422–42435, 2020.

[134] Jose Pagliery. Tesla car doors can be hacked. `https://money.cnn.com/2014/03/31/technology/security/tesla-hack/`, March 2014.

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

[136] A. Perrig, R. Canetti, J. Tygar, and D. Song. Approaches for secure and efficient in-vehicle key management. In *Proceedings of the IEEE Symposium on Security and Privacy (SP 2000)*, pages 56–73, 2000.

[137] Mert Pese, Kang Shin, Josiah Bruner, and Amy Chu. Security analysis of android automotive. *SAE International Journal of Advances and Current Practices in Mobility*, 2(2020-01-1295):2337–2346, 2020.

[138] Mert D. Pesé, Arun Ganesan, and Kang G. Shin. Carlab: Framework for vehicular data collection and processing. In *Proceedings of the 2Nd ACM International Workshop on Smart, Autonomous, and Connected Vehicular Systems and Services*, CarSys '17, pages 43–48, New York, NY, USA, 2017. ACM.

[139] Mert D Pesé, Jay W Schauer, Junhui Li, and Kang G Shin. S2-can: Sufficiently secure controller area network. *Annual Computer Security Applications Conference (ACSAC'21)*, 2021.

[140] Mert D Pesé, Karsten Schmidt, and Harald Zweck. Hardware/software codesign of an automotive embedded firewall. Technical report, SAE Technical Paper, 2017.

[141] Mert D Pesé, Troy Stacer, C Andrés Campos, Eric Newberry, Dongyao Chen, and Kang G Shin. Librecan: Automated can message translator. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2283–2300, 2019.

[142] PYMNTS. Who controls data in web-connected vehicles? `https://www.pymnts.com/innovation/2018/data-sharing-smart-cars-privacy/`, June 2018.

[143] Adnan Qayyum, Muhammad Usama, Junaid Qadir, and Ala Al-Fuqaha. Securing connected & autonomous vehicles: Challenges posed by adversarial machine learning and the way forward. *IEEE Communications Surveys & Tutorials*, 22(2):998–1026, 2020.

[144] Raul Quinonez, Jairo Giraldo, Luis Salazar, Erick Bauman, Alvaro Cardenas, and Zhiqiang Lin. {SAVIOR}: Securing autonomous vehicles with robust physical invariants. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 895–912, 2020.

[145] A.-I. Radu and F.D. Garcia. Leia: a lightweight authentication protocol for can. *Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016*, 878, 2016.

[146] Varsha Raghuwanshi and Simmi Jain. Denial of service attack in vanet: a survey. *International Journal of Engineering Trends and Technology (IJETT)*, 28(1):15–20, 2015.

[147] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[148] Maxim Raya, Panagiotis Papadimitratos, Imad Aad, Daniel Jungels, and Jean-Pierre Hubaux. Eviction of misbehaving and faulty nodes in vehicular networks. *IEEE Journal on Selected Areas in Communications*, 25(8):1557–1568, 2007.

[149] SAE. *Dedicated Short Range Communications (DSRC) Message Set Dictionary*, March 2016.

[150] SAE. *On-Board System Requirements for V2V Safety Communications*, March 2016.

[151] SAE. *Diagnostic Link Connector Security*, jun 2018.

[152] Fatih Sakiz and Sevil Sen. A survey of attacks and detection mechanisms on intelligent transportation systems: Vanets and iov. *Ad Hoc Networks*, 61:33–50, 2017.

[153] Steffen Sanwald, Liron Kaneti, Marc Stöttinger, and Martin Böhner. Secure boot revisited: challenges for secure implementations in the automotive domain. *17th Escar Europe: Embedded Security in Cars*, pages 113–127, 2020.

[154] Matthias Schäfer, Patrick Leu, Vincent Lenders, and Jens Schmitt. Secure motion verification using the doppler effect. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 135–145, 2016.

[155] S. Seifert and R. Obermaisser. Secure automotive gateway — secure communication for future cars. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 213–220, July 2014.

[156] Khaled Serag, Rohit Bhatia, Vireshwar Kumar, Z Berkay Celik, and Dongyan Xu. Exposing new vulnerabilities of error handling mechanism in {CAN}. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4241–4258, 2021.

