Reverse Engineering of Multiplexed CAN Frames

Alessio Buscemi
University of Luxembourg (alumnus)
alessio.buscemi@alumni.uni.lu

Thomas Engel
University of Luxembourg
thomas.engel@uni.lu

Kang G. Shin
University of Michigan
kgshin@umich.edu

***Abstract*—The Controller Area Network (CAN) is widely deployed as the *de facto* global standard for the communication between Electronic Control Units (ECUs) in the automotive sector. Despite being unencrypted, the data transmitted over CAN is encoded according to the Original Equipment Manufacturers (OEMs) specifications, and their formats are kept secret from the general public. Thus, the only way to obtain accurate vehicle information from the CAN bus is through reverse engineering. Aftermarket companies and academic researchers have focused on automating the CAN reverse-engineering process to improve its speed and scalability. However, the manufacturers have recently started *multiplexing* the CAN frames primarily for platform upgrades, rendering state-of-the-art (SOTA) reverse engineering ineffective. To overcome this new barrier, we present *CAN Multiplexed Frames Translator* (CAN-MXT), the first tool for the identification of new-generation multiplexed CAN frames. We also introduce *CAN Multiplexed Frames Generator* (CAN-MXG), a tool for the parsing of standard CAN traffic into multiplexed traffic, facilitating research and app development on CAN multiplexing.**

I. Introduction

CAN is a message-based protocol which allows ECUs inside vehicles to communicate with each other without relying on a central bus master [18], [19], [20]. The popularity of CAN has soared to the point of becoming the *de facto* world standard for in-vehicle communication. The CAN communication data is not encrypted and no authentication is implemented for the ECUs. However, the CAN data is encoded in different formats that depend on the specific design choices made by the OEM and are kept secret to the general public.

Companies that offer aftermarket solutions and consumers are interested in using automotive data as an enabler of safety, convenience and infotainment features. However, given the OEMs's unwillingness to disclose the formats of most CAN signals of commercial vehicles (passenger cars in particular) to the general public, a common way to acquire such information is through *reverse engineering*. The state-of-the-art (SOTA) CAN reverse-engineering methodologies [22], [26], [25], [30], [17], [38], [32], [9], [15] have achieved good performance in terms of speed, accuracy and coverage.

Recently, however, OEMs have been investigating the *multiplexing* of CAN frames, enabling ECUs to send messages with the same CAN frame ID but carrying different payloads/signals [13], [37]. In other words, frame multiplexing is the process of combining multiple data frames into a single frame. As recognized by the world leading suppliers of in-vehicle networking software [3], [6], this frame multiplexing enables system designers to provide sufficient headroom for future platform upgrades. In fact, a new network design, including the assignment of frame IDs, is only implemented for significant platform revisions. However, many vehicle models undergo periodic facelifts, which are incremental updates that involve the replacement of some ECUs.

OEMs are moving from a centralized architecture to toward a zonal architecture that divides a vehicle's electrical and electronic systems into distinct zones or domains [7]. Since the zonal architecture allows for greater control over the vehicle's hardware and software, these facelifts will likely occur frequently in the future. Frame multiplexing is thus expected to play a significant role in the future of CAN communications.

The surge in physical and remote attacks on CAN threatens vehicle and driver/passenger safety, with proven efficacy [23], [27], [28], [40], [21]. While the automation of CAN reverse-engineering is essential for research and business, by granting fast and easy access to in-vehicle data, it also exposes vulnerabilities, potentially enabling attacks involving multiple vehicles and automated driver fingerprinting at scale [12].

Multiplexing data in each CAN frame can greatly affect the effectiveness of SOTA CAN reverse engineering, in particular, tokenization. This paper explores the feasibility of reverse-engineering multiplexed CAN frames with SOTA methodologies. In particular, this paper makes the following contributions:

1) Development of CAN-MXT, a translator of multiplexed CAN frames. This is the first algorithm capable of identifying multiplexed CAN frames and can be integrated in any reverse-engineering tool between the data collection and the tokenization step;
2) Development of CAN-MXG, a generator of multiplexed CAN frames. This is a tool for generating realistic multiplexed CAN traffic based on real CAN traces collected from vehicles;
3) Evaluation of CAN-MXTon multiplexed CAN traces generated with CAN-MXGfrom standard CAN traces gathered in real-world scenarios. The evaluation dataset is made publicly available;
4) Identification of the security threats associated with the automation of CAN reverse engineering.

II. Background

This section introduces the preliminary concepts and notions necessary for understanding the technical details about

Symposium on Vehicles Security and Privacy (VehicleSec) 2024
26 February 2024, San Diego, CA, USA
ISBN 979-8-9894372-7-6
https://dx.doi.org/10.14722/vehiclesec.2024.23011
www.ndss-symposium.org

CAN-MXT and CAN-MXG.

A. Controller Area Network

Inter-ECU communication is achieved via messages/frames over the CAN bus. Each ECU periodically transmits a CAN frame, which, in the absence of collision, reaches all ECUs. While an ECU may receive or transmit frames with distinct IDs, all the frames with the same ID are transmitted by the same ECU. The frames contain no information about its originator or receiver ECU.

A CAN frames is composed of multiple fields: 1) Start of Frame 2) ID, 3) RTR (Remote Transmission Request) bit, 3) Control Field, 4) Data Field, or Payload, 5) CRC Field - Checksum, 6) EOF (End of Frame). The two fields that are most pertinent to our work are the ID and the payload. The ID uniquely identifies the frame and carries the frame's priority used to resolve collision with other frames. An ECU can transmit or receive CAN frames with different IDs, but different ECU cannot send frames with the same ID.

The actual data contained within the frame is found in the payload. In the standard version of the CAN protocol, the payload of messages that are sent with the same ID always has the same length, which is specified by the Data Length Code (DLC). One or more signals may be contained within a payload. A *signal* is a portion of the payload that completely encapsulates a vehicle function.

B. CAN Reverse Engineering

In CAN, finding the boundaries of the signals contained within a frame payload is the first step in the process of reverse engineering. This step is referred to as *tokenization*. It is followed by the decoding of the signals' formats (such as scale factor and offset) and their semantic meanings (i.e., what vehicle function they encapsulate), known as *translation*. The process of manual reverse engineering is carried out by a human operator, who physically connects and disconnects the ECUs and monitors the CAN transmission in order to identify changes in the payload [33]. Injecting diagnostic messages through the On-Board Diagnosticss (OBD-IIs) port in order to generate a response from the CAN bus is another method that can be used to retrieve certain signals.

By contrast, in automated reverse engineering, the tokenization is accomplished in the majority of cases by studying the flipping of the bits that compose the frames over the time [30], [25]. Following the process of tokenization, the tokens are typically compared with GPS/IMU data, which provides a ground truth about the current status of the vehicle while it is being driven, in order to discover correlations [32]. It is a common practice to automate not only the injection of particular diagnostic messages but also the analysis of consecutive responses from the CAN bus.

The translation, or at least part of it, can be accomplished with a variety of different approaches that span from using commercial companion apps [39] to Machine Learning (ML) [29], as described in [12]. For example, in ML-based approaches, a classifier is trained to recognize the characteristics (or features) of each signal in CAN traces from known vehicles, so as to exploit this knowledge later on an unknown vehicle. Note that the level of automation as well as the equipment and time required for the reverse engineering vary greatly across the SOTA methodologies.

The output of a reverse engineering process is typically stored in a Database CAN (DBC) file. The DBC file format is widely accepted as the standard for storing data that facilitates the interpretation of CAN signals from their raw format into a version that is easily comprehensible to humans. DBC files can be used as input to a variety of CAN diagnostic tools, which are used by OEMs and aftermarket companies to interpret the data exchanged via the bus in real time or offline.

C. Multiplexing CAN Frames

In traditional CAN traffic, the number of signals that can be allocated in a frame is bounded by the DLC field. The purpose of multiplexing frames in CAN is to overcome this restriction, thus allowing frames to carry more signals than their DLC usually permits. At the time of this writing, multiplexed CAN frames have not yet been deployed in commercial vehicles on the road. However, according to the information we obtained from a prominent supplier of in-vehicle networking software, one of the primary German OEMs is currently in the testing phase for CAN frame multiplexing. Moreover, some of widely-used commercial CAN traffic analyzers [3], [4], [2] now support CAN frame multiplexing and provide a means to represent multiplexed CAN frames in the DBC format. We, therefore, expect CAN frame multiplexing to be implemented and deployed in commercial vehicles in the near future.

CAN frame multiplexing can be either *simple* or *extended*. In the former, the value of a reference signal, called *multiplexor*, is used to identify the set of signals that can be found in the frame. We define the frame associated with a specific multiplexor value as a *subframe*. The number and the position of the signals in each subframe are independent of the other subframes. Similarly to the standard frames, subframes are sent periodically. However, to avoid collisions, each subframe is sent after a certain *delay* relative to the *base period* of the *cycle* [13]. The base period of the cycle indicates how long it takes for the first subframe in the sequence to be sent again.

