IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 12 August 2022, accepted 2 September 2022, date of publication 6 September 2022, date of current version 15 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3204746

RESEARCH ARTICLE

Android Head Units vs. In-Vehicle ECUs: Performance Assessment for Deploying In-Vehicle Intrusion Detection Systems for the CAN Bus

TUDOR ANDREICA, CHRISTIAN-DANIEL CURIAC, CAMIL JICHICI, AND BOGDAN GROZA, (Member, IEEE)
Faculty of Automatics and Computers, Politehnica University of Timisoara, 300223 Timişoara, Romania
Corresponding author: Bogdan Groza (bogdan.groza@aut.upt.ro)

ABSTRACT Following the numerous attacks that exploited vulnerabilities of Controller Area Networks (CAN), intrusion detection systems have become a topic of prime importance for in-vehicle buses. Newer in-vehicle communication layers, such as CAN-FD, despite the larger payloads which can easily integrate cryptographic elements, need similar attention. But detecting intrusions may call for demanding algorithms that are not computationally cheap while timely detection is necessary in order to process frames in real-time and take the appropriate actions. In this work we evaluate the performance of several binary classifiers on traditional in-vehicle Electronic Control Units (ECUs) and compare them to modern Android devices which have become widespread inside cars with the adoption of Android-capable infotainment systems. Needless to say, these modern devices benefit from higher computational and memory resources while cloud connectivity may alleviate computational costs even further. Contrasting between traditional controllers and Android devices has become necessary and so far there have been little efforts in this direction. To create a realistic testbed, we use collected in-vehicle CAN bus traffic from an SUV as well as more demanding logs from Advanced Driver-assistance Systems (ADAS) implemented on CAN-FD which we augment with adversarial activity.

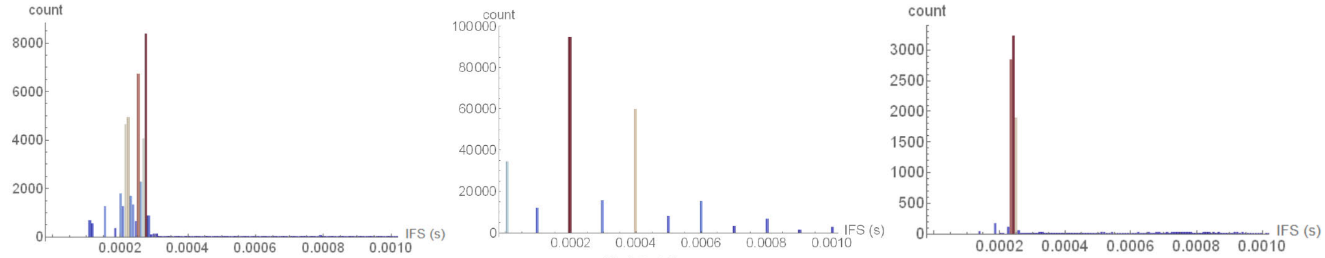
INDEX TERMS CAN bus, electronic control unit, intrusion detection systems, machine learning.

I. INTRODUCTION

Modern cars are equipped with a high number of Electronic Control Units (ECUs) that are used to accomplish various functions, e.g., breaking and stability control, advanced driver assistance, comfort features, etc. Depending on the specific market segment, i.e., economy or luxury, vehicles may be equipped with more or less features and consequently the number of ECUs may range from a dozen or less up to more than a hundred. As the automotive industry is heading toward new trends, such as electrification, autonomous driving or vehicle-to-vehicle communication, we can only expect the number of ECUs and their interconnectivity to increase. The same is implied by recent regulations which are pushing the vehicle industry to evolve in terms of electronics by developing new technologies that will make the driving experience safer and decrease the environmental pollution or energy consumption. But as a side-effect to the increased complexity of in-vehicle electronics and interconnectivity, the number of attack surfaces will increase as well.

More than three decades after its introduction by BOSCH, the Controller Area Network (CAN) is still the most commonly used communication protocol inside vehicles which makes it one of the most important assets that requires protection against malicious attacks. But the security of CAN buses is lacking since no mechanisms were put in place at design time and CAN offers no protection against malicious adversaries. The potential of attacking in-vehicle networks

The associate editor coordinating the review of this manuscript and approving it for publication was Parul Garg.



