IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Digital Object Identifier

Toward In-Vehicle Network Digital Twins: Timing Accuracy of CAN Traffic Reproduction Using Industry-Grade Interfaces and API-Based Solutions on Windows IoT Enterprise

EMILIA CARAGEA[1], BOGDAN GROZA[2]
[1]Faculty of Automatics and Computers, Politehnica University of Timisoara, Timişoara, 300223 Timiş, Romania, e-mail: emilia.caragea@student.upt.ro, bogdan.groza@aut.upt.ro

Corresponding author: Bogdan Groza (e-mail: bogdan.groza@aut.upt.ro)

Acknowledgement. This research was supported by the project "Romanian Hub for Artificial Intelligence - HRIA", Smart Growth, Digitization and Financial Instruments Program, 2021-2027, MySMIS no. 351416.

ABSTRACT Realistic experimental evaluation of time-sensitive communication mechanisms and intrusion detection systems that exploit temporal properties of CAN traffic requires precise control over CAN frame transmission timing, as variations in frame arrival times can critically affect both system behaviour and anomaly detection accuracy. However, in development environments, test engineers, hardware-in-the-loop specialists, and researchers rely heavily on toolchains native to Windows platforms, which lack real-time guarantees. This creates a practical challenge in constructing digital twins of in-vehicle network traffic. In this work, we compare the timing behaviour of production-grade Vector interfaces operating within their proprietary CANoe environment against a trace-driven scheduler implemented in C++ and Python using the Vector XL Driver Library. Replay accuracy is mainly characterized using the timing absolute deviation at the trace level, complemented by analyses of jitter, ID switching ratio, deadline miss ratio, and longest burst miss. We compare Python, an interpreted language, and C++, a compiled language, as representative endpoints of contemporary programming paradigms. In addition, we contrast a standard Windows 11 distribution with Windows IoT Enterprise, which is designed for embedded and industrial applications. Finally, our trace-driven scheduler is evaluated against a high-end reference platform comprising Vector Breakout Boxes D62Y9, connected to dedicated PCI Express acquisition cards, as well as against the more practical to use Vector VN5610A, connected on a regular USB interface, both operating within the native CANoe environment. The results highlight the practical limits of user-space scheduling on general-purpose operating systems and provide quantitative insight into the trade-offs between programming language choice, operating system configuration, and achievable timing precision.

INDEX TERMS CAN bus, digital twins, inter-arrival time, scheduler, soft real-time, timing analysis, trace replay, Vector XL Driver, Windows IoT

I. INTRODUCTION

As CAN traffic becomes increasingly complex due to the tighter integration of sensing, actuation, and software-defined functionalities, the ability to capture and replay network traces with high temporal fidelity has become indispensable. State-of-the-art research in real-time scheduling across embedded-system domains and IoT consistently emphasises that timing correctness is as critical as functional correctness. Real-time schedulers must manage periodic, sporadic, and event-driven tasks with guarantees on deadlines, latency, and jitter. Systematic reviews of real-time scheduling literature highlight that timing constraints such as execution time, latency, and deadline satisfaction remain the dominant evaluation metrics in distributed real-time environments [1]. In automotive CAN buses, from a networking perspective, these same metrics ultimately translate into message transmission or reception intervals.

Concretely, with respect to in-vehicle networks, the mo-

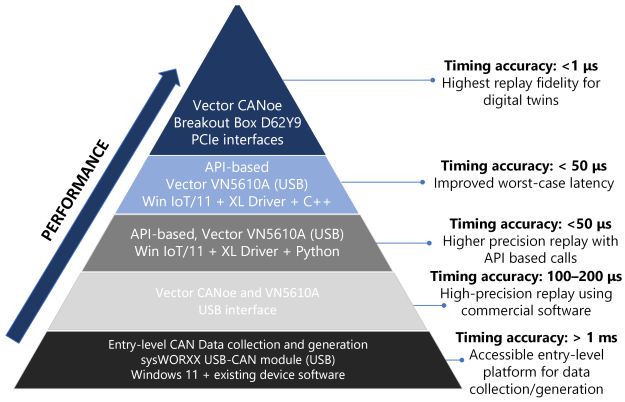


FIGURE 1: Hierarchy of observed mean timing accuracies across various CAN traffic reproduction platforms from our experiments

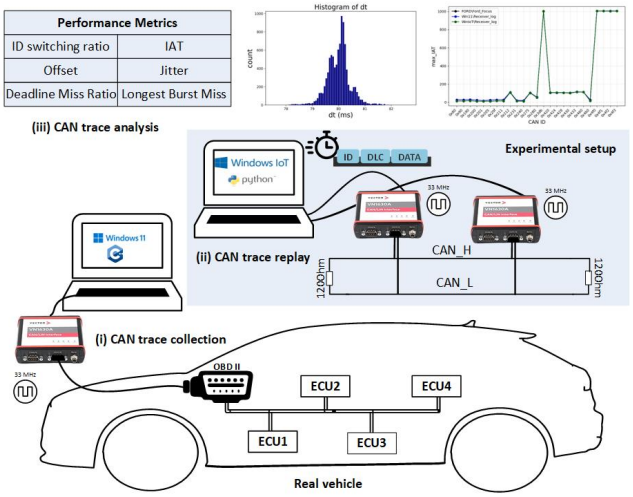


FIGURE 2: Setup and procedures: (i) CAN trace collection, (ii) CAN trace replay with scheduler in Python or C++ on Windows IoT, (iii) CAN trace analysis based on performance metrics.

tivation for pursuing strict timing characteristics is twofold. First, as the industry moves toward Software Defined Vehicles (SDV), recent architectural paradigms place Time-Sensitive Networking (TSN) at their core. While the IEEE 802.1 family of TSN standards is intended for Ethernet-based communication, CAN buses are likely to coexist with Ethernet in many modern vehicles. Currently, several studies have addressed time synchronisation on CAN buses, achieving microsecond-level accuracy with hardware-based [2], [3] or software-based approaches [4], [5], [6], [7], [8]. Second, reliable intrusion detection systems for in-vehicle CAN communication require development systems where timing semantics, i.e., message periodicity, inter-arrival time, and event-triggered latencies, are preserved, which is essential for network replication, system design, and validation. When evaluating intrusion-detection systems, or even Electronic Control Units (ECU) behaviour under high computational load, it is critical that the replay system reproduces not only the correct sequence of frames but also the exact inter-arrival patterns observed during a real attack. Even small replay-time distortions introduced by the host operating system may cause an IDS to trigger too early, fail to trigger at all, or mischaracterise the threat model, undermining the validity of the experiment. A specific application is the use of clock skews for detecting network intrusion [9], [10], [11], [12], [13] or for mapping ECUs from a CAN bus [14], [15], [16]. Subtle offset-drift attacks can clone the clock skew of a legitimate sender by a shift of a hundred microseconds or less, i.e., cloaking attacks [17]. Needless to say, a poorly designed experimental setup may lead to timestamp contamination, compromising the accuracy of the results.

However, achieving deterministic timing on commodity or general-purpose computing platforms is challenging. As illustrated in Figure 1, the timing accuracy achievable for CAN traffic reproduction spans several orders of magnitude, ranging from sub-microsecond deviations with industry-grade PCIe interfaces, to tens of microseconds with USB-based replay solutions and API-based implementations in Python and C++, to even larger deviations for low-end off-the-shelf devices. This hierarchy highlights the fundamental trade-off between timing fidelity and deployment cost. Our work proposes a minimal trace-driven scheduler architecture that offers an excellent cost-performance trade-off by leveraging API-based replay through Vector's XL Driver Library from C++ or Python applications to Vector VN interfaces. While such a software-based architecture does not achieve the sub-microsecond timing accuracy of dedicated PCIe industry-grade interfaces, it consistently limits timing deviations to a few tens of microseconds, even more competitive than those obtained with the commercial CANoe software suite. This enables faithful reproduction of CAN message logs as originally observed in an operational vehicle network, thereby supporting research on experimentation, design, and validation of automotive security mechanisms. By ensuring that both logical message content and temporal execution semantics are preserved, the scheduler contributes to improved reproducibility and reliability in automotive cybersecurity engineering. The overall setup and experimental procedures are depicted in Figure 2.

As timing fidelity is mission-critical for the methodology proposed in this work, the selection of the operating system for deployment is equally important. Real-time simulation of CAN and other IVN traffic requires an execution platform that minimises jitter, provides predictable task scheduling, and ensures limited non-essential interference. In such experimental settings, uncontrolled background activity or frequent platform changes can introduce timing variability that affects the validity and reproducibility of experimental results. Based on these considerations, this work evaluates two closed-source Microsoft platforms that represent different

deployment assumptions: Windows IoT Enterprise LTSC and Windows 11 Pro. Windows IoT Enterprise LTSC is, in particular, designed for fixed-function or special-purpose devices whose functionality is expected to remain stable throughout the device lifecycle. Windows IoT LTSC supports several features that improve soft real-time capabilities, including CPU isolation, Interrupt Service Routine/Deferred Procedure Call pinning, priority inheritance for mutexes, and real-time thread priorities. While these mechanisms are present in Windows 11 as well, several documentations from Microsoft emphasize the support for these in Windows IoT Enterprise [18], [19]. By contrast, Windows 11 serves as a general-purpose PC baseline platform. While both platforms share the same Windows kernel architecture and priority-based pre-emptive scheduling model, Windows 11 is primarily optimised for interactive desktop workloads rather than controlled embedded deployments, without an equivalent IoT-focused soft real-time workflow. To the best of our knowledge, there is no white paper from Microsoft that compares Windows 11 and IoT, in regards to soft real-time operation, therefore our work also provides some insights into this comparison. We did not consider Linux-based operating systems due to our dependence on Vector's hardware that is industry-grade and the XL Driver Library, which has no native support in Linux. Consequently, we arrived at Windows IoT Enterprise LTSC as an immediate choice. Linux PREEMPT_RT and SocketCAN remain a strong option for future exploration due to their support for hard and soft real-time computing.

In addition to the operating-system platform, the selection of programming language can influence timing behaviour in replay-based experiments. For the CAN trace replay scheduler implemented in this work, both C++ and Python were considered as candidate programming languages. C++ enables direct interaction with operating-system APIs, explicit memory management, and minimal runtime overhead, allowing timing-critical logic to be implemented with low execution latency and reduced scheduling jitter. By contrast, Python, while is widely used in experimental and data-analysis environments, relies on an interpreted runtime and automatic memory management, which can introduce additional execution variability through interpreter overhead and runtime scheduling effects. While prior studies have compared Python and C/C++ in embedded and real-time settings, these evaluations have largely been conducted on generic embedded [20] or robotic platforms [21]; to the best of our knowledge, comparable evidence for timing-sensitive automotive replay workloads remains limited.

The contributions of our work can be summarised as follows:

1) We implement a trace-driven scheduler capable of replaying collected in-vehicle CAN traffic using two programming environments: Python, an interpreted language, and C++, a compiled language. The implementation leverages industry-grade CAN interfaces from Vector Informatik and the Vector XL driver library.
2) We characterise performance using metrics at both the trace level (ID switching ratio, absolute median offset and timing absolute error) and the individual ID level (inter-arrival time, jitter, deadline miss ratio, and longest burst miss).
3) We deploy the scheduler on both Windows 11 and Windows IoT Enterprise to observe the effect of the operating system on network delay, allowing us to derive a clear comparison of the practical limits associated with the programming environment and the operating system, with and without the presence of adversarial activity.
4) We provide a comparison of the scheduler with a high-end industry solution, a Vector Breakout Box D62Y9 connected via PIC Express, and a regular VN5610A connected via USB, both of them operated from their native CANoe environment.

The rest of this paper is organised as follows. Section II reviews the related work. Section III outlines the main challenges related to timestamp accuracy in CAN traffic generation. Section IV describes the vehicle data collection procedure and the traffic characteristics relevant for scheduler design. Section V presents the scheduler design and its implementations in Python and C++. Section VI details the experimental setup for traffic reproduction. Section VII introduces the performance indicators, reports the results obtained under Windows 11 and Windows IoT, and discusses the main findings and their implications. Finally, in Section VIII, we conclude.

II. RELATED WORK

The rapidly growing field of CAN security research that seeks to detect intrusions and anomalies has produced a lot of timing and payload-based detection mechanisms. However, previous work has focused primarily on analysing CAN bus traffic and design detection algorithms, paying less attention to the quality of the datasets used for evaluation and to the experimental environments in which CAN traffic is generated, processed, or analysed. This gap is important because many of the CAN traffic characteristics are highly sensitive to timing distortions introduced outside the vehicle network itself.

The fact that non-real-time kernels introduce unpredictable latencies due to interrupt handling, deferred procedure calls (DPCs), and non-pre-emptive kernel sections is well-known. Experimental evaluations of widely used operating systems, such as Windows NT [22], have already proved this decades ago. These limitations lead to jitter and blocking that can compromise the precise replay of recorded timing traces. Recent surveys on IoT operating systems consistently identify scheduling behaviour, timing predictability, power-management mechanisms, and system overhead as major criteria in OS selection for constrained or time-sensitive applications. In particular, [23] highlights that energy efficiency and timing behaviour are strongly affected by timer models, sleep–state transitions, device power management, and coordination between the kernel and networking stack. The review in [1] emphasises latency, execution time, deadline, makespan and energy consumption as evaluation factors. Previ-

ous surveys [24] have emphasised architecture, scheduling, memory management, communication support, and power consumption as the main differentiators between IoT-oriented platforms.

The faithful reproduction of CAN traffic timing characteristics aligns with the concept of digital twins, as it enables the replication of the temporal behaviour observed in the physical system. Digital twins provide an architectural foundation for vehicle simulation, modelling, verification, and validation, and are key enablers of life-cycle-oriented vehicle development [25]. From a maturity perspective [26], the replication of assets, the connection of models and systems, and their synchronisation are the fundamental levels to achieve a digital shadow of the autonomous vehicle. At the CAN-network level, [27] derives ECU functionalities from models that are run in a laboratory setup constructed around a real-world in-vehicle wiring harness. The work in [28] proceeds toward the evaluation of CAN IDS through a digital-twin-based framework designed to generate more realistic attack traffic.

Prior work on timing-based intrusion detection systems relies on modelling CAN traffic as periodic, using deviations in inter-arrival time, cycle time, or frequency deviations as indicators of anomalies [29]. Representative examples include inter-arrival time differences of the same CAN ID to flag anomalies [30], frequency-based methods that monitor per-ID packet rates and inter-arrival time statistics [31], [32], [33], [34] and anomaly detectors that model frame interval arrival time deviations within fixed time windows [35]. However, as we will show later, real CAN communication is only nominally periodic: the frames exhibit deviations from strict cycle time and bounded jitter because of the high-computation ECU functions, different operating modes, arbitration, and retransmission delays.

In parallel, synthetic datasets [36], [37] have been introduced to alleviate data scarcity and support broader attack experimentation. However, analysis of currently available intrusion-detection datasets [38], [39] shows that such synthetic resources often exhibit unrealistic temporal patterns, reduced identifier coverage, and insufficient temporal fidelity for deployment-relevant evaluation. Consequently, temporal fidelity, real-time constraints, attack diversity, and validation remain key open challenges in synthetic CAN data generation, as does designing the tooling that preserves real-time traffic characteristics during experimentation and attack fabrication. Broadly speaking, these works suggest that digital-twin CAN experimentation is promising for IDS development, but its validity depends on the faithful reproduction of ECU interactions and timing fidelity of the CAN traffic. In the light of the previous works and based on the taxonomy from [26], the proposed trace-driven scheduler can be positioned as a digital shadow, a precursor to a full digital twin.

Scheduling has long been a central topic in automotive systems. Early works primarily focused on bus-level scheduling to improve bandwidth utilisation and reduce latencies on in-vehicle networks [40], [41] including security-aware scheduling strategies that account for the transmission of cryptographic material [42]. More recently, the emergence of Software-Defined Vehicles (SDVs) has considerably broadened the role of scheduling beyond in-vehicle communication to include edge-assisted [43], cloud-assisted [44], [45] and edge-cloud-assisted [46] computation. Specific applications of scheduling include efficient content delivery [47] and timely incident reporting in the Internet of Vehicles (IoV) [48]. The trace-driven scheduling that we address in this work may help in providing practical means of reproducing realistic communication workloads in SDVs under representative timing conditions.

III. CAN BUS TRAFFIC AND CHALLENGES WITH TIMESTAMP ACCURACY

In this section we discuss some background on the CAN bus traffic and give details on the traffic collected from a real vehicle that is compared to another trace from a public dataset.

A. OVERVIEW OF CAN BUS TRAFFIC

The Controller Area Network (CAN) is fundamentally built around a broadcast-oriented and cyclic communication paradigm. CAN frames are transmitted on a shared bus where every Electronic Control Unit (ECU) simultaneously receives all messages. Bus access is governed by identifier-based arbitration, where identifiers with lower numerical values have higher priority. Once a frame wins arbitration, it is transmitted in a non-pre-emptive manner and is observed by all ECUs. This shared-medium behaviour introduces strict timing interdependencies: a delay, advancement, or jitter in the transmission of any ECU directly affects the temporal context of all other messages on the network. The timing behaviour of CAN traffic is primarily determined by *message identifier* priority and the *Data Length Code (DLC)*. The identifier defines the frame's priority and determines the queuing of messages. The DLC specifies payload size and directly influences the physical transmission time through protocol overhead and bit-stuffing mechanisms. For example, in an 11-bit identifier format at 500 kbps, transmission times range from approximately 94μs for a remote frame to roughly 260μs for full 8-byte data frames [49]. These properties, combined with CAN's fixed-priority arbitration and non-pre-emptive transmission, yield well-defined latency bounds and interference patterns as formalised in [50]. At the system level, the vehicle communication matrix maps vehicle functions to periodic and event-triggered signals. Most ECUs transmit *messages at fixed periods* – commonly scheduled at 10ms, 20ms, 50ms, 100ms or 1000ms – depending on the update requirements of the underlying control loop. These periodic signals establish a time-triggered backbone, forming a repeating communication pattern with well-defined inter-arrival times and tightly bounded jitter. *Event-triggered frames* occur sporadically, often in response to driver activities or actuator events (e.g., switching lights on or off). These messages represent only a minor fraction of the total traffic and, due to their significantly

lower frequency, do not disrupt the regularity of the dominant cyclic communication schedule.

B. CHALLENGES WITH TIMESTAMP ACCURACY

When assessing the quality of the considered CAN datasets, several observations are relevant with respect to the validity of timing-dependent analyses, such as the design of intrusion detection methods.

First, missing [51], incomplete or duplicate timestamps [37] prevent verification of key temporal properties such as periodicity, jitter, and message loss, restricting analysis to signal correlations inside the data field. For CAN communication, where communication is strongly periodic, this represents a major shortcoming.

Second, very short inter-arrival times (IAT) between consecutive data frames, particularly below $100\mu s$, which are also present in [52] [53] [54], are relevant for both replay feasibility and attack detectability. Such strict timing requirements may be difficult to reproduce faithfully in non-real-time operating environments due to scheduler-induced latency, buffering, and software-based timestamp effects, e.g. performance counters with resolution above $100ns$ may not allow reproduction of CAN traffic at such accuracies. Consequently, attacks targeting periodic traffic, including periodic replay, are more likely to perturb characteristic IAT, jitter, and short-window bus load patterns. Therefore, detection may be based on simple timing-based algorithms that expose deviations from legitimate traffic behaviour, rather than on complex inference mechanisms.

Figure 3 visually illustrates the first (i) and second (ii) challenges related to duplicate timestamps and very short IATs, while (iii) and (iv) display the residual inter-arrival time, which we define as the difference between the observed inter-arrival time and the estimated transmission time of the preceding CAN frame based on its DLC. For a low bus utilization, sufficient spacing exists between frames, whereas a high bus load scenario shows back-to-back transmission with reduced idle time, close to hardware-level timing in a domain sometimes below $50\mu s$.

Third, low-end devices may also cause issues in the transmission and reception time. To highlight the importance of the selection of different CAN interfaces for time-dependent analysis, we evaluated the performance of a low-end CAN interface: a *sysWORXX USB-CAN module* [55], operated using the Systec USB-CANmodul Network Driver version 4.0.2.7 dated 23 September 2008, which is the original driver that came on a CD with the device. This was set to transmit periodic messages with nominal periods of $10ms$ and $100ms$. The reception of the messages was done with an industry-grade Vector VN1630 using the same application developed for in-vehicle data collection. For each message, the observed IAT time distribution was characterised by its relevant descriptive statistics: mean, standard deviation of the mean and median. For a cycle time of $10ms$ the measured IAT has a mean of $9.997ms$, a high standard deviation of $22.709ms$, and a low median of $0.278ms$ which are caused by the bursty nature of

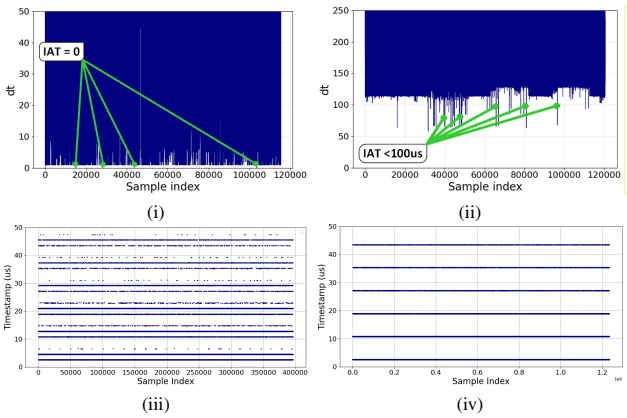


FIGURE 3: Representative timestamp anomalies in analysed CAN traces: (i) duplicate timestamps, (ii) sub-threshold inter-arrival times, and (iii) residual inter-arrival time for Ford Focus and (iv) Ford Kuga

TABLE 1: CAN periodic traffic generation statistics for a low-end off-the-shelf device

			IAT		
ID	T_{cycle} *ms*	**Size**	μ(IAT) *ms*	σ(IAT) *ms*	Med(IAT) *ms*
0x07	10	8B	9.997	22.709	0.278
0x07	100	8B	99.975	133.890	116.79

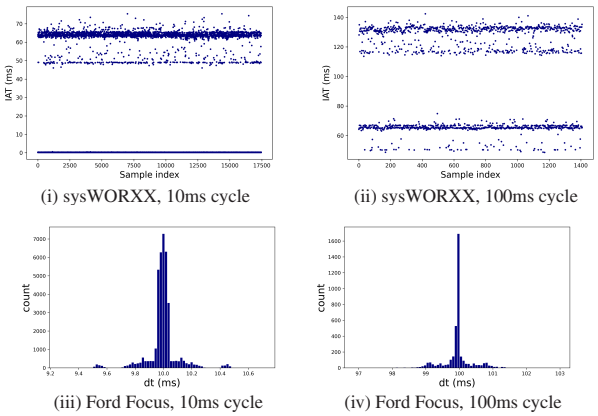
the transmission, where longer idle periods are followed by closely spaced frames (instead of the expected periodic transmission pattern). The same can be observed for the $100ms$ cycle with an IAT having a mean of 99.975 *ms*, a standard deviation of $133.890ms$, and a median of $116.79ms$. We do not know if a better driver or software can improve this as we could not identify a newer compatible driver. This result should not be interpreted as a general assessment of low-cost CAN interfaces, but serve as a cautionary example that older or low-cost USB-CAN interfaces may be unsuitable for timing-sensitive transmissions unless their timing behaviour is experimentally verified. The delays can be visualised in Figure 4, and the descriptive statistics are presented in Table 1 against two messages with the same period from the dataset that we collected from the Ford Focus.

Fourth, synthetically generated datasets as presented in some works [37] may not align with the more complex timing dependencies in the CAN bus traffic from real-world vehicles.

Collectively, these limitations motivate our development of a trace-driven scheduler, capable of reproducing realistic traffic dynamics, allowing controlled and repeatable experimentation.

IV. DATA COLLECTION AND TRAFFIC ANALYSIS

To begin with, we first use the Vector XL Driver Library with a VN1630 to collect data from a Ford Focus and analyse the collected data. This data will serve as a baseline in our



(i) sysWORXX, 10ms cycle (ii) sysWORXX, 100ms cycle

(iii) Ford Focus, 10ms cycle (iv) Ford Focus, 100ms cycle

FIGURE 4: Quality difference in CAN frames arrival time with a sysWORXX USB-CAN module device (i) 10ms cycle, (ii) 100ms cycle, and from a Ford Focus (iii) 10ms cycle, (iv) 100ms cycle.

experiment, and we use a more complex trace from a Ford Kuga coming a public dataset [16] in order to see how our scheduler scales up with more demanding traffic.

A. DATA COLLECTION

The trace was collected from a passenger vehicles via the OBD-II diagnostic port using a Vector VN1630 USB-to-CAN interface, a CAN cable, and a C++ logging application developed with the Vector XL Driver Library [56] running on a laptop with Windows 11 operating system as presented in Figure 5

The VN1630 hardware provides multiple galvanically isolated CAN channels and includes integrated high-speed CAN transceivers. For the data acquisition setup used in this paper, the physical connection to the vehicle's high-speed CAN network was established using the VN1630's built-in NXP TJA1051 high-speed CAN transceiver [57]. This transceiver is an ISO 11898-2 compliant high-speed CAN transceiver specially designed for automotive applications. For every received CAN frame, the system captured a nanosecond-resolution timestamp delivered by the CAN driver, the identifier (ID), the data length code (DLC), and the payload, all of which were stored for subsequent analysis.

B. TRAFFIC ANALYSIS FOR THE COLLECTED DATA

Traffic evaluation has been performed on two traces. One own collected dataset from a Ford Focus (MY 2008) and one Ford Kuga (MY 2013-2019) from a publicly available dataset [16]. Figure 6 shows the IDs and their cycle distribution in the two cars.

The first dataset contains about 390,000 frames recorded over 6 minutes of stationary traffic, comprising of 25 periodic CAN IDs from 6 ECUs (Powertrain Module, Fuel Module, ABS, Headlamp Module, Electronic-Controlled Steering, Instrument Cluster) with nominal recurrences listed in Table 2, average message rate of 240*ms* and a total bus utilisation of 21%. The second dataset contains about 1,230,000 frames recorded over approximately 8 minutes of traffic, comprising of 70 periodic CAN IDs with nominal recurrences listed



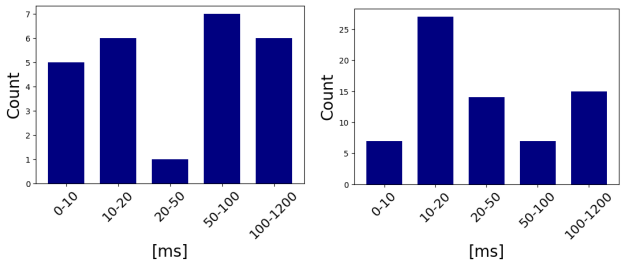
FIGURE 5: In vehicle data collection using Vector VN1630

TABLE 2: CAN bus traffic summary for Ford Focus

				Real	
ID	**Rank**	Size	T_{cycle} *ms*	$\mu(\text{IAT})$ *ms*	$\sigma(\text{IAT})$ *ms*
0x80	1	7B	14	14.007860	0.064702
0x90	2	8B	14	14.007860	0.064829
0x190	3	8B	16	16.000612	0.118893
0x200	4	8B	10	9.995106	0.114491
0x201	5	8B	10	9.995106	0.186936
0x205	6	8B	10	9.995106	0.163965
0x20f	7	8B	14	14.007860	0.283216
0x211	8	8B	14	14.007860	0.327919
0x212	9	8B	105	105.058944	0.955509
0x231	10	7B	10	9.995120	0.864406
0x240	11	2B	10	10.000226	0.270339
0x275	12	3B	100	99.939764	0.508158
0x280	13	8B	50	49.975532	1.027782
0x3db	14	8B	1000	999.831287	0.276074
0x420	15	8B	100	99.950647	1.560934
0x424	16	8B	100	99.950647	1.574219
0x428	17	4B	100	99.950649	1.582055
0x430	18	2B	100	100.003582	0.306311
0x433	19	8B	100	99.939760	1.246593
0x460	20	8B	100	99.936328	1.194552
0x4b0	21	8B	14	14.007868	0.402689
0x4f0	22	6B	1000	999.506620	1.483853
0x4f1	23	8B	1000	999.506620	1.492041
0x4f2	24	4B	1000	1000.035267	0.521859
0x4f3	25	8B	1000	999.506642	1.497376
				Average Bus load: 21%	

TABLE 3: CAN bus traffic summary for Ford Kuga

ID	Rank	Size	T_{cycle} *ms*	Real μ(IAT) *ms*	Real σ(IAT) *ms*
0x10	1	8	10	9.999973	0.116394
0x30	2	8	10	9.999824	0.199587
0x4a	3	8	1002	1002.185361	0.132087
0x4b	4	8	1002	1002.185361	0.156222
0x50	5	8	10	10.000078	0.148831
0x60	6	8	15	15.000116	0.265372
0x6a	7	8	20	20.000151	0.105461
0x70	8	8	20	20.000171	0.266854
0x80	9	8	20	20.000171	0.274126
0x90	10	8	10	10.000086	0.627645
0xa0	11	8	15	15.000107	0.815713
0xb0	12	8	20	19.993223	0.234485
0xc0	13	8	20	20.000171	0.478204
0xc8	14	8	20	19.999646	0.253977
0xd0	15	8	20	20.000150	0.112892
0xe0	16	8	20	20.000150	0.113182
0xf0	17	8	10	10.000093	0.703692
0xf5	18	8	10	10.000093	0.690686
0xf8	19	8	20	20.000202	0.532038
0x100	20	8	20	20.000156	0.134376
0x120	21	8	20	20.000201	0.750347
0x130	22	8	20	20.000244	1.029333
0x138	23	8	20	20.000244	1.101015
0x140	24	8	20	19.993223	0.442559
0x150	25	8	20	19.999649	0.507429
0x160	26	8	20	19.999117	0.479203
0x170	27	8	20	19.991993	0.481834
0x17e	28	8	100	99.998246	0.410776
0x180	29	8	20	19.999106	0.340945
0x190	30	8	10	9.999542	1.309934
0x1a0	31	8	20	20.000264	1.044474
0x1b0	32	8	30	30.000493	3.547167
0x1c0	33	8	20	19.999128	0.537715
0x1d0	34	8	20	19.999159	0.548890
0x1e0	35	8	20	19.999159	0.594847
0x200	36	8	25	25.000169	2.975373
0x208	37	8	25	25.054845	1.858533
0x210	38	8	20	19.999169	0.651840
0x213	39	8	20	19.999169	0.658597
0x218	40	8	30	29.998535	2.909015
0x229	41	8	40	39.983736	0.883560
0x252	42	8	20	19.999095	0.950329
0x260	43	8	25	24.999543	2.106767
0x269	44	8	30	30.015877	2.061869
0x270	45	8	30	30.000573	4.171935
0x280	46	8	30	30.000462	4.123170
0x290	47	8	30	29.999664	2.114722
0x2a0	48	8	40	39.983527	1.400882
0x2a5	49	8	40	39.984176	1.483591
0x2b0	50	8	40	40.000311	0.563247
0x2d0	51	8	40	39.998043	1.300716
0x2d4	52	8	60	59.997412	0.878102
0x2d8	53	8	60	60.001016	0.889459
0x2e0	54	8	70	70.064681	2.566629
0x2f0	55	8	90	90.001481	4.571898
0x310	56	8	100	99.998076	1.702895
0x340	57	8	120	120.002026	1.002462
0x360	58	8	150	149.996373	2.424620
0x380	59	8	300	299.995218	0.778598
0x388	60	8	801	801.745138	2.604533
0x3b4	61	8	300	299.995224	1.145644
0x3e2	62	8	1000	999.995011	2.025272
0x3ea	63	8	1000	999.995011	2.321766
0x400	64	8	250	249.995849	2.490313
0x405	65	8	250	257.226941	42.098793
0x40a	66	8	125	124.996949	2.544420
0x420	67	8	600	599.990729	1.065909
0x435	68	8	300	299.990827	1.410456
0x455	69	8	100	99.999533	2.391197
0x581	70	8	1000	999.974708	1.382289
				Average Bus load: 59%	



(i) Ford Focus (ii) Ford Kuga

FIGURE 6: CAN ID distribution in the two experimental datasets: (i) Ford Focus and (ii) Ford Kuga

in Table 3, average message rate of approximately 140*ms* and a total bus utilisation of 59%. For each CAN identifier, we characterise scheduling performance from the recorded transmit timestamps by computing the inter-arrival time (IAT) sequence $IAT = t_k - t_{k-1}$ and its most common descriptive statistics, mean and variance. We note that ID 0x405 from the Ford Kuga has a bimodal distribution and for this reason it appears with a high standard deviation of around 42 *ms* from its average IAT of 257 *ms*.

For the detailed evaluation, we selected messages from 3 categories from the Ford Focus, representing high, medium and lower priorities, all of them with a worst-case payload of 8 bytes. A visual representation of the IAT histogram and distribution characteristics can be seen in Figure 7. The following can be observed:

- message ID 0x200 exhibits a mean IAT of approximately 14.008*ms* and demonstrates high timing stability, which is consistent with a message whose priority is sufficiently high to avoid sustained arbitration-induced interference. The small, bounded deviations above the mean are consistent with transient delays (e.g., arbitration/queueing under competing traffic or timestamping/processing effects), whereas shorter intervals may reflect compensation following a prior delay.
- message ID 0x280 exhibits a different IAT profile, reflecting its lower arbitration priority and increased susceptibility to bus-level interference. The mean IAT is approximately 50*ms*, but the distribution is considerably broader and multimodal. The histogram and scatter plot reveal distinct bands (approximately around 48*ms*, 50*ms*, and 52*ms*), and the time-evolution plot shows regime-like shifts. This structure is consistent with recurring bus-access delays (arbitration and blocking effects by higher priority IDs). Shorter IATs are just artifacts of a previously delayed transmission, which causes the following frame to exhibit an apparently shorter IAT.
- message ID 0x4f1 exhibits a relatively narrow but structured distribution with a mean at approximately 999.507*ms*, with several clusters around the mean. This multimodal structure reflects intermittent arbitration and blocking effects that arise when higher-priority frames contend for bus availability. The timing deviations corre-

TABLE 4: Soft Real-Time Mechanisms Supported by Windows 11 Pro and Windows IoT Enterprise

Feature	Win 11 Pro	Win IoT
Process Affinity	✓	✓
Thread Affinity	✓	✓
REAL_TIME_PRIORITY_CLASS	✓	✓
THREAD_PRIORITY_TIME_CRITICAL	✓	✓
QPC Counter	✓	✓

spond to delayed transmission until the channel becomes idle, local scheduling jitter and timer-advance effects internal to the transmitter ECU.

In addition to the ID-specific analyses, we characterise the aggregate traffic of each dataset by computing (i) the empirical cumulative distribution function (CDF) of inter-arrival times (IATs) over the full trace, and (ii) the bus utilisation (bus load) estimated over consecutive, non-overlapping blocks of 500 frames. Although there is no universally accepted classification of CAN bus load, prior work provides some useful reference points. The well-known work on CAN response time from [58] reports that bus utilisation of up to approximately 30–40% is typical in automotive applications and may therefore be regarded as low, while careful schedulability can increase this to around 80% while still meeting the deadlines. More conservative guidelines from the automotive industry [59] state that, as a rule of good design, the average bus load should not exceed 50% and the stricter context of avionics from ARINC 825 recommends the same limitation [60]. Based on these references, we classify the bus load from the Ford Focus as low at 21% and that from the Ford Kuga as a high bus load at 59%.

Based on the empirical CDF of inter-event delays dt and bus load computed over the entire trace in both datasets displayed in Figure 8, we can derive the following constraint. Regarding the *minimum inter-arrival time*, the trace-driven scheduler shall support a timing resolution fine enough to reproduce inter-frame separations in the low μs range, including cases near $100\mu s$, while preserving frame transmission ordering. A higher concentration of the IAT in the region of $200 - 300\mu s$ is expected.

V. DESIGNING THE CAN TRAFFIC TRANSMISSION SCHEDULER IN PYTHON AND C++

This section outlines the design of the scheduler that allows us to accurately reproduce the CAN bus traffic in Windows IoT. We separately discuss the Python and C++ approach. To maximize timing performance, the software-based mechanisms from Table 4 were applied in both Windows 11 and IoT implementations.

A. PYTHON DEPLOYMENT

Python is widely used in data science, machine learning, automation scripting, etc., but was not designed for hard real-time execution, which may limit its applicability in automotive systems. Nevertheless, its accessibility, rich ecosystem and rapid prototyping capabilities make it an attractive option for analysing in-vehicle communication, while as we investigate later, many performance issues can be alleviated with proper design choices, e.g., hardware timestamping, busy-wait only loops, process/thread scheduling controls, garbage-collection control, buffered I/O and batching, etc.

Implementing a high-fidelity scheduler in Python for the purpose of CAN message processing using Vector XL driver library [56] presents several challenges, e.g., protocol-specific constants and structures, memory allocation, callbacks and asynchronous events, etc., rooted in the cross-language integration, Python's execution model and the operating system's scheduling.

On Windows, the Vector XL driver library uses the `__stdcall` calling convention, we loaded the library at startup via `ctypes.WinDLL` and declare callbacks using `ctypes.WINFUNCTYPE`. The APIs resource handlers (eg. `XLPortHandle`) are treated as opaque handlers and passed through the API without interpretation. To interface safely with the driver data structures, we replicate the required C types, like the critical struct and union definitions, using `ctypes.Structure` and `ctypes.Union`. CAN channel selection and routing is performed using the driver's access mask (`XLaccess`), which uniquely identifies the physical channel used for message reception and transmission.

Figure 9 shows an overview of the software architecture – the schematic applies to Python but most of the design is followed in the C++ implementations too, one notable difference is the attacker process, which we implemented only in Python. In addition to the presented architecture, the figure shows the timings of interest for the performance evaluation as follows: (i) Inter Arrival Time (IAT) that represents the ground truth, the inter-frame time in the original in-vehicle trace, reflecting network dynamics (ECU scheduling, arbitration, bus load); (ii) Inter Transmission Time represents the inter-frame timing realised by the transmitter; deviations from IAT include the transmitter path processing and OS scheduling jitter; (iii) Inter Reception Time (IRT) represents the inter-arrival time observed by the receiver application; deviation from IAT includes receiver-path effects like ISR/DPC handling, driver buffering, thread scheduling and user-space queueing/logging delays.

