# *SmrtComp*: Intelligent and Online CAN Data Compression

Dipika Tanwar[†], Priyanka Singla[‡], Soham Nag[*], Vireshwar Kumar[†] and Smruti R. Sarangi[†]

***Abstract*—In modern vehicles, efficiently storing CAN bus data is crucial for on-board diagnostics, performance monitoring, analysis, and the investigation of failures and accidents. In past work, an EDR (Event Data Recorder) akin to the ones in aircrafts has been mooted. The state-of-the-art in this field comprises proposals that propose efficient lossless compression of CAN data for such analyses – this limits the data storage capacity. Our contribution *SmrtComp* aims to achieve a much larger storage efficiency by storing recent data in a lossless format and compressing older data in increasingly lossy formats. The system tries to approximately adhere to an expected accuracy vs time curve (a QoS metric) that is specified a priori. Our study evaluates *SmrtComp*'s performance using various metrics such as the compression gain (CG), root mean square error (RMSE), total data storage and the preservation of key features. *SmrtComp* was implemented on an ARM Beaglebone board; it was used to store realistic traces and synthetic traces obtained from CAN bus simulators. We achieve a 3.2× higher data storage efficiency as compared to the closest competing work and outperform a popular lossy algorithm by 94.33% in terms of the RMSE. *SmrtComp* achieves line-speed compression of CAN bus data making it a promising solution for managing large data volumes, and we also show that our compression method preserves anomalies. To the best of our knowledge, this is the first hybrid and tunable compression system in this domain.**

***Index Terms*—Data compression, CAN bus, Embedded system, Event data recorder, Memory efficiency**

## I. Introduction

The automotive industry is seeing a move towards more connected and intelligent cars [1]. A modern car has a network of over 100 electronic control units (ECUs) interconnected through a controller area network (CAN) bus [1]. These ECUs collect and analyze data from vehicle sensors generating significant data traffic on the CAN bus [2]. Thus, akin to aircraft, there are proposals to record this information using an EDR [3] (Event Data Recorder) for various purposes such as diagnosing vehicle issues, real-time monitoring, scheduling maintenance, studying component behavior (particularly electric vehicle batteries) and analyzing driving patterns. Notably, EDRs hold legally cognizable data, particularly in accidents involving self-driving or automated systems.

There are predominantly two kinds of EDRs proposed in the literature. The first category captures video data using on-board cameras [3], while the second kind logs CAN bus data [4], [5]. The former retains video recordings for a short period, typically about 30 seconds [6], sufficient for accident reconstruction. Video compression technologies enable efficient storage but are not suitable for long-term analysis of driving patterns and identification of abnormal behavior – critical in legal cases.

ECUs can generate data at a rate of 500 kbps on the CAN bus [2], which poses challenges for analysis without adequate processing. Effective techniques are required to manage ECU data inflow and efficient storage (compressed or uncompressed). This area is sparse. Hence, in this paper, our focus is on the second type of EDRs that log CAN bus data. We propose an improved CAN bus data logger that surpasses state-of-the-art technology. This research area has gained momentum due to the advent of automated driving, electric vehicles (EVs), and hydrogen-based vehicles that are heavily reliant on electronic systems [7].

### A. Motivation

Before designing an efficient solution for logging CAN bus data, it is crucial to understand the significance of storing such data and the inadequacies of traditional storage methods. The intention is to propose a novel and effective CAN bus data logger that can overcome the existing challenges and offer better performance in terms of data storage and processing speed without losing the *key* features of the logged data.

❶ ***Importance of Data Storage***:

Due to the growing complexity of ECUs a significant amount of data is generated in modern cars (around 25 GB per hour [8]), which necessitates efficient storage solutions. Efficient ECU log storage enables real-time analysis, issue diagnosis, and long-term benefits such as enhanced vehicle performance, reliability, safety, and reduced maintenance costs and downtime. Of late cyber attack detection and courtroom evidence have emerged as important use cases.

In the event of an ECU failure, recorded data helps identify the root cause [9]. Vehicles may employ selective logging to minimize data storage by capturing specific data relevant to expected faults like the notion of the *frozen frame* [9]. While selective logging partially solves the problem, it relies on the predictability of faults, which may seldom be very realistic.

Coming to attacks, CAN injection is a popular cyber attack where unauthorized entities maliciously transmit messages through a vehicle's CAN bus by illegitimately getting access to it. The lack of authentication and encryption in the CAN bus protocol allows attackers to send false or unauthorized messages, potentially resulting in unintended actions such as vehicle theft [9]. Detecting such attacks again requires monitoring ECU data and analyzing all CAN message traffic for abnormal patterns or unexpected behavior [10].

[†]Dipika Tanwar, Vireshwar Kumar, and Smruti R. Sarangi are with the Department of Computer Science and Engineering, IIT Delhi, New Delhi, India, {dipika.mcs20, viresh, srsarangi}@cse.iitd.ac.in

[‡]Priyanka Singla is with the School of Information Technology, IIT Delhi, New Delhi, India, priyanka@cse.iitd.ac.in

[*]Soham Nag is with the Center of Excellence in Cyber Systems and Information Assurance, IIT Delhi, New Delhi, India, jcs222663@csia.iitd.ac.in

Globally, courts have recognized recorded CAN bus data as admissible evidence when proven to be reliable and tamper-proof [11]. This aids in accident reconstruction and identifying abnormal driving behavior. However, we often need more than 250 hours of data for proving that a given driving pattern was *anomalous* and *inconsistent* [12]. The dictum is "more the data better it is" [13]. Furthermore, this information is also proving to be useful to insurance companies [14], because it facilitates anomalous pattern identification and compliance with driving standards.

The aforementioned case studies indicate the importance of the long-term storage of data. An EDR aids this, but due to memory capacity constraints, it can only store data for a limited time. As a result, there is a need to increase the effective capacity of an EDR [13]. Let us elaborate.

❷ ***Requirements in terms of nonvolatile memory***: Automotive components are exposed to harsh ambient conditions, including elevated temperatures and strong electromagnetic fields, necessitating the implementation of robust EDR storage systems capable of withstanding such extreme conditions.