(i) compact SUV (Dacia Duster)
(ii) ADAS system
(iii) mid-size sedan (Hyundai Sonata [1])

FIGURE 1. **Inter-frame space as recorded in three in-vehicle traces: (i) the compact SUV Dacia Duster, (ii) an ADAS system and (iii) the Hyundai Sonata from [1].**

was demonstrated by many works, e.g., in [2], [3], and [4], which proved that vehicles are extremely vulnerable to cyber attacks. Because of this, manufacturers need to carefully develop and implement proper security mechanisms on forthcoming cars, as cyber attacks can easily lead to catastrophic situations. In addition to the preventive measures, e.g., cryptographic authentication which was commonly proposed in the literature [5], vehicles should be capable to monitor their subsystems and detect potential attacks. Within this scope, Intrusion Detection Systems (IDS) offer an additional protection layer that strengthens the vehicles security architectures.

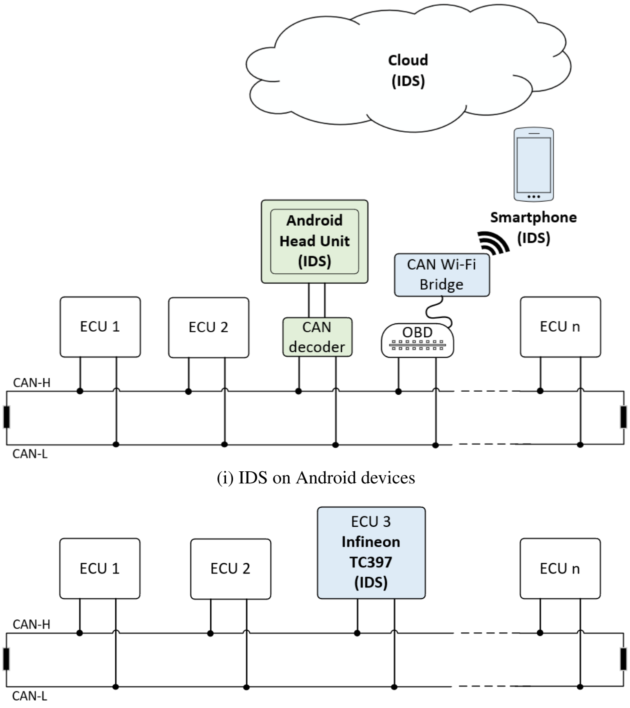
Overview of contribution. While there are many recent works focusing on the design of in-vehicle intrusion detection systems (briefly surveyed by us in the following section), most of these works evaluate the performance of these systems on regular computers. While this is fine for assessing the detection rates, it is not really effective in assessing their behavior on real-world in-vehicle ECUs. This is especially problematic as in-vehicle controllers have to cope with real-time delays. To get a more accurate image, in Figure 1, we depict the delays between consecutive frames, i.e., the inter-frame space (IFS), as recorded in the three in-vehicle traces that we use in our experiments: the compact SUV Dacia Duster (i), an Advanced Driver-assistance Systems (ADAS) from a high-end sedan (ii) from which we collected data and a Hyundai Sonata from [1] (iii) which we keep as a reference in our experiments. Notably, in all three cases the IFS is generally around 200 μs. But in the worst case, the IFS can be as low as 3 recessive bits, i.e., 6 μs on a 500 kbps CAN bus. The IDS has to cope with such small delays and be fast enough in order to be effective for real world needs. Timing is not the only constraint of the problem since the IDS must also fit in the controller memory and it may also need to be updated to learn new attacks in a similar manner to anti-virus software, etc.

For this purpose, in our work, we test the performance of automotive-grade controllers and compare them with Android-based devices such as car head units, in the context of detecting intrusions with machine-learning algorithms. Specifically, our work accounts for the following obvious setups that can be deployed inside a vehicle:

1) *IDS deployed on the Android capable devices.* This setup is outlined in Figure 2 (i). There are two potential variations. First, the IDS can be deployed on an Android head unit which is already a common component in modern vehicles. Besides exhibiting increased computational resources, these units are also equipped with 5G communication which can be used for remote diagnosis (possibly via cloud-based services, which can be used to enhance even further the intrusion detection mechanism inside vehicles by more demanding algorithms and large data pools). Second, the IDS can be deployed on the user device, e.g., a smartphone, that collects CAN bus data by using WiFi connectivity to the OBD port as also outlined in Figure 2 (i). This would allow similar capabilities to the case of Android head units. However, there are additional advantages since users may easily change their smartphone thus benefiting from increased computational and communication capabilities over the years (changing the Android head unit is less convenient). Also, this may open room for third-party software that may be published in Android application stores and may be aquired by users similar to existing anti-virus software. An immediate disadvantage however is that Android head units or smartphones may become more easily corrupted than in-vehicle controllers. For example, the authors of [6] performed some attack experiments on real vehicles by repackaging Android commercial apps. Another demonstration of possible attacks on Android devices is made in [7], in which an Android infotainment unit is hacked and enables attackers to inject messages on the CAN bus. Further, applications vulnerabilities are discