LibreCAN: Automated CAN Message Translator

Mert D. Pesé, Troy Stacer, C. Andrés Campos, Eric Newberry, Dongyao Chen, and Kang G. Shin
University of Michigan
Ann Arbor, MI, USA
{mpese,trstacer,andcmps,emnewber,chendy,kgshin}@umich.edu

ABSTRACT

Modern Connected and Autonomous Vehicles (CAVs) are equipped with an increasing number of Electronic Control Units (ECUs), many of which produce large amounts of data. Data is exchanged between ECUs via an in-vehicle network, with the Controller Area Network (CAN) bus being the *de facto* standard in contemporary vehicles. Furthermore, CAVs have not only physical interfaces but also increased data connectivity to the Internet via their Telematic Control Units (TCUs), enabling remote access via mobile devices. It is also possible to tap into, and read/write data from/to the CAN bus, as data transmitted on the CAN bus is not encrypted. This naturally generates concerns about automotive cybersecurity. One commonality among most vehicular security attacks reported to date is that they ultimately require write access to the CAN bus. In order to cause targeted and intentional changes in vehicle behavior, malicious CAN injection attacks require knowledge of the CAN message format. However, since this format is proprietary to OEMs and can differ even among different models of a single make of vehicle, one must manually reverse-engineer the CAN message format of each vehicle they target — a time-consuming and tedious process that does not scale. To mitigate this difficulty, we develop `LibreCAN`, which can translate most CAN messages with minimal effort. Our extensive evaluation on multiple vehicles demonstrates `LibreCAN`'s efficiency in terms of accuracy, coverage, required manual effort and scalability to any vehicle.

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security**.

KEYWORDS

Reverse Engineering; CAN Bus; Automotive Security

ACM Reference Format:
Mert D. Pesé, Troy Stacer, C. Andrés Campos, Eric Newberry, Dongyao Chen, and Kang G. Shin. 2019. LibreCAN: Automated CAN Message Translator. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3363190

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) [39]. 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 [46], data security and privacy are among the most critical drivers and inhibitors of next-generation mobility services. Automotive cybersecurity is a relatively young field, with the first major publications appearing in 2010 [16, 33]. In 2015, several attacks were reported, including three major wireless attacks: an attack on BMW Connected Drive [49], an attack on GM OnStar [15], and the Tesla Door Attack [43]. Although the first two attacks received some attention, it was not until Miller and Valasek's Jeep attack [42] 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) [18, 27, 30, 52],

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '19, November 11–15, 2019, London, United Kingdom

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3363190

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* [32]. 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 paper, 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 paper is organized as follows. Sec. 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. 3 details the design of LibreCAN, while Sec. 4 evaluates the accuracy, coverage, and required manual and computation time for reverse-engineering CAN messages. Sec. 5 discusses the limitations and potential other use-cases of LibreCAN, as well as possible countermeasures. Sec. 6 discusses related efforts in manual and automated CAN reverse-engineering, while Sec. 7 concludes the paper.

2 BACKGROUND

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 depicts the structure of a CAN 2.0A data frame — the most common data-frame type used on the CAN bus.

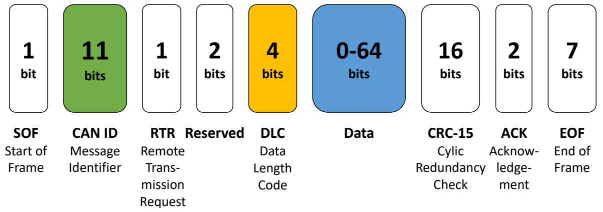


Figure 1: CAN data frame structure

Highlighted with non-white color in this figure are the three fields that are essential to the understanding of LibreCAN:

- **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 their source or destination ECUs, but instead each frame carries a unique message identifier (ID) that represents its meaning and priority. Lower CAN IDs have higher priority (e.g., powertrain- vs. body-related information) and will "win" the distributed arbitration process that occurs when multiple messages are sent on the CAN bus at the same time. 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, and thus allows for up to 2048 different CAN IDs.
- **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 ranging from 0 to 8 bytes.
- **Data:** This is the payload field of a CAN message containing the actual message data. It can contain 0–8 bytes of data depending on the value of the DLC field.

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 [37, 38], such as:

- **Constants:** Values that do not change over time.
- **Multi-Values:** Values with a domain consisting of only a few constant values. [38] 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. 2 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.

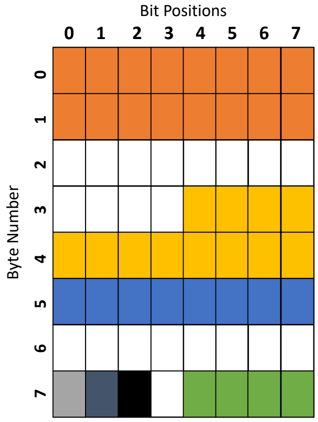


Figure 2: Example of CAN signals

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 [29]) and ARXML (AUTOSAR [1]) files. However, the most common format used for this purpose is DBC [24], a standard created by German automotive supplier company Vector Informatik.