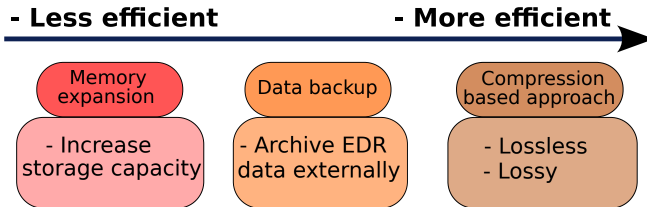


Fig. 1: EDR data storage solutions

Therefore, an EDR storage system must have a high endurance and also a fast write speed to accommodate the substantial volume of data generated, particularly in systems with frequent writes and many read-write cycles. Furthermore, it should possess resilience against security attacks [15].

F-RAM and automotive SD cards are preferred over traditional EEPROM (Electrically Erasable Programmable Read-Only Memory) and flash memory in vehicular environments [16], [17] due to their durability and superior performance [18]. For instance, F-RAMs offer faster write times (<50 ns) and a significantly higher PE (Program Erase) endurance (100 trillion cycles) compared to EEPROM's limited endurance (100,000 cycles). F-RAM devices operate at a lower voltage (1.5V), provide resistance against power analysis attacks, and exhibit good temperature resistance (up to 125°C). AEC-Q100 [19] Grade 3 automotive SD cards are a practical alternative to F-RAMs [20]. These are tailored for vehicular use, offering high temperature tolerance (-40 °C to 85°C), ECC and wear leveling [20], and reliable storage with high endurance (over 500,000 cycles). In our experiments we have used a functionally similar option.

Both F-RAMs and automotive SD cards have much lower storage capacities (typically KBs to 64/256 GB) as compared to standard flash. Hence, **efficient data compression techniques are very important** in this space.

❸ ***Solutions***: We will now explore potential methods for storing such massive data (see Figure 1). Expanding memory capacity is the natural solution to accommodate increasing data storage requirements. However, the high price of F-RAM makes this option highly unfeasible. While an 8 GB automotive SD card costs around 17 USD, an F-RAM with only 256 Kb capacity costs the same [21], [22]. In developing countries where cars are bought in the range of 7,000-12,000 USD [23], an F-RAM based EDR may account for 20-25 % of the total cost of the car. While automotive SD cards are cost-effective, the generation of substantial data volumes at short intervals [8] imposes significant storage capacity demands, resulting in substantial cumulative costs (an order of magnitude less than F-RAM though).

An alternative solution for storing large data volumes is transferring it periodically to an external drive like a disk drive or a solid-state drive (SSD) for retention, when the EDR memory reaches its maximum capacity. However, this approach increases the EDR cost and may have limited reliability in automotive environments [24], [25]. With numerous parameters recorded from over 100 ECUs, the memory quickly fills up, requiring frequent data retrieval and storage. Thus, this solution is neither feasible nor scalable. Also note that cloud-based storage is also not an option due to continuous internet connection requirements, associated costs, and privacy concerns.

Given the storage limitations and our preference for robustness, employing data compression techniques proves advantageous for storing more data without adding more storage. Traditional approaches use lossless compression, but for incident analyses, storing all data is unnecessary. Instead, a subset of the recorded data suffices to identify the root cause [26]. Therefore, recent research [27], [28] has focused on performing lossy compression and storing only the relevant signals. Building upon this concept, we propose *SmrtComp*- a highly efficient solution for storing CAN bus data. *SmrtComp* uses an intelligent and tunable algorithm to store recent data in a lossless format while older data is stored in a lossy format with varying degrees of compression, i.e., information loss. In addition, *SmrtComp* dynamically attempts to compress old data based on a pre-defined quality of service (QoS) metric, such as the expected accuracy versus time curve.

### *B. Contributions*

To the best of our knowledge, this is the ❶ first contribution in this space that stores CAN bus data in a hybrid fashion (lossless + lossy). ❷ Second, we propose a *novel* online compression algorithm based on principal component analysis (PCA) and the discrete cosine transform (DCT) where we can dynamically reduce the resolution of the data. ❸ Our tunable algorithm varies the degree of information loss in data without uncompressing it. ❹ The system is implemented on an ARM Beaglebone board; it can compress and store CAN data at line speed. ❺ The efficacy of our proposed algorithm can be seen from the results, which show that we can store 3.2× more data than the state-of-the-art algorithms while preserving most anomalies in the data.

The paper is organized as follows: we discuss related work in §II to provide the context for our proposed algorithm.

In §III, we provide the necessary background information followed by our proposed algorithm in §IV. We evaluate the performance of our algorithm in §V and and finally, conclude in §VI.

## II. Related Work

To the best of our knowledge, we are not aware of any hybrid and dynamically tunable compression algorithm for CAN data.

❶***Lossless compression*** References [4] and [5] present a GD (Generalized Deduplication)-based algorithm for lossless compression of CAN data. However, the compression ratios are not tunable; we can thus store data only for a fixed duration.

❷***Lossy compression*** Similar to *SmrtComp*, Yao et al. [27] propose a two-level lossy compression approach - $high$ and $low$. In contrast, our approach employs 10 compression levels, gradually reducing the data quality. Furthermore [27] focuses only on some *events of interest* (e.g., hard braking) and discards low-priority data, regardless of age. In contrast, *SmrtComp* considers age and priority while compressing, using a user-specific QoS curve. Havers et al. [29] use piecewise-linear approximations for lossy compression but do not leverage correlations in vehicular data. Khelifati et al. [28] exploit correlations for storing multivariate data but incur high overheads for dictionary creation and storage.

## III. Background

❶ ***CAN Protocol*:** CAN [30] is the de-facto standard for communication between ECUs in modern vehicles. The typical CAN bus bandwidth is 500 kbps [2].

When multiple ECUs concurrently try to send messages on the CAN bus, an arbitration based on message IDs is done to allow only one ECU. The process is equivalent to scanning the IDs of the concurrent messages from the left, and finding the index of the first 0 bit. The message with the lowest such index has the highest priority. For example, consider two ECUs, $ECU_1$ and $ECU_2$, with message IDS $(11010000001)_2$ and $(11100100011)_2$, respectively. $ECU_1$ is allowed to send because its first 0 bit is at index 3 (index 1 refers to the MSB position), while it is at index 4 for $ECU_2$.