Next, we describe each element of the architecture: (1) Transmitter process, (2) Receiver process and (3) Attacker process , with a focus on design decisions critical for realising the critical transmitter time. The architecture of the C++ scheduler is similar to the one in Figure 9, with the remark that instead of processes, it uses threads to realise the same implementation. The Python scheduler has, in addition to the C++, a separate process for adversarial activity that is used only in the later part of our work. We implemented a process-isolated architecture to minimise jitter and prevent I/O stalls: transmitter and receiver are executed as independent OS processes, each started through `subprocess.Popen()` calls. These create independent Python interpreter instances for the three components, avoiding the sharing of the same GIL:

- *Transmitter (replay) process (1)* – drives the determinis-

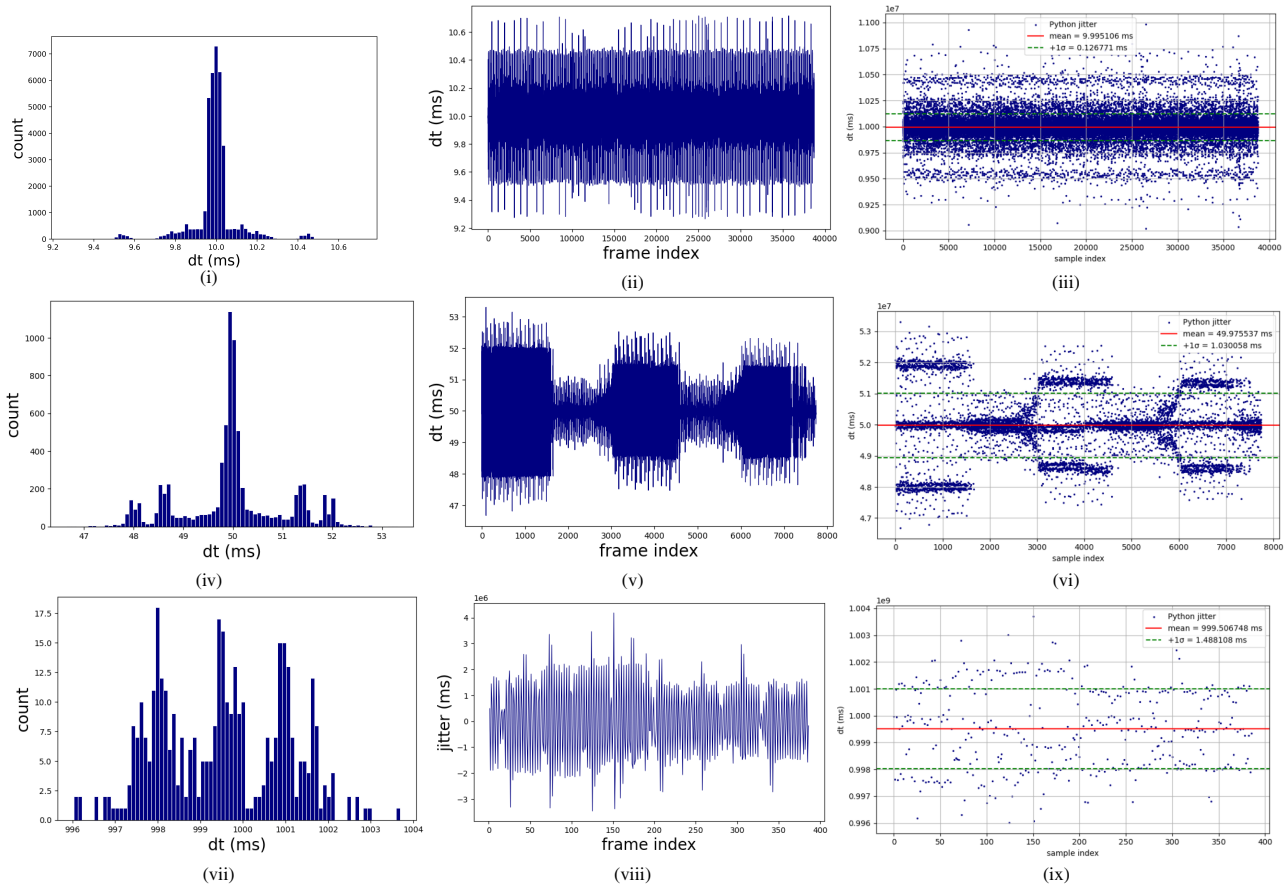


FIGURE 7: IAT histogram, temporal IAT evolution, mean, and standard deviation for representative CAN IDs. (i) to (iii) Statistics for ID 0x200, (iiii) to (vi) Statistics for ID 0x280, and (vii) to (ix) Statistics for ID 0x4F1.

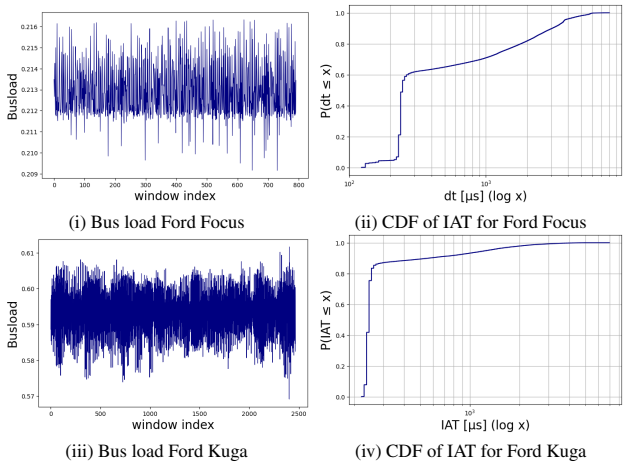


FIGURE 8: Empirical CDF and bus load analysis for the two datasets used in this paper: (i) Bus load Ford Focus, (ii) CDF Ford Focus, (iii) Bus load Ford Kuga, (iv) CDF Ford Kuga

tic CAN transmission for either replaying an in-vehicle CAN trace or periodic/aperiodic message generation (the latter functionality is intended for security testing, like generating intrusions on the bus). The process is configured with `REALTIME_PRIORITY_CLASS` and `CPU affinity`, which allows pinning to dedicate a physical core to reduce pre–emption and cache disruption.

- *Receiver process (2)* – polls the library function call for message receive `xlReceive` or waits on a driver event, copies minimal fields from Xlevent structure returned by the driver, i.e., most importantly the hardware timestamp with a resolution of 8*ns*, CAN ID, channel, DLC, special flags and payload, into a ring buffer. The process is configured with `HIGH_PRIORITY_CLASS` and`CPU affinity`.
- *Attacker process (3)* – polls the library function call for message receive `xlReceive` in order to intercept the ID under attack. After a successful intercept, it transmits configured attack frames using `xlCanTransmit`.

In addition, each process is assigned to a dedicated CPU core and configured with an explicit scheduling priority as explained in the following section. The transmitter that faithfully reproduces the collected CAN traffic is implemented as described in Algorithm1. The procedure starts by recording the `replayStart`, the current time returned by

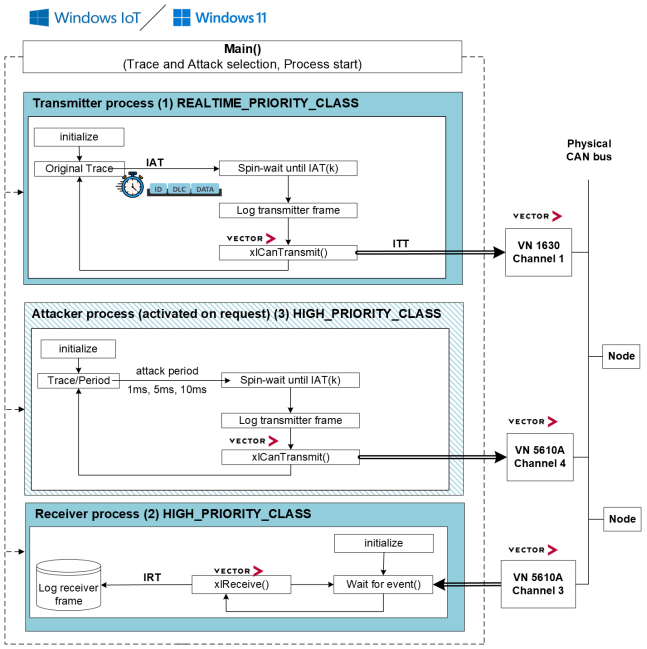


FIGURE 9: Overview of the Python scheduler

a high respolutin counter. Next, for each frame, the transmission is scheduled on the basis of the frame offset, which is translated into an absolute target transmission time relative to the common `replayStart`. The scheduler then computes the residual time to the target transmission time, and if the scheduled release time has not yet been reached, the algorithm calls the `SpinWait` function – by this call, a calibrated busy-wait routine is started that continuously polls the Windows high-resolution hardware-backed performance counter `QueryPerformanceCounter()`. QPC is used to read the current counter value in ticks, while `QueryPerformanceFrequency()` is queried once at initialisation to obtain machine-specific counter frequency and convert the scheduled release time in counter ticks. By scheduling each frame against a fixed global reference rather than incrementally from the previous transmission, the algorithm limits drift accumulation and better preserves the timing profile of the original trace. This timing mechanism controls only the software release time; the remaining end-to-end timing error includes the Vector XL library, driver, interface, and CAN hardware transmit path.

B. C++ DEPLOYMENT

In C++ the Vector XL driver library has native support and calls to the API are done directly, e.g., with the `xlOpenDriver`, `xlOpenPort`, `xlActivateChannel`, `xlCanTransmit`, `xlReceive` functions. Unlike the Python implementation, no foreign-function interface (FFI) is required: driver structs are consumed in their original definition and the control path avoids dynamic marshalling, type conversions, and interpreter bookkeeping. This eliminates overhead sources that are typical for interpreted runtimes,

Algorithm 1 Scheduler for the Transmitter in Python

Require: $xlEvent$, $portHandle$, $accessMask$, $channelIndex$, $msgCount$, $frames$
Ensure: st – status of the retransmission
procedure ReplayFrames
  $replayStart \leftarrow$ PerfCounter()
  for $i \leftarrow 0$ **to** $|frames|$ **do**
    $frame \leftarrow frames[i]$
    $offsetSec \leftarrow frame.offset$
    $target \leftarrow replayStart + offsetSec$
    $msgCount \leftarrow 1$
    $xlEvent \leftarrow frame$
    $now \leftarrow$ PerfCounter()
    $remaining \leftarrow target - now$
    if $remaining > 0$ **then**
      SpinWait($remaining$)
    end if
    $xlStatus \leftarrow$ xlCanTransmit($port, mask, msgCount, xlEvent$)
  end for
  return $xlStatus$
end procedure

Algorithm 2 Scheduler for the Transmitter in C++

Require: $xlChanIndex, frames$
Ensure: $xlStatus$ – status of the retransmission
procedure ReplayFrames
  $xlStatus \leftarrow XL_SUCCESS$
  InitQpcOnce()
  $t_0 \leftarrow frames[0].timestamp$
  $replayStart \leftarrow$ QpcNow
  for $i \leftarrow 0$ **to** $len(frames)$ **do**
    $xlEvent \leftarrow$ ZeroEvent
    $msgCount \leftarrow 1$
    $xlEvent \leftarrow frames[i]$
    $target \leftarrow frames[i].timestamp - t_0$
    $targetTick \leftarrow replayStart +$ NsToQpcTicks($target$)
    while QpcNow $< targetTick$ **do**
      skip
    end while
    $msgCount \leftarrow 1$
    $xlStatus \leftarrow$ xlCanTransmit($port, mask, msgCount, xlEvent$)
  end fors
  return $xlStatus$
end procedure

such as dynamic dispatch, reference counting/GC interactions, and `ctypes` boundary costs. Thus, a more stable instruction footprint and cache behaviour are expected.

Both the receive and transmit threads follow the same logic as in Python:

- *Transmitter thread:* the thread is pinned to a dedicated core and configured with real time scheduling priority to minimise pre–emption, having the objective to replay the traffic collected from the actual vehicle.
- *Receiver thread*: the thread is pinned to a dedicated core and configured with high scheduling priority to minimise pre–emption. It blocks on the driver-provided event handle and only executes on frame availability. Upon wake-up, it issues a single direct `xlReceive()` call and extracts only the fields required for analysis. The extracted frame is immediately serialised in the driver-provided textual representation by the use of `xlGetEventString`. The resulting string is written to a log file using standard buffered I/O.

The critical transmitter timer is achieved by the implementation of Algorithm2. First, the frequency of the high–resolution performance counter is obtained once using the

QueryPerformanceFrequency() call and stored for subsequent conversion of timing intervals into counter ticks. Next, the algorithm records a common timestamp, which defines the temporal origin of the replay process. For each frame, the stored offset is converted into an absolute target counter value relative to the start timestamp. The scheduler then polls the current counter value until the target deadline is met, after which the frame is passed to the CAN driver for transmission. This absolute-time scheduling strategy implemented in both languages avoids the cumulative error associated with incremental delay chaining and therefore improves the fidelity of trace replay. The proposed scheduler, both in its Python and C++ implementation, combines a trace-driven frame release with a lightweight timing-correction mechanism that adjusts the target time tick with the offset of the current frame timestamp relative to t_0 (line 6 of Algorithm 1 and line 10 of Algorithm 2, respectively), in order to improve replay fidelity. By employing a common time reference and high-resolution monotonic timing with busy-wait synchronisation, the scheduler minimises cumulative drift and scheduling-induced latencies, enabling accurate reproduction of the temporal characteristics for the replayed CAN traffic across diverse execution environments.

As a summary of the advantages, compared to the Python scheduler, the C++ variant improves determinism primarily by removing runtime variability and library call overhead: (i) direct driver calls, without inter-language data conversion overhead, xlReceive/xlCanTransmit are invoked without ctypes wrappers, argument conversions, or per-call object creation; (ii) no interpreter/GIL effects: implementation executes natively without a global interpreter lock, dynamic dispatch, or reference counting.

VI. SETUP FOR EXPERIMENTS

This section presents the experimental setup and results of the comparison between software and hardware timestamps during CAN traffic replay. The data collected from our setups and Python code used for determining trace performance metrics are publicly available[1].

A. EXPERIMENTAL SETUP

The experimental setup presented in Figure 10 uses one Vector VN1630 and one Vector VN5610A for CAN/CAN-FD interfaces connected to a properly terminated high-speed CAN bus. Our tests were done in a dual-boot system with Windows 11 and Windows IoT LTSC, running on a laptop with Intel i7-1165G7, base frequency 1.2-2.8 GHz, turbo boost 4.7 GHz, 16GB of RAM.

In all the experiments with the C++ or Python scheduler and receivers, the replay configuration uses the VN1630 for transmission and the VN5610A for reception. The use of the two Vector hardware units enables a controlled multi-node environment in which one interface can perform measurement or replay while the second emulates an additional bus node.

[1]https://github.com/EmiliaCaragea/trace-scheduler

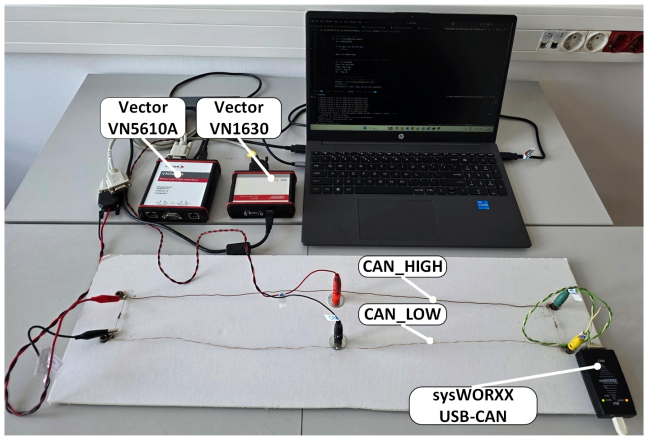


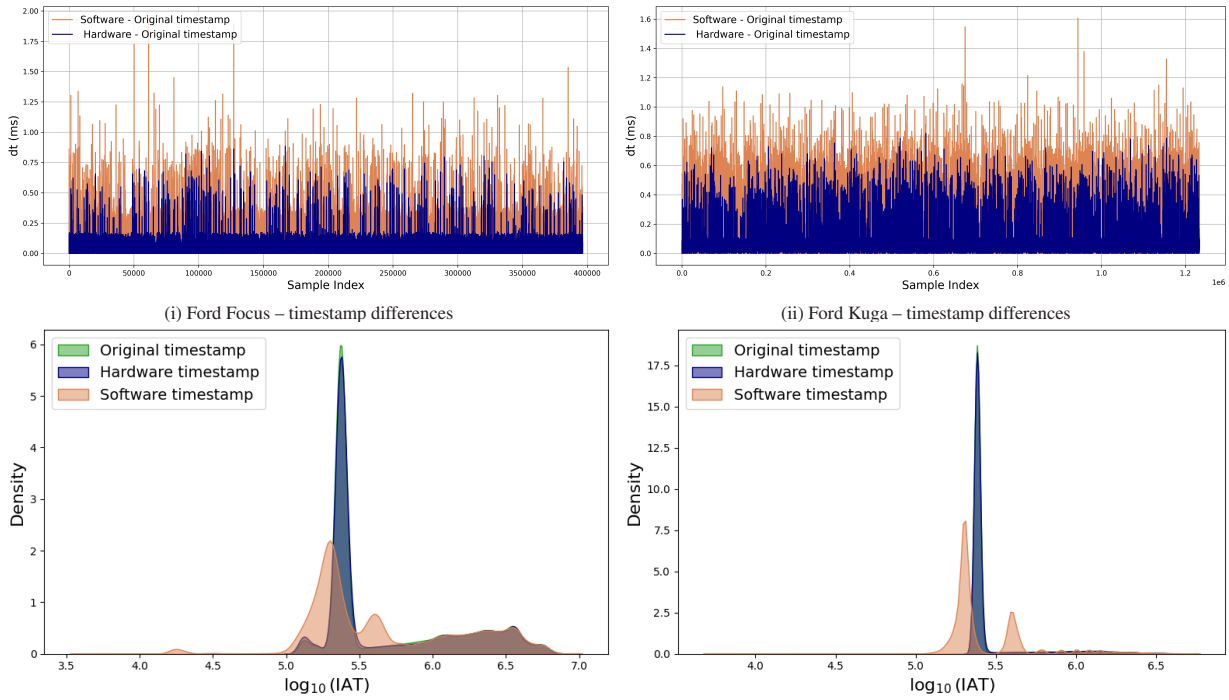
FIGURE 10: Experimental setup for the C++ and Python implementations of the trace-driven scheduler

Importantly, each interface operates with its own hardware clock, introducing independent timing behaviour between transmitters (e.g., clock drift, phase offsets, and scheduling jitter), which reflects more closely the in-vehicle network traffic, allowing evaluation of time-sensitive effects under non-ideal synchronisation.

B. SOFTWARE VS. HARDWARE TIMESTAMPING

To better interpret the replay mode results, hardware and software timestamps were compared at the inter-frame level. While hardware timestamps reflect the time of the communication interface, software timestamps additionally include delays introduced by the operating system and software execution path. Therefore, the difference between both, provides an indirect measure of software-induced timing variability. We conduct the experiments on the two datasets from the two vehicles of the same manufacturer, the Ford Focus at 21% and the Ford Kuga at 59% bus load. Varying CAN bus utilisation enables us to assess how a higher bus load amplifies scheduling effects and whether these effects manifest primarily in the mean inter-arrival time or in worst-case timing deviations (tail latency).

To obtain a meaningful comparison between software and hardware timestamping, we computed the difference of the hardware and software timestamps with respect to the original in-vehicle trace. Both timestamps are collected with the transmitter running C++ under Windows IoT. The software timestamps were obtained using xlGetSyncTime() library function which returns the current high precision PC time (in ns). For each frame i, in the two traces containing software and hardware timestamps, the differences to the original timestamps are computed as follows: $dt_i^{\mathrm{SW}} = t_i^{\mathrm{SW}} - t_i^{\mathbb{O}}$, $dt_i^{\mathrm{HW}} = t_i^{\mathrm{HW}} - t_i^{\mathbb{O}}$ where $t_i^{\mathbb{O}}$ denotes the timestamps from the original trace. The time differences calculated over the two datasets can be visualised in Figure 11. To quantify the magnitude of the timestamp error between measurements, we use the absolute values and compare the median values. For the Ford Focus dataset, the absolute software to original



(i) Ford Focus – timestamp differences

(ii) Ford Kuga – timestamp differences

(iii) Ford Focus – log-scaled IAT KDE

(iv) Ford Kuga – log-scaled IAT KDE

FIGURE 11: Hardware vs. software timestamp distribution in the two experimental datasets: (i) and (ii) Timestamp differences for the Ford Focus and Ford Kuga datasets, and (iii) and (iv) kernel density estimates (KDEs) of the log-scaled inter-arrival times derived from the original, hardware-based, and software-based timestamps for the Ford Focus and Ford Kuga datasets

difference is about 149.7μs, whereas the corresponding hardware timestamp error is only 32.8μs. We can conclude that for the Ford Focus, the typical timing error of the software timestamp is approximately 4.6 times larger than that of the hardware timestamp. For the Ford Kuga dataset, the absolute software to original difference is about 49.8μs, whereas the corresponding hardware timestamp error is 49.1μs. This shows that for higher bus utilization the error between software and hardware timestamps remains quantifiable.

These results demonstrate that, although the average replay behaviour remains close to the hardware reference, the software path introduces significant timing jitter. Overall, the hardware timestamps provide a more faithful representation of the actual frame timing, whereas software timestamps additionally capture non-deterministic delays caused by the operating system and the software execution path. For the remaining analysis, focusing on the impact of the operating system and programming language, we used only the hardware timestamps due to their higher accuracy.