In the case of *extended* multiplexing, there are multiple hierarchically-ordered multiplexors that define the content of the frame. Figure 1 shows an example of extended multiplexed frame with 4 subframes and 2 multiplexors. Besides the multiplexors, the frame contains 7 signals. The base period of the cycle is 100 µs and the subframes are sent with separation of 20 µs from each other. Note that there is a time gap of 40 µs between the transmission of the 4-th subframe and the start of the following cycle. As illustrated in the example, the extended multiplexing allows multiplexor $S0$ to have the same value in different subframes by virtue of the fact that $S1$ is the distinguishing element of the payload contents. Note that the 3rd and 4th subframes could also share the same value for $S0$ (different from the first two subframes). In such a case, a 3rd multiplexor would be required to distinguish the contents of the two subframes. Finally, we define the temporally-ordered sequence of subframes associated with the same multiplexor value (and hence, payload content) as *subframe series*. Figure 2 shows an example series of subframes.

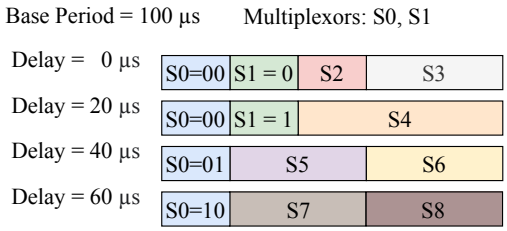


Fig. 1: An example of extended multiplexing. S0 and S1 correspond to the multiplexor signals. Each combination of values uniquely identifies a subset of signals.

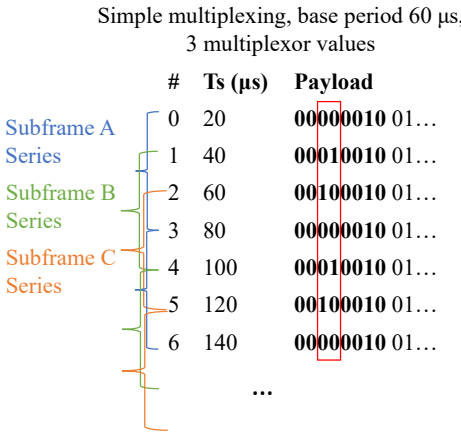


Fig. 2: An example series of subframes. Each color indicates a different subframe series. The red rectangle in the payload identifies the bits corresponding to the multiplexor.

III. CAN-MXT

This section presents the technical details of Multiplexed CAN Frames Translator (CAN-MXT), and discusses its underlying objective and assumptions.

A. Objective

As mentioned in Section I, multiplexing does not allow SOTA tokenization algorithms to identify the boundaries of signals within a frame's payload. However, we argue that SOTA tokenization algorithms can be efficiently adapted to deal with the frame multiplexing. The first step is to split the target CAN trace in subtraces each of which is a series of frames associated with an ID. Then, each subtrace is processed independently to check whether it is multiplexed and, if yes, locate the multiplexor signal(s). Once subframes are known, it suffices to reverse-engineer each series of subframes independently. CAN-MXT is an algorithm that can identify multiplexed CAN frames. Its goal is to enable SOTA reverse-engineering algorithms to work on multiplexed CAN traffic.

B. Assumptions

Based on the documents related to CAN multiplexing published thus far by prominent software suppliers for in-vehicle networks utilizing CAN Multiplexing [37], [13], we design CAN-MXT under the following assumptions.

A1. The multiplexors are located in the first byte of the payload, as their purpose is to indicate the content of the rest of the payload.

A2. In the case of extended multiplexing, distinct multiplexors are placed consecutively in the payload and ordered according to the multiplexing hierarchy, as shown in Figure 1.

Given that the purpose of Multiplexing is to enable ECUs to transmit more signals subject to the stringent payload constraints, and based on the available documents, we infer multiplexing to be applied only to cyclic frames, not event-driven frames. In this scope, we consider only the case without any time gap between the sending of the last subframe in a cycle and the subsequent cycle. In other words, given a multiplexed frame M of N subframes and a base period P, the time gap between two consecutive cycles is equal to the delay between subframes, i.e., P/N.

The reason for placing this constraint is that we want to build CAN-MXT to identify multiplexed frames under the most challenging scenarios. As an example, consider a CAN trace composed of all standard frames, but one multiplexed frame, with ID A, a base period of 100 µs and 4 subframes with a delay of 20 µs. Given that the standard frames are sent periodically, it would be easy to label ID A as multiplexed, since it is the only frame in the trace with a cyclic time gap between the sending of consecutive frames. By contrast, if the delay was 25 µs, there will not be a time gap, and hence will not be possible to identify the multiplexed frame without a further analysis of the payload, such as the one provided by CAN-MXT.

C. Overview

Algorithm 1 describes the main steps of CAN-MXT. It requires in input a CAN trace T. Initially, T is split in N subtraces, one for each CAN ID (Step 1). Each subtrace corresponds to the time-series of frames associated with the same ID.

Each subtrace ST is then evaluated independently (Steps 2–8). The mean period P of ST is calculated (Step 3). Then, the function *identify_multiplexing* tests whether ST is a sequence of multiplexed frames (Step 4). This is done by iteratively assuming cycles with different base periods and verifying that there is a signal within the first byte of the payloads that meet the conditions to be a multiplexor. The function outputs the base period B and the list of candidate multiplexors CM.

If CM is not null, the subframes of ST undergo additional processing to extract information regarding the multiplexing type MT, as well as defining the final set of multiplexors M, i.e., a refined version of CM (Steps 5–7).

D. Calculate Period

To calculate the period of a list *ts_diff* containing the differences between the timestamps of consecutive frames in ST is calculated. Then, the raw period P_{raw} is computed as the mean of *ts_diff*. Finally, the reference period P is obtained by rounding P_{raw} to its order-of-magnitude. For instance, if the calculated P_{raw} is 101.3 μs, P is rounded to 100 μs. This step is necessary to identify the actual sending

Algorithm 1 CAN-MXT

Require: CAN Trace T
Ensure: Base Period B, Multiplexors M, Multiplexing Type MT
1: *subtraces* ← extract_subtrace(T)
2: **for** ST **in** *subtraces* **do**
3: p ← calculate_period(*ST*)
4: B, CM ← identify_multiplexing(*ST*, p)
5: **if** CM **is not** null **then**
6: M, MT ← extract_info_multiplexing(*ST*, CM)
7: **end if**
8: **end for**

period P of the frames associated with the current ID as the manufacturer intended. According to our experience with more than 450 CAN traces [9], the manufacturers define the period of the CAN IDs as a multiple of 10 μs. However, due to the imprecision of the quarz clocks installed in the ECUs, the actual sending period recorded in the traces typically drifts away from the intended sending period.

E. Identify Frame Multiplexing

Algorithm 2 illustrates how to assess whether the current subtrace ST is a sequence of multiplexed frames.

Algorithm 2 identify_multiplexing

Require: Subtrace ST, Period P, Max number of subframes N
Ensure: Candidate Multiplexors CM, Base Period B
1: CM ← []
2: **for** n **in** range(2, N) **do**
3: **for** *start_frame* **in** range(0, n-1) **do**
4: *start_ts* ← timestamp(*start_frame*)
5: *curr_series* ← extract_series(ST, n, *start_ts*, P)
6: *xnor_res* ← XNOR(*curr_series*)
7: *c_mux_value* ← extract_seq(*xnor_res*)
8: **if** eligible_sequence(*c_mux_value*, CM) **then**
9: CM.append(*c_mux_value*)
10: **else**
11: **break**
12: **end if**
13: **end for**
14: **if** is_complete(CM, n) **then**
15: B ← info_baseperiod(ST, n)
16: **break**
17: **end if**
18: **end for**
19: **if** multiple_candidate_sets(CM) **then**
20: CM ← longest_candidate(CM)
21: **end if**

First, CM is initialized (Step 1). CM is the list that will contain output of this function, i.e., the first byte of each subframe, if ST is found to be multiplexed.

Then, the algorithm iteratively tests the assumption that ST contains n subframes (Steps 2–17). n ranges from 2 to N, which is defined by the user. Under the assumption **A1** in Section III-B that the first byte of the payload is reserved for multiplexing CAN frames, N can be at maximum 256. Hence, 256 is the theoretical upper bound of the number of distinct multiplexor values associated with ST. However, it is reasonable/practical to assume that the number of multiplexor values is much lower. Thus, it is logical to set N few orders-of-magnitude smaller than the theoretical maximum to reduce the computation time.

Xnor_res	False	False	True	True	True	False	True	False
C_mux_value	X	X	1	0	0	X	1	X

Fig. 3: Example of the XNOR's output and the *extract_seq* functions. The rectangles indicate the positions occupied by static bits.