❷ ***Principal Component Analysis (PCA)*:** PCA is a popular dimensionality reduction technique that projects $n$-dimensional data onto a transformed space using a few principal components. These components capture the majority of the data's variance. The process involves computing the covariance matrix, decomposing it to obtain eigenvalues and eigenvectors. The eigenvectors represent orthogonal directions with maximum variance, while the eigenvalues indicate the magnitude of variance. The principal components, defined by the eigenvectors, exhibit the highest variance.

❸ ***Discrete Cosine Transform (DCT)*:** DCT transforms the data by expressing it as a weighted sum of cosine functions with different frequencies. The weights (or coefficients) are indicative of the amount of information stored in the corresponding cosine functions. The lower-valued coefficients store less information, hence, they can be removed to reduce the data size without significantly affecting the data quality.

## IV. SmrtComp: An Intelligent Online Data Recorder

### A. Overview

***Ideal Recorder*:** Ideally, we would like to log the vehicle's data since it was last serviced. However, in practice, due to the recorder's limited storage space, such a large amount of data cannot be stored. In addition, lowering the cost of the recorder is essential to ensure widespread acceptance in the market while meeting cost constraints. Typically, an EDR can cost around 1000 USD (based on market estimates for similar products); hence, reducing the price will improve its acceptability and make it more appealing to potential buyers.

While lossless compression is ideal, memory efficiency can be achieved through lossy compression. However, accuracy is compromised in this trade-off. To strike the right balance, *recent* data is stored in a lossless format, while older data undergoes increasingly aggressive lossy compression over time (older data is compressed more). The inherent redundancy in CAN data, both temporal and spatial, makes it well-suited for this compression approach. Eventually, extremely old data can be discarded.

***Intelligent Recorder***: *SmrtComp*, intelligently takes compression decisions at runtime based on the amount of memory available for logging (log size) and recency of the data. Our goal is to maximize the amount of logged data (for a fixed log size) while still adhering to pre-decided QoS criteria.

### B. Frames and their Storage

Let us define a *signal* as a numeric representation of the state of some vehicular component. A single ECU may be responsible for monitoring multiple signals. Whenever an ECU senses a change, it collects all the relevant signals and generates a CAN bus data packet, which is transmitted on the bus. The recorder *snoops* it and stores it in a *buffer*. Along with this set of signals, it also stores the current time stamp and the latest values of all other signals as well (as required by the vehicle's user). Thus, a particular entry in the buffer comprises the values of all the signals at a particular timestamp (similar to [29]). Similarly, when a new message from any other component arrives, it is stored in the buffer as a separate entry. A collection of $k$ such entries is called a *frame*, and a collection of $l$ such frames is called a *macro frame*.

***Original Frame***: As shown in Figure 2(a), a frame can be represented as a $2D$ array with $m$ rows and $n$ columns, where each column corresponds to a specific signal. A row represents the state of all the signals at a given instant in time.

***Compressed Frame***: To perform lossless compression on the frame, we use the Deflate [31] algorithm. We use a combination of PCA and DCT for lossy compression because of their respective benefits.

PCA transforms the frame into a sequence of components with the first few having the most variance and hence capturing the maximum information. PCA then retains the first $n'$ components and removes the remaining components (Figure 2(b)). The value $n'$ is chosen such that the sum of the variance of these components is $\geq X\%$ of the

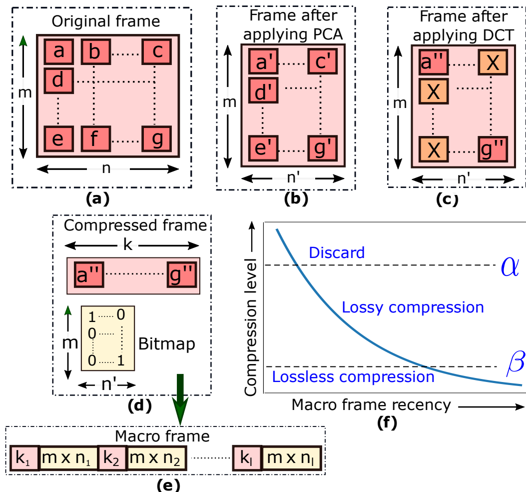


Fig. 2: **(a)** Original frame, **(b)** frame after applying PCA, **(c)** frame after applying DCT, **(d)** compressed frame with the corresponding bitmap, **(e)** macro frame and **(f)** function to determine the macro frame's compression level.

variance of the entire frame (with $n$ columns) [32]. We set $X = 100 - 0.1\times$ current compression level, so as to ensure a faithful reconstruction of all $n$ dimensions of data from $n'$ principal components. The *compression level*, in this context, refers to the extent of information loss.

Next, DCT transforms the updated frame and removes values smaller than a particular *threshold*. This is shown in Figure 2(c), where $\times$ indicates the removed values. The remaining values are stored in a row-major order to get the compressed frame. Figure 2(d) shows a compressed frame with $k$ entries.

***Storing the Frames***: The black box recorder buffers the incoming CAN data in its volatile memory, and when the number of entries equals the frame size, the frame is *finalized*. A frame's size is typically in kilobytes (KBs) (in our experiment, the frame size was set to 20KB); thus, writing a new frame repeatedly to the secondary storage device (automotive SD card) is expensive. So, we group frames into macro frames. A compressed macro frame (Figure 2(e)) is a list of compressed frames, wherein each frame is augmented with its bitmap (described next). A sequence of compressed macro frames *forms* the *log*.

***Frame Reconstruction***: Reconstruction requires three steps: (i) converting the compressed frame back to the matrix form (similar to Figure 2(c)), (ii) performing an inverse DCT, and (iii) performing an inverse PCA operation. We call the process of converting a compressed frame to its matrix form *frame reshaping*. Frame reshaping requires keeping track of indexes in the frame's $2D$ representation that have been removed during DCT. This can be done by maintaining a $2D$ *bitmap* (Figure 2(d)). A 0 in the bitmap indicates that the entry at the corresponding index was removed during the DCT transform, while a 1 indicates that the corresponding entries are still present in the compressed frame.