VII. RESULTS AND EVALUATION

In this section, we introduce the performance metrics and then experimentally evaluate different pairings of Python or C++ with Windows 11 or Windows IoT. Finally, we evaluate retransmission in the presence of adversarial behaviour.

A. PERFORMANCE METRICS

Let the arrival timestamps and IDs in a trace be ordered pairs of the form $(t_i, ID_j), i = 1..n, ID_j \in \{1..\ell\}$ where n denotes the number of timestamps and ℓ the number of IDs. Let $\mathbb{O} = \{(t_i^{\mathbb{O}}, ID_i^{\mathbb{O}})\}_{i=1}^{n}$ denote the original, collected in-vehicle trace and $\mathbb{R} = \{(t_i^{\mathbb{R}}, ID_i^{\mathbb{R}})\}_{i=1}^{n}$ the reproduced trace as recorded by the receiver from our experimental setup. We assume that all frames are ultimately transmitted and received, thus $|\mathbb{O}| = |\mathbb{R}| = n$, though the order cannot be guaranteed (which is a reasonable assumption for the CAN bus). In the measurements that follow, both the original and replay timestamp sequences are normalized to their respective first timestamp. This normalization is necessary because the timestamps are referenced to independent start timers originating from different acquisition environments: the original trace is recorded during in-vehicle data collection, whereas the replay trace is generated in the laboratory setup. Consequently, the absolute timestamp origins are unrelated, and only the relative timing between frames is meaningful for comparison.

We characterise the performance of our CAN communication scheduler based on metrics at the trace and ID level. Firstly, at the trace level, we compare the offsets between the original $\mathbb{O}$ and the receiver trace $\mathbb{R}$ based on the following metrics:

1) *ID switching ratio* which refers to the number of IDs that

differ in the original trace compared to the receiver trace, and which should be ideally zero, i.e.,

$$\mathrm{SWR} = \left| \left\{ (t_i^{\mathbb{O}}, ID_i^{\mathbb{O}}) \in \mathbb{O}, (t_i^{\mathbb{R}}, ID_i^{\mathbb{R}}) \in \mathbb{R} \middle| ID_i^{\mathbb{R}} \neq ID_i^{\mathbb{O}} \right\} \right|$$

2) *absolute median offset (AMO)*, which is the absolute value of the median of differences between the original and replay timestamps, i.e.,

$$\mathrm{AMO} = \left| \mathrm{median}\left(t_i^{\mathbb{R}} - t_i^{\mathbb{O}}\right) \right|, \qquad i = 1 \ldots n,$$

3) *timing absolute deviation (TAD)*, which is the absolute residual difference between the original and replay timestamps after compensating for the temporal alignment with the median offset, i.e.,

$$\mathrm{TAD} = \left\{ \left| t_i^{\mathbb{O}} - t_i^{\mathbb{R}} - \mathrm{median}\left(t_i^{\mathbb{R}} - t_i^{\mathbb{O}}\right) \right| \middle| i = 1 \ldots n \right\}$$

for which we derive the following descriptive statistics: mean, standard deviation, median, p95, p99, p99.9 and maximum.

Note that the AMO quantifies the magnitude of the constant offset correction required to align the original and receiver timestamp sequences, whereas the TAD quantifies the residual timing error of each replayed frame after this constant offset has been removed. Consequently, the two metrics are complementary: AMO characterizes the global alignment between the traces, while TAD characterizes the per-frame replay accuracy following alignment. As stated previously, both the original and receiver timestamp sequences are expressed relative to their respective first timestamp.

Secondly, at the ID level, let $I_{\mathrm{id}} = \{i | ID_i^{\mathbb{R}} = \mathrm{id}\} = \{r_1 < r_2 < \ldots < r_{N_{\mathrm{id}}}\}, N_{\mathrm{id}} = |I_{\mathrm{id}}|$ denote the set of indexes from the receiver trace $\mathbb{R}$ for which the identifier *ID* has value id, and let the nominal cycle of that identifier be C_{id}. We define the following for a target identifier id:

4) *mean IAT* which is the average inter-arrival time for the targeted identifier, i.e.,

$$\mathrm{IAT}_{\mathrm{mean}}^{\mathrm{id}} = \frac{1}{|I_{\mathrm{id}}|} \sum_{i=2,|I_{\mathrm{id}}|} \left| t_i^{\mathbb{R}} - t_{i-1}^{\mathbb{R}} \right|,$$

5) *maximum IAT* which is the maximum inter-arrival time for the targeted identifier, i.e.,

$$\mathrm{IAT}_{\mathrm{max}}^{\mathrm{id}} = \max(|t_i^{\mathbb{R}} - t_{i-1}^{\mathbb{R}}|), i = 2, |I_{\mathrm{id}}|$$

6) *average Jitter* which is the average deviation of the IAT from the nominal period for the targeted identifier, i.e.,

$$\mathrm{J}_{\mathrm{mean}}^{\mathrm{id}} = \left| \mathrm{IAT}_{\mathrm{mean}}^{\mathrm{id}} - C_{\mathrm{id}} \right|$$

7) *maximum Jitter* - maximum deviation of the the IAT from the nominal period for the targeted identifier, i.e.,

$$\mathrm{J}_{\mathrm{max}}^{\mathrm{id}} = \left| \mathrm{IAT}_{\mathrm{max}}^{\mathrm{id}} - C_{\mathrm{id}} \right|$$

8) *The deadline miss ratio* is the fraction of frames for the targeted identifier where the IAT is larger than the cycle time by at least λ percent, i.e. having the indexes of the frames that miss the aforementioned deadline:

$$\mathbb{M}_{\lambda}^{\mathrm{id}} = \left\{ r \in \{2..|I_{\mathrm{id}}|\} \middle| t_{i_r}^{\mathbb{R}} - t_{i_{r-1}}^{\mathbb{R}} > (1+\lambda) C_{\mathrm{id}} \right\},$$

the *deadline miss ratio* is:

$$\mathrm{DMR}_{\lambda}^{\mathrm{id}} = \frac{|\mathbb{M}_{\lambda}^{\mathrm{id}}|}{|I_{\mathrm{id}}|},$$

9) *longest burst miss* which is the longest continuous sequence of frames for the targeted identifier where the IAT is larger than the cycle time C_{id} by at least λ percents, i.e.,

$$\mathrm{LBM}_{\lambda}^{\mathrm{id}} = \max \left\{ k \middle| \exists i \text{ s.t. } \{i, ..., i+k-1\} \in \mathbb{M}_{\lambda}^{\mathrm{id}} \right\}.$$

The following subsection reports the experimental results of the proposed CAN communication scheduler under realistic bus load conditions across two execution environments: Windows 11 and Windows IoT. Using the earlier defined metrics, we assess performance both at the trace level and at the identifier level. The goal is to quantify not only average timing accuracy but also determinism through worst-case tail behaviour.

B. RESULTS FOR TRAFFIC RETRANSMISSION

In this section, we interpret the obtained results in relation to the main question of this paper, namely, how retransmission in a non-real-time operating system, comprising both the scheduler and the transmission hardware, compares to real automotive real-time traffic. For clarity, we refine this question into two sub-questions, which we address based on the results of our study: (i) how does the operating system (Windows 11 vs. Windows IoT Enterprise) affect real-time behaviour, and (ii) how does the programming language (Python vs. C++) influence real-time performance? We discuss this in what follows, both at the trace-level and individual ID–level.

1) Trace-level Performance

Characterisation of the trace-level performance including statistics and run variability, is done for both datasets from 5 executions and is summarised in Tables 5 and 6 . The findings can be summarised as follows:

a) SWR – the packet switching ratio is 0 in all cases, which indicates the replay of the original trace in all environments causes no structural degradation.
b) AMO – the C++ implementation achieves substantially closer alignment of the replay trace to the original trace than Python, with offsets below $0.1ms$ for both datasets,

TABLE 5: Replay timing statistics and run variability with Python in Windows 11 and Windows IoT

OS	Impl.	Run	Car	Frames	SWR	AMO	Timing Absolute Deviation (TAD) – ms						
							Mean	Std.	Med.	P95	P99	P99.9	Max
Win 11	Python/XLDriverAPI	1	Focus	396436	0	6.104	0.028	0.039	0.017	0.091	0.148	0.312	6.104
Win 11	Python/XLDriverAPI	2	Focus	396436	0	5.800	0.029	0.038	0.017	0.098	0.155	0.206	5.800
Win 11	Python/XLDriverAPI	3	Focus	396436	0	6.046	0.029	0.039	0.017	0.098	0.156	0.279	6.047
Win 11	Python/XLDriverAPI	4	Focus	396436	0	5.972	0.029	0.038	0.017	0.091	0.156	0.238	5.972
Win 11	Python/XLDriverAPI	5	Focus	396436	0	6.063	0.030	0.038	0.018	0.098	0.155	0.205	6.062
Win IoT	Python/XLDriverAPI	1	Focus	396436	0	5.530	0.030	0.038	0.024	0.091	0.149	0.417	5.529
Win IoT	Python/XLDriverAPI	2	Focus	396436	0	5.522	0.030	0.041	0.017	0.098	0.156	0.476	5.522
Win IoT	Python/XLDriverAPI	3	Focus	396436	0	5.882	0.030	0.041	0.017	0.099	0.156	0.476	5.882
Win IoT	Python/XLDriverAPI	4	Focus	396436	0	5.858	0.029	0.038	0.017	0.091	0.148	0.324	5.857
Win IoT	Python/XLDriverAPI	5	Focus	396436	0	0.058	0.030	0.036	0.018	0.098	0.155	0.395	1.009
Win 11	Python/XLDriverAPI	1	Kuga	1233545	0	17.376	0.022	0.107	0.013	0.066	0.096	0.516	17.373
Win 11	Python/XLDriverAPI	2	Kuga	1233545	0	15.959	0.021	0.095	0.013	0.062	0.095	0.532	15.954
Win 11	Python/XLDriverAPI	3	Kuga	1233545	0	16.540	0.035	0.214	0.022	0.081	0.101	0.543	25.541
Win 11	Python/XLDriverAPI	4	Kuga	1233545	0	16.147	0.025	0.097	0.014	0.081	0.100	0.519	16.146
Win 11	Python/XLDriverAPI	5	Kuga	1233545	0	17.245	0.020	0.106	0.013	0.050	0.092	0.538	17.240
Win IoT	Python/XLDriverAPI	1	Kuga	1233545	0	15.500	0.027	0.096	0.016	0.075	0.097	0.585	15.500
Win IoT	Python/XLDriverAPI	2	Kuga	1233545	0	15.524	0.029	0.095	0.018	0.075	0.096	0.591	15.520
Win IoT	Python/XLDriverAPI	3	Kuga	1233545	0	16.073	0.024	0.100	0.015	0.073	0.097	0.589	16.080
Win IoT	Python/XLDriverAPI	4	Kuga	1233545	0	16.114	0.030	0.098	0.021	0.075	0.092	0.582	16.112
Win IoT	Python/XLDriverAPI	5	Kuga	1233545	0	16.237	0.031	0.106	0.019	0.077	0.346	0.615	16.240

TABLE 6: Replay timing statistics and run variability with C++ in Windows 11 and Windows IoT

OS	Impl.	Run	Car	Frames	SWR	AMO	Timing Absolute Deviation (TAD) – ms						
							Mean	Std.	Med.	P95	P99	P99.9	Max
Win 11	C++/XLDriverAPI	1	Focus	396436	0	0.017	0.032	0.038	0.024	0.093	0.160	0.451	1.982
Win 11	C++/XLDriverAPI	2	Focus	396436	0	0.058	0.031	0.036	0.023	0.097	0.156	0.361	0.844
Win 11	C++/XLDriverAPI	3	Focus	396436	0	0.050	0.032	0.036	0.024	0.095	0.157	0.388	0.840
Win 11	C++/XLDriverAPI	4	Focus	396436	0	0.082	0.031	0.038	0.025	0.091	0.156	0.443	1.795
Win 11	C++/XLDriverAPI	5	Focus	396436	0	0.066	0.031	0.038	0.023	0.093	0.159	0.444	1.050
Win IoT	C++/XLDriverAPI	1	Focus	396436	0	0.025	0.035	0.040	0.027	0.094	0.159	0.464	3.418
Win IoT	C++/XLDriverAPI	2	Focus	396436	0	0.058	0.032	0.038	0.025	0.091	0.156	0.460	0.845
Win IoT	C++/XLDriverAPI	3	Focus	396436	0	0.074	0.033	0.046	0.025	0.097	0.171	0.647	0.999
Win IoT	C++/XLDriverAPI	4	Focus	396436	0	0.033	0.031	0.037	0.024	0.098	0.156	0.444	0.852
Win IoT	C++/XLDriverAPI	5	Focus	396436	0	0.058	0.033	0.045	0.024	0.098	0.171	0.598	1.147
Win 11	C++/XLDriverAPI	1	Kuga	1233545	0	0.050	0.025	0.065	0.016	0.075	0.271	0.738	3.081
Win 11	C++/XLDriverAPI	2	Kuga	1233545	0	0.066	0.026	0.048	0.016	0.075	0.116	0.591	2.090
Win 11	C++/XLDriverAPI	3	Kuga	1233545	0	0.033	0.026	0.047	0.016	0.082	0.114	0.614	0.844
Win 11	C++/XLDriverAPI	4	Kuga	1233545	0	0.050	0.017	0.043	0.009	0.033	0.099	0.574	2.016
Win 11	C++/XLDriverAPI	5	Kuga	1233545	0	0.025	0.021	0.046	0.012	0.064	0.112	0.579	4.509
Win IoT	C++/XLDriverAPI	1	Kuga	1233545	0	0.009	0.022	0.061	0.012	0.056	0.362	0.752	2.140
Win IoT	C++/XLDriverAPI	2	Kuga	1233545	0	0.000	0.019	0.051	0.011	0.041	0.139	0.664	1.344
Win IoT	C++/XLDriverAPI	3	Kuga	1233545	0	0.017	0.022	0.084	0.009	0.050	0.353	1.557	2.622
Win IoT	C++/XLDriverAPI	4	Kuga	1233545	0	0.050	0.018	0.048	0.009	0.034	0.099	0.621	2.080
Win IoT	C++/XLDriverAPI	5	Kuga	1233545	0	0.025	0.017	0.048	0.009	0.033	0.123	0.607	0.852

compared to approximately $6ms$ for the Ford Focus and even $17ms$ for the Ford Kuga when using the Python scheduler. This indicates a larger systematic gap between languages, which can be attributed to scheduling and processing overhead. For the exact individual frame delay, the individual ID-level metrics shall be consulted.

c) the *mean* value of TAD – for both programming languages, Python and C++ and on both operating systems, Windows 11 and Windows IoT, the timing deviation across all runs is kept well below $50\mu s$, with a minimum of $17\mu s$ for two replay runs in C++. This metric indicates a very similar replay behaviour across all languages and operating systems.

d) the *standard deviation* of TAD – in Python, for the Ford Focus, the standard deviation in both configurations, Windows 11 and Windows IoT, is bounded to approximately $50\mu s$. For the Ford Kuga, the standard deviation is slightly increased to about $100\mu s$, with a first differentiator factor when replay is performed under Windows 11, the maximum observed standard deviation exceeds $200\mu s$ for one of the runs. In C++, for both traces and in both operating systems, the TAD value remains well below $100\mu s$, indicating a more stable replay behaviour. This metric shows a slight advantage for C++ in front of Python in case of the more demanding traffic from the Ford Kuga.

e) the *median* value of TAD – appears to follow the mean behaviour closely. For both programming languages, Python and C++ and on both operating systems, Windows 11 and Windows IoT the median of the TAD

across all runs is kept at about $20\mu s$, with the lowest values achieved for C++ execution under Windows IoT. This indicates that C++ provides more stable typical timing behaviour, particularly for the more demanding Ford Kuga trace.

f) the *P95, P99* and *P99.9* – for Python, the Ford Focus trace remains mostly in the μs range on both operating systems, with a maximum of the P99 of $156\mu s$ in both Windows 11 and Windows IoT. In contrast, the Ford Kuga trace shows several runs with a slight increase of the p99. In both C++ and Python and across Windows 11 and Windows IoT runs the operating systems the values are bounded to about $300\mu s$.

g) the *maximum* TAD – shows significantly increased variations, in the order of several milliseconds. First, for the Ford Focus trace and Python, the absolute maximum is of $6.104ms$ in Windows 11. A minimum of $1.009ms$ is reached for one of the Windows IoT runs. When replaying the same trace using the C++ scheduler, the maximum value of the TAD is reduced significantly, with most of the values at $1ms$ and one absolute maximum of $3.418ms$ for one run under Windows IoT. For the Ford Kuga trace, the variations are increased even further. With Python scheduler under both operating systems the maximum of the TAD exceeds $15ms$. In contracts, the replay of the same trace using C++ schedule reduces the values significantly, with most of the values at about $2ms$ and a maximum of $4.509ms$ for one execution under Windows 11. This metric clearly favours C++, however, it is determined solely by the single sample exhibiting the largest timing deviation.

Figure 12 illustrates the maximum value of the TAD, which provides the largest difference in our comparison, but the overall timing behaviour is better reflected by the mean and median of the TAD which shows the performance of both C++ and Python implementation to be quite close. The maximum value of the TAD appears to have the strongest configuration dependency. The maximum error is reported for Python and Windows 11 configuration. A significant improvement is achieved when changing the scheduler implementation to C++. The value doesn't appear to be influenced by the use of Windows 11 or IoT.

Overall, the evaluation of all the trace-level statistics indicates that the replay timing accuracy is influenced first of all by the development environment, C++ outperforming Python in the tail behaviour, although the mean and median are close, and no significant variations were observed between Windows 11 or IoT. These results are based on 5 runs, for 2 cars, 2 operating systems and 2 programming environments, resulting in a total of 40 replay traces. All experiments were conducted using original traces collected under standstill conditions, providing a controlled setup for evaluating replay timing behaviour. While dynamic driving conditions were not evaluated, the CAN traffic considered in this work is cyclic, with message periods fixed during production. Consequently, different driving conditions would affect only the payload contents, not the message sizes or transmission periods.

2) ID–level performance

We now evaluate our defined set of ID–level metrics under low and high bus load, in the Ford Focus and Ford Kuga, to isolate platform effects from overload-induced artefacts. For brevity, the main body of this paper presents the performance highlights in the selected environments, while Appendix A contains the complete results for all experimental configurations and should be consulted for the full numerical results.

The following conclusions can be drawn regarding the ID–level performance:

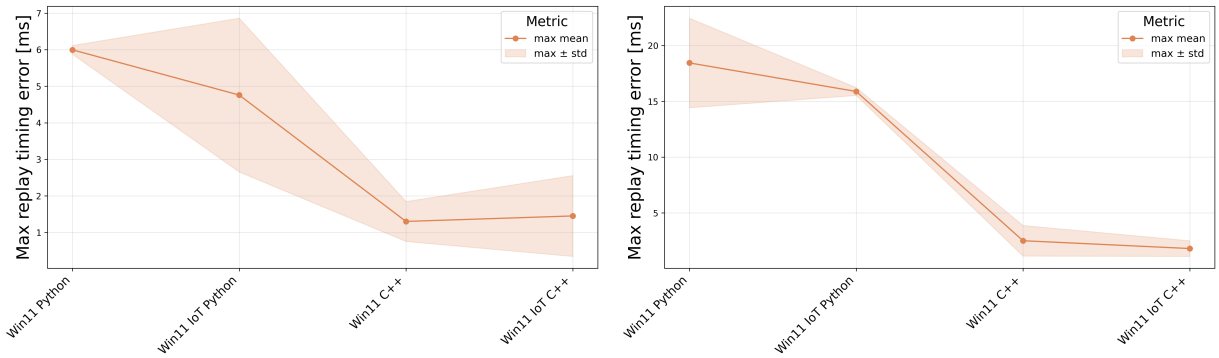
a) IAT^{id}_{mean} – is preserved at ID level under all environments down to μs level, indicating that average timing behaviour alone is not sufficient to assess real-time capability of the solutions. A more in-depth analysis is required by evaluating the following metrics.

b) $\text{IAT}^{id}_{\text{max}}$ – revealed more implementation-dependent variation than IAT^{id}_{mean}. Differences were visible for several short-cycle messages. For the Ford Focus, Python produced higher maximum values for several messages, including 0x200, 0x201, 0x205, 0x20f, and 0x4b0. For the Ford Kuga, the result was mixed; both Python and C++ exhibited lower $\text{IAT}^{id}_{\text{max}}$, making the differences ID and traffic-dependent. Regarding the operating systems, no consistent advantage was observed between the two evaluated platforms.

c) $\text{J}^{\text{id}}_{\text{mean}}$ – remained very low for most messages under both Python and C++, indicating comparable typical timing stability. Several isolated differences were observed in both directions, but they do not indicate a general language-dependent change in typical jitter. Between Windows 11 and Windows IoT results were nearly identical for most CAN IDs. Platform configuration therefore had little influence on average timing variability.

d) $\text{J}^{\text{id}}_{\text{max}}$ – was the most sensitive metric to implementation language and deployment platform. For the Ford Focus trace, Python produced higher maximum jitter for several short-cycle CAN messages with a maximum deviation across implementations of about $300\mu s$ for ID 0x200. For the Ford Kuga trace, results were more variable; both Python and C++ showed differences in hundred μs range for a subset of messages with 10 ms and 20 ms periods. These results indicate that the language effect on worst-case jitter is trace-dependent. A comparable pattern was observed for the platform comparison. In the Ford Focus dataset, the platform-dependent reductions were generally modest. More pronounced variations were observed for the Ford Kuga bus load. Windows IoT reduced $\text{J}^{\text{id}}_{\text{max}}$ by as much as 1.385 *ms* for ID 0x4a, while other $20ms$ cyclic messages improved by about $500\mu s$.

e) $\text{DMR}^{\text{id}}_{\lambda}$ – the deadline miss ratio, evaluated at $\lambda = 10\%$, is ID-dependent and is largely preserved across design solutions and across both operating systems. Most DMR



(i) Ford Focus (ii) Ford Kuga

FIGURE 12: Maximum TAD across the 5 runs for (i) Ford Focus and (ii) Ford Kuga

differences were marginal and bidirectional, providing little evidence of a systematic language or platform effect.

f) $\mathrm{LBM}^{\mathrm{id}}_{\lambda}$ – the longest burst miss, evaluated at $\lambda = 10\%$, aims to evaluate the persistence of the deadline miss and whether it occurs in clusters. The metric showed robust results in both language and operating-system configurations. The limited category changes were threshold-dependent and occurred in both directions, without consistently favouring one language or platform.

Overall, the ID-level analysis shows that the solutions are largely similar in terms of average timing behaviour and deadline miss ratio, while the main performance differences appear in worst–case timing metrics. $\mathrm{IAT}^{id}_{\mathrm{mean}}$, $\mathrm{J}^{\mathrm{id}}_{\mathrm{mean}}$, $\mathrm{DMR}^{\mathrm{id}}_{\lambda}$, and $\mathrm{LBM}^{\mathrm{id}}_{\lambda}$ were mostly preserved across the implementations, indicating that typical timing performance is not strongly affected by the deployment configuration. However, $\mathrm{IAT}^{id}_{\mathrm{max}}$ and $\mathrm{J}^{\mathrm{id}}_{\mathrm{max}}$ reveal more pronounced differences at the ID level too, with C++ generally exhibiting better worst-case timing behaviour than Python. These findings suggest again that the programming language and platform primarily influences tail behaviour, e.g., worst-case deviations, rather than general timing stability.

C. COMPARISON WITH AN INDUSTRY-GRADE SOFTWARE ENVIRONMENT: CANOE

We use specialized Vector measurement hardware to determine whether the observed timing deviations can be attributed primarily to the sender thread, the receiver thread, or both. Since the sender relies on the more fragile trace-driven scheduler, while the receiver depends mainly on the timestamps returned by the XL Driver library, we expect the sender side to be the dominant source of error. Moreover, we compare with the timing accuracy when the VN5610A is used as a transmitter in the CANoe environment from Vector. We note that in the previous experiments, VN5610 was the receiver and VN1630 the transmitter, however, to simplify the following experiments which introduce a new device, the Breakout Box D62Y9, we used just the more advanced VN5610 as receiver or transmitter. All configurations with the respective transmitter and receiver side implementations are described in Table 7 and are briefly explained as follows.

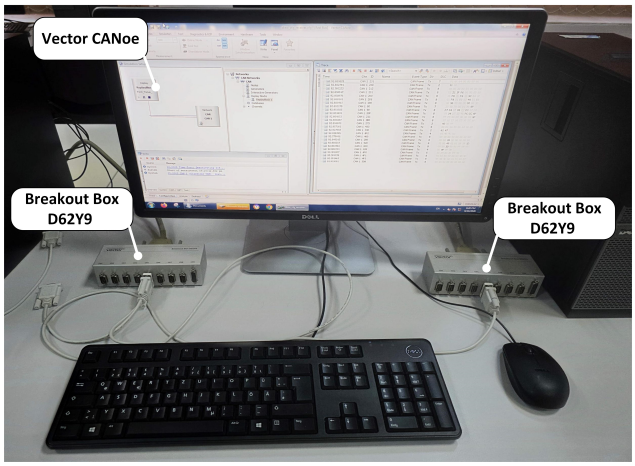


FIGURE 13: Experimental setup with two Vector Breakout Boxes D62Y9 and the native CANoe environment

As a reference, we replay the trace using two Vector Breakout Boxes D62Y9 connected to PCI Express acquisition cards. The setup presented is presented in Figure 13. These devices provide a clock skew below 1 ppm. The results shown in Table 7 for configurations (1) and (2) indicate that the timing deviations remain essentially below $1\mu s$. Note that these devices are connected through dedicated PCI Express hardware rather than the USB interfaces used by the more portable VN devices. For configurations (3) and (4), we replay the two traces using one Vector Breakout Box D62Y9 connected to PCI Express and replace the receiver with a VN5610A connected via USB. For these configurations, both the transmitter and the receiver are implemented using Vector's CANoe application. In configurations (5) and (6), we use the same devices as previously but swap the transmitter and receiver sides. In configurations (7) and (8), we retain one Breakout Box D62Y9 as the sender and replace the receiver

with a VN5610A running the C++ implementation on Windows IoT. Since the sender and receiver now rely on different clocks, the relative clock skew between the VN5610A and the D62Y9 had to be estimated. This skew was determined to be approximately 39ppm and adjusted accordingly. We note that skew adjustment was not necessary when two VNs were connected to the same computer, in the experiments from the previous section, because their local clocks were synchronized. Finally, in configurations (9) and (10), we use the same setup as in configurations (7) and (8) but swap the transmitter and receiver sides.

Overall, the results in Table 7 show a global trace alignment dependent on both hardware and software transmission paths. Best AMO, approximately $1\mu s$, is achieved when both transmitter and receiver are realised with D62Y9 devices. For all other configurations, the reported AMO remains at *ms* level. The best mean timing deviation is again obtained in configuration (1), with a low standard deviation of approximately $1\mu s$. In the mixed configurations, when replay is performed from VN5610A to D62Y9 or vice versa, the mean TAD stays within tens of microseconds range, close to the results obtained when we used the same receiver and our trace-driven scheduler. In configurations (5) and (6), where CANoe is used to replay the traces from VN5610A to D62Y9, the mean TAD is increased to hundreds of microseconds, while the maximum TAD reaches millisecond values. These results suggest that the dominant source of timing error lies in the replay scheduler and the associated Windows and USB communication stack, rather than in the receiver's timestamping capabilities. As shown by the D62Y9-based setup, timing deviations below $1\mu s$ are achievable, albeit with specialised laboratory hardware.

D. THEORETICAL WORST-CASE RESPONSE TIME

For a deeper analysis of the worst-case transmission delays, we use the well-established CAN WCRT (Worst-Case Response Time) framework introduced by [58]. This framework was also recently extended to CAN-FD by [61]. Note that this analysis characterises the worst-case response time imposed by CAN arbitration, identifier priorities, blocking, frame transmission times, and periodic interference. It does not provide a formal upper bound for the host-side replay delay introduced by Windows, the Vector driver, USB communication, or the replay scheduler. Therefore, the WCRT can be used only as a bus-level reference model against which the measured replay timing deviations can be interpreted.

Based on [58], the worst–case transmission time for a frame i of n bytes, considering 11-bit identifiers and the maximum number of stuffing bits, is defined as: $C_i = t_{\text{bit}} \cdot (47 + 8n + \lfloor \frac{34+8n-1}{4} \rfloor)$. The value of t_{bit} is set to $2\mu s$ in our case since we are working with a 500 kbps CAN. Then the *worst-case response time* (WCRT) for each message i is: $R_i = J_i + w_i + C_i$, where J_i is the queueing jitter, w_i is the worst-case transmission delay, and C_i is the worst-case transmission time of message i. In this relation, J_i is the actual queueing jitter for which the software stack is responsible, and since we want to characterise the response of the network itself, and later compare to the experimental values from the Windows measurements, we set $J_i = 0$ to get a lower bound of the worst-case response time, attributable solely to arbitration, blocking and frame transmission time. The worst-case transmission delay w_i is obtained by solving the recurrence: $w_i^{n+1} = B_i + \sum_{k<i} \lceil \frac{w_i^n + J_k + \tau_{bit}}{T_k} \rceil C_k$. Here, B_i is the maximum blocking time caused by a message with lower priority and T_k the transmission period.

Figure 14 shows the resulting WCRT estimates for the Ford Focus and Ford Kuga traces. Due to the higher bus load, the estimated worst-case response time increases from below $7ms$ for the Ford Focus to almost $35ms$ for the Ford Kuga. The largest measured Timing Absolute Deviation (TAD) values are $2.572ms$ for the Ford Focus and $7.274ms$ for the Ford Kuga, both observed with the Python replay scheduler on Windows 11. These values remain below the corresponding CAN bus-level WCRT estimates. However, this comparison should not be interpreted as a formal guarantee for the replay software. Rather, it indicates that the observed software-induced replay deviations are smaller than the timing delays that can arise from worst-case arbitration and interference on the CAN bus itself.

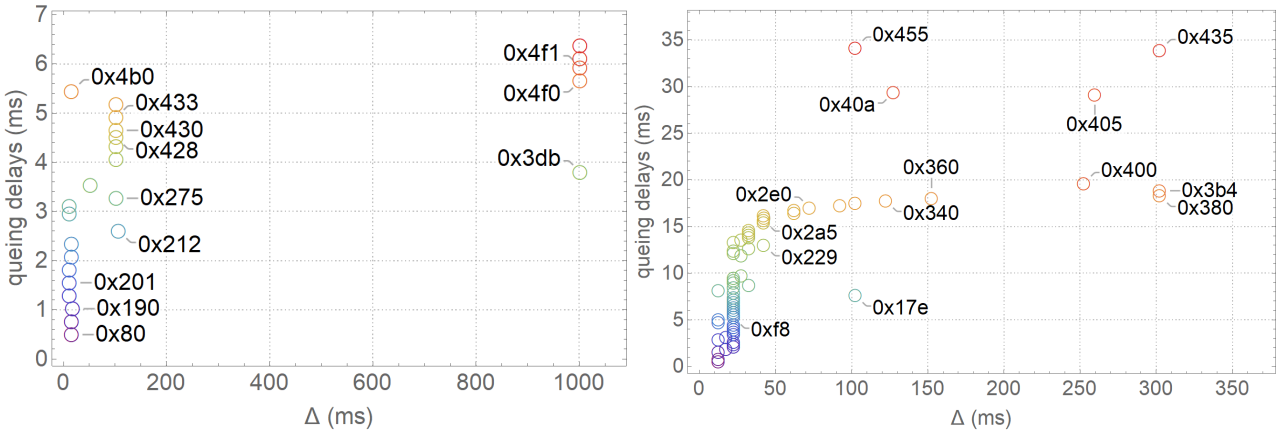
E. RESULTS WITH ADVERSARIAL ACTIVITY

In this section, we briefly assess the capability of the scheduler under several attack scenarios. The motivation for comparing its behaviour across different execution environments is to determine whether the attacker model is platform-robust or whether its timing characteristics, i.e., clock skews, are substantially influenced by the operating system or the transmission stack. This analysis is particularly relevant for automotive cybersecurity experiments, where timing deviations of only a few milliseconds may affect ID switching, burst miss ratio, ID-distribution statistics, and more importantly, the clock skew, which is also used by intrusion detection systems. For this evaluation, we extended the Python scheduler since the programming environment is more promising for future extensions with machine learning algorithms. At the physical level, to intercept the periodic CAN traffic and inject attack frames, we used a separate channel of the VN5610A. At the logical level, we used an isolated process, created through `subprocess.Popen()` call, which was assigned a dedicated core and `HIGH_PRIORITY_CLASS` execution. For both cars, we cover 2 attack scenarios i.e., replay and fuzzing performed on two CAN IDs. The replay attack consists of resending the frame with an identical payload, whereas for fuzzing, the frame is sent with a randomised payload.

For the Ford Focus dataset, which is representative for a slightly lower bus load, we selected IDs 0x200 and 0x275 for the replay attacks and IDs 0x205 and 0x3DB for fuzzing attacks, covering nominal periods of $10ms$, $100ms$ and $1000ms$. For the Ford Kuga dataset, which represents the higher bus load scenario, we selected shorter period messages, IDs 0x10 and 0x130 for the replay attacks and IDs 0x140 and 0x160 for fuzzying attacks, which have $10ms$ or $20ms$ periods. For

TABLE 7: Replay timing statistics with Vector tools and the C++ trace-driven scheduler in Windows IoT

No.	Transmitter (Scheduler)	Receiver	Car	Frames	SWR	AMO	Timing Absolute Deviation (TAD) – ms						
							Mean	Std.	Med.	P95	P99	P99.9	Max
(1)	CANoe / D62Y9	CANoe / D62Y9	Focus	396,436	0	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
(2)	CANoe / D62Y9	CANoe / D62Y9	Kuga	1,233,545	0	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
(3)	CANoe/D62Y9	CANoe/VN5610A	Focus	396,436	0	4.632	0.035	0.033	0.026	0.116	0.141	0.152	0.330
(4)	CANoe/D62Y9	CANoe/VN5610A	Kuga	1,233,545	0	5.535	0.023	0.020	0.017	0.064	0.091	0.126	0.385
(5)	CANoe/VN5610A	CANoe/D62Y9	Focus	396,436	0	10.117	0.215	0.157	0.184	0.506	0.600	0.689	4.847
(6)	CANoe/VN5610A	CANoe/D62Y9	Kuga	1,233,545	0	7.242	0.172	0.138	0.141	0.462	0.583	0.685	1.213
(7)	CANoe/D62Y9	C++/XLDriver VN5610A	Focus	396,436	0	7.168	0.039	0.035	0.029	0.116	0.147	0.169	0.468
(8)	CANoe/D62Y9	C++/XLDriver VN5610A	Kuga	1,233,545	0	8.397	0.026	0.022	0.020	0.068	0.096	0.131	0.383
(9)	C++/XLDriver VN5610A	CANoe/D62Y9	Focus	396,436	0	7.007	0.041	0.095	0.026	0.104	0.328	0.914	6.426
(10)	C++/XLDriver VN5610A	CANoe/D62Y9	Kuga	1,233,545	0	8.416	0.022	0.041	0.014	0.069	0.098	0.662	0.874



(i) Ford Focus (ii) Ford Kuga

FIGURE 14: WCRT analysis for the IDs and cycle times in the Ford Focus (i) and Ford Kuga (ii)

both cars and all IDs, the attack is performed for the duration of the entire trace. Whenever a frame associated with a targeted CAN ID was intercepted by the attacker process, a corresponding malicious frame was immediately injected on the bus – thus, unless there are delays from the attacker process, the skew will be preserved. The experiments were conducted on both platforms, Windows 11 and Windows IoT, and the timing behaviour of the targeted ID was evaluated using the mean and standard deviation of the IAT together with the skew expressed in ppm, which was computed based on the slope extracted with Python's `linregress`.

The results are summarised in Table 8. Across both Windows 11 and Windows IoT, the attacked frames retain nearly identical mean inter-arrival times and the standard deviation of the attack frames remains on the same order of magnitude with moderate variations from the legitimate frames. The skews are also identical to legitimate traffic with the exception of IDs 0x3db and 0x130 where a difference of 1ppm can be observed. This difference is likely due to small rounding errors. In previous works, even larger differences were seen on IDs originating from the same ECU [16], thus the 1ppm deviation should not be a significant concern. This demonstrates that the temporal signature of the legitimate traffic is preserved, thereby increasing attack stealthiness.

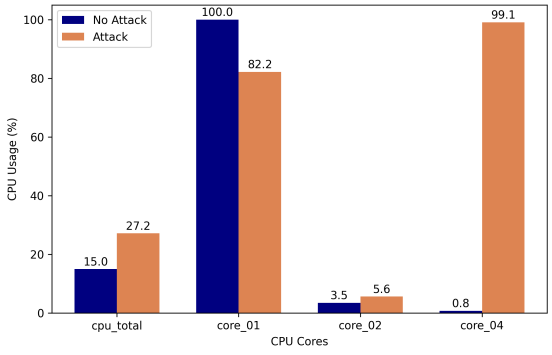
F. RUNTIME OVERHEAD CONSIDERATIONS

In this section, we briefly evaluate the runtime overhead of the trace-driven scheduler designed in Python by measuring both total CPU utilisation and per-core CPU utilisation under normal operation and under mixed attack mode, replay and fuzzy, for the two CAN traces.

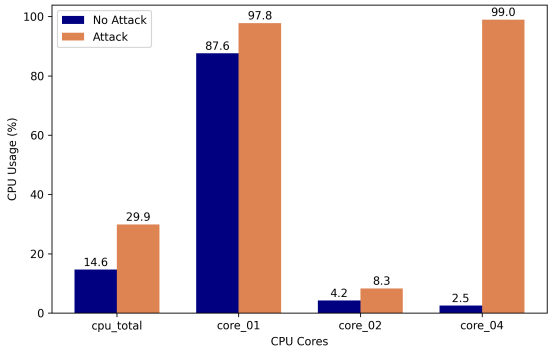
Figure 15 shows the CPU utilisation for (i) the Transmitter thread pinned to CPU 1, (ii) the Receiver thread pinned to CPU 2 and (iii) the Attacker thread pinned to CPU 4, which is isolated only in case an attack scenario is enabled. Core 3 and cores 5–8 are not used by our application and their load is negligible; therefore, they are omitted from the figure. For the low bus load condition, the Ford Focus trace, the total CPU utilisation remains relatively low in both normal and attack scenarios, with a maximum usage of 27.2%. However, the per-core measurements show that the attack workload is concentrated on the isolated core, which shows a CPU increase to almost 100% under attack. A similar behaviour is observed for the high bus load trace, the Ford Kuga trace, where attacker core reaches nearly 100%, while total CPU remains at 29.9%. The increased CPU load can be attributed to the scheduling approach that uses busy-wait loops to reduce wake-up latency. Such a design, which tries to offer precise timing, comes with the cost of increased CPU utilisation. Therefore, the approach is suitable for systems where a dedicated core can be reserved for timing-critical replay or attack

TABLE 8: Skew preservation in case of replay and fuzzing attacks on Ford Focus and Ford Kuga

				Windows 11						Windows IoT					
				Original			Attacked			Original			Attacked		
Car	ID	Period	Attack Type	Mean	σ	Skew	Mean	σ	Skew	Mean	σ	Skew	Mean	σ	Skew
		ms		*ms*	*ms*	*ppm*	*ms*	*ms*	*ppm*	*ms*	*ms*	*ppm*	*ms*	*ms*	*ppm*
Focus	0x200	10	Replay	9.995	0.134	-489	9.995	0.192	-489	9.995	0.126	-489	9.995	0.254	-489
Focus	0x205	10	Fuzzing	9.995	0.245	-489	9.995	0.304	-489	9.995	0.254	-489	9.995	0.284	-489
Focus	0x275	100	Replay	99.940	0.525	-601	99.940	0.567	-601	99.940	0.529	-601	99.940	0.577	-601
Focus	0x3db	1000	Fuzzing	999.832	0.281	-167	999.831	0.400	-166	999.831	0.275	-167	999.831	0.421	-167
Kuga	0x10	10	Replay	10.000	0.212	-3	10.000	0.385	-3	10.000	0.213	-3	10.000	0.394	-3
Kuga	0x160	20	Fuzzing	19.998	0.557	-46	19.998	0.541	-46	19.999	0.555	-46	19.999	0.446	-46
Kuga	0x130	20	Replay	20.000	1.046	9	20.000	1.025	10	20.000	1.127	9	20.000	1.112	10
Kuga	0x140	20	Fuzzing	19.993	0.472	-340	19.993	0.539	-340	19.993	0.649	-340	19.993	0.664	-340



(i) Ford Focus



(ii) Ford Kuga

FIGURE 15: Total CPU utilisation and per-core CPU utilisation under normal and attack condition (i) Ford Focus and (ii) Ford Kuga.

generation. In platforms where resources are constrained, a hybrid sleep-busy-wait strategy may be used to reduce CPU-load, but our experiments showed that sleep intervals increase the timing deviations compared to the busy-wait approach that we used.