DBC files contain a myriad of information. However, to understand this paper, 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 [22]. 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:

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

2.3 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. [19] provides an overview of the automotive electronic systems present in a typical vehicle. After analyzing multiple sources [40–42], we derived a list of ECUs (Table 8 in Appendix A) 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 [20]. 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:

$$d_s = m_s \cdot r_s + t_s. \tag{2}$$

2.4 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. 3.

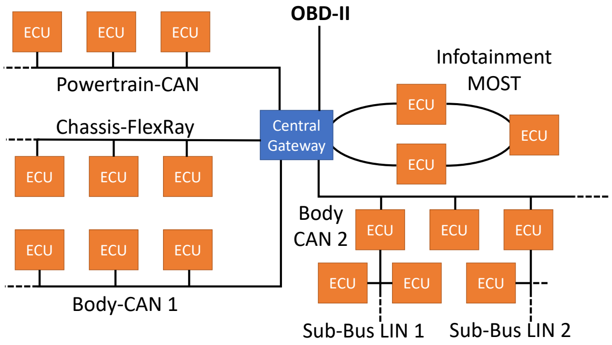


Figure 3: Common automotive E/E architecture (adapted from [54])

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 [13]. 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 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 [13]). 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 [3]. Furthermore, proprietary diagnostic protocols are used in EVs (instead of SAE J/1979).

OBD-II data cae n be accessed by anyone through aftermarket dongles [25]. 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 [13].

Note that the OBD-II protocol is public and does not make any use of DBC files at all. As stated in [13], 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). 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 [33] to erroneously steering the vehicle [40]. 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 justified as a security countermeasure, but the OBD-II connector has only 16 pins, with some pins already assigned [14], and thus only up to three CAN buses can be monitored through the OBD-II port.

3 SYSTEM DESIGN

Fig. 4 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 [13].
- 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.

Data from sets P and V are only used in Phase 1. As shown in Table 9, 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.

3.1 Phase 0: Signal Extraction

As described in Sec. 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 [37].

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 consecutive 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

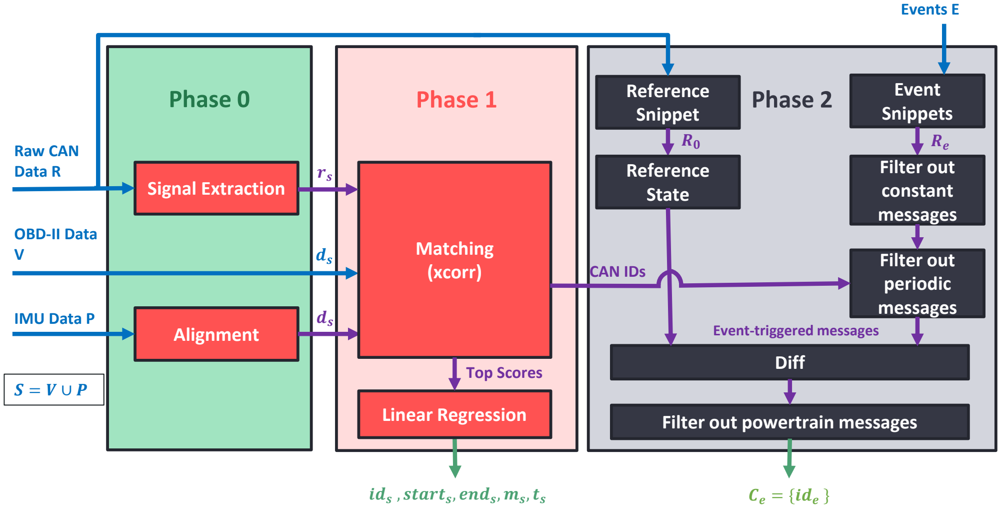


Figure 4: System design overview

a percentage decrease allowed us to extract more signals correctly than the original READ.