An internal loop is used to iterate over ST and identify subframe series (Step 3). In particular, for each iteration, the timestamp *start_ts* of the frame at position *start_frame* is retrieved (Step 4). Subsequently, the function *extract_series* outputs a series *t_series* containing all frames whose timestamp is *start_ts* + c*n*P, where c serves as a counter for the cycles (Step 5).

Inspired by the READ algorithm [25], the algorithm then performs an XNOR operation between the initial byte of the first payload and the corresponding byte of the second payload in the *t_series*. The XNOR operation is then repeated between the outcome of the preceding XNOR and the initial byte of the subsequent payload. This iterative process continues until the conclusion of the *t_series*. The aim is identify identifying *static* bits, i.e., bits which never flip (change value) throughout the series (Step 6). The output of the XNOR operation, *xnor_res*, is passed to the function *extract_seq* (Step 7).

This function outputs the list *c_mux_value*, which displays the value of the static bits within the byte. For ease of understanding, an example of the difference between *xnor_res* and *c_mux_value* is shown in Figure 3. A check is performed on *c_mux_value* (Step 8) to verify that it meets all of the following requirements:

R1. There is a sequence of consecutive static bits whose length is at least $log_2(n)$. The existence of such a sequence indicates that the *t_series* may be a subframe series. On the other hand, its absence validates that the *t_series* under consideration is not a subframe series.

R1. The position of the sequence is overlapping with the other sequences in CM. Based on the assumption **A2** in Section III-B, multiplexors in different subframes are located in the same portion of the payload.

R3. The sequence is not yet present in CM. Given that a distinct multiplexor value is associated with each subframe, a unique sequence of static bits for each *c_mux_value* is necessary for ST to be multiplexed.

If the above check is successful, the current *c_mux_value* is appended to CM. At the end of the inner loop (step 14), the algorithm verifies that:

1) There is a fully formed set of subframes that identify ST as potentially multiplexed;

2) The set CM does not consist of counter signals. Counter signals and multiplexors can be mistakenly identified as each other because of their cyclic behavior. This verification is made by comparing the Bit Flip Rate (BFR) [25], [32], [9] of the payload sequence linked to each element in CM with the BFR of all payloads in ST. If the BFR of each individual element in the set CM consistently deviates from the overall BFR, the signals found in CM are more likely multiplexors rather than counters.

If both conditions are met, the information related to the base period B is inferred from ST and n (Step 16). Then, the outer loop is interrupted (Step 17).

Finally, if more than one sequence meet the condition to be considered as a multiplexor, only the longest one is chosen (Steps 20–21). This criterion was designed solely to ensure a deterministic single output. Nevertheless, the presence of sequences that satisfy all inclusion criteria for CM other than the multiplexor is highly improbable. In fact, our evaluation of CAN-MXT found that this final check was not necessary for all of the multiplexed frames we tested.

F. Extract Information on Multiplexors

After assessing the multiplexing of a subtrace ST,CAN-MXT analyzes the output, CM, of Algorithm 2 to find the multiplexing type (i.e., simple or extended) and obtain more details of the multiplexors.

Algorithm 3 initially computes the minimum number of bits *min_bits* necessary to represent the multiplexor values in the case of simple multiplexing (Step 1). This number can be calculated as $log_2(CM.\text{length})$. *min_bits* is then employed to generate the *target*, a list of multiplexor values we expect to find in the case of simple multiplexing. Based on assumption **A3** in Section III-B, the target is the binary representation of numbers between 0 and N, where N is the number of subframes.

Algorithm 3 extract_information_multiplexing

Require: Candidate Multiplexors CM
Ensure: Multiplexors M, Multiplexing Type MT
1: *min_bits* ← compute_minbits(CM)
2: *target* ← generate_target(*min_bits*)
3: MT, M ← match(*target*, CM)

Finally, CM and the *target* are inputted to the *match* function, whose purpose is to identify a match between the *target* and any portion of CM. The function adopts a sliding window approach, i.e., it scans CM from the start bit of the candidate multiplexor and iteratively checks if the current window matches the *target*. Figure 4 illustrates how the match function works, based on two example multiplexed frames, each with 3 subframes. In these examples, the target is [00, 01, 10]. In Example A, a match is found. Hence, the multiplexing is simple, and all of the static bits which do not belong to the multiplexor are discarded from the final set of multiplexors M. In Example B, no match with the target is identified. In this case, $M = CM$.

Note that in the case of extended multiplexing, the algorithm cannot identify the boundary between the multiplexors.

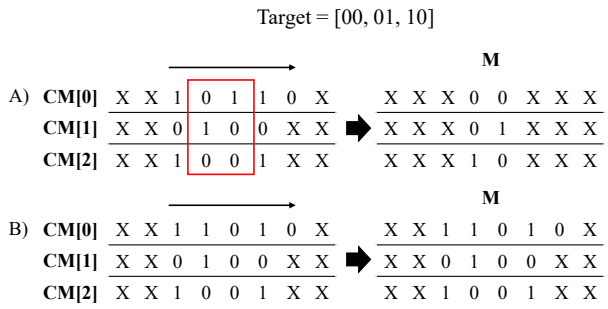


Fig. 4: Two examples showing the functioning of Algorithm 3. On the left, the target is matched against CM. In scenario A, the red rectangle indicates that a match is found between the target and CM. On the right, the final set of multiplexors M.

This is a limitation of the current CAN-MXT, and addressing it is part of our future work.

This section presents CAN-MXG, a tool to generate synthetic traces with multiplexed frames from real traces of non-multiplexed frames.

G. Objective

Comma AI was the first to release an extensive dataset of DBC files. While the collection of DBC files contained in Comma AI's open repository OpenDBC [1] is impressive, no CAN trace is present in its dataset. Hence, it is up to the researchers to collect the data from the vehicle model related to the DBC file of interest.

Zago et al. [41] released a dataset consisting of several hours of CAN traces related to 5 vehicle models suitable for reverse engineering. Nonetheless, no ground truth is associated with the traces, as the repository does not contain any DBC file. Unfortunately, OpenDBC does not contain any DBC related to the vehicle models present in this dataset. As a result, it is not possible to use these two datasets jointly.

At the time of this writing, none of the publicly available CAN traces or DBC files are related to multiplexed CAN traffic. Given the unavailability of real multiplexed data to evaluate CAN-MXT, we implemented CAN-MXG, a software tool that can generate multiplexed CAN traffic frames from standard CAN traces.

H. Overview

Algorithm 4 provides an overview of CAN-MXG. The algorithm receives in input a standard CAN trace T and the parameters described in Table I, which are set by the user. Through the configuration of these parameters, users can simulate a wide range of scenarios that an OEM might contemplate when multiplexing the frames. The first step is to identify IDs whose frames can be used as subframes for the multiplexed traffic (Step 1). The output of this operation is the list E of IDs whose frames are eligible for the task. Subsequently, the IDs in E are divided into groups or *pools* based on the common characteristics of the frames associated with them (Step 2). Finally, the pools PL are used to generate the multiplexed trace MT (Step 3).

TABLE I: CAN-MXG Parameters

Parameter	Description
Percentage of eligible frames to multiplex	The percentage of frame IDs to multiplex over the total number of eligible frame IDs present in the original trace.
Multiplexing Type	Simple or extended multiplexing.
Max number of subframes	The maximum number of subframes.
Bits behavior	The behavior of the other bits in the bytes containing the multiplexors, where *static* corresponds to bits that never change throughout the trace and *dynamic* are bits that change randomly. *Half dynamic* indicates that 50% of the bits other than the multiplexor are static, while the other 50% are dynamic.

Algorithm 4 CAN-MXG

Require: CAN Trace T, Percentage of eligible frames to multiplex P, Max number of subframes M, Multiplexing Type MT, Bits Behavior BB
Ensure: A multiplexed trace MT
1: $E \leftarrow$ identify_eligible_frames(T, P)
2: $PL \leftarrow$ pool_frames(T, E, M)
3: $MT \leftarrow$ generate_mux_trace(T, M, BB, PL)

I. Identification of Eligible Frames

This phase is concerned with the identification of frames in the original trace that are eligible for multiplexing. CAN-MXG aims at preserving the information contained in the original trace, i.e., the set of signals contained in the payload. For this reason, frames whose DLC is maximum (i.e., the payload is 8 bytes long) should in principle not be considered for multiplexing. Given that the first byte of the payloads in the generated multiplexed frames will contain the multiplexors, there would be only two ways to generate such a payload:

1) Produce a 9-byte payload, thus exceeding the maximum payload length (=8 bytes) imposed by the standard CAN;
2) Remove one byte from the original payload, in order to allocate the necessary space for the multiplexors. This would normally result in loss of information. Alternatively, the removed byte could be appended to frames with a different IDs in the generated trace. However, this could be realistically done only for other frames sent by the same ECU, if any, and would alter the original stream of information.

Nonetheless, as reported in [32], [9], the frames' payloads can also contain bits which are *unused*, i.e., they neither flip nor contain any information, but rather function as a buffer between signals. In this regard, CAN-MXG can remove bytes containing only unused bits to allocate space for the multiplexors. In such a case, even frames with 8-byte payloads are eligible for multiplexing.