VIII. CONCLUSION

Our experimental evaluation leads to the following conclusions that may guide future research on software-based CAN traffic reproduction:

- Both Windows 11 and Windows IoT Enterprise provide several mechanisms for improving soft real-time behaviour, including process affinity `REAL_TIME_PRIORITY_CLASS`, and the high-resolution Query Performance Counter (QPC) – these mechanisms were employed throughout our implementation and contributed to the timing performance.
- For both schedulers, in Python and C++, Windows 11 and Windows IoT Enterprise LTSC exhibited comparable typical replay timings, as indicated by the mean and median timing deviations. The P99 and the maximum of the TAD exhibited more run-to-run variability but did not favour one specific OS.
- As expected, the C++ generally exhibited tighter timing distributions and better worst-case behaviour than Python. This advantage was most apparent in the upper-tail metrics, especially the maximum timing deviations.
- The proposed trace-driven scheduler based on the Vector XL Driver Library achieved better average replay accuracy than the native CANoe environment using the same Vector VN5610A interface, i.e., a mean TAD of 22–41μs vs. 172–215μs. However, our implementation exhibited a number of larger timing spikes that were absent with CANoe, suggesting that while the proposed scheduler provides better average accuracy, CANoe offers more stable tail behaviour with the VN5610A.
- The combination of CANoe with Vector Breakout Boxes connected through dedicated PCI Express acquisition hardware achieved replay accuracies below 1μs, a level of precision that we could not match by the software trace-driven scheduler. Nevertheless, our objective was to provide a portable and flexible replay framework based on USB-connected Vector VN interfaces that can be integrated directly into C++ and Python applications, rather than to replace specialised laboratory hardware.
- In all experimental configurations, the packet switching rate remained zero, confirming that both the Python and C++ implementations preserved the original packet ordering throughout the replay. For applications where preserving packet order and payload is sufficient, both implementations provide functionally correct trace reproduction; timing accuracy, however, remains the primary differentiating factor.
- The experiments with an entry-level off-the-shelf CAN

interface demonstrate that timing performance should not be assumed and CAN interfaces intended for timing-sensitive replay should therefore be experimentally characterised before being used in research or validation studies.

The natural extension of this work toward higher bandwidth communication interfaces like CAN-FD and Ethernet would be of great interest, especially for CAN-FD, where the arbitration mechanism at CAN ID level remains unchanged. Automotive Ethernet is also of immediate interest for us as future work, particularly because of its support for Time-Sensitive Networking (TSN). Evaluating how laboratory tools reproduce Automotive Ethernet traffic would provide valuable insight into their suitability for time-sensitive communication.

REFERENCES

[1] S. Abolhassani Khajeh, M. Saberikamarposhti, and A. M. Rahmani, "Real-time scheduling in iot applications: A systematic review," *Sensors*, vol. 23, no. 1, p. 232, 2022.

[2] F. C. Carvalho and C. E. Pereira, "A runtime stability analysis of clock synchronization precision on a time-triggered bus prototype," *Sba: Controle & Automação Sociedade Brasileira de Automatica*, vol. 20, pp. 45–52, 2009.

[3] G. Rodriguez-Navas, S. Roca, and J. Proenza, "Orthogonal, fault-tolerant, and high-precision clock synchronization for the controller area network," *IEEE Transactions on Industrial Informatics*, vol. 4, no. 2, pp. 92–101, 2008.

[4] M. Akpınar, E. G. Schmidt, and K. Werner Schmidt, "Drift correction for the software-based clock synchronization on controller area network," in *IEEE Symposium on Computers and Communications*, 2020, pp. 1–6.

[5] S. Einspieler, N. Rathakrishnan, A. Prabhakara, B. Steinwender, and W. Elmenreich, "High accuracy software-based clock synchronization over can," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 52, no. 7, pp. 4438–4446, 2022.

[6] M. Akpınar, E. G. Schmidt, and K. W. Schmidt, "Highly accurate clock synchronization with drift correction for the controller area network," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4071–4082, 2022.

[7] F. Luckinger and T. Sauter, "Software-based autosar-compliant precision clock synchronization over can," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 10, pp. 7341–7350, 2022.

[8] M. Akpınar and K. W. Schmidt, "Predictable timestamping for the controller area network: Evaluation and effect on clock synchronization accuracy," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 54, no. 3, pp. 1926–1935, 2024.

[9] H. Ji, Y. Wang, H. Qin, X. Wu, and G. Yu, "Investigating the effects of attack detection for in-vehicle networks based on clock drift of ecus," *IEEE Access*, vol. 6, pp. 49 375–49 384, 2018.

[10] J. Zhou, G. Xie, S. Yu, and R. Li, "Clock-based sender identification and attack detection for automotive can network," *IEEE Access*, vol. 9, pp. 2665–2679, 2021.

[11] O. Schell, C. Oechsler, and M. Kneib, "Asymmetric symbol and skew sender identification for automotive networks," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 3959–3971, 2022.

[12] S. Lee, W. Choi, H. J. Jo, and D. H. Lee, "Errids: An enhanced cumulative timing error-based automotive intrusion detection system," *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 11, pp. 12 406–12 421, 2023.

[13] H. Wei, Q. Ai, W. Zhao, and Y. Zhang, "Real-time security warning and ecu identification for in-vehicle networks," *IEEE Sensors Journal*, vol. 23, no. 17, pp. 20 258–20 266, 2023.

[14] K.-T. Cho and K. G. Shin, "Fingerprinting electronic control units for vehicle intrusion detection," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 911–927.

[15] J. Zhou, G. Xie, H. Zeng, W. Zhang, L. T. Yang, M. Alazab, and R. Li, "A model-based method for enabling source mapping and intrusion detection on proprietary can bus," *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 11, pp. 12 922–12 932, 2023.

[16] L. Popa, B. Groza, C. Jichici, and P.-S. Murvay, "Ecuprint—physical fingerprinting electronic control units on can buses inside cars and sae j1939 compliant vehicles," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 1185–1200, 2022.

[17] S. U. Sagong, X. Ying, A. Clark, L. Bushnell, and R. Poovendran, "Cloaking the clock: Emulating clock skew in controller area networks," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2018, pp. 32–42.

[18] Developing a soft real-time application. [Online]. Available: https://learn.microsoft.com/en-us/windows/iot/iot-enterprise/soft-real-time/soft-real-time-application?source=recommendations

[19] Soft real-time on windows iot enterprise. [Online]. Available: https://learn.microsoft.com/en-us/windows/iot/iot-enterprise/soft-real-time/soft-real-time?source=recommendations

[20] S. Y. Cho, R. Delgado, and B. W. Choi, "Feasibility study for a python-based embedded real-time control system," *Electronics*, vol. 12, no. 6, p. 1426, 2023.

[21] H. K. Lim and T. R. Clites, "Lithe: Bridging best-effort python and real-time c++ for hot-swapping robotic control laws on commodity linux," *arXiv preprint arXiv:2603.07442*, 2026.

[22] K. Ramamritham, C. Shen, O. González, S. Sen, and S. Shirgurkar, "Using windows nt for real-time applications: Experimental observations and recommendations," in *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No. 98TB100245)*. IEEE, 1998, pp. 102–111.

[23] U. Şimşek and A. H. Zaim, "A survey on real-time operating systems for the internet of things from an energy efficiency perspective," *Discover Internet of Things*, 2026.

[24] A. Al-Sakran, M. H. Qutqut, F. Almasalha, H. S. Hassanein, and M. Hijjawi, "An overview of the internet of things closed source operating systems," in *2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC)*. IEEE, 2018, pp. 291–297.

[25] M. R. Kabir, B. B. Y. Ravi, and S. Ray, "Digital twin technologies for vehicular prototyping: A survey," *IEEE Open Journal of Intelligent Transportation Systems*, 2025.

[26] R. Klar, N. Arvidsson, and V. Angelakis, "Maturity of vehicle digital twins: From monitoring to enabling autonomous driving," in *2024 IEEE 27th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2024, pp. 3673–3679.

[27] L. Popa, A. Berdich, and B. Groza, "Cartwin—development of a digital twin for a real-world in-vehicle can network," *Applied Sciences*, vol. 13, no. 1, p. 445, 2022.

[28] S. Sharmin, H. Mansor, A. F. A. Kadir, and A. R. Ismail, "Digital twin-based evaluation of vehicular controller area network intrusion detection systems," *International Journal on Perceptive and Cognitive Computing*, vol. 11, no. 1, pp. 67–80, 2025.

[29] F. Pollicino, D. Stabili, and M. Marchetti, "Performance comparison of timing-based anomaly detectors for controller area network: A reproducible study," *ACM Transactions on Cyber-Physical Systems*, vol. 8, no. 2, pp. 1–24, 2024.

[30] H. M. Song, H. R. Kim, and H. K. Kim, "Intrusion detection system based on the analysis of time intervals of can messages for in-vehicle network," in *2016 international conference on information networking (ICOIN)*. IEEE, 2016, pp. 63–68.

[31] A. Taylor, N. Japkowicz, and S. Leblanc, "Frequency-based anomaly detection for the automotive can bus," in *2015 World Congress on Industrial Control Systems Security (WCICSS)*. IEEE, 2015, pp. 45–49.

[32] M. Gmiden, M. H. Gmiden, and H. Trabelsi, "An intrusion detection method for securing in-vehicle can bus," in *2016 17th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*. IEEE, 2016, pp. 176–180.

[33] M. R. Moore, R. A. Bridges, F. L. Combs, M. S. Starr, and S. J. Prowell, "Modeling inter-signal arrival times for accurate detection of can bus signal injection attacks: a data-driven approach to in-vehicle intrusion detection," in *Proceedings of the 12th annual conference on cyber and information security research*, 2017, pp. 1–4.

[34] C. Young, H. Olufowobi, G. Bloom, and J. Zambreno, "Automotive intrusion detection based on constant can message frequencies across vehicle driving modes," in *Proceedings of the ACM Workshop on Automotive Cybersecurity*, 2019, pp. 9–14.

[35] A. Tomlinson, J. Bryans, S. A. Shaikh, and H. K. Kalutarage, "Detection of automotive can cyber-attacks by identifying packet timing anomalies in

time windows," in *2018 48th Annual IEEE/IFIP international conference on dependable systems and networks workshops (DSN-W)*. IEEE, 2018, pp. 231–238.
[36] M. Hanselmann, T. Strauss, K. Dormann, and H. Ulmer, "Canet: An unsupervised intrusion detection system for high dimensional can bus data," *Ieee Access*, vol. 8, pp. 58 194–58 205, 2020.
[37] G. Dupont, A. Lekidis, J. J. den Hartog, and S. S. Etalle, "Automotive controller area network (can) bus intrusion dataset v2," 2019. [Online]. Available: https://data.4tu.nl/articles/dataset/Automotive_Controller_Area_Network_CAN_Bus_Intrusion_Dataset/12696950/2
[38] M. E. Verma, R. A. Bridges, M. D. Iannacone, S. C. Hollifield, P. Moriano, S. C. Hespeler, B. Kay, and F. L. Combs, "A comprehensive guide to can ids data and introduction of the road dataset," *PLoS one*, vol. 19, no. 1, p. e0296879, 2024.
[39] J. Li, N. Ersotelos, M.-A. Tsompanas, and G. Epiphaniou, "Challenges and opportunities of synthetic data generation for machine learning-based intrusion detection systems in in-vehicle networks," *Vehicular Communications*, p. 100998, 2026.
[40] M. Lenc and P. Pisa, "Scheduling of can message transmission when multiple fifos with assigned priorities are used in rtos drivers," in *18th International CAN Conference*, 2024.
[41] M. Di Natale, "Scheduling the can bus with earliest deadline techniques," in *Proceedings 21st IEEE Real-Time Systems Symposium*, 2000, pp. 259–268.
[42] C.-W. Lin, Q. Zhu, and A. Sangiovanni-Vincentelli, "Security-aware modeling and efficient mapping for can-based real-time distributed automotive systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 11–14, 2014.
[43] P. Li, X. Wang, C. Li, M. Iqbal, A. Al-Dulaimi, P. Casaseca-de-la Higuera *et al.*, "Deep reinforcement learning-based task scheduling and resource allocation for vehicular edge computing: A survey," *IEEE Transactions on Intelligent Transportation Systems*, 2025.
[44] L. Yang, "Task offloading and resource scheduling in a vehicle-rsu-cloud resource environment," *Journal of Grid Computing*, vol. 24, no. 3, p. 21, 2026.
[45] A. Ashok, P. Steenkiste, and F. Bai, "Vehicular cloud computing through dynamic computation offloading," *Computer Communications*, vol. 120, pp. 125–137, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366417304140
[46] P. Arthurs, L. Gillam, P. Krause, N. Wang, K. Halder, and A. Mouzakitis, "A taxonomy and survey of edge cloud computing for intelligent transportation systems and connected vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 7, pp. 6206–6221, 2021.
[47] Q. Yuan, H. Zhou, J. Li, Z. Liu, F. Yang, and X. S. Shen, "Toward efficient content delivery for automated driving services: An edge computing solution," *IEEE Network*, vol. 32, no. 1, pp. 80–86, 2018.
[48] S. Sedaghat and M. Conti, "Cfea-iov: An sdn-enabled iov architecture for time-critical accident notification in automated vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 10, no. 2, pp. 752–761, 2025.
[49] Embedded Systems Academy (EmSA). (2026) Can best and worst case calculator. Accessed: 2026-02-18. [Online]. Available: https://www.esacademy.com/en/library/calculators/can-best-and-worst-case-calculator.html
[50] K. Tindell and A. Burns, "G uaranteeing message latencies on control a rea network (can) 1," in *Proceedings of the 1st International CAN Conference*, 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:12609941
[51] E. C. P. Neto, H. Taslimasa, S. Dadkhah, S. Iqbal, P. Xiong, T. Rahman, and A. A. Ghorbani, "Ciciov2024: Advancing realistic ids approaches against dos and spoofing attack in iov can bus," *Internet of Things*, vol. 26, p. 101209, 2024.
[52] H. Kang, B. I. Kwak, Y. H. Lee, H. Lee, H. Lee, and H. K. Kim, "Car hacking: Attack & defense challenge 2020 dataset," 2021. [Online]. Available: https://dx.doi.org/10.21227/qvr7-n418
[53] M. L. Han, B. I. Kwak, and H. K. Kim, "Anomaly intrusion detection method for vehicular networks based on survival analysis," *Vehicular Communications*, vol. 14, pp. 52–63, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214209618301189
[54] HCRL, "M-can intrusion dataset," https://ocslab.hksecurity.net/Datasets/m-can-intrusion-dataset, 2022, accessed: 2026-03-19.
[55] Sysworxx. [Online]. Available: https://www.systec-electronic.com/en/products/interfaces-gateways-amp-controls/sysworxx-usb-can-module2
[56] Vector Informatik GmbH, *XL Driver Library*, Vector Informatik GmbH, Stuttgart, Germany, 2026, available at https://www.vector.com/int/en/download/xl-driver-library/.
[57] ——, *VN1600 Interface Family Manual*, Vector Informatik GmbH, Stuttgart, Germany, 2025, accessed: 2026-02-18. [Online]. Available: https://cdn.vector.com/cms/content/products/VN16xx/docs/VN1600_Interface_Family_Manual_EN.pdf
[58] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (can) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.
[59] C. in Automation (CiA), "Designing a can network," *CAN knowledge*, 1992-2026.
[60] R. Knueppel, "Standardization of can networks for airborne use through arinc 825," *Airbus Operations GmbH*, 2012.
[61] O. Ikumapayi, H. Olufowobi, J. Daily, T. Hu, I. C. Bertolotti, and G. Bloom, "Canasta: Controller area network authentication schedulability timing analysis," *IEEE Transactions on Vehicular Technology*, vol. 72, no. 8, pp. 10 024–10 036, 2023.



EMILIA CARAGEA is currently a PhD student at Politehnica University of Timisoara. She received her B.Sc. and M.Sc. degrees from the same institution in 2009 and 2011. She has more than 17-years experience in automotive industry as software developer and system technical lead for telematic, ADAS and powertrain controllers. She currently serves as a Technical Project Lead for Battery Management System (BMS) projects at Schaeffler Romania.



BOGDAN GROZA is Professor at Politehnica University of Timisoara (UPT). He received his Dipl.Ing. and Ph.D. degree from UPT in 2004 and 2008 respectively. In 2016 he successfully defended his habilitation thesis having as core subject the design of cryptographic security for automotive embedded devices and networks. He has been actively involved inside UPT with the development of laboratories by Continental Automotive and Vector Informatik. Besides regular participation as a member in national and international research projects in information security, he led the CSEAMAN (2015-2017) and PRESENCE (2018-2020) projects, two research programs dedicated to automotive security funded by the Romanian National Authority for Scientific Research and Innovation.

APPENDIX A - DETAILED MEASUREMENTS

In this appendix, we provide the detailed experimental results that complement the synthesis presented in the main text.

Tables 9, 10, and 11 present the Ford Focus detailed retransmission results of the trace-driven scheduler in C++ and Python for both Windows platforms, Windows 11 and Windows IoT. Tables 12, 13, and 14 present the Ford Kuga detailed retransmission results of the trace-driven scheduler in C++ and Python for both Windows platforms, Windows 11 and Windows IoT.

TABLE 9: Timing performance of the receiver in vehicle and C++ implementation under Windows IoT for Ford Focus

		--						Windows IoT					
		Original						Cpp					
CAN ID	T_{cycle} *ms*	IAT_{mean} *ms*	IAT_{max} *ms*	J_{mean} *ms*	J_{max} *ms*	DMR $\lambda = 0.1$	$\text{LBM}^{\text{id}}_{\lambda}$ $\lambda = 0.1$	IAT_{mean} *ms*	IAT_{max} *ms*	J_{mean} *ms*	J_{max} *ms*	DMR $\lambda = 0.1$	$\text{LBM}^{\text{id}}_{\lambda}$ $\lambda = 0.1$
0x80	14	14.008	14.418	0.000	0.410	0.039	1.000	14.008	14.877	0.000	0.868	0.054	2.000
0x90	14	14.008	14.418	0.000	0.410	0.039	1.000	14.008	14.868	0.000	0.860	0.053	2.000
0x190	16	16.001	17.883	0.002	1.884	0.070	2.000	16.001	17.916	0.002	1.917	0.068	2.000
0x200	10	9.995	10.715	0.001	0.721	0.088	2.000	9.995	10.977	0.001	0.983	0.100	3.000
0x201	10	9.995	10.945	0.001	0.950	0.102	2.000	9.995	11.010	0.001	1.016	0.115	2.000
0x205	10	9.995	10.748	0.001	0.754	0.131	3.000	9.995	11.198	0.001	1.204	0.131	3.000
0x20f	14	14.008	15.163	0.000	1.155	0.135	2.000	14.008	15.270	0.000	1.262	0.144	2.000
0x211	14	14.008	15.155	0.000	1.147	0.167	3.000	14.008	15.450	0.000	1.442	0.176	3.000
0x212	105	105.059	106.881	0.005	1.827	0.096	1.000	105.059	107.389	0.005	2.335	0.090	1.000
0x231	10	9.995	12.870	0.001	2.875	0.312	3.000	9.995	12.927	0.007	2.925	0.318	3.000
0x240	10	10.000	12.042	0.002	2.040	0.108	3.000	10.000	12.149	0.002	2.146	0.148	3.000
0x275	100	99.940	102.990	0.003	3.047	0.031	1.000	99.940	103.096	0.003	3.154	0.034	1.000
0x280	50	49.976	53.297	0.013	3.334	0.230	2.000	49.976	53.305	0.013	3.342	0.230	2.000
0x3db	1000	999.831	1001.153	0.002	1.319	0.000	0.000	999.831	1001.046	0.006	1.221	0.000	0.000
0x420	100	99.951	106.193	0.025	6.218	0.321	2.000	99.951	106.201	0.025	6.226	0.321	2.000
0x424	100	99.951	106.193	0.020	6.222	0.320	2.000	99.951	106.201	0.016	6.234	0.318	2.000
0x428	100	99.951	106.078	0.008	6.119	0.319	2.000	99.951	106.086	0.008	6.128	0.316	2.000
0x430	100	100.004	101.941	0.004	1.942	0.010	1.000	100.004	102.015	0.004	2.015	0.016	1.000
0x433	100	99.940	113.025	0.003	13.083	0.039	2.000	99.940	113.033	0.003	13.091	0.041	2.000
0x460	100	99.936	112.222	0.006	12.280	0.025	1.000	99.936	112.255	0.002	12.321	0.026	1.000
0x4b0	14	14.008	15.704	0.000	1.696	0.304	3.000	14.008	15.720	0.009	1.704	0.309	3.000
0x4f0	999	999.507	1003.684	0.021	4.198	0.000	0.000	999.507	1003.725	0.025	4.243	0.000	0.000
0x4f1	1000	999.507	1003.684	0.005	4.182	0.000	0.000	999.507	1003.717	0.003	4.207	0.000	0.000
0x4f2	1000	1000.035	1002.103	0.054	2.122	0.000	0.000	1000.035	1002.111	0.005	2.081	0.000	0.000
0x4f3	999	999.507	1003.684	0.009	4.186	0.000	0.000	999.507	1003.725	0.021	4.239	0.000	0.000

TABLE 10: Timing performance of the receiver in Python and C++ under Windows IoT for Ford Focus

		Windows IoT						Windows IoT					
		Python						C++					
CAN ID	T_{cycle} *ms*	IAT_{mean} *ms*	IAT_{max} *ms*	J_{mean} *ms*	J_{max} *ms*	DMR $\lambda = 0.1$	$\text{LBM}^{\text{id}}_{\lambda}$ $\lambda = 0.1$	IAT_{mean} *ms*	IAT_{max} *ms*	J_{mean} *ms*	J_{max} *ms*	DMR $\lambda = 0.1$	$\text{LBM}^{\text{id}}_{\lambda}$ $\lambda = 0.1$
0x80	14	14.008	14.967	0.000	0.958	0.053	2.000	14.008	14.877	0.000	0.868	0.054	2.000
0x90	14	14.008	14.967	0.000	0.958	0.052	2.000	14.008	14.868	0.000	0.860	0.053	2.000
0x190	16	16.001	17.940	0.002	1.942	0.065	2.000	16.001	17.916	0.002	1.917	0.068	2.000
0x200	10	9.995	11.313	0.001	1.319	0.100	2.000	9.995	10.977	0.001	0.983	0.100	3.000
0x201	10	9.995	11.280	0.001	1.286	0.112	2.000	9.995	11.010	0.001	1.016	0.115	2.000
0x205	10	9.995	11.313	0.001	1.319	0.130	3.000	9.995	11.198	0.001	1.204	0.131	3.000
0x20f	14	14.008	15.417	0.000	1.409	0.143	2.000	14.008	15.270	0.000	1.262	0.144	2.000
0x211	14	14.008	15.417	0.000	1.409	0.175	3.000	14.008	15.450	0.000	1.442	0.176	3.000
0x212	105	105.058	106.906	0.061	1.909	0.104	1.000	105.059	107.389	0.005	2.335	0.090	1.000
0x231	10	9.995	12.927	0.001	2.933	0.327	3.000	9.995	12.927	0.007	2.925	0.318	3.000
0x240	10	10.000	12.149	0.002	2.146	0.146	3.000	10.000	12.149	0.002	2.146	0.148	3.000
0x275	100	99.940	103.047	0.003	3.105	0.035	1.000	99.940	103.096	0.003	3.154	0.034	1.000
0x280	50	49.976	53.297	0.013	3.334	0.230	2.000	49.976	53.305	0.013	3.342	0.230	2.000
0x3db	1000	999.831	1001.128	0.002	1.294	0.000	0.000	999.831	1001.046	0.006	1.221	0.000	0.000
0x420	100	99.951	106.226	0.025	6.250	0.322	2.000	99.951	106.201	0.025	6.226	0.321	2.000
0x424	100	99.951	106.226	0.025	6.250	0.319	2.000	99.951	106.201	0.016	6.234	0.318	2.000
0x428	100	99.951	106.086	0.008	6.128	0.318	2.000	99.951	106.086	0.008	6.128	0.316	2.000
0x430	100	100.004	102.031	0.004	2.032	0.017	1.000	100.004	102.015	0.004	2.015	0.016	1.000
0x433	100	99.940	113.017	0.003	13.074	0.040	2.000	99.940	113.033	0.003	13.091	0.041	2.000
0x460	100	99.936	112.230	0.002	12.296	0.027	1.000	99.936	112.255	0.002	12.321	0.026	1.000
0x4b0	14	14.008	15.966	0.000	1.958	0.313	3.000	14.008	15.720	0.009	1.704	0.309	3.000
0x4f0	999	999.507	1003.700	0.029	4.223	0.000	0.000	999.507	1003.725	0.025	4.243	0.000	0.000
0x4f1	999	999.507	1003.708	0.017	4.219	0.000	0.000	999.507	1003.717	0.003	4.207	0.000	0.000
0x4f2	1000	1000.035	1002.185	0.046	2.195	0.000	0.000	1000.035	1002.111	0.005	2.081	0.000	0.000
0x4f3	999	999.507	1003.708	0.025	4.227	0.000	0.000	999.507	1003.725	0.021	4.239	0.000	0.000

TABLE 11: Timing performance of the receiver in C++ under Windows 11 and IoT for Ford Focus

		Windows 11						Windows IoT					
		C++						C++					
CAN ID	T_{cycle} *ms*	IAT_{mean} *ms*	IAT_{max} *ms*	J_{mean} *ms*	J_{max} *ms*	DMR $\lambda = 0.1$	LBM^{id}_{λ} $\lambda = 0.1$	IAT_{mean} *ms*	IAT_{max} *ms*	J_{mean} *ms*	J_{max} *ms*	DMR $\lambda = 0.1$	LBM^{id}_{λ} $\lambda = 0.1$
0x80	14	14.008	14.868	0.000	0.860	0.054	2.000	14.008	14.877	0.000	0.868	0.054	2.000
0x90	14	14.008	14.868	0.000	0.860	0.053	2.000	14.008	14.868	0.000	0.860	0.053	2.000
0x190	16	16.001	17.916	0.002	1.917	0.068	2.000	16.001	17.916	0.002	1.917	0.068	2.000
0x200	10	9.995	10.994	0.001	0.999	0.102	3.000	9.995	10.977	0.001	0.983	0.100	3.000
0x201	10	9.995	11.223	0.001	1.229	0.118	3.000	9.995	11.010	0.001	1.016	0.115	2.000
0x205	10	9.995	11.002	0.001	1.008	0.132	3.000	9.995	11.198	0.001	1.204	0.131	3.000
0x20f	14	14.008	15.507	0.000	1.499	0.144	2.000	14.008	15.270	0.000	1.262	0.144	2.000
0x211	14	14.008	15.507	0.000	1.499	0.176	3.000	14.008	15.450	0.000	1.442	0.176	3.000
0x212	105	105.059	106.971	0.001	1.913	0.089	1.000	105.059	107.389	0.005	2.335	0.090	1.000
0x231	10	9.995	12.935	0.007	2.933	0.319	3.000	9.995	12.927	0.007	2.925	0.318	3.000
0x240	10	10.000	12.149	0.002	2.146	0.150	3.000	10.000	12.149	0.002	2.146	0.148	3.000
0x275	100	99.940	103.023	0.003	3.080	0.034	1.000	99.940	103.096	0.003	3.154	0.034	1.000
0x280	50	49.976	53.297	0.013	3.334	0.229	2.000	49.976	53.305	0.013	3.342	0.230	2.000
0x3db	1000	999.831	1001.112	0.002	1.278	0.000	0.000	999.831	1001.046	0.006	1.221	0.000	0.000
0x420	100	99.951	106.250	0.020	6.279	0.322	2.000	99.951	106.201	0.025	6.226	0.321	2.000
0x424	100	99.951	106.250	0.016	6.283	0.320	2.000	99.951	106.201	0.016	6.234	0.318	2.000
0x428	100	99.951	106.086	0.008	6.128	0.318	2.000	99.951	106.086	0.008	6.128	0.316	2.000
0x430	100	100.004	102.031	0.004	2.032	0.017	1.000	100.004	102.015	0.004	2.015	0.016	1.000
0x433	100	99.940	113.033	0.003	13.091	0.041	2.000	99.940	113.033	0.003	13.091	0.041	2.000
0x460	100	99.936	112.189	0.006	12.247	0.025	1.000	99.936	112.255	0.002	12.321	0.026	1.000
0x4b0	14	14.008	15.737	0.009	1.720	0.308	3.000	14.008	15.720	0.009	1.704	0.309	3.000
0x4f0	999	999.507	1003.725	0.013	4.231	0.000	0.000	999.507	1003.725	0.025	4.243	0.000	0.000
0x4f1	1000	999.507	1003.725	0.005	4.223	0.000	0.000	999.507	1003.717	0.003	4.207	0.000	0.000
0x4f2	1000	1000.035	1002.185	0.005	2.154	0.000	0.000	1000.035	1002.111	0.005	2.081	0.000	0.000
0x4f3	999	999.507	1003.725	0.009	4.227	0.000	0.000	999.507	1003.725	0.021	4.239	0.000	0.000

TABLE 12: Timing performance of the receiver in real busload conditions Ford Kuga original vs Windows IoT C++

		--						Windows IoT					
		Original						C++					
CAN ID	T_{cycle} *ms*	IAT_{mean} *ms*	IAT_{max} *ms*	J_{mean} *ms*	J_{max} *ms*	DMR $\lambda = 0.1$	Λ $\lambda = 0.1$	IAT_{mean} *ms*	IAT_{max} *ms*	J_{mean} *ms*	J_{max} *ms*	DMR $\lambda = 0.1$	LBM_{λ}^{id} $\lambda = 0.1$
0x10	10	10.000	10.306	0.002	0.303	0.189	2.000	10.000	11.944	0.002	1.942	0.202	3.000
0x30	10	10.000	10.805	0.006	0.811	0.296	3.000	10.000	11.198	0.006	1.204	0.299	3.000
0x4a	1002	1002.185	1002.660	0.009	0.483	0.000	0.000	1002.185	1002.717	0.001	0.532	0.000	0.000
0x4b	1002	1002.185	1002.766	0.009	0.590	0.000	0.000	1002.185	1002.848	0.001	0.664	0.000	0.000
0x50	10	10.000	10.953	0.002	0.950	0.232	3.000	10.000	11.543	0.002	1.540	0.259	3.000
0x60	15	15.000	16.138	0.001	1.139	0.386	3.000	15.000	16.171	0.001	1.171	0.386	3.000
0x6a	20	20.000	20.890	0.003	0.893	0.042	2.000	20.000	20.914	0.003	0.918	0.052	2.000
0x70	20	20.000	21.234	0.003	1.237	0.295	2.000	20.000	22.110	0.004	2.114	0.301	2.000
0x80	20	20.000	21.234	0.003	1.237	0.298	2.000	20.000	22.061	0.004	2.064	0.302	2.000
0x90	10	10.000	11.747	0.002	1.745	0.484	3.000	10.000	12.149	0.002	2.146	0.483	3.000
0xa0	15	15.000	17.105	0.008	2.097	0.493	3.000	15.000	17.236	0.008	2.228	0.492	3.000
0xb0	20	19.993	21.692	0.003	1.696	0.104	2.000	19.993	22.127	0.003	2.130	0.118	2.000
0xc0	20	20.000	21.799	0.003	1.802	0.348	2.000	20.000	22.053	0.003	2.056	0.349	2.000
0xc8	20	20.000	21.742	0.003	1.745	0.172	3.000	20.000	22.520	0.003	2.523	0.184	3.000
0xd0	20	20.000	20.881	0.003	0.885	0.044	2.000	20.000	20.914	0.003	0.918	0.054	2.000
0xe0	20	20.000	20.832	0.003	0.836	0.047	2.000	20.000	20.955	0.003	0.958	0.057	2.000
0xf0	10	10.000	11.674	0.002	1.671	0.500	2.000	10.000	11.682	0.006	1.688	0.499	3.000
0xf5	10	10.000	11.674	0.006	1.679	0.500	2.000	10.000	11.682	0.006	1.688	0.498	3.000
0xf8	20	20.000	22.422	0.004	2.425	0.350	2.000	20.000	22.446	0.004	2.449	0.352	2.000
0x100	20	20.000	21.135	0.003	1.139	0.072	3.000	20.000	22.036	0.003	2.040	0.085	3.000
0x120	20	20.000	22.675	0.004	2.679	0.350	2.000	20.000	23.020	0.004	3.023	0.352	2.000
0x130	20	20.000	22.675	0.004	2.679	0.351	2.000	20.000	22.774	0.004	2.777	0.353	2.000
0x138	20	20.000	22.675	0.004	2.679	0.359	2.000	20.000	22.782	0.004	2.785	0.361	2.000
0x140	20	19.993	24.117	0.003	4.121	0.152	2.000	19.993	24.109	0.003	4.112	0.162	2.000
0x150	20	20.000	23.888	0.003	3.891	0.170	3.000	20.000	24.199	0.003	4.202	0.175	3.000
0x160	20	19.999	23.871	0.002	3.875	0.096	3.000	19.999	23.912	0.002	3.916	0.110	3.000
0x170	20	19.992	24.314	0.005	4.317	0.141	2.000	19.992	24.732	0.005	4.735	0.154	2.000
0x17e	100	99.998	104.014	0.007	4.022	0.005	1.000	99.998	104.047	0.001	4.047	0.006	1.000
0x180	20	19.999	24.371	0.002	4.375	0.123	3.000	19.999	24.904	0.002	4.907	0.133	3.000
0x190	10	10.000	14.844	0.003	4.841	0.485	3.000	10.000	15.352	0.003	5.349	0.484	4.000
0x1a0	20	20.000	23.871	0.004	3.875	0.367	2.000	20.000	23.904	0.004	3.908	0.369	3.000
0x1b0	30	30.000	35.308	0.023	5.284	0.497	1.000	30.000	35.553	0.023	5.530	0.497	1.000
0x1c0	20	19.999	24.928	0.002	4.932	0.189	3.000	19.999	24.928	0.002	4.932	0.195	3.000
0x1d0	20	19.999	24.863	0.002	4.866	0.206	3.000	19.999	24.986	0.002	4.989	0.213	3.000
0x1e0	20	19.999	24.855	0.002	4.858	0.228	3.000	19.999	24.871	0.002	4.874	0.235	3.000
0x200	25	25.000	32.178	0.088	7.266	0.423	4.000	25.000	32.276	0.096	7.373	0.425	4.000
0x208	25	25.055	32.154	0.012	7.111	0.369	4.000	25.055	32.154	0.020	7.119	0.373	4.000
0x210	20	19.999	25.092	0.002	5.095	0.276	3.000	19.999	25.076	0.002	5.079	0.281	3.000
0x213	20	19.999	25.584	0.002	5.587	0.291	3.000	19.999	25.592	0.002	5.595	0.296	3.000
0x218	30	29.999	37.265	0.008	7.274	0.481	1.000	29.999	37.405	0.001	7.406	0.480	2.000
0x229	40	39.984	47.284	0.010	7.291	0.163	2.000	39.984	47.292	0.010	7.299	0.170	2.000
0x252	20	19.999	25.559	0.002	5.562	0.354	3.000	19.999	25.559	0.002	5.562	0.354	3.000
0x260	25	25.000	32.752	0.055	7.807	0.424	3.000	25.000	32.784	0.055	7.840	0.425	3.000
0x269	30	30.016	37.782	0.000	7.766	0.300	2.000	30.016	37.773	0.000	7.758	0.304	2.000
0x270	30	30.001	37.470	0.034	7.504	0.491	2.000	30.001	37.724	0.034	7.758	0.490	2.000
0x280	30	30.000	37.634	0.018	7.651	0.484	2.000	30.000	37.716	0.018	7.733	0.484	2.000
0x290	30	30.000	35.881	0.009	5.890	0.385	2.000	30.000	36.389	0.001	6.390	0.382	2.000
0x2a0	40	39.984	48.562	0.002	8.577	0.244	3.000	39.984	48.579	0.002	8.593	0.247	3.000
0x2a5	40	39.984	48.284	0.007	8.307	0.271	3.000	39.984	48.300	0.007	8.323	0.274	3.000
0x2b0	40	40.000	43.426	0.007	3.432	0.149	2.000	40.000	43.434	0.007	3.441	0.152	2.000
0x2d0	40	39.998	46.260	0.005	6.267	0.279	2.000	39.998	46.293	0.005	6.300	0.283	2.000
0x2d4	60	59.997	65.602	0.001	5.603	0.182	2.000	59.997	65.634	0.001	5.636	0.185	2.000
0x2d8	60	60.001	64.897	0.003	4.899	0.163	2.000	60.001	64.897	0.003	4.899	0.167	2.000
0x2e0	70	70.065	80.355	0.064	10.355	0.313	3.000	70.065	80.372	0.064	10.371	0.315	3.000
0x2f0	90	90.001	99.222	0.078	9.298	0.433	2.000	90.001	99.238	0.078	9.314	0.433	2.000
0x310	100	99.998	109.855	0.015	9.871	0.195	2.000	99.998	109.871	0.015	9.888	0.199	2.000
0x340	120	120.002	124.838	0.003	4.833	0.050	2.000	120.002	124.838	0.003	4.833	0.054	2.000
0x360	150	149.996	158.851	0.001	8.856	0.235	2.000	149.996	158.843	0.013	8.860	0.238	2.000
0x380	300	299.995	307.470	0.004	7.479	0.006	1.000	299.995	307.479	0.004	7.487	0.006	1.000
0x388	801	801.745	810.975	0.248	9.478	0.021	1.000	801.745	810.967	0.244	9.466	0.021	1.000
0x3b4	300	299.995	307.470	0.004	7.479	0.015	1.000	299.995	307.479	0.004	7.487	0.015	1.000
0x3e2	1000	999.995	1007.297	0.014	7.315	0.000	0.000	999.995	1007.329	0.006	7.340	0.000	0.000
0x3ea	1000	999.995	1007.305	0.022	7.332	0.000	0.000	999.995	1007.329	0.018	7.352	0.000	0.000
0x400	250	249.996	257.376	0.001	7.381	0.185	1.000	249.996	257.327	0.001	7.332	0.184	1.000
0x405	250	257.227	502.546	7.207	252.527	0.186	2.000	257.227	502.546	7.199	252.518	0.187	2.000
0x40a	125	124.997	134.324	0.061	9.388	0.313	3.000	124.997	134.308	0.061	9.372	0.313	3.000
0x420	600	599.991	607.150	0.009	7.168	0.003	1.000	599.991	607.166	0.005	7.180	0.003	1.000
0x435	300	299.991	307.454	0.000	7.463	0.026	1.000	299.991	307.462	0.000	7.471	0.026	1.000
0x455	100	100.000	108.806	0.016	8.823	0.205	2.000	100.000	108.806	0.016	8.823	0.206	2.000
0x581	1000	999.975	1005.101	0.002	5.124	0.000	0.000	999.975	1005.117	0.002	5.145	0.000	0.000