[157] Alexandru Constantin Serban, Erik Poll, and Joost Visser. A security analysis of the etsi its vehicular communications. In *International Conference on Computer Safety, Reliability, and Security*, pages 365–373. Springer, 2018.

[158] Abhishek B Sharma, Leana Golubchik, and Ramesh Govindan. Sensor faults: Detection methods and prevalence in real-world datasets. *ACM Transactions on Sensor Networks (TOSN)*, 6(3):1–39, 2010.

[159] Junjie Shen, Jun Yeon Won, Zeyuan Chen, and Qi Alfred Chen. Drift with devil: Security of multi-sensor fusion based localization in high-level autonomous driving under {GPS} spoofing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 931–948, 2020.

[160] Ali Shuja Siddiqui, Yutian Gui, Jim Plusquellic, and Fareena Saqib. Secure communication over canbus. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1264–1267. IEEE, 2017.

[161] Craig Smith. *The car hacker's handbook: a guide for the penetration tester*. No Starch Press, 2016.

[162] Steven So, Jonathan Petit, and David Starobinski. Physical layer plausibility checks for misbehavior detection in v2x networks. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, pages 84–93, 2019.

[163] Steven So, Prinkle Sharma, and Jonathan Petit. Integrating plausibility checks and machine learning for misbehavior detection in vanet. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 564–571. IEEE, 2018.

[164] Christoph Sommer, David Eckhoff, Alexander Brummer, Dominik S Buse, Florian Hagenauer, Stefan Joerer, and Michele Segata. Veins: The open source vehicular network simulation framework. In *Recent Advances in Network Simulation*, pages 215–252. Springer, 2019.

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

[166] Dieter Spaar and Fabian A. Scherschel. Beemer, open thyself! – security vulnerabilities in bmw's connecteddrive. https://www.heise.de/ct/artikel/Beemer-Open-Thyself-Security-vulnerabilities-in-BMW-s-ConnectedDrive-2540957.html, February 2015.

[167] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[168] Stephen Stachowski, Ron Gaynier, David J LeBlanc, et al. An assessment method for automotive intrusion detection system performance. Technical report, United States. Department of Transportation. National Highway Traffic Safety, 2019.

[169] Jan Peter Stotz, Norbert Bißmeyer, Frank Kargl, Stefan Dietzel, Panos Papadimitratos, and Christian Schleiffer. Security requirements of vehicle security architecture. *Deliverable, PRESERVE consortium*, 2011.

[170] Takeshi Sugashima, Dennis Kengo Oka, and Camille Vuillaume. Approaches for secure and efficient in-vehicle key management. *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, 9(2016-01-0070):100–106, 2016.

[171] Beshr Sultan and Mike McDonald. Assessing the safety benefit of automatic collision avoidance systems (during emergency braking situations). In *Proceedings of the 18th International Technical Conference on the Enhanced Safety of Vehicle.(DOT HS 809 543)*, 2003.

[172] Mingshun Sun, Ming Li, and Ryan Gerdes. A data trust framework for vanets enabling false data detection and secure vehicle tracking. In *2017 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9. IEEE, 2017.

[173] Masaru Takada, Yuki Osada, and Masakatu Morii. Counter attack against the bus-off attack on can. In *2019 14th Asia Joint Conference on Information Security (AsiaJCIS)*, pages 96–102. IEEE, 2019.

[174] Tecsynt Solutions. How to reach the new business niche: Connected car app development approaches. https://medium.com/swlh/how-to-reach-the-new-business-niche-connected-car-app-development-approaches-7e4d3849b4fb, April 2018.

[175] A. S. Thangarajan, M. Ammar, B. Crispo, and D. Hughes. Towards bridging the gap between modern and legacy automotive ecus: A software-based security framework for legacy ecus. In *2019 IEEE 2nd Connected and Automated Vehicles Symposium (CAVS)*, pages 1–5, 2019.

[176] Nataša Trkulja, David Starobinski, and Randall A Berry. Denial-of-service attacks on c-v2x networks. *arXiv preprint arXiv:2010.13725*, 2020.