The pseudocode related to this phase is presented in Algorithm 5. The algorithm requires in input a CAN trace T. It initially identifies as eligible only the IDs whose DLC is less than 8, and insert them in a list E (Step 1). Then, the algorithm searches for frames whose payload contains at least a byte composed of unused bits only and appends them to E (Steps 2–3). In our implementation of CAN MX-Multiplexor this operation is hard-coded. However, it can be easily modified to function as an optional feature.

Finally, the algorithm reduces the number of eligible frames by removing $1 - P$ randomly selected elements from E. Note that if $P = 1$ then no element is removed from E (Step 4).

Algorithm 5 identify_eligible_frames

Require: CAN Trace T, Percentage of eligible frames to multiplex P
Ensure: List of eligible IDs E
1: $E \leftarrow$ eligible_IDs(T)
2: $U \leftarrow$ unused_first_byte(E)
3: E.extend(U)
4: $E \leftarrow$ remove(T, P)

J. Pooling Frames

The goal of this phase is to identify groups of IDs which will constitute the new multiplexed frames, based on the characteristics that their frames have in common. The algorithm receives a CAN trace T, the list of eligible frame IDs E identified in Algorithm 5, and the max number of subframes M.

Initially, the mean sending periods P of the frame sequences associated with each ID in T are extracted (Step 1). The IDs in E are initially divided into G groups based on the DLC (Step 2). The IDs in each group are then sorted in ascending order according to P (Step 3). A data structure PL is initialized in Step 4. The purpose of PL is to contain subgroups of IDs whose frames will be blended to form a multiplexed frame, which we call *pools*. Operations are then performed independently on each group G_i of G (Steps 5–14). For each G_i, the IDs in position 0 to N are extracted iteratively, inserted into PL and removed from G_i (Steps 8–9). N is randomly generated between 2 (the minimum number of frames to form a multiplexed frame) and M. Let l be the length of G_i, if $l \geq M$, a pool is generated considering the whole G_i. Therefore, the number of pools per G_i lies between 1 and l/M.

Note that in the current implementation of CAN-MXG, the parameter *Multiplexing Type* determines all the eligible frames in a given run to be processed according to either simple or extended multiplexing. Given that 3 subframes are the minimum set to form an extended multiplexed frame, the pools composed of 2 subframes are always processed as simple-multiplexed, regardless of the *Multiplexing Type*.

Finally, the mean period associated with the first ID, i.e., the lowest mean period in the pool, is selected as a reference to calculate the base period of the cycle (Step 11). The reason for taking this approach is detailed in Section III-K.

K. Generation of Multiplexed Traces

The goal of this phase, as described in Algorithm 7, is to generate the final trace MT of multiplexed frames. The algorithm requires in input the CAN trace T, the pools PL

Algorithm 6 pool_frames

Require: CAN Trace T, Set of eligible IDs E, max number of subframes M
Ensure: Pools PL
1: $P \leftarrow$ calculate_base_periods(T)
2: $G \leftarrow$ divide_on_length(E)
3: $G \leftarrow$ sort_groups(G)
4: $PL \leftarrow \{\}$
5: **for** G_i **in** G **do**
6: $j \leftarrow 0$
7: **while** G_i **is NOT** empty **do**
8: $N \leftarrow$ random(2, M)
9: $PL[j] \leftarrow$ pop(G_i, N)
10: $PL[j]$.period $\leftarrow$ assign_period($PL[j]$)
11: j++
12: **end while**
13: **end for**

Algorithm 7 generate_mux_trace

Require: CAN Trace T, Pools PL, Bits behavior BB
Ensure: Multiplexed Trace MT
1: *list_MFS* $\leftarrow$ []
2: **for** pl **in** PL **do**
3: $FS \leftarrow$ extract_series(T, pl)
4: $eq\_FS \leftarrow$ equalize_length(T, FS)
5: *timestamped_FS* $\leftarrow$ assign_timestamps(pl, eq_FS)
6: $MFS \leftarrow$ form_mux_frame_series($timestamped\_FS$)
7: *mux_values* $\leftarrow$ create_mux_values(MFS)
8: *MFS_with_mux* $\leftarrow$ add_first_byte(MFS, mux_values, BB)
9: *final_MFS* $\leftarrow$ assign_ID(*MFS_with_mux*)
10: *list_MFS*.append(*final_MFS*)
11: **end for**
12: $MT \leftarrow$ produce_mux_trace(T, *list_MFS*)

extracted in Algorithm 6, and the bits behavior BB defined by the user.

Initially, the list that will contain all the multiplexed frame series *list_MFS* is initialized (Step 1). Operations are then performed independently on each pool pl (Step 2). The first step is to extract all frame series associated with each ID in the pl (Step 3). The output of this operation is a list of frame series FS. Then, for each series in FS, all the frames after position N are discarded, where N is the length of the shorter sequence (i.e., the one with the highest mean period in the original trace, thus less frames in the same time span) (Step 4). This way, we obtain equal-length series.

Subsequently, new timestamps are assigned to all frame series in FS based on the period identified in Algorithm 6 for the pool from which FS was extracted. In this step, each frame series in FS is treated as a subframe series and the timestamps are assigned to form cycles accordingly. Then, all the *timestamped_FS* are concatenated together, and the frames are sorted according to the new timestamps (Step 6). The result, MFS, is a new series of multiplexed frames. MFS does not yet contain the multiplexor signals, which are created by the function *create_mux_value* (Step 7).

In *create_mux_value*, the multiplexors are assigned as follows. A new empty byte is added at the beginning of each frame's payload in MFS. In the case of simple multiplexing, a list of binary values representing decimals between 0 and N is created, where N is the number of subframes. Note that this is the same process adopted for the creation of the *target* in Algorithm 3. In the case of extended multiplexing, a binary tree data structure with branches of different lengths is created. Each node in the tree, except for the root (which is null), corresponds to a bit. Each path from the root to the leaves corresponds to a multiplexor value or the concatenation of the values of hierarchically-ordered multiplexors. Starting from the root, the tree is created randomly until it has N leaves, where N is the number of subframes in the current multiplexed frame. In the current implementation of CAN-MXG, in order not to create trees which are heavily unbalanced, the maximum length of the longest branch is limited to the length of the shortest branch $+2$.

Once the multiplexor values have been created, they are inserted into an empty byte at a random position (Step 8). The value of the other bits in the byte are assigned according to the bit behavior BB. The thus-formed bytes are added in front of all payloads in the current multiplexed frame series. The ID of the newly-formed multiplexed frame series is then randomly chosen among the IDs of the original frames that compose it (Step 9). The output, *final_MFS*, is then appended to *list_MFS* (Step 10). Figure 5 illustrates an example output produced in Steps 4–9 of Algorithm 7. In this example, a multiplexed frame series is generated from the frames belonging to 3 different IDs.

Finally, when *list_MFS* is complete, it is passed to the trace T to the function *produce_mux_trace*, along with the trace T. The output is the final trace containing multiplexed frames MT (Step 12).

IV. Performance Evaluation

We now evaluate the performance of CAN-MXT on a set of traces generated with CAN-MXG. Both CAN-MXT and CAN-MXG were implemented in Python 3.10. All the tests presented hereafter were run on a Dell Latitude 5490 laptop, equipped with Intel(R) Core(TM) i5-8250U CPU @ 1.60 GHz, 1800 MHz, 4 Core(s) with 8 Logical Processor(s).

A. Dataset

The dataset utilized in this study comprises traces obtained from a total of seven real vehicles. Specifically, three of them were sourced from ReCAN [41], while the remaining four traces were collected by us. Our data was obtained utilizing a Raspberry Pi 3 device, which was equipped with a PiCAN2 Duo shield and connected to the OBD-II port of various commercial vehicles.

The data-collection process involved driving each of the following commercial cars: Peugeot 108, Peugeot 208 and Renault Captur for a duration of 60 seconds. The ReCAN repository utilizes two traces pertaining the following vehicles: Alfa Romeo Giulia and Opel Corsa. To ensure a fair evaluation and maintain consistency in trace length, we truncated the collected traces to the initial 60 seconds.

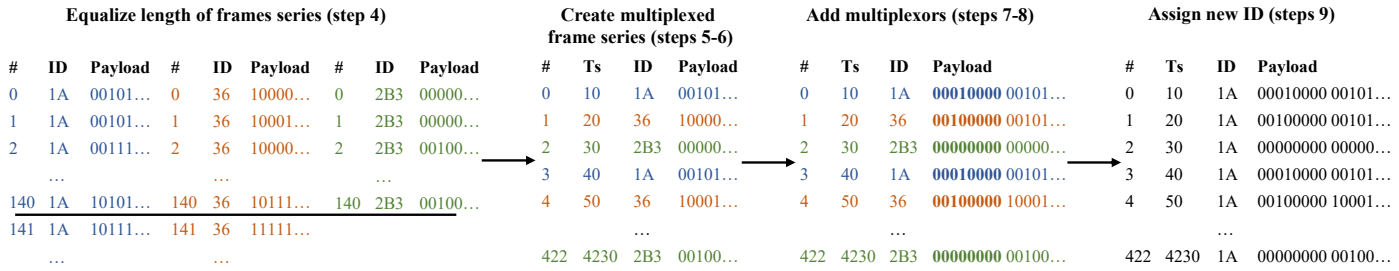


Fig. 5: An example of simply-multiplexed traffic generated by CAN-MXG from the frames belonging to 3 different IDs. The figure illustrates the output generated by Steps 4–9 of Algorithm 7.