***Using the QoS Function to Determine Compression Levels***: Since the logged data loses its importance with time, it's a good idea to compress different macro frames with varying degrees of compression, which can be done using different *thresholds* in DCT (please see §IV-B). The varied compression levels would result in different error values. In particular, there is a direct correlation between error values and compression levels. So, we can use a *function*, $f(\mathrm{x})$, to control the amount of error. This function determines the macro frame's compression level (i.e., the degree of compression) based on its recency. In particular, it is a decreasing function with higher compression levels for older frames (as shown in Figure 2(f)). Similar to [33], we have used the exponential function ($f(x) = \lambda e^{-\lambda x}$) as our QoS function, where $x$ is the macro frame index (older frames have a lower index) and $\lambda$ is a parameter that is provided by the user. Please note that any decreasing function can be used instead of an exponential function.

Aggressively compressing very old macro frames, drastically reduces their quality, making it difficult to reconstruct them. Thus, it is wise to discard such frames. Similarly, preserving very recent frames with lossless compression is a good choice. The number of frames to be discarded/preserved (lossless) can be decided based on two user-provided hyper-parameters, $\alpha$, and $\beta$. The frames with compression levels greater than $\alpha$ are discarded, while new frames with compression levels less than $\beta$ are preserved without any loss. In our case, since we limit our lossy compression to 10 levels, we assign the values of $\alpha$ and $\beta$ as 10 and 1, respectively.

The selection of $\alpha$ and $\beta$ relies on user preferences for the trade-off between storage efficiency and quality. A higher $\alpha$ value permits quality reduction in older frames to save space, while a smaller $\beta$ value decreases the number of preserved lossless frames, sacrificing data quality for improved efficiency. Thus, these parameters should be chosen carefully based on applications' requirements and available resources.

TABLE I: Notations used in *SmrtComp*

| Notations | Definition |
|---|---|
| $Msg$ | Incoming CAN message. |
| $F_b, MF_b$ | Frame and macro frame buffers (lists), respectively. |
| $F_{size}$ | Max. no. of messages in a frame. |
| $M_{size}$ | Max. no. of frames in a macro frame. |
| $log$ | List of compressed macro frames. |
| $log_N$ | No. of macro frames in the log. |
| $log_{max}$ | Max. size of the log (in bytes). |
| $cl_i$ | Compression level for the $i^{th}$ macro frame. |
| $f(i)$ | Function to get the $cl_i$ of the macro frame. |
| $\alpha, \beta$ | Discard and lossless thresholds, respectively. |
| $F_r, F_c$ | Reshaped and compressed (lossy) frame, respectively. |
| $MF_c, MF_{c'}$ | Lossy compressed macro frames (list of $F_r$ or $F_c$). |

**Algorithm 1** Macro Frame Creation

1: **procedure** CONSTRUCT_MF($Msg$)
2: $F_b.add(Msg)$
3: **if** $F_b.size() = F_{size}$ **then**
4: $MF_b.add(compress_{Lossless}(F_b))$
5: $F_b \leftarrow \{\}$
6: **end if**
7: **if** $MF_b.size() = M_{size}$ **then**
8: $log(MF_b)$
9: $MF_b \leftarrow \{\}$
10: **end if**
11: **end procedure**

## C. Compression Algorithm

Our proposed approach *SmrtComp* comprises two processes: (i) macro frame creation and (ii) log compression. The first process handles incoming CAN data and creates macro frames to be stored in the log (secondary storage). The second process continuously monitors the log in real time and recompresses them iteratively to make room for new incoming macro frames. Algorithms 1 and 2 present the pseudo codes for these processes, and Table I summarizes the notations used in them.

***Algorithm 1 - Macro Frame Creation***: This process maintains a list ($F_b$) to store the incoming CAN data. Whenever a new CAN message (packet) arrives, it is appended to $F_b$ until the number of entries in $F_b$ equals the frame size, $F_{size}$ (Lines 2-3). A lossless compression is performed on this frame (using the function $compress_{Lossless}()$), and the compressed frame is added to a macro frame buffer, $MF_b$ (Line 4). $F_b$ is cleared to store new data (Line 5). When the size of a macro frame buffer ($MF_b$) reaches the macro frame size ($M_{size}$), the data is logged to the secondary storage and the macro frame buffer is emptied (Lines 7-9).

***Algorithm 2 - Log Compression***: This process monitors the instantaneous size of the log. When the size exceeds the maximum log size, $log_{max}$ (Line 2), all the macro frames in the log are recompressed (Line 3). The compression level is determined using the *function* $f(i)$ (Line 4), which gives higher compression levels for older macro frames and lower compression levels for newer macro frames. The macro frames with compression levels greater than $\alpha$ are discarded (Lines 5-6), while the macro frames with compression levels greater than $\beta$ are recompressed using the procedure $Compress_MF()$, and are replaced in the log (Line 8).

**Algorithm 2** Log Compression

```
1:  procedure COMPRESS_LOG(log, log_max, f(i), α, β)
2:      if sizeof(log) ≥ log_max then
3:          for i ← 1, log_N do
4:              cl_i ← f(i)
5:              if cl_i ≥ α then          ▷ discard i^th macro frame
6:                  log.remove(i)
7:              else if cl_i ≥ β then     ▷ recompress log(i)
8:                  log(i) ← Compress_MF(log(i), cl_i)
9:              end if
10:         end for
11:     end if
12: end procedure
13: procedure COMPRESS_MF(MF_c, cl)
14:     MF_c' ← {}
15:     for all F ∈ MF_c do
16:         F_r ← reshape(F)
17:         F_c ← compress_Lossy(F_r, cl)
18:         MF_c'.add(F_c)
19:     end for
20:     return MF_c'
21: end procedure
```

At a high level, recompressing a macro frame necessitates reconstructing the macro frame and then compressing it as per a specified compression level. Reconstruction, however, involves three steps: frame reshaping (see §IV-B), inverse DCT, and inverse PCA, making it an expensive operation. We observed that during recompression, as the new compression level is higher than the previous, the new DCT threshold (for a macro frame) would also be higher. So, instead of uncompressing the frame, we can simply *reshape and compress* it using the new DCT thresholds, thus **considerably reducing** the overheads. Please note that reshaping is sufficient when we recompress a lossy frame. To create a lossy frame from a lossless one, we must first uncompress the frame and then compress it. This, however, is done only once for a frame.