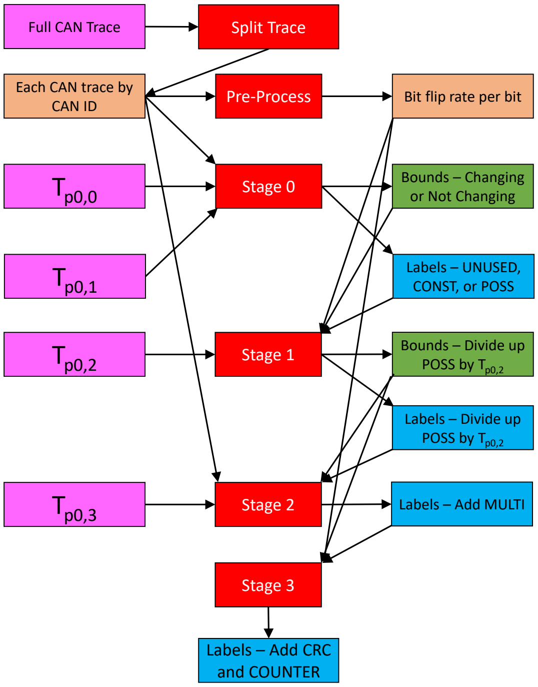


Figure 5: Flowchart of Phase 0 algorithm

A flowchart of the algorithm for this phase is provided in Fig. 5. 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 [37].

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, 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

Algorithm 1 Stage 0

```
procedure $stage0(trace\_file, T_{p0,0}, T_{p0,1})$
    $bits\_that\_dont\_change\_label \leftarrow []$
    for $l, r \in bits\_that\_dont\_change$ do
        if $True \in changes[l : r]$ then
            $bits\_that\_dont\_change\_label.append(\texttt{CONST})$
            break
        else if $r - l < T_{p0,0}$ then
            $reinserted \leftarrow false$
            for $l\_c, r\_c \in bits\_that\_change$ do
                if $l\_c == r + 1$and $r\_c - l\_c > T_{p0,1}$ then
                    $l\_c \leftarrow l$
                    $reinserted \leftarrow false$
                    delete $l, c$
                    break
            if $reinserted == false$ then
                $bits\_that\_dont\_change\_label.append(\texttt{UNUSED})$
```

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.

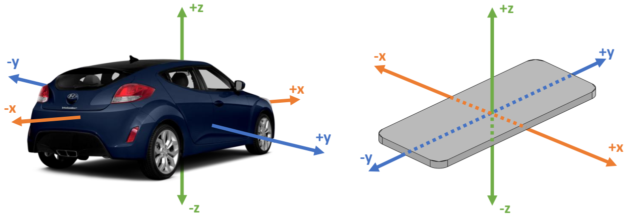


Figure 6: Alignment of phone's coordinate system (right) with vehicular coordinate system (left)

Alignment: Phase 0 also encompasses phone alignment. As Fig. 6 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 [17]. 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. 6, the phone should be located on the center console, with the short edge parallel to the direction of the vehicle's motion.

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. 4) — are depicted in Table 9. 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), 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. 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 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 1.

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 10.

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. (3), 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 $j \in [0, 63]$:

$$BFC_{n,j} = \sum_{i=0}^{|id_n|-1} 1, \forall j \in [0, 63] \text{ and if } m_{n,i,j} \neq m_{n,i-1,j}. \quad (3)$$

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

$$BFA_{n,j} = \frac{BFC_{n,j}}{|id_n|}. \quad (4)$$

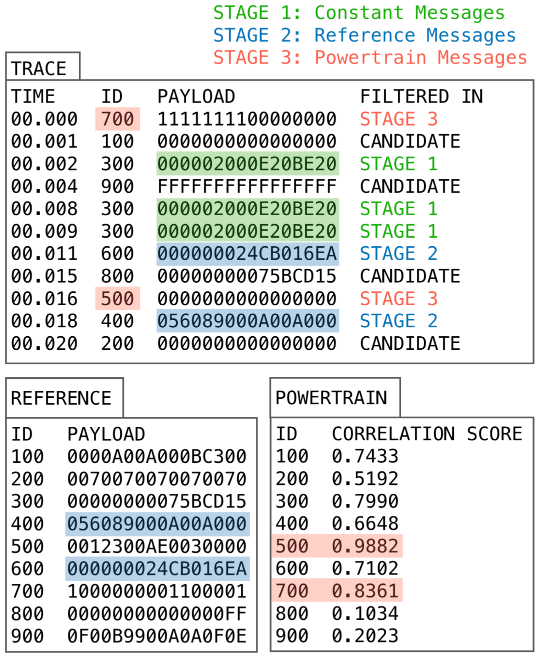
Finally, we define the bit-flip rate (BFR_n) of a CAN ID (id_n) as:

$$BFR_n = \frac{\sum_{j=0}^{63} BFA_{n,j}}{64}. \quad (5)$$

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.

Fig. 7 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: 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 7: Phase 2 Filtering Example

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. 7, 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. 7, 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

Table 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

a threshold ($T_{p2,3}$). The correlation scores for each CAN ID in the example in Fig. 7 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. 7, 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 1.

4 EVALUATION

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 9). For Phase 2, we solely used the OpenXC dongle to record raw CAN data.