The traces we extracted were collected in two different locations capturing distinct traffic and environmental scenarios. Each car was driven by a distinct driver to ensure that different driving styles and habits were included in the dataset.

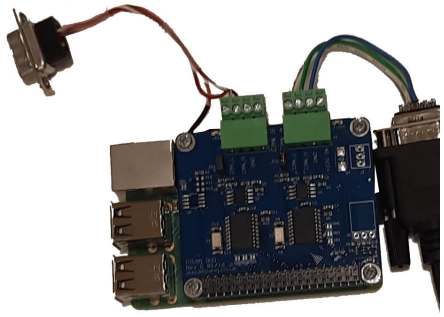


Fig. 6: Dongle used for data collection — a Raspberry Pi 3, equipped with a PiCAN2 Duo shield.

As to the traces from ReCAN [41], the dataset has numerous traces for each individual vehicle. In this study, we utilize the first trace (Exp-1) that is available for each vehicle. Note that ReCAN includes data pertaining to three other vehicles, namely, an Isuzu M55, a Mitsubishi Fuso Canter and a Piaggio Porter. The Isuzu M55 was not considered in our evaluation due to its lack of frames with a DLC < 8, while the Mitsubishi Fuso Canter only has one frame with a DLC < 8. Finally, the Piaggio Porter trace has too few unique IDs (19) to perform a comprehensive evaluation based on the parameters presented in Section IV.

Table II illustrates the composition of the test traces in terms of the number of unique IDs and the total number of frames. As shown in the table, the number of IDs and frames varies consistently across the test traces, despite the fact that they were all collected during the same time interval.

Figure 7 shows the percentage of eligible frames in the original CAN traces extracted from the test vehicles. The figure highlights a high variance in the percentage of eligible frames in the test set.

In this work, we used CAN-MXG to generate 10 distinct multiplexed traces for each combination of parameter values indicated in Table III. Note that the maximum number of subframes is limited to 8. This limitation arises from the fact that CAN-MXG was unable to discover more than 8 IDs that

TABLE II: Test traces overview

Vehicle	N. IDs	N. frames
Peugeot 108, 2010	79	61468
Peugeot 208, 2011	51	90411
Renault Captur, 2013	72	61863
Alfa Romeo Giulia	76	158316
Opel Corsa	78	122112

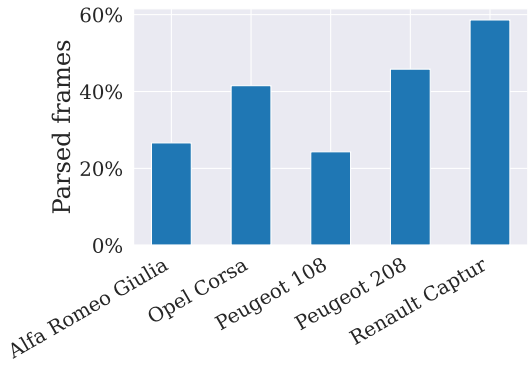


Fig. 7: Percentage of frames eligible for multiplexing in the tested vehicles.

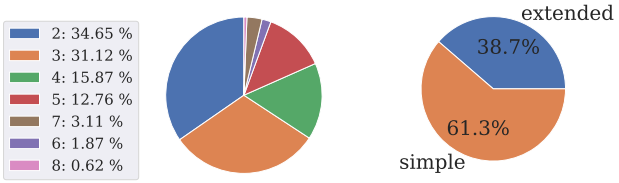
could be grouped together in the same multiplexed frame for any of the tested traces. The size of this dataset is comparable with that of the datasets utilized in related studies on CAN reverse engineering.

8 provides an overview of the characteristics of the test set used for the evaluation of CAN-MXT.

Chart a) in Figure 8 shows the composition of the test set in relation to the number of subframes associated with

TABLE III: Range of parameter values used for our evaluation

Parameter	Considered parameter range
Percentage of eligible frames to multiplex	[0.1-1], step 0.1
Multiplexing type	simple or extended
Max number of subframes	[2-8]
Bits behavior	Static, dynamic, half dynamic



(a) Percentage of frames in the test set with the same number of subframes.

(b) Percentage of frames in the test set belonging to each type.

Fig. 8: Composition of the generated multiplexed traces.

each multiplexed frame. The chart highlights that the majority of frames in our test set are composed of 5 subframes or less. In fact, as described in Section III-H, *Max number of subframes* only represents an upper bound of the number of subframes. Therefore, there may be only n frames associated to a certain DLC in the original trace, where $n <$*Max number of subframes*. The lower bound, instead, is the minimum number of bits required for multiplexing, i.e., 2 for simple multiplexing, and 3 for extended multiplexing.

Chart b) shows the composition of the test set in relation to the multiplexing type. According to the figure, approximately two thirds of the multiplexed frames in the test set were generated according to simple multiplexing. This is due to the fact that even in the case of creation of traces based extended multiplexing, pools composed of 2 subframes are treated as simple multiplexing, as explained in Section III-J.

Finally, from a preliminary analysis, we discovered that the *percentage of eligible frames to multiplex* does not affect the performance of CAN-MXT. Thus, we set this parameter to 1, i.e., all eligible frames are transformed into multiplexed frames. As a result, CAN-MXT was evaluated on 1,920 generated traces. The various combinations of parameters tested in this work reflect the choices available to a manufacturer for multiplexing frames, thus enabling us to provide a generalized assessment of CAN-MXT performance.

We made the generated traces publicly available on the Github repository `Reverse Engineering of Multiplexed CAN Frames` [5].

B. Evaluation Metrics

CAN-MXG outputs a *ground truth* file containing the information necessary to identify and interpret the newly-generated multiplexed trace. For each multiplexed frame, the ground truth contains the information on i) a mapping between its ID and the IDs of the original frames that compose it, ii) the multiplexing type, iii) the start position of the multiplexor, and iv) all the values of the multiplexors. To evaluate the performance of CAN-MXT on each tested trace, we compare the output it produced, with the trace's ground truth.

multiplexed IDs will henceforth refer to IDs whose associated frames are multiplexed, as opposed to *standard IDs*, whose associated frames are not multiplexed. Let the True Positives (TPs) be the multiplexed IDs correctly identified as multiplexed by CAN-MXT; the False Positives (FPs) be the IDs corresponding to standard IDs incorrectly identified as

Fig. 9: Recall, Precision, and Base Period Accuracy achieved by CAN-MXT for each test vehicle.

multiplexed by CAN-MXT; the True Negative (TN) be the standard IDs frames correctly identified as non-multiplexed by CAN-MXT; and the False Negatives (FNs) be the multiplexed IDs incorrectly identified as non-multiplexed by CAN-MXT.

Then, we evaluate CAN-MXT using the following metrics:

- *Recall*: is equal to $\frac{TP}{TP+FN}$;
- *Precision*: is equal to $\frac{TP}{TP+FP}$;
- *Base Period Accuracy*: the percentage of IDs whose base period has been correctly identified, over the total number of TPs. Note that this metric also implies the accuracy in identifying the correct number of the subframes, given that this information is necessary to calculate correctly the base period;
- *Multiplexor Type Accuracy*: the percentage of IDs whose multiplexor's type bit has been correctly identified, over the total number of TPs.

C. Results

Figure 9 shows the mean Recall, Precision, and Base Period Accuracy achieved on each vehicle by CAN-MXT. The average Recall of 98.4% indicates that almost all multiplexed frame IDs are correctly labeled, while the Precision of 99.3% indicates that few standard IDs are incorrectly identified as multiplexed. Based on these results, Algorithm 2 is determined to be highly reliable. Furthermore, Algorithm 2 achieves a high Base Period Accuracy of 99.2%.

Figure 10 illustrates the Multiplexor Type Accuracy achieved by CAN-MXT on each of the tested vehicles, divided according to the dynamicity. A confidence interval of 95% is reported for each measurement.