$Compress_MF()$ compresses a macro frame based on this **insight**. It sequentially iterates over all the frames in the macro frame, and every frame is reshaped (using $reshape()$) and compressed (using $compress_{Lossy}()$) according to the new compression levels (Lines 15-17). The compressed frame, $F_c$, is appended to a new compressed macro frame $MF_{c'}$ (Line 18), which is returned to the $compress_log$ procedure (Line 20). Please note that only for a lossless frame, the $reshape()$ function will uncompress the frame.

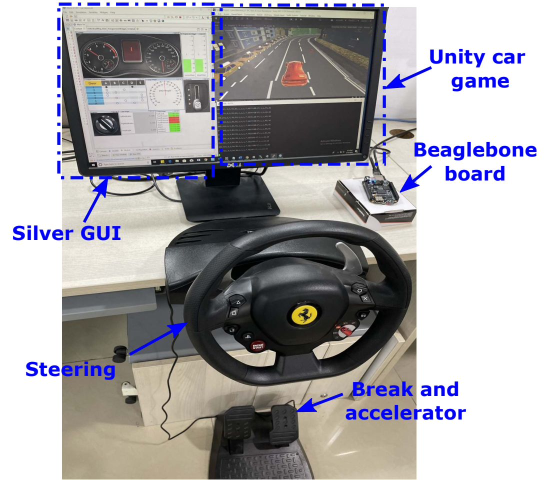

Fig. 3: Simulation setup

# V. Experiments and Results

## A. Experimental Setup

To evaluate *SmrtComp*'s resilience and adaptability, we utilized three distinct datasets: one from a previous study [34], synthetic data resembling autonomous driving scenarios generated with the Carla simulator, and data captured from a Unity car game, encompassing diverse manual driving behaviors, including normal and aggressive driving styles. Figure 3 shows the setup used for collecting data and performing all our experiments.

*1) Software Setup:* We use multiple software: (i) Synopsys Silver [35] (a virtual ECU platform), (ii) Unity car game, and (iii) Carla simulator [36]. We used a Windows 10 system with 32 GB RAM and a 6-core Intel® Core™ i7-8700 processor running at 3.2 GHz.

***Synopsys Silver***: It is an industry-standard tool that simulates virtual ECUs, which monitor the various components of a car. When a component's state changes, the corresponding ECU senses the change and generates a CAN message that

contains this information. The Silver GUI (shown in Figure 3) provides knobs to change various parameters (e.g., speed, gear, etc). In addition to the Silver GUI, the component states can be changed via a Python script. The final output is a set of CAN bus messages.

***Unity Car Game***: This is used for capturing realistic driving behavior. The car game is interfaced with equipment to control the steering, acceleration, and brakes ($T80$ Thrustmaster $488$ GTB Edition). We employ a socket-based communication channel to connect the car game to Silver, allowing real-time input control from the user, driving the car in the game.

***Carla Simulator***: It is an open-source simulator for autonomous driving research. In contrast to the car game that captures human driving, Carla implements an in-house autonomous driving algorithm. The driving decisions are communicated to Silver to generate the CAN bus traces.

*2) Hardware components:* Synopsys Silver generates CAN bus data that is stored in our prototype system: a ***Beaglebone Black Board [37]*** with the *SmrtComp* software running on it. Beaglebone Black is an open-source single-board computer with a $1GHz$ ARM Cortex-A8 processor, 512 MB RAM, and 16 GB microSD card. This board is a good choice for a data recorder, given its present pricing and various configurations. On the board, we installed Ubuntu (version 5.10.100-ti-r38) and used Python 3.8 and GCC version 9.3.0 to implement the recorder. The recorder is considered to have $4$ GB of storage for the logs. Owing to price considerations, we did not opt for an F-RAM based solution.

### B. Datasets

We utilized the ***Real*** dataset from [34], containing real-world driving data from 10 drivers over 23 hours, and generated additional data using our in-house setups. The Carla simulator provided the ***Autonomous*** dataset, while the Unity car game captured the ***Normal*** and ***Rash*** datasets.

1) *Normal driving*: The behavior in a typical driving scenario is emulated with a legal driving speed and a few lane changes.
2) *Rash driving*: In this, the lanes are changed more often and the acceleration pedal is pressed more eagerly.

We sent all the traces to Synopsys Silver, and generated CAN bus data, which was streamed to the EDR via a socket-based channel in real time.

### C. Performance Metrics

Given that we do not have any direct competitors, we show the efficacy of our proposed algorithm by comparing it against the best algorithms at the two ends of the memory-accuracy spectrum, i.e., the best lossless and lossy compression algorithms. To choose the best lossless (or lossy) algorithm, we performed an extensive experimental analysis and compared the *compression gain* and *RMSE* (root mean square error) of different algorithms. The compression gain (CG) is the ratio of the original size of the entire log to its compressed size (i.e., $CG = original_size/compressed_size$).

### D. Experimental Results

❶ ***Choosing the best lossless algorithm***: Figure 4 compares various state-of-the-art lossless algorithms: Deflate, BZip2, Lempel-Ziv-Markov (LZMA) and Lempel-Ziv-Storer-Szymanski (LZSS). The results demonstrate Deflate's superiority, indicated by the shortest compression time and largest compression gain across all datasets. We also compared Deflate to a recent work by Yazdani et al. [5], which employs a combination of the Generalized Deduplication (GD) algorithm with a dictionary. Deflate exhibited a superior compression gain over GD without a dictionary, while GD with a dictionary showed slightly better performance. However, GD with a dictionary incurred a $1.4\times$ higher decompression time and an additional dictionary storage overhead, outweighing its marginal improvement over Deflate.