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 3.1, we performed a brute-force search through all possible combinations as depicted in Table 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 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. 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: 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 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 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 1, we defined the following metrics to assess for Phase 1:

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

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. 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. 8 shows the PR curve for both vehicles. Each data point depicts a value of $T_{p1} \in [0, 1]$.

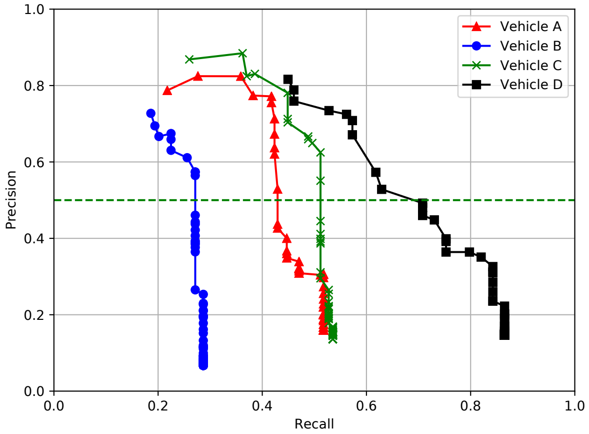


Figure 8: Precision-Recall Curve for Phase 1

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. 8 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 4 sums up the precision and recall values using the optimal threshold T_{p1} (see Table 3) obtained from the PR curve analysis. entering The precision and recall values reflect the evaluation of Phase 1 with correct bounds 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.

Table 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%

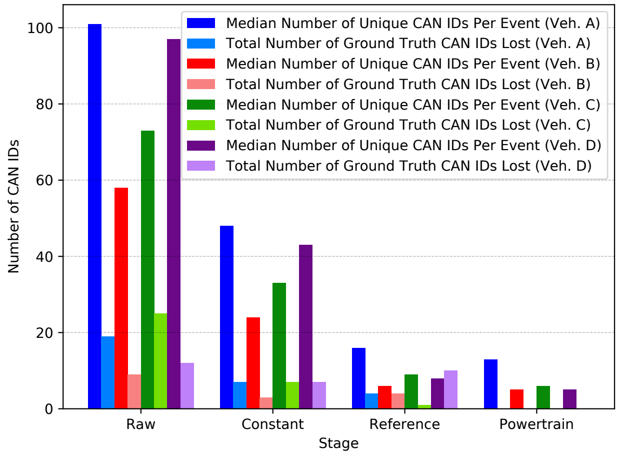


Figure 9: Filtering out CAN IDs in each stage

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 10. 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 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. 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 3.

The second part of Table 4 summarizes the mean values of our metrics for all 53 events while Fig. 9 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. 9 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.

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.

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. 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. 10.

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

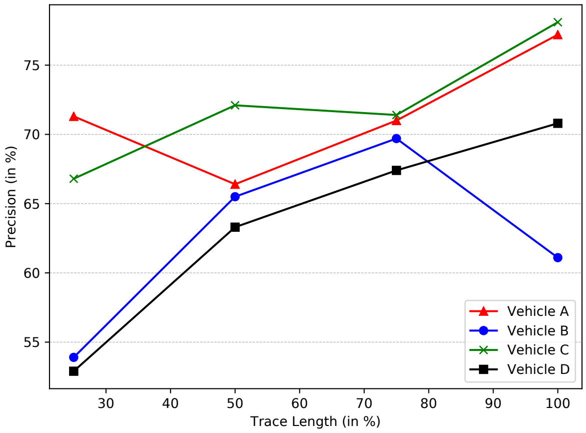


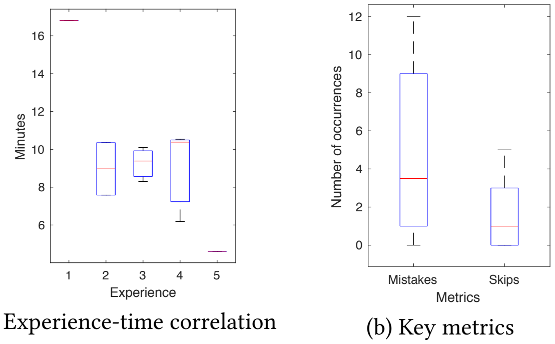
Figure 10: Precision of Phase 1 with varying trace lengths

(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 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 10, 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 [44]. 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 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. 11 (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. 11 (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 [48] imply significant effort is required. However, researchers from the well-known Jeep hack [42] 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.



(a) Experience-time correlation

(b) Key metrics

Figure 11: Results in user-study experiment

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 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 5.

The entire three phase automated process takes 79 seconds for Vehicle A, 74 seconds for Vehicle B, 70 seconds for Vehicle C and 72 seconds for Vehicle D. All vehicles have a similar computation time, indicating that LibreCANis 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).