As expected, the Multiplexor Type Accuracy reaches the highest ($> 95\%$) when all the other bits in the byte are dynamic. By contrast, it attains the lowest (80%-94%) when a half the bits are dynamic. In fact, Algorithm 3 is designed to distinguish stationary bits that are not part of the multiplexors in addition to excluding bits with a dynamic behavior that is obviously distinguishable from the behavior of multiplexors. As a consequence, the algorithm performs worse when bits adjacent to the multiplexors have mixed behaviors. This is also demonstrated by the wide confidence interval associated with

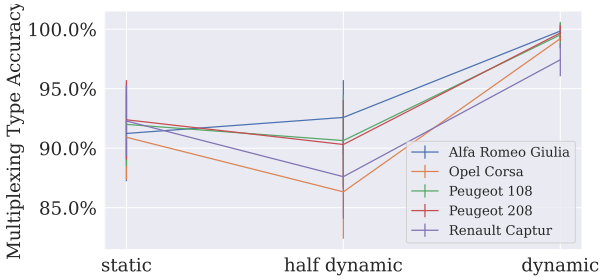


Fig. 10: Multiplexor Type Accuracy achieved by CAN-MXT for each test vehicle and according to the dynamicity of the bits other than the multiplexors.

TABLE IV: DBC trace coverage

Vehicle	% of IDs covered	% of eligible IDs covered
Peugeot 108	10.1%	9.5%
Peugeot 208	29.4%	16%
Renault Captur	27.8%	13.6%

the case when a half of the bits exhibit dynamic behavior, showing a high variance in the results. The additional results provided in Section A serve to further our understanding of the aforementioned findings.

Finally, the time taken by CAN-MXG for generating multiplexed traces and the time taken by CAN-MXT for translating the multiplexors were also measured. These results are provided in Section B.

D. Multiplexed Frames Reverse Engineering

As described in Section III-A, the ultimate purpose of CAN-MXT is to enable SOTA tokenization algorithms on multiplexed frames. Thus, we compare the results obtained with the tokenizer from CANMatch [9] on multiplexed frames alone vs. using CAN-MXT as a preliminary step. As a ground truth, for each of the three vehicles whose traces were collected by us, we have a DBC file generated by a human operator with expertise in manual reverse engineering. No DBC file is publicly available for the two traces from ReCAN. It is, therefore, not possible to benchmark the results with a ground truth for these two traces, and hence we exclude them from this evaluation.

Note that the DBC files used in this test cover only partially the signals that can be found in the traces, as reported in Table IV. As shown in the table, this also means that only a part of the IDs eligible for multiplexing with CAN-MXG can be matched against our DBC files. So, our evaluation of the tokenization exclusively relies on the signals which are present in the ground truth and found in eligible IDs. Let CE be the correctly extracted tokens and $TDBC$ be the total number of signals in the DBC file, then we use the ratio $CE/TDBC$ as a metric to evaluate the outcome of the tokenization for each tested vehicle [32], [9].

The results in Figure 11 show that without CAN-MXT the tokenization produces spurious results (i.e., a $\frac{CE}{TBDC}$ close or equal to 0). In fact, the chosen tokenization

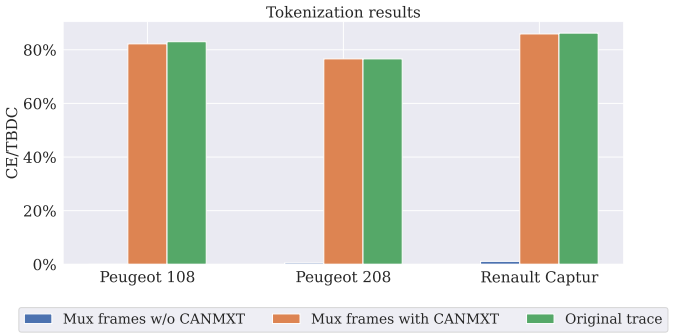


Fig. 11: Results obtained by the CANMatch's tokenization algorithm when CAN-MXT is not employed vs when CAN-MXT is employed. The performance of the algorithm on the same frames in the original frames is also reported to serve as a benchmark.

algorithm tries to identify consistency in the behavior of consecutive bits, based on their flipping across consecutive frames in the trace. However, no consistency can be found given that consecutive frames do not carry the same signals. Therefore, in the context of multiplexed frames, the tokenization method mostly generates tokens that consist of a single bit. In contrast, the utilization of CAN-MXT allows for the independent processing of each frame series within the multiplexed frame by the tokenization method. This enables the algorithm to achieve a performance comparable to that obtained for the target frames in the original trace.

V. Security Threats Associated with CAN Reverse Engineering

An alarming number of physical and remote attacks against CAN posing a serious threat to the safety of vehicles and passengers have been shown to be feasible [23], [27], [28], [40], [21]. The automation of CAN reverse-engineering, while being an essential tool for research and businesses, can also be exploited by adversaries. In fact, granting easy and fast access to cleartext in-vehicle data in Connected and Automated Mobility (CAM) scenarios may enable large-scale attacks and driver fingerprinting [12], [14]. As discussed in [12], the ability to reverse-engineer CAN data formats in less than one minute may enhance the adversaries' capabilities in congested environments, such as parking garages and road intersections. In particular, CAN-MXT requires a negligible amount of time to reverse-engineer CAN frames while allowing adversaries to exploit the new generation of (multiplexed) CAN frames.

A variety of countermeasures have been proposed to secure CAN, often using cryptographic algorithms [31], [36], [24] and Intrusion Detection System (IDS) [34], [16], [35]. However, cryptographic algorithms usually incur significant resource and latency overheads, and hence may not be practicable for the cost-conscious automotive industry. In contrast, IDS do not affect bandwidth and latency significantly and may thus seem to meet the OEMs's financial concerns. However, they are ineffective against most reverse-engineering approaches, as they are passive processes which do not alter the traffic and are, therefore, undetectable.

To counter the reverse engineering of multiplexed CAN frames, a first intuitive solution is to anonymize the IDs of the frames. However, as shown in [10], anonymizing IDs is insufficient to prevent reverse engineering, since they can be successfully de-anonymized through ML classification. A subsequent work [11] attempts to counter this de-anonymization through traffic mutation techniques, such as padding and morphing. The results show that the de-anonymization accuracy can be halved, but at the cost of a consistent traffic overhead. In addition to traffic mutations, we need to investigate the prevention of automated reverse engineering in the case of multiplexed CAN traffic, including:

1) Place the multiplexors in bytes of the payload other than the first, with the aim of making the identification of multiplexed frames harder;

2) In extended multiplexing, place the multiplexors in non-contiguous positions, in order to make their location more difficult;

3) At the time of this writing, OEMs have exploited endianness to encode CAN payloads only at byte level. As an alternative to the conventional use of bit-level big endianness, the byte that contains the multiplexors might be encoded also using bit-level little endianness (i.e., ordering the bits from the least to the most significant) As a result of this, the process of reverse-engineering multiplexed frames would become more difficult.

The main objective of this study was to assess the impact of multiplexing on reverse-engineering processes, with a focus on the potential enhancement of security provided by this mechanism. As illustrated in , without the integration of the proposed tool, CAN-MXT, state-of-the-art algorithms fail to reconstruct the underlying data structures embedded within the multiplexed payloads. However, the introduction of CAN-MXT provides a viable solution, demonstrating the possibility to circumvent these limitations and enhance the efficiency of reverse engineering in the presence of multiplexed frames.

In conclusion, our research on reverse-engineering multiplexed CAN frames is a double-edged sword. On one hand, it provides potential adversaries with the knowledge and means to exploit CAN data, even in the context of future multiplexed frames. On the other hand, our research functions as a warning to OEM, underscoring the limitations of multiplexing as an additional security layer. By doing so, it encourages OEMs to further bolster the security of CAN systems, fostering continuous improvement in safeguarding the integrity of vehicle communication and control.

VI. Conclusion

OEMs have recently been conducting tests on multiplexed CAN frames to support upcoming vehicle platform upgrades. These upgrades will occur more frequently as in-vehicle networks transition from a centralized to a zonal architecture. However, SOTA CAN reverse-engineering tools are shown to achieve good performance on standard CAN traffic, but are not effective in handling multiplexed CAN frames. Given the current security and privacy risks associated with automated CAN reverse engineering, multiplexed CAN frames are likely to enable further protection to the vehicle.

We are the first to investigate the feasibility of reverse-engineering this new type of multiplexed CAN frames which are expected to be deployed in near-future vehicles. CAN-MXT identifies multiplexed CAN frames and translates their multiplexors. It can handle both simple and extended frame-multiplexing, and is easily integrateable with SOTA CAN reverse-engineering schemes between the data collection and the tokenization phases.

We tested CAN-MXT on a set of 1,920 distinct traces containing multiplexed frames which were generated by another tool, CAN-MXG, we developed from standard non-multiplexed real traces collected from 5 distinct vehicles. CAN-MXG can parse standard CAN traces into multiplexed traffic in real-world settings and according to multiple parameters that can be set by the user. The numerous parameter combinations evaluated in this study provide the options available to a manufacturer for multiplexing frames according to our comprehensive evaluation of CAN-MXT performance.

Given the lack of publicly available DBC files and multiplexed CAN traces, we released the dataset used for our evaluation on Github [5]. We also plan to open-source CAN-MXG with the aim of providing the research and business communities with a useful tool to investigate CAN frame multiplexing.