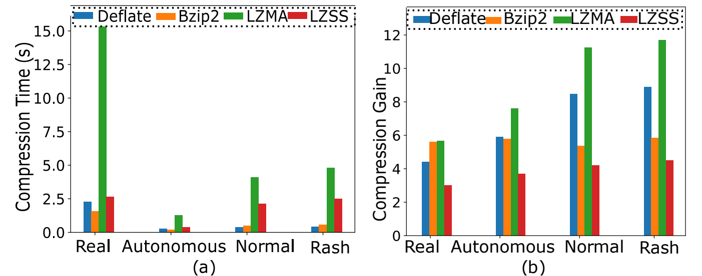


Fig. 4: Comparison between various lossless algorithms **(a)** compression time and **(b)** compression gain

❷ ***Choosing the best lossy algorithm***:

Storage efficiency requires a high CG; however, Figure 5(a) highlights that a high CG leads to increased RMSE impacting data accuracy. The simultaneous pursuit of a high CG and minimized RMSE is crucial. We evaluated several transformation-based lossy algorithms: Fast Fourier Transform (FFT), Discrete Wavelet Transform (DWT), Discrete Cosine Transform (DCT), Fast Hankel Transform (FHT), and Hilbert transform, focusing on their CG vs. RMSE performance. As seen in Figure 5(a), DCT outperformed the others, exhibiting both the highest CG and the lowest RMSE. Due to space constraints, we have only shown findings for the *autonomous* dataset; other datasets behaved similarly.

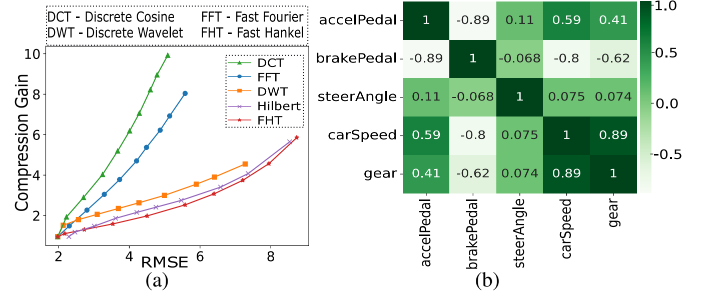


Fig. 5: **(a)** Compression gain vs RMSE for different lossy algorithms and **(b)** correlation of various signals between themselves

Apart from compressing the CAN data, we also studied the correlations between its various signals and found that several of them are highly correlated. For example, the *carSpeed* and *gear* signals in the heatmap (Figure 5(b)). Thus, we can conclude that we can compress CAN data using PCA.

❸ ***Performance evaluation of* SmrtComp**: Figure 6 compares - *SmrtComp* to the best lossless (Deflate) and lossy

(*PCA-DCT*) algorithms. Along with *SmrtComp*, we also implemented the other two algorithms on the Beaglebone board. All three algorithms see the same number of macro frames at every time instance. The experiments are performed with a frame size of 20 KB, a macro frame size of 32 MB, and a log size of 4 GB. The data rate of CAN is 500 kbps.

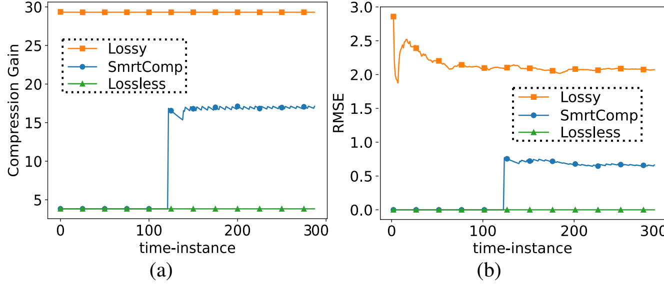


Fig. 6: Comparison of the three approaches in terms of mean **(a)** compression gain and **(b)** RMSE

Based on the findings in Figure 6, it is apparent that the lossy algorithm yields the highest compression gain. However, this comes at the expense of a substantially high RMSE. Conversely, the lossless algorithm demonstrates the lowest RMSE but results in poor compression gain. Therefore, it is imperative to consider both the compression gain and RMSE while evaluating compression algorithms. Accordingly, as indicated in Figure 6, *SmrtComp* delivers a desirable (and tunable) outcome in the middle of the spectrum by effectively balancing the compression gain and RMSE.

Apart from the compression gain (CG) and RMSE, the number of macro frames stored is an important metric, as it quantifies the amount of *information* captured in the log. Considering the log size of 4 GB, *SmrtComp* was able to store more than 400 macro frames, while the lossless algorithm could store only 128 macro frames. In general, considering the CAN bus data rate of 500 kbps [2], *SmrtComp* fills the 4 GB log in ten days, whereas the lossless approach fills it in three days.

In contrast to the lossless algorithm, the lossy algorithm supports storing a large number of macro frames (around 800) but at a higher RMSE cost. A large RMSE value indicates high information loss, making the highly compressed data unusable. *SmrtComp*, on the other hand, maintains the data quality. Figure 7 (a) illustrates that we were able to accurately reconstruct the original *acceleratorPedal* signal by using *SmrtComp*'s compressed data. *SmrtComp* avoids the aforementioned data quality loss issue by adjusting the compression levels of the frames based on the *QoS* function (please see §IV-B).