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

Table 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
LibreCAN	Total	79	74	70	72

of the parameters, we would like to introduce an accuracy analysis similar to the one in Sec. 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 6. A comparison with the optimal results for each vehicle in Table 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.

5 DISCUSSION

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).

Table 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%

Second, for Phase 2, not all vehicles may have the 53 events defined in Table 10. 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). 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 [34].

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 LibreCANis 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 [50]. 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 [13]). In particular, usage-based insurance (UBI) companies [4, 5, 8, 11] 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.

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 [33, 40, 42]. 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. 3).

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 [12]. In the corresponding SAE standard [10], 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 [26], when coverage of car hacking by news outlets increased significantly [9]. [7] also suggests enhancing existing gateway designs by adding additional security measures, such as a firewall. The aforementioned SAE standard [10] 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 [36, 47]. 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 [28, 36] or via access control/firewalls [35, 45].

6 RELATED WORK

6.1 Manual CAN Reverse Engineering

[21] 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 [40, 42], 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, [23] 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.

6.2 Automating CAN Reverse-Engineering

[38] 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 [37] 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 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 [51] 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.

7 CONCLUSION

In this paper, 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.

Table 7: Comparison to Related Work

	LibreCAN			READ [37]			ACTT [51]		
	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%	-

ACKNOWLEDGMENTS

The work reported in this paper was supported in part by NSF under Grant CNS-1646130. Assistance from undergraduate researcher Alice C. Ying is also gratefully acknowledged.

REFERENCES

[1] [n.d.]. AUTOSAR XML Schema. https://automotive.wiki/index.php/AUTOSAR_XML_Schema

[2] [n.d.]. Barometric Formula. https://www.math24.net/barometric-formula/

[3] [n.d.]. Diagnostic Port Index. https://teslamotorsclub.com/tmc/threads/diagnostic-port-index.98663/

[4] [n.d.]. Drivewise - Allstate. https://www.allstate.com/drive-wise/drivewise-device.aspx

[5] [n.d.]. Esurance Insurance Company. https://www.esurance.com/drivesense

[6] [n.d.]. Power vs. Torque. https://x-engineer.org/automotive-engineering/internal-combustion-engines/performance/power-vs-torque/

[7] [n.d.]. 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

[8] [n.d.]. What is Snapshot and How You Can Save. https://www.progressive.com/auto/discounts/snapshot/

[9] 2018. A Brief History of Car Hacking 2010 to the Present. https://smart.gi-de.com/2017/08/brief-history-car-hacking-2010-present/

[10] . 2018. *Diagnostic Link Connector Security.* https://doi.org/10.4271/J3138_201806

[11] 2018. Drive Safe & Save™ – State Farm®. https://www.statefarm.com/insurance/auto/discounts/drive-safe-save

[12] 2018. Sharpening the focus on OBD-II security. https://www.sae.org/news/2017/02/sharpening-the-focus-on-obd-ii-security

[13] 2019. OBD-II PIDs. https://en.wikipedia.org/wiki/OBD-II_PIDs

[14] 2019. On-board diagnostics. https://en.wikipedia.org/wiki/On-board_diagnostics#OBD-II_diagnostic_connector

[15] Gabriel Brindusescu. 2015. 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

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

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

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

[19] Clemson Vehicular Electronic Laboratory. [n.d.]. Clemson Vehicular Electronics Laboratory: Automotive Electronic Systems. https://cecas.clemson.edu/cvel/auto/systems/auto-systems.html

[20] CSS Electronics. [n.d.]. CAN Bus Explained - A Simple Intro (2019). https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/language/en

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

[22] Ebroecker. [n.d.]. ebroecker/canmatrix. https://github.com/ebroecker/canmatrix/wiki/signal-Byteorder

[23] CSS Electronics. [n.d.]. CAN Bus Sniffer - Reverse Engineering Vehicle Data (Wireshark). https://www.csselectronics.com/screen/page/reverse-engineering-can-bus-messages-with-wireshark/language/en

[24] CSS Electronics. [n.d.]. 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

[25] Elm Electronics, Inc. [n.d.]. OBD. https://www.elmelectronics.com/products/ics/obd/

[26] Equipment and Tool Institute. 2015. The case for a Vehicle Gateway. URL: http://www.eti-home.org/TT-2015/Presos/ETI-ToolTech_2015_Gateway.pdf.

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

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

[29] Julietkilo. 2017. julietkilo/kcd. https://github.com/julietkilo/kcd

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

[31] Kalwinder Kaur. 2019. Accelerator Pedal Position Sensors vs. Throttle Position Sensors. https://www.azosensors.com/article.aspx?ArticleID=51

[32] Auguste Kerckhoffs. 1883. La cryptographie militaire. *Journal des sciences militaires* 9 (1883), 5–38.

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

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

[35] Feng Luo and Shuo Hou. 2019. Security Mechanisms Design of Automotive Gateway Firewall. In *WCX SAE World Congress Experience*. SAE International. https://doi.org/10.4271/2019-01-0481

[36] Feng Luo and Qiang Hu. 2018. Security Mechanisms Design for In-Vehicle Network Gateway. In *WCX World Congress Experience*. SAE International. https://doi.org/10.4271/2018-01-0018

[37] Mirco Marchetti and Dario Stabili. 2019. READ: Reverse Engineering of Automotive Data Frames. *IEEE Transactions on Information Forensics and Security* 14, 4 (April 2019), 1083–1097. https://doi.org/10.1109/TIFS.2018.2870826

[38] Moti Markovitz and Avishai Wool. 2017. Field classification, modeling and anomaly detection in unknown CAN bus networks. *Vehicular Communications* 9 (2017), 43–52.

[39] Kirsten Matheus and Thomas Königseder. 2017. *Automotive ethernet.* Cambridge University Press.

[40] Charlie Miller and Chris Valasek. 2014. Adventures in Automotive Networks and Control Units.

[41] Charlie Miller and Chris Valasek. 2014. A survey of remote automotive attack surfaces. *black hat USA* 2014 (2014), 94.

[42] Charlie Miller and Chris Valasek. 2015. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA* 2015 (2015), 91.

[43] Jose Pagliery. 2014. Tesla car doors can be hacked. https://money.cnn.com/2014/03/31/technology/security/tesla-hack/

[44] Mert D. Pesé, Arun Ganesan, and Kang G. Shin. 2017. 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)*. ACM, New York, NY, USA, 43–48. https://doi.org/10.1145/3131944.3133940

[45] Mert D. Pesé, Karsten Schmidt, and Harald Zweck. 2017. Hardware/Software Co-Design of an Automotive Embedded Firewall. In *WCX™ 17: SAE World Congress Experience*. SAE International. https://doi.org/10.4271/2017-01-1659

[46] PYMNTS. 2018. Who Controls Data In Web-Connected Vehicles? https://www.pymnts.com/innovation/2018/data-sharing-smart-cars-privacy/

[47] S. Seifert and R. Obermaisser. 2014. Secure automotive gateway — Secure communication for future cars. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. 213–220. https://doi.org/10.1109/INDIN.2014.6945510

[48] Craig Smith. 2016. *The car hacker's handbook: a guide for the penetration tester.* No Starch Press.

[49] Dieter Spaar and Fabian A. Scherschel. 2015. 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.

[50] Tecsynt Solutions. 2018. 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.

[51] Miki E Verma, Robert A Bridges, and Samuel C Hollifield. 2018. ACTT: Automotive CAN Tokenization and Translation. *arXiv preprint arXiv:1811.07897* (2018).

[52] Armin R Wasicek, Mert D Pesé, André Weimerskirch, Yelizaveta Burakova, and Karan Singh. 2017. Context-aware intrusion detection in automotive control systems. In *5th ESCAR USA Conference, USA*. 21–22.

[53] Saheed Wasiu, Rashid Abdul Aziz, and Hanif Akmal. 2018. 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 (2018), 691–696.

[54] Werner Zimmermann and Ralf Schmidgall. 2006. *Bussysteme in der Fahrzeugtechnik*. Springer.

A VEHICULAR SIGNALS

Table 8 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 [6]. 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:

$$\tau \underset{\sim}{\propto} rpm. \tag{6}$$

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

$$load_{engine} \propto \tau. \tag{7}$$

- For engine speed values up to approximately 2000 RPM, torque (τ) and pressure boost (p_{boost}) are linearly related [53]. 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}:

$$\tau \underset{\sim}{\propto} p_{boost} \underset{\sim}{\propto} p_{map}. \tag{8}$$

- The electrical circuitry in the Accelerator Pedal Position (APP) and Throttle Position (TPS) sensors is identical [31]. 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:

$$ACC\_PED \underset{\sim}{\propto} THR\_POS. \tag{9}$$

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

$$a_y \propto \omega_z v. \tag{10}$$

- 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) [2]. Via the *barometric formula*:

$$p \underset{\sim}{\propto} e^{-k \cdot h \cdot M}. \tag{11}$$

  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.

Table 8: 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

B PHASE 1

Table 9 depicts a complete list of all signals in set S that we are considering for correlation in Phase 1. Table 10 shows all 53 events that were analyzed for Phase 2.

Table 9: 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*

Table 10: 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

C PHASE 2

Fig. 12 depicts which CAN IDs have been filtered out at what stage for each of the 53 events for Vehicle A. Fig. 13, Fig. 14, and Fig. 15 are similar, but for Vehicles B, C, and D, respectively.

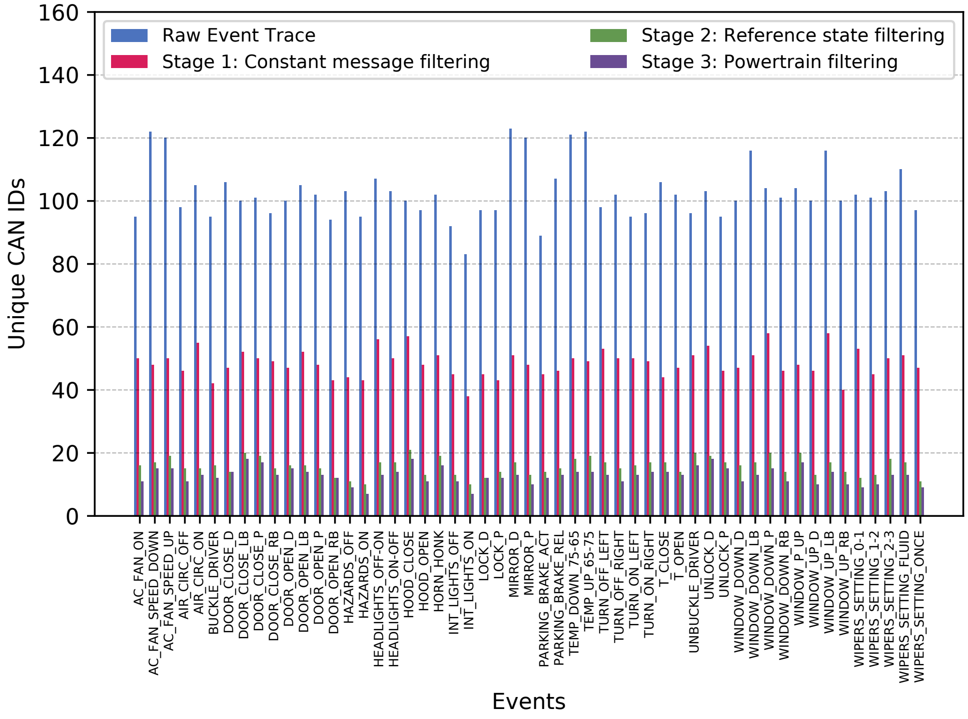


Figure 12: Number of Unique CAN IDs Remaining After Each Stage for all 53 Events for Vehicle A

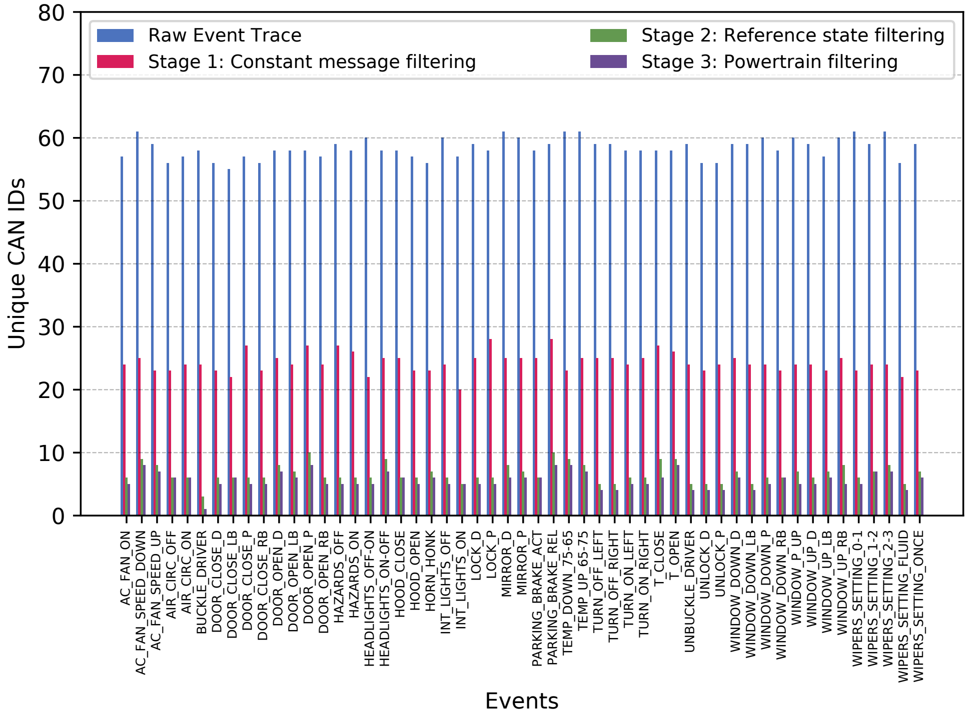


Figure 13: Number of Unique CAN IDs Remaining After Each Stage for all 53 Events for Vehicle B