These results show that, while CAN-MXT is a useful resource for the reverse engineering of the new generation and type of CAN traffic, it can also be exploited by adversaries to gather/extract information about the target vehicle's data. This information can then be exploited to mount tailored attacks. Finally, we discussed potential countermeasures against CAN-MXT's identification of multiplexed frames. In future, we would like to:

1) distinguish multiplexors in extended multiplexing – CAN-MXT can identify the start and the end position of the multiplexor signals but, in the case of extended multiplexing, it cannot distinguish the boundary between two multiplexors in the same subframe. In practice, this is not required for reverse engineering on the remainder of the payload (see Section III-A), but it is necessary to complete the understanding of the frame's content;
2) extend our data-collection capabilities, thus enabling CAN-MXG and CAN-MXT to process traces collected with a variety of dongles;
3) perform a comparative analysis of protocols employed to enhance CAN's data throughput and their impact on reverse engineering (see Section C).

Acknowledgement

Kang Shin's part of the work reported in this paper was supported in part by the US Office of Naval Research under Grant N00014-22-1-2622 and the US National Science Foundation under Grant CNS-2245223.

References

[1] OpenDBC. https://github.com/commaai/opendbc, 2021.

[2] CANAnalyzer. https://www.vector.com/int/en/products/products-a-z/software/canalyzer/, 2023.

[3] CANEasy. https://www.caneasy.de/startseite, 2023.

[4] CANoe. https://www.vector.com/int/en/products/products-a-z/software/canoe/, 2023.

[5] Reverse engineering of multiplexed can frames. https://github.com/CANMultiplexing/Reverse-Engineering-of-Multiplexed-CAN-Frames/, 2023.

[6] Vector Group. https://www.vector.com/int/en/, 2023.

[7] ALPARSLAN, O., ARAKAWA, S., AND MURATA, M. Next generation intra-vehicle backbone network architectures. In *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)* (2021), IEEE, pp. 1–7.

[8] AUTOSAR. Specification of CAN Transport Layer. Tech. Rep. R21-11, 11 2021.

[9] BUSCEMI, A., TURCANU, I., CASTIGNANI, G., CRUNELLE, R., AND ENGEL, T. Canmatch: A fully automated tool for can bus reverse engineering based on frame matching. *IEEE Transactions on Vehicular Technology 70*, 12 (2021), 12358–12373.

[10] BUSCEMI, A., TURCANU, I., CASTIGNANI, G., AND ENGEL, T. On frame fingerprinting and controller area networks security in connected vehicles. In *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)* (2022), IEEE, pp. 821–826.

[11] BUSCEMI, A., TURCANU, I., CASTIGNANI, G., AND ENGEL, T. Preventing frame fingerprinting in controller area network through traffic mutation. In *2022 IEEE International Conference on Communications Workshops (ICC Workshops)* (2022), IEEE, pp. 385–390.

[12] BUSCEMI, A., TURCANU, I., CASTIGNANI, G., PANCHENKO, A., ENGEL, T., AND SHIN, K. G. A survey on controller area network reverse engineering. *IEEE Communications Surveys & Tutorials* (2023).

[13] CANEASY. Multiplex messages, 2021.

[14] ENEV, M., TAKAKUWA, A., KOSCHER, K., AND KOHNO, T. Automobile driver fingerprinting. *Proc. Priv. Enhancing Technol. 2016*, 1 (2016), 34–50.

[15] EZEOBI, U., OLUFOWOBI, H., YOUNG, C., ZAMBRENO, J., AND BLOOM, G. Reverse engineering controller area network messages using unsupervised machine learning. *IEEE Consumer Electronics Magazine* (2020).

[16] HAMADA, Y., INOUE, M., UEDA, H., MIYASHITA, Y., AND HATA, Y. Anomaly-based intrusion detection using the density estimation of reception cycle periods for in-vehicle networks. *SAE International Journal of Transportation Cybersecurity and Privacy 1*, 11-01-01-0003 (2018), 39–56.

[17] HUYBRECHTS, T., VANOMMESLAEGHE, Y., BLONTROCK, D., VAN BAREL, G., AND HELLINCKX, P. Automatic reverse engineering of CAN bus data using machine learning techniques. In *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing* (2017), Springer, pp. 751–761.

[18] ISO 11898-1:2015. Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling. Standard, International Organization for Standardization, 12 2015.

[19] ISO 11898-2:2016. Road vehicles — Controller area network (CAN) — Part 2: High-speed medium access unit. Standard, International Organization for Standardization, 12 2016.

[20] ISO 11898-3:2006. Road vehicles — Controller area network (CAN) — Part 3: Low-speed, fault-tolerant, medium-dependent interface. Standard, International Organization for Standardization, 06 2006.

[21] JAFARNEJAD, S., CODECA, L., BRONZI, W., FRANK, R., AND ENGEL, T. A car hacking experiment: When connectivity meets vulnerability. In *2015 IEEE Globecom Workshops (GC Wkshps)* (2015), IEEE, pp. 1–6.

[22] JAYNES, M., DANTU, R., VARRIALE, R., AND EVANS, N. Automating ECU identification for vehicle security. In *15th IEEE International Conference on Machine Learning and Applications (ICMLA)* (2016), IEEE, pp. 632–635.

[23] KOSCHER, K., CZESKIS, A., ROESNER, F., PATEL, S., KOHNO, T., CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., ET AL. Experimental security analysis of a modern automobile. In *2010 IEEE symposium on security and privacy* (2010), IEEE, pp. 447–462.

[24] LU, Z., WANG, Q., CHEN, X., QU, G., LYU, Y., AND LIU, Z. Leap: A lightweight encryption and authentication protocol for in-vehicle communications. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)* (2019), IEEE, pp. 1158–1164.

[25] MARCHETTI, M., AND STABILI, D. READ: Reverse engineering of automotive data frames. *IEEE Transactions on Information Forensics and Security 14*, 4 (2018), 1083–1097.

[26] MARKOVITZ, M., AND WOOL, A. Field classification, modeling and anomaly detection in unknown can bus networks. *Vehicular Communications 9* (2017), 43–52.

[27] MILLER, C., AND VALASEK, C. Adventures in automotive networks and control units. *Def Con 21*, 260-264 (2013), 15–31.

[28] MILLER, C., AND VALASEK, C. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA 2015*, S 91 (2015).

[29] MOORE, M. R., BRIDGES, R. A., COMBS, F. L., AND ANDERSON, A. L. Data-Driven Extraction of Vehicle States From CAN Bus Traffic for Cyberprotection and Safety. *IEEE Consumer Electronics Magazine 8*, 6 (2019), 104–110.

[30] NOLAN, B. C., GRAHAM, S., MULLINS, B., AND KABBAN, C. S. Unsupervised time series extraction from controller area network payloads. In *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)* (2018), IEEE, pp. 1–5.

[31] PESÉ, M. D., SCHAUER, J. W., LI, J., AND SHIN, K. G. S2-can: Sufficiently secure controller area network. In *Annual Computer Security Applications Conference* (New York, NY, USA, 2021), ACSAC, Association for Computing Machinery, p. 425–438.

[32] PESÉ, M. D., STACER, T., CAMPOS, C. A., NEWBERRY, E., CHEN, D., AND SHIN, K. G. LibreCAN: Automated CAN Message Translator. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2019), ACM, pp. 2283–2300.

[33] QUIGLEY, C., CHARLES, D., AND MCLAUGHLIN, R. CAN Bus Message Electrical Signatures for Automotive Reverse Engineering, Bench Marking and Rogue ECU Detection. In *SAE Technical Paper* (04 2019), SAE International.

[34] SAGONG, S. U., YING, X., CLARK, A., BUSHNELL, L., AND POOVENDRAN, R. Cloaking the clock: emulating clock skew in controller area networks. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)* (2018), IEEE, pp. 32–42.

[35] SEO, E., SONG, H. M., AND KIM, H. K. Gids: Gan based intrusion detection system for in-vehicle network. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)* (2018), IEEE, pp. 1–6.

[36] VAN BULCK, J., MÜHLBERG, J. T., AND PIESSENS, F. Vulcan: Efficient component authentication and software isolation for automotive control networks. In *Proceedings of the 33rd Annual Computer Security Applications Conference* (2017), pp. 225–237.

[37] VECTOR. Extended Signal Multiplexing in DBC Databases. Tech. Rep. 1.2, 2019.

[38] VERMA, M., BRIDGES, R., AND HOLLIFIELD, S. ACTT: Automotive CAN tokenization and translation. In *International Conference on Computational Science and Computational Intelligence (CSCI)* (2018), IEEE, pp. 278–283.

[39] WEN, H., ZHAO, Q., CHEN, Q. A., AND LIN, Z. Automated cross-platform reverse engineering of can bus commands from mobile apps. In *Proceedings 2020 Network and Distributed System Security Symposium (NDSS'20)* (2020).