❹ ***Assessing* SmrtComp*'s Accuracy and Efficacy***: Having evaluated *SmrtComp*'s performance, it is necessary to look at the preservation of vital features in the lossy compressed data, notably anomalies indicating potential accident-related aspects. We detected anomalies in the original data and tried to find if they still remain with different levels of compression. The reliable Isolation Forest algorithm [38], adept at detecting anomalies in high-dimensional time series data, was employed for this purpose. Hyperparameter tuning was performed using GridsearchCV, setting n_estimators $= 100,000$

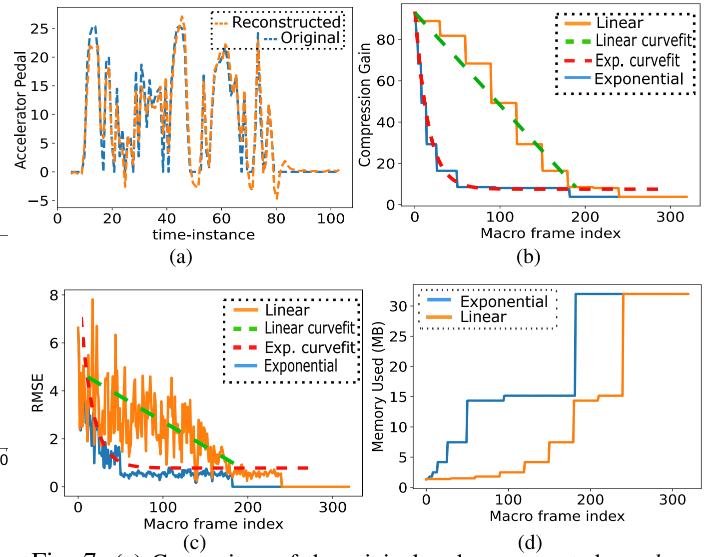


Fig. 7: **(a)** Comparison of the original and reconstructed *acceleration pedal* signal and **(b-d)** distributions of different metrics for different QoS functions. **(b)** compression gain, **(c)** RMSE and **(d)** memory consumption (in MBs)

(number of isolation trees), contamination $= 0.01$ (expected proportion of anomalies), and random_state $= 42$ (seed for reproducibility), ensuring the model's generalizability to new data and avoiding overfitting.

Using our machine learning model, we detected anomalies in the uncompressed data followed by decompression of the lossy data at varying levels. Each decompressed dataset underwent anomaly detection with the same model to assess the presence of original anomalies. Experimental findings showed misclassification rates ranging from $15.1\%$ (compression level = 1) to $29.8\%$ (compression level = 10), indicating increased vulnerability to misclassification with higher lossy compression. Table II summarizes the misclassification percentages corresponding to each compression level. *SmrtComp* achieved an average misclassification rate of $24.37\%$ below the acceptable threshold [39]. This demonstrates its efficacy in preserving critical anomaly information while reducing data size, making it a valuable data compression tool. Between levels 5 and 10, we achieved an $80\%$ space gain with a mere $15\%$ misclassification increase. However, beyond level 10 (specifically levels 11 to 15), misclassification surged to $40\%$. This defeats the purpose of storing CAN bus data, thereby restricting lossy compression to 10 levels.

❺**SmrtComp*'s flexible QoS function***: For our experiments, we used the *exponential* QoS function($f(x) = \lambda e^{-\lambda x}$) is a tunable parameter and $x$ is the macro frame index, which is indicative of the age of the frame.

To demonstrate *SmrtComp*'s flexibility, we also experimented with a *linear* QoS function, $f(x) = a * x + b$ where $a$ and $b$ are the tunable parameters. The compression gain, RMSE, and memory consumption (in the log) for both cases are shown in Figures 7(b-d). The graphs depict how the metrics vary across different macro frames. The dashed lines (curve-fits) in Figure 7(c) clearly show that the error curve is in accordance with the compression gain (Figure 7(b)), which is proportional to the QoS values ($f(x)$) provided by

the user.

TABLE II: Misclassification rates at different compression levels

| Lossy Compression Levels | Misclassification (%) at each level | | | | |
|---|---|---|---|---|---|
| 1-5 | 15.1 | 19.4 | 21.6 | 22.4 | 25.5 |
| 6-10 | 26.1 | 27.2 | 28.3 | 28.3 | 29.8 |
| 11-15 | 31.8 | 33.2 | 35.3 | 38.2 | 40.4 |

❻**SmrtComp*'s overhead***: The time taken by *SmrtComp* to recompress all the macro frames (in the log) was compared to the macro frames' inter-arrival time. Ideally, the recompression should finish before a new macro frame reaches the log. The minimum inter-arrival time for 32 MB CAN macro frames is roughly 34 minutes (assuming that the practical maximum CAN data rate is 500 kbps [2]). *SmrtComp* recompresses the entire log in $\sim 4$ minutes, thus ensuring a minimal overhead.

## VI. Conclusion

In this paper, we showed the feasibility of implementing a hybrid compression scheme with CAN bus data. We employed our proposed scheme – *SmrtComp*– to implement an intelligent EDR. *SmrtComp* dynamically changes the quality and the size of the logged data while conforming to pre-specified QoS metrics. Our recorder is able to process CAN data at line speed (500 kbps) and can record data continuously for 10 days, considering a log size of 4 GB.

## References


[1] "Shifting toward software-defined vehicles," [Online] Available: https://semiengineering.com/shifting-toward-software-defined-vehicles/, Accessed on: May 16 2023.

[2] P. Park, H. Yim, H. Moon, and J. Jung, "An OSGi based in-vehicle gateway platform architecture for improved sensor extensibility and interoperability," in *33rd Annu. IEEE Int. Comput. Soft. and Appl. Conf.*, vol. 2. IEEE, 2009, pp. 140–147.

[3] R. Feng, Y. Yao, and E. Atkins, "Smart Black Box 2.0: Efficient High-Bandwidth Driving Data Collection Based on Video Anomalies," *Algorithms*, vol. 14, no. 2, p. 57, 2021.

[4] S. Vaucher, N. Yazdani, P. Felber, D. E. Lucani, and V. Schiavoni, "ZipLine: in-network compression at line speed," in *Proc. 16th Int. Conf. Emerg. Netw. Exp. Technol.*, 2020, pp. 399–405.

[5] N. Yazdani, L. Nielsen, and D. E. Lucani, "Memory-aware online compression of CAN bus data for future vehicular systems," in *IEEE Global Commun. Conf.(GLOBECOM)*. IEEE, 2020, pp. 1–6.

[6] "Video Event Data Recorder (VEDR)," [Online] Available: https://www.government-fleet.com/297532/video-event-data-recorder-vedr, Accessed on: May 16 2023.

[7] "How Do Fuel Cell Electric Vehicles Work Using Hydrogen?," [Online] Available: https://afdc.energy.gov/vehicles/how-do-fuel-cell-electric-cars-work, Accessed on: May 16 2023.