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

[178] Franco van Wyk, Yiyang Wang, Anahita Khojandi, and Neda Masoud. Real-time sensor anomaly detection and identification in automated vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 21(3):1264–1276, 2019.

[179] Miki E Verma, Robert A Bridges, and Samuel C Hollifield. Actt: Automotive can tokenization and translation. *arXiv preprint arXiv:1811.07897*, 2018.

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

[181] Xiaoyang Wang, Ioannis Mavromatis, Andrea Tassi, Raul Santos-Rodriguez, and Robert J Piechocki. Location anomalies detection for connected and autonomous vehicles. In *2019 IEEE 2nd Connected and Automated Vehicles Symposium (CAVS)*, pages 1–5. IEEE, 2019.

[182] Yiyang Wang, Neda Masoud, and Anahita Khojandi. Real-time sensor anomaly detection and recovery in connected automated vehicle sensors. *IEEE Transactions on Intelligent Transportation Systems*, 22(3):1411–1421, 2020.

[183] Armin R Wasicek, Mert D Pesé, André Weimerskirch, Yelizaveta Burakova, and Karan Singh. Context-aware intrusion detection in automotive control systems. In *5th ESCAR USA Conference, USA*, pages 21–22, 2017.

[184] Saheed Wasiu, Rashid Abdul Aziz, and Hanif Akmal. Effects of pressure boost on the performance characteristics of the direct injection spark ignition engine fuelled by gasoline at various throttle positions. *International Journal of Applied Engineering Research*, 13(1):691–696, 2018.

[185] Haohuang Wen, Qi Alfred Chen, and Zhiqiang Lin. Plug-n-pwned: Comprehensive vulnerability analysis of obd-ii dongles as a new over-the-air attack surface in automotive iot. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 949–965, 2020.

[186] Marko Wolf, André Weimerskirch, and Christof Paar. Security in automotive bus systems. In *Workshop on Embedded Security in Cars*, pages 1–13. Citeseer, 2004.

[187] David A. Wood. Jeep hacking lawsuit dismissed, Mar 2020.

[188] Wufei Wu, Renfa Li, Guoqi Xie, Jiyao An, Yang Bai, Jia Zhou, and Keqin Li. A survey of intrusion detection for in-vehicle networks. *IEEE Transactions on Intelligent Transportation Systems*, 2019.

[189] Zhihong Wu, Jianning Zhao, Yuan Zhu, Ke Lu, and Fenglue Shi. Research on in-vehicle key management system under upcoming vehicle network architecture. *Electronics*, 8(9):1026, 2019.

[190] Yuan Yao, Bin Xiao, Gaofei Wu, Xue Liu, Zhiwen Yu, Kailong Zhang, and Xingshe Zhou. Voiceprint: A novel sybil attack detection method based on rssi for vanets. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 591–602. IEEE, 2017.

[191] Yuan Yao, Bin Xiao, Gaofei Wu, Xue Liu, Zhiwen Yu, Kailong Zhang, and Xingshe Zhou. Multi-channel based sybil attack detection in vehicular ad hoc networks using rssi. *IEEE Transactions on Mobile Computing*, 18(2):362–375, 2018.

[192] Chaitanya Yavvari, Zoran Duric, and Duminda Wijesekera. Vehicular dynamics based plausibility checking. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–8. IEEE, 2017.

[193] Clinton Young, Joseph Zambreno, Habeeb Olufowobi, and Gedare Bloom. Survey of automotive controller area network intrusion detection systems. *IEEE Design & Test*, 36(6):48–55, 2019.

[194] Michael Ziehensack. Safe and secure communication with automotive ethernet, Oct 2015.

[195] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik*. Springer, 2006.

[196] Qingwu Zou, Wai Keung Chan, Kok Cheng Gui, Qi Chen, Klaus Scheibert, Laurent Heidt, and Eric Seow. The study of secure can communication for automotive applications. In *SAE Technical Paper*. SAE International, 03 2017.

[197] Baozhu Zuo. Can-bus shield v2.0, 2020. `https://wiki.seeedstudio.com/CAN-BUS_Shield_V2.0/`.