[40] WOO, S., JO, H. J., AND LEE, D. H. A practical wireless attack on the connected car and security protocol for in-vehicle can. *IEEE Transactions on Intelligent Transportation Systems 16*, 2 (2015), 993–1006.

[41] ZAGO, M., LONGARI, S., TRICARICO, A., GIL PÉREZ, M., CARMINATI, M., MARTÍNEZ PÉREZ, G., AND ZANERO, S. Recan source - reverse engineering of controller area networks, 01 2020.

APPENDIX A
ADDITIONAL RESULTS

In this appendix, we aim to further explain the results presented in Section IV-C. Specifically, let us consider the extended multiplexors as the positives, and the simple multiplexors as the negatives. As shown in the confusion matrix on the left, when the dynamicity mode is *static*, the results are

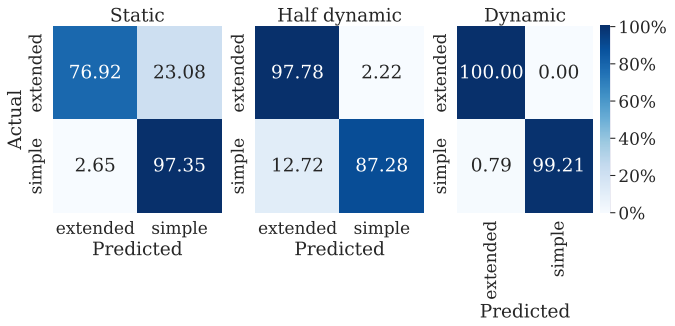


Fig. 12: Confusion matrices illustrating the type of errors made by CAN-MXG in identifying the multiplexing type according to the division of bits behavior defined in the input.

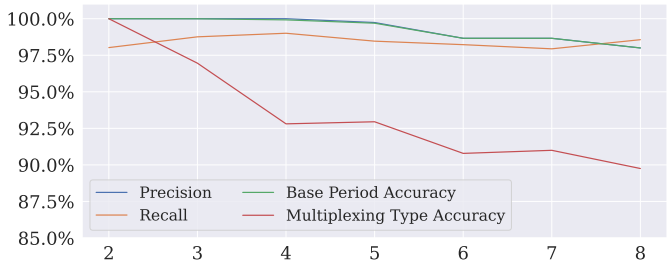


Fig. 13: Impact of the number of subframes on Recall, Base Period Accuracy and Multiplexing Type Accuracy achieved by CAN-MXT.

penalized by False Negative Rate (FNR) (i.e., FN/(TP+FN)) of almost 20% and False Positive Rate (FPR) (i.e., FP/(TP+FN)) of almost 3%. By contrast, when the dynamicity mode is *half-dynamic*, the FNR is around 15%, while the FPR is around 3%. As highlighted in Figure 8, around two thirds of the total multiplexors in our test set are simple, the FPR makes a more significant impact on the overall accuracy than FNR.

Finally, Figure 13 illustrates the impact that the number of multiplexors have on CAN-Multiplexor performance with respect to the metrics considered thus far. As shown in the figure, the number of multiplexors does not affect Precision, Recall and Base Period Accuracy much. In contrast, the Multiplexing Type Accuracy decreases with the number of multiplexors, from 100% when the subframes are 2, to 92.5% circa when the subframes are 4, and it is stable between 90% and 92% for a higher number of subframes.

Appendix B
CAN-MXT and CAN-MXG Execution Time

Here we analyze the time required for CAN-MXG and CAN-MXT to, respectively, generate and identify multiplexed frames. A confidence interval of 95% is reported for each measurement.

Figure 14 illustrates the mean execution time of CAN-MXG for each test vehicle. Every multiplexed trace is generated between 2 and 4s. Based on the figure and Table II, there appears to exist a positive linear correlation between the execution time and the number of frames in the original trace.

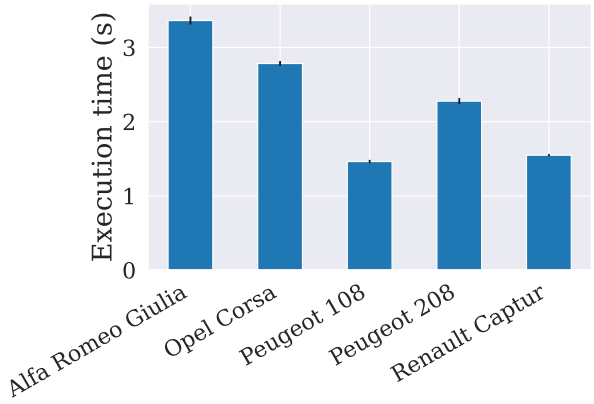


Fig. 14: Time required to generate multiplexed traces for each test vehicle.

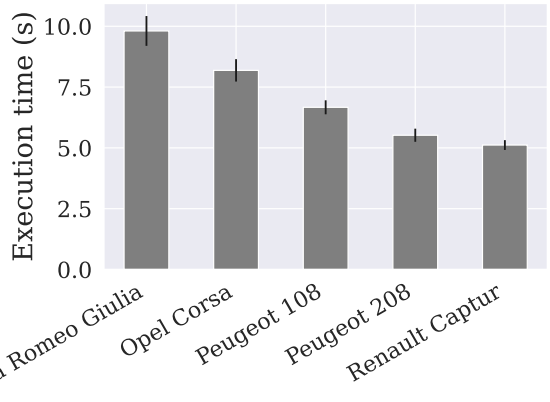
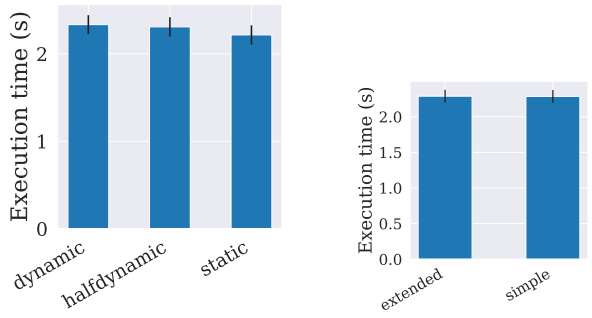


Fig. 15: Time required to translate multiplexed frames for each test vehicle.

Figure 15 illustrates the mean execution time of CAN-MXT for each test vehicle, which spans between 5 and 10s. Similarly to the generation of the multiplexed traces, CAN-MXT's execution time seems to be linearly correlated to the number of frames in the original trace.

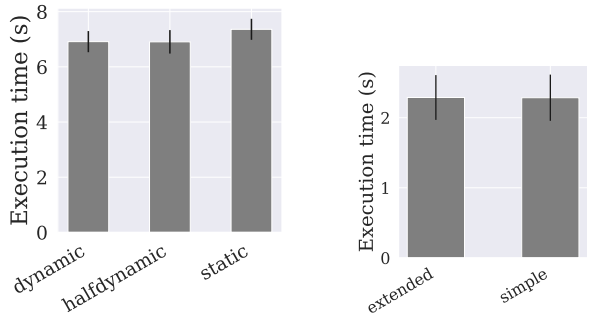
Figure 16 shows the mean execution time of CAN-MXG with respect to the dynamicity and the multiplexing type. As highlighted in plot a), CAN-MXG is the fastest in creating traces where the other bits in the multiplexor's byte are static, and slowest when they are dynamic. In contrast, the multiplexing type does not seem to impact the runtime of CAN-MXG, as shown in plot b).

Section B shows the mean execution time of CAN-MXT based on the dynamicity and the multiplexing type. As shown in plot a), CAN-MXT gets slowest when translating multiplexed frames whose multiplexor is inserted in a half-dynamic context, whereas it gets fastest when the multiplexor is in a strictly dynamic context. In contrast, the multiplexing type does not make any discernible difference.



(a) Dynamicity. (b) Multiplexing type.

Fig. 16: Time required to generate multiplexed traces with respect to the dynamicity and the multiplexing type.



(a) Dynamicity. (b) Multiplexing type.

Fig. 17: Time required to translated multiplexed frames with respect to the dynamicity and the multiplexing type.

APPENDIX C
ALTERNATIVES TO CAN MULTIPLEXING

In the scope of our research, we have focused on exploring and developing techniques for the reverse engineering of CAN multiplexed traffic. However, it is important to acknowledge that there are other protocols that, similarly to frame multiplexing, also aim at increasing the traffic throughput on CAN networks. Two notable examples in this regard are CAN Flexible Data-Rate (CAN FD) – ISO11898-1:2015 [18] – and CAN Transport Protocol (CANtp) –AUTOSAR CP R21-11 [8].

CAN FD is an extension of the traditional CAN protocol and introduces higher data rates, larger frame (up to 64 bytes), and enhanced error handling capabilities. CANtp is designed primarily for applications requiring deterministic, time-critical communication. It facilitates the transmission of larger data payloads by segmenting and reassembling them efficiently.

In future work, we aim at comparing CAN multiplexing with CAN FD and CANtp. More specifically, our goal is to evaluate how these protocols affect reverse engineering and security.