[8] "Your car is watching you. Who owns the data?," [Online] Available: https://rollcall.com/2019/04/09/your-car-is-watching-you-who-owns-the-data/, Accessed on: May 16 2023.

[9] Ken and Tindell, "CAN Injection: keyless car theft," [Online] Available: https://kentindell.github.io/2023/04/03/can-injection/, Accessed on: May 16 2023.

[10] M. Jedh, L. B. Othmane, N. Ahmed, and B. Bhargava, "Detection of message injection attacks onto the can bus using similarities of successive messages-sequence graphs," vol. 16. IEEE, 2021, pp. 4133–4146.

[11] "How Can Event Data Recorders (EDRs) Improve Your Case?," [Online] Available: https://www.smileylaw.com/blog/event-data-recorders-edrs-improve-your-case/, Accessed on: May 16 2023.

[12] Y. Qiu, T. Misu, and C. Busso, "Driving Anomaly Detection with Conditional Generative Adversarial Network using Physiological and CAN-Bus Data," in *IEEE Trans. Inf. Forensics Secur.*, vol. 16. ICMI, 2021, pp. 4133 – 4146.

[13] Int. Org. of Motor Vehi. Manuf.(OICA), "Event Data Recorder (EDR) and Data Storage System for Automated Driving (DSSAD)s," Tech. Rep., 2019.

[14] B. Smith and R. Miller, "The warranty process flow within the automotive industry: An investigation of automotive warranty processes and issues," *Center for Auto. Res.(CAR)*, 2005.

[15] L. Sharma, P. Chandankhede, and M. Khanapurkar, "Secured Event Data Recorder (EDR) System for Analysis of Data." IJSR, 2017.

[16] "Why Use F-RAM for Event Data Recorders?," [Online] Available: https://www.eetimes.com/why-use-f-ram-for-event-data-recorders/, Accessed on: May 16 2023.

[17] M. Alwais, "FRAM technology puts the "intelligence" in smart airbag systems," *ATZelektronik worldwide*, vol. 2, no. 2, pp. 11–12, 2007.

[18] "FRAM FAQs Texas Instruments," [Online] Available: https://www.ti.com/lit/ml/slat151/slat151.pdf?ts=1680089453794, Accessed on: May 16 2023.

[19] "Failure Mechanism Based Stress Test Qualification For Integrated Circuits," in *Automotive Electronics Council*, 2017.

[20] "SanDisk® Automotive SD™ Card Storage Solutions," [Online] Available: https://www.mouser.com/pdfdocs/Automotive_SD_ProdBrief.pdf, Accessed on: May 16 2023.

[21] "AVNET," [Online] Available: https://www.avnet.com/, Accessed on: May 16 2023.

[22] "Future Electronics," [Online] Available: https://www.futureelectronics.com/, Accessed on: May 16 2023.

[23] CARS24, "The Mileage Report by CARS24 for 2022-2023," Tech. Rep., 2023.

[24] "CollisionData 2023," [Online] Available: https://www.collisiondata.com/, Accessed on: May 16 2023.

[25] "InsideEVs 2021," [Online] Available: https://insideevs.com/news/503245/tesla-event-data-recorder-information/, Accessed on: May 16 2023.

[26] "Deadly Tesla crash was just pedal confusion, despite social media rumors," [Online] Available: https://electrek.co/2023/03/02/deadly-tesla-crash-was-just-pedal-confusion-despite-social-media-rumors/, Accessed on: May 16 2023.

[27] Y. Yao and E. Atkins, "The Smart Black Box: A Value-Driven Automotive Event Data Recorder," in *Proc. IEEE 21th Int. Conf. Intell. Transp. Syst. (ITSC)*, 2018, pp. 973–978.

[28] A. Khelifati, M. Khayati, and P. Cudré-Mauroux, "CORAD: Correlation-Aware Compression of Massive Time Series using Sparse Dictionary Coding," in *IEEE Int. Conf. on Big Data*. IEEE, 2019, pp. 2289–2298.

[29] B. Havers, R. Duvignau, H. Najdataei, V. Gulisano, M. Papatriantafilou, and A. C. Koppisetty, "Driven: A framework for efficient data retrieval and clustering in vehicular networks," *Future Gener. Comput. Syst.(FGCS)*, vol. 107, pp. 1–17, 2020.

[30] C. P. Szydlowski, "Can specification 2.0: Protocol and implementations," SAE Technical Paper 921603, Tech. Rep., 1992.

[31] P. Deutsch, "RFC1951: Deflate compressed data format specification version 1.3," Tech. Rep., 1996.

[32] M. R. Chowdhury, S. Tripathi, and S. De, "Adaptive Multivariate Data Compression in Smart Metering Internet of Things," *IEEE Trans. on Industr. Inform.*, vol. 17, no. 2, pp. 1287–1297, 2021.

[33] Y. An, Y. Su, Y. Zhu, and J. Wang, "TVStore: Automatically bounding time series storage via Time-Varying compression," in *20th USENIX Conf. on File and Storage Technol.(FAST 22)*, 2022, pp. 83–100.

[34] B. I. Kwak, J. Woo, and H. K. Kim, "Know Your Master: Driver Profiling-based Anti-theft Method," in *Proc. 14th Annu. Conf. on Priv., Secu. and Trus. (PST)*, 2016, pp. 211–218.

[35] R. Linssen, F. Uphaus, and J. Mauss, "Simulation of Networked ECUs for Drivability Calibration," *ATZ-Elektron. worldw.*, vol. 11, no. 4, pp. 16–21, 2016.

[36] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An Open Urban Driving Simulator," in *Conference on robot learning*, vol. 78. PMLR, 2017, pp. 1–16.

[37] G. Coley, "Beaglebone Black System Reference Manual," *TI, Dallas*, vol. 5, no. 2, pp. 1–108, 2013.

[38] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest." Eighth IEEE International Conference on Data Mining, 2008, pp. 413–422.

[39] S. Ucar, B. Hoh, and K. Oguchi, "Abnormal driving behavior detection system," in *2021 IEEE 93rd Vehi. Tech. Conf. (VTC2021-Spring)*. IEEE, 2021, pp. 1–6.