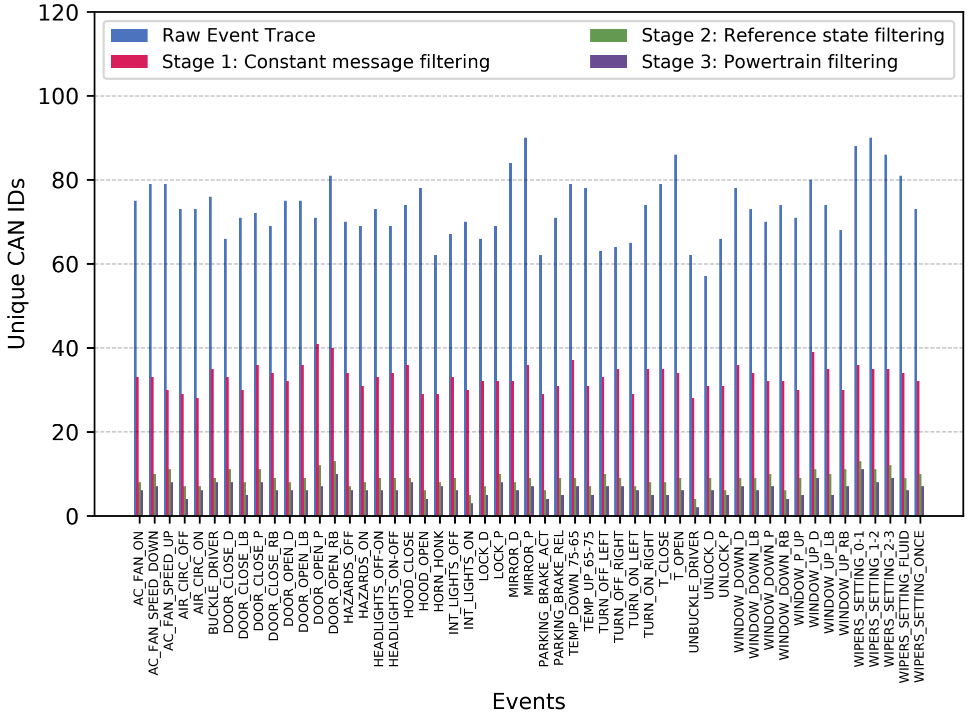


Figure 14: Number of Unique CAN IDs Remaining After Each Stage for all 53 Events for Vehicle C

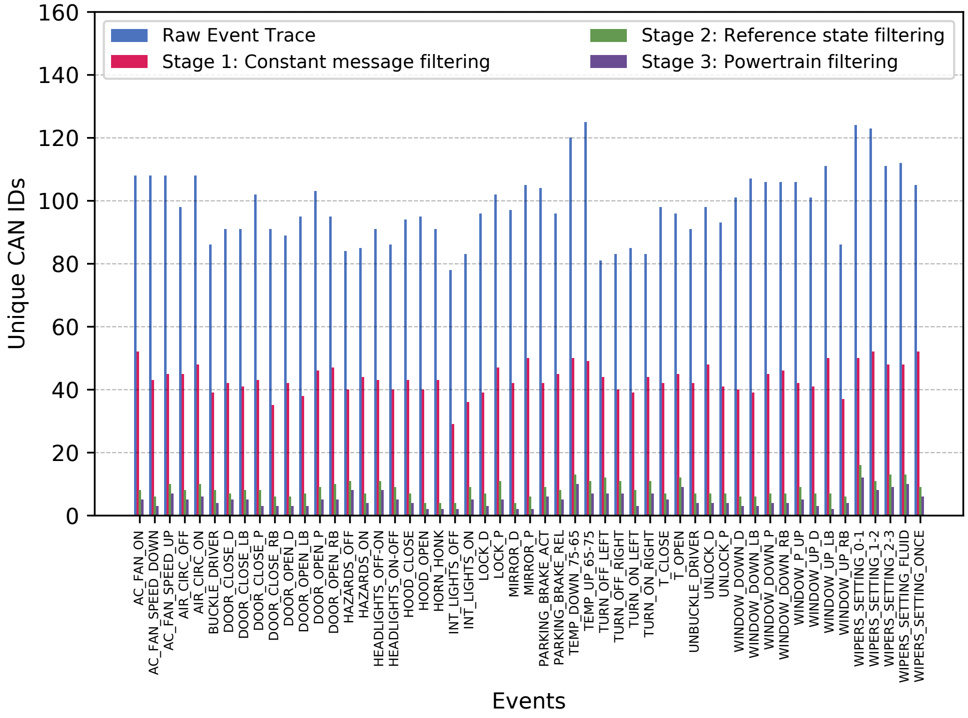


Figure 15: Number of Unique CAN IDs Remaining After Each Stage for all 53 Events for Vehicle D