TABLE 13: Timing performance of the receiver in Python and C++ under Windows IoT for Ford Kuga

		Windows IoT						Windows IoT					
		Python						C++					
CAN ID	T_{cycle} *ms*	IAT_{mean} *ms*	IAT_{max} *ms*	J_{mean} *ms*	J_{max} *ms*	DMR $\lambda = 0.1$	Λ $\lambda = 0.1$	IAT_{mean} *ms*	IAT_{max} *ms*	J_{mean} *ms*	J_{max} *ms*	DMR $\lambda = 0.1$	LBM_{λ}^{id} $\lambda = 0.1$
x10	10	10.000	10.928	0.003	0.926	0.216	3.000	10.000	11.944	0.002	1.942	0.202	3.000
0x30	10	10.000	11.166	0.005	1.171	0.304	3.000	10.000	11.198	0.006	1.204	0.299	3.000
0x4a	1002	1002.185	1002.693	0.001	0.508	0.000	0.000	1002.185	1002.717	0.001	0.532	0.000	0.000
0x4b	1002	1002.185	1002.807	0.001	0.623	0.000	0.000	1002.185	1002.848	0.001	0.664	0.000	0.000
0x50	10	10.000	11.231	0.003	1.229	0.282	3.000	10.000	11.543	0.002	1.540	0.259	3.000
0x60	15	15.000	16.318	0.008	1.311	0.387	3.000	15.000	16.171	0.001	1.171	0.386	3.000
0x6a	20	20.000	20.922	0.003	0.926	0.054	2.000	20.000	20.914	0.003	0.918	0.052	2.000
0x70	20	20.000	21.340	0.005	1.335	0.297	2.000	20.000	22.110	0.004	2.114	0.301	2.000
0x80	20	20.000	21.340	0.005	1.335	0.300	2.000	20.000	22.061	0.004	2.064	0.302	2.000
0x90	10	10.000	11.829	0.003	1.827	0.482	3.000	10.000	12.149	0.002	2.146	0.483	3.000
0xa0	15	15.000	17.162	0.024	2.138	0.492	3.000	15.000	17.236	0.008	2.228	0.492	3.000
0xb0	20	19.993	21.766	0.004	1.769	0.122	2.000	19.993	22.127	0.003	2.130	0.118	2.000
0xc0	20	20.000	21.815	0.005	1.810	0.349	2.000	20.000	22.053	0.003	2.056	0.349	2.000
0xc8	20	19.999	21.848	0.002	1.851	0.188	3.000	20.000	22.520	0.003	2.523	0.184	3.000
0xd0	20	20.000	20.922	0.003	0.926	0.057	2.000	20.000	20.914	0.003	0.918	0.054	2.000
0xe0	20	20.000	20.832	0.003	0.836	0.059	2.000	20.000	20.955	0.003	0.958	0.057	2.000
0xf0	10	10.000	11.649	0.006	1.655	0.499	3.000	10.000	11.682	0.006	1.688	0.499	3.000
0xf5	10	10.000	11.649	0.006	1.655	0.499	3.000	10.000	11.682	0.006	1.688	0.498	3.000
0xf8	20	20.000	22.413	0.005	2.408	0.351	2.000	20.000	22.446	0.004	2.449	0.352	2.000
0x100	20	19.999	21.185	0.003	1.188	0.086	3.000	20.000	22.036	0.003	2.040	0.085	3.000
0x120	20	20.000	23.216	0.005	3.211	0.352	2.000	20.000	23.020	0.004	3.023	0.352	2.000
0x130	20	20.000	23.224	0.005	3.219	0.352	2.000	20.000	22.774	0.004	2.777	0.353	2.000
0x138	20	20.000	23.216	0.005	3.211	0.360	2.000	20.000	22.782	0.004	2.785	0.361	2.000
0x140	20	19.993	24.175	0.004	4.178	0.165	2.000	19.993	24.109	0.003	4.112	0.162	2.000
0x150	20	19.999	24.412	0.002	4.415	0.174	3.000	20.000	24.199	0.003	4.202	0.175	3.000
0x160	20	19.999	23.929	0.002	3.932	0.121	3.000	19.999	23.912	0.002	3.916	0.110	3.000
0x170	20	19.991	24.330	0.005	4.334	0.156	2.000	19.992	24.732	0.005	4.735	0.154	2.000
0x17e	100	99.998	104.055	0.007	4.063	0.007	1.000	99.998	104.047	0.001	4.047	0.006	1.000
0x180	20	19.999	24.412	0.002	4.415	0.140	3.000	19.999	24.904	0.002	4.907	0.133	3.000
0x190	10	9.999	15.532	0.005	5.538	0.486	3.000	10.000	15.352	0.003	5.349	0.484	4.000
0x1a0	20	20.000	23.880	0.005	3.875	0.367	2.000	20.000	23.904	0.004	3.908	0.369	3.000
0x1b0	30	30.000	35.611	0.009	5.620	0.497	1.000	30.000	35.553	0.023	5.530	0.497	1.000
0x1c0	20	19.999	25.338	0.002	5.341	0.199	3.000	19.999	24.928	0.002	4.932	0.195	3.000
0x1d0	20	19.999	25.035	0.002	5.038	0.216	3.000	19.999	24.986	0.002	4.989	0.213	3.000
0x1e0	20	19.999	24.912	0.002	4.915	0.239	3.000	19.999	24.871	0.002	4.874	0.235	3.000
0x200	25	24.999	32.227	0.104	7.332	0.425	4.000	25.000	32.276	0.096	7.373	0.425	4.000
0x208	25	25.054	32.195	0.020	7.160	0.374	4.000	25.055	32.154	0.020	7.119	0.373	4.000
0x210	20	19.999	25.207	0.002	5.210	0.281	3.000	19.999	25.076	0.002	5.079	0.281	3.000
0x213	20	19.999	25.625	0.002	5.628	0.296	3.000	19.999	25.592	0.002	5.595	0.296	3.000
0x218	30	29.998	37.249	0.015	7.266	0.483	1.000	29.999	37.405	0.001	7.406	0.480	2.000
0x229	40	39.983	47.317	0.011	7.324	0.171	2.000	39.984	47.292	0.010	7.299	0.170	2.000
0x252	20	19.999	25.633	0.002	5.636	0.352	3.000	19.999	25.559	0.002	5.562	0.354	3.000
0x260	25	24.999	32.842	0.054	7.897	0.427	3.000	25.000	32.784	0.055	7.840	0.425	3.000
0x269	30	30.015	37.863	0.000	7.848	0.305	2.000	30.016	37.773	0.000	7.758	0.304	2.000
0x270	30	30.000	37.552	0.025	7.578	0.489	2.000	30.001	37.724	0.034	7.758	0.490	2.000
0x280	30	30.000	37.683	0.017	7.700	0.483	2.000	30.000	37.716	0.018	7.733	0.484	2.000
0x290	30	29.999	35.889	0.000	5.890	0.382	2.000	30.000	36.389	0.001	6.390	0.382	2.000
0x2a0	40	39.982	48.644	0.003	8.659	0.249	3.000	39.984	48.579	0.002	8.593	0.247	3.000
0x2a5	40	39.983	48.349	0.014	8.380	0.276	3.000	39.984	48.300	0.007	8.323	0.274	3.000
0x2b0	40	40.000	43.434	0.007	3.441	0.152	2.000	40.000	43.434	0.007	3.441	0.152	2.000
0x2d0	40	39.997	46.285	0.004	6.291	0.283	2.000	39.998	46.293	0.005	6.300	0.283	2.000
0x2d4	60	59.997	65.618	0.002	5.620	0.185	2.000	59.997	65.634	0.001	5.636	0.185	2.000
0x2d8	60	60.001	64.954	0.003	4.956	0.165	2.000	60.001	64.897	0.003	4.899	0.167	2.000
0x2e0	70	70.065	80.445	0.064	10.445	0.315	3.000	70.065	80.372	0.064	10.371	0.315	3.000
0x2f0	90	89.999	99.312	0.059	9.372	0.433	2.000	90.001	99.238	0.078	9.314	0.433	2.000
0x310	100	99.998	109.937	0.015	9.953	0.199	2.000	99.998	109.871	0.015	9.888	0.199	2.000
0x340	120	120.002	124.854	0.003	4.850	0.052	2.000	120.002	124.838	0.003	4.833	0.054	2.000
0x360	150	149.996	158.835	0.001	8.839	0.236	2.000	149.996	158.843	0.013	8.860	0.238	2.000
0x380	300	299.995	307.470	0.004	7.479	0.006	1.000	299.995	307.479	0.004	7.487	0.006	1.000
0x388	802	801.745	811.057	0.236	9.548	0.021	1.000	801.745	810.967	0.244	9.466	0.021	1.000
0x3b4	300	299.995	307.462	0.004	7.471	0.015	1.000	299.995	307.479	0.004	7.487	0.015	1.000
0x3e2	1000	999.995	1007.313	0.010	7.328	0.000	0.000	999.995	1007.329	0.006	7.340	0.000	0.000
0x3ea	1000	999.995	1007.337	0.014	7.356	0.000	0.000	999.995	1007.329	0.018	7.352	0.000	0.000
0x400	250	249.996	257.409	0.012	7.401	0.183	1.000	249.996	257.327	0.001	7.332	0.184	1.000
0x405	250	257.227	502.546	7.199	252.518	0.186	2.000	257.227	502.546	7.199	252.518	0.187	2.000
0x40a	125	124.997	134.365	0.061	9.429	0.312	2.000	124.997	134.308	0.061	9.372	0.313	3.000
0x420	600	599.991	607.109	0.009	7.127	0.003	1.000	599.991	607.166	0.005	7.180	0.003	1.000
0x435	300	299.991	307.388	0.000	7.397	0.026	1.000	299.991	307.462	0.000	7.471	0.026	1.000
0x455	100	99.997	108.872	0.013	8.888	0.207	2.000	100.000	108.806	0.016	8.823	0.206	2.000
0x581	1000	999.975	1005.191	0.002	5.218	0.000	0.000	999.975	1005.117	0.002	5.145	0.000	0.000

TABLE 14: Timing performance of the receiver in real busload conditions Ford Kuga C++ in Windows11 and Windows IoT

		Windows 11						Windows IoT					
		C++						C++					
CAN ID	T_{cycle} *ms*	IAT_{mean} *ms*	IAT_{max} *ms*	J_{mean} *ms*	J_{max} *ms*	DMR $\lambda = 0.1$	Λ $\lambda = 0.1$	IAT_{mean} *ms*	IAT_{max} *ms*	J_{mean} *ms*	J_{max} *ms*	DMR $\lambda = 0.1$	LBM_{λ}^{id} $\lambda = 0.1$
0x10	10	10.000	11.436	0.002	1.434	0.201	3.000	10.000	11.944	0.002	1.942	0.202	3.000
0x30	10	10.000	11.149	0.006	1.155	0.299	3.000	10.000	11.198	0.006	1.204	0.299	3.000
0x4a	1002	1002.185	1004.102	0.001	1.917	0.000	0.000	1002.185	1002.717	0.001	0.532	0.000	0.000
0x4b	1002	1002.185	1004.102	0.001	1.917	0.000	0.000	1002.185	1002.848	0.001	0.664	0.000	0.000
0x50	10	10.000	11.428	0.002	1.425	0.258	3.000	10.000	11.543	0.002	1.540	0.259	3.000
0x60	15	15.000	17.146	0.001	2.146	0.386	3.000	15.000	16.171	0.001	1.171	0.386	3.000
0x6a	20	20.000	21.479	0.003	1.483	0.052	2.000	20.000	20.914	0.003	0.918	0.052	2.000
0x70	20	20.000	21.307	0.003	1.311	0.302	2.000	20.000	22.110	0.004	2.114	0.301	2.000
0x80	20	20.000	21.553	0.003	1.556	0.302	2.000	20.000	22.061	0.004	2.064	0.302	2.000
0x90	10	10.000	12.034	0.002	2.032	0.482	3.000	10.000	12.149	0.002	2.146	0.483	3.000
0xa0	15	15.000	17.244	0.008	2.236	0.492	3.000	15.000	17.236	0.008	2.228	0.492	3.000
0xb0	20	19.993	22.028	0.003	2.032	0.118	2.000	19.993	22.127	0.003	2.130	0.118	2.000
0xc0	20	20.000	21.897	0.003	1.901	0.349	2.000	20.000	22.053	0.003	2.056	0.349	2.000
0xc8	20	20.000	21.758	0.003	1.761	0.184	3.000	20.000	22.520	0.003	2.523	0.184	3.000
0xd0	20	20.000	21.479	0.003	1.483	0.054	2.000	20.000	20.914	0.003	0.918	0.054	2.000
0xe0	20	20.000	21.447	0.003	1.450	0.057	2.000	20.000	20.955	0.003	0.958	0.057	2.000
0xf0	10	10.000	12.132	0.011	2.122	0.499	3.000	10.000	11.682	0.006	1.688	0.499	3.000
0xf5	10	10.000	12.870	0.011	2.859	0.498	2.000	10.000	11.682	0.006	1.688	0.498	3.000
0xf8	20	20.000	22.634	0.004	2.638	0.351	2.000	20.000	22.446	0.004	2.449	0.352	2.000
0x100	20	20.000	22.045	0.003	2.048	0.084	3.000	20.000	22.036	0.003	2.040	0.085	3.000
0x120	20	20.000	22.774	0.004	2.777	0.351	2.000	20.000	23.020	0.004	3.023	0.352	2.000
0x130	20	20.000	23.011	0.004	3.015	0.353	2.000	20.000	22.774	0.004	2.777	0.353	2.000
0x138	20	20.000	23.011	0.004	3.015	0.360	2.000	20.000	22.782	0.004	2.785	0.361	2.000
0x140	20	19.993	24.117	0.003	4.121	0.162	2.000	19.993	24.109	0.003	4.112	0.162	2.000
0x150	20	20.000	23.896	0.003	3.899	0.173	3.000	20.000	24.199	0.003	4.202	0.175	3.000
0x160	20	19.999	23.904	0.002	3.908	0.109	3.000	19.999	23.912	0.002	3.916	0.110	3.000
0x170	20	19.992	24.322	0.005	4.325	0.152	2.000	19.992	24.732	0.005	4.735	0.154	2.000
0x17e	100	99.998	104.022	0.007	4.030	0.006	1.000	99.998	104.047	0.001	4.047	0.006	1.000
0x180	20	19.999	24.371	0.002	4.375	0.132	3.000	19.999	24.904	0.002	4.907	0.133	3.000
0x190	10	10.000	14.877	0.003	4.874	0.484	3.000	10.000	15.352	0.003	5.349	0.484	4.000
0x1a0	20	20.000	23.904	0.004	3.908	0.368	3.000	20.000	23.904	0.004	3.908	0.369	3.000
0x1b0	30	30.000	35.406	0.001	5.407	0.498	1.000	30.000	35.553	0.023	5.530	0.497	1.000
0x1c0	20	19.999	24.912	0.002	4.915	0.195	3.000	19.999	24.928	0.002	4.932	0.195	3.000
0x1d0	20	19.999	24.928	0.002	4.932	0.212	3.000	19.999	24.986	0.002	4.989	0.213	3.000
0x1e0	20	19.999	24.863	0.002	4.866	0.235	3.000	19.999	24.871	0.002	4.874	0.235	3.000
0x200	25	25.000	32.801	0.088	7.889	0.423	4.000	25.000	32.276	0.096	7.373	0.425	4.000
0x208	25	25.055	32.186	0.020	7.152	0.374	4.000	25.055	32.154	0.020	7.119	0.373	4.000
0x210	20	19.999	25.084	0.002	5.087	0.279	3.000	19.999	25.076	0.002	5.079	0.281	3.000
0x213	20	19.999	25.510	0.002	5.513	0.294	3.000	19.999	25.592	0.002	5.595	0.296	3.000
0x218	30	29.999	37.306	0.008	7.315	0.481	1.000	29.999	37.405	0.001	7.406	0.480	2.000
0x229	40	39.984	47.342	0.010	7.348	0.170	2.000	39.984	47.292	0.010	7.299	0.170	2.000
0x252	20	19.999	25.821	0.002	5.825	0.354	3.000	19.999	25.559	0.002	5.562	0.354	3.000
0x260	25	25.000	32.768	0.055	7.823	0.425	3.000	25.000	32.784	0.055	7.840	0.425	3.000
0x269	30	30.016	37.806	0.000	7.791	0.304	2.000	30.016	37.773	0.000	7.758	0.304	2.000
0x270	30	30.001	37.618	0.026	7.643	0.490	2.000	30.001	37.724	0.034	7.758	0.490	2.000
0x280	30	30.000	37.618	0.018	7.635	0.484	2.000	30.000	37.716	0.018	7.733	0.484	2.000
0x290	30	30.000	35.881	0.009	5.890	0.385	3.000	30.000	36.389	0.001	6.390	0.382	2.000
0x2a0	40	39.984	48.570	0.002	8.585	0.248	3.000	39.984	48.579	0.002	8.593	0.247	3.000
0x2a5	40	39.984	48.341	0.007	8.364	0.274	3.000	39.984	48.300	0.007	8.323	0.274	3.000
0x2b0	40	40.000	43.393	0.007	3.400	0.152	2.000	40.000	43.434	0.007	3.441	0.152	2.000
0x2d0	40	39.998	46.277	0.005	6.283	0.282	2.000	39.998	46.293	0.005	6.300	0.283	2.000
0x2d4	60	59.997	65.593	0.001	5.595	0.187	2.000	59.997	65.634	0.001	5.636	0.185	2.000
0x2d8	60	60.001	64.922	0.003	4.923	0.168	2.000	60.001	64.897	0.003	4.899	0.167	2.000
0x2e0	70	70.065	80.380	0.064	10.379	0.316	3.000	70.065	80.372	0.064	10.371	0.315	3.000
0x2f0	90	90.001	99.230	0.070	9.298	0.433	2.000	90.001	99.238	0.078	9.314	0.433	2.000
0x310	100	99.998	109.871	0.015	9.888	0.199	2.000	99.998	109.871	0.015	9.888	0.199	2.000
0x340	120	120.002	124.813	0.003	4.809	0.054	2.000	120.002	124.838	0.003	4.833	0.054	2.000
0x360	150	149.996	158.876	0.009	8.888	0.236	2.000	149.996	158.843	0.013	8.860	0.238	2.000
0x380	300	299.995	307.511	0.004	7.520	0.006	1.000	299.995	307.479	0.004	7.487	0.006	1.000
0x388	801	801.745	810.967	0.248	9.470	0.023	1.000	801.745	810.967	0.244	9.466	0.021	1.000
0x3b4	300	299.995	307.495	0.004	7.504	0.015	1.000	299.995	307.479	0.004	7.487	0.015	1.000
0x3e2	1000	999.995	1007.313	0.014	7.332	0.000	0.000	999.995	1007.329	0.006	7.340	0.000	0.000
0x3ea	1000	999.995	1007.321	0.014	7.340	0.000	0.000	999.995	1007.329	0.018	7.352	0.000	0.000
0x400	250	249.996	257.368	0.001	7.373	0.185	1.000	249.996	257.327	0.001	7.332	0.184	1.000
0x405	250	257.227	502.776	7.203	252.752	0.188	2.000	257.227	502.546	7.199	252.518	0.187	2.000
0x40a	125	124.997	134.341	0.069	9.413	0.313	3.000	124.997	134.308	0.061	9.372	0.313	3.000
0x420	600	599.991	607.166	0.005	7.180	0.003	1.000	599.991	607.166	0.005	7.180	0.003	1.000
0x435	300	299.991	307.462	0.000	7.471	0.026	1.000	299.991	307.462	0.000	7.471	0.026	1.000
0x455	100	100.000	108.814	0.016	8.831	0.205	2.000	100.000	108.806	0.016	8.823	0.206	2.000
0x581	1000	999.975	1005.117	0.002	5.145	0.000	0.000	999.975	1005.117	0.002	5.145	0.000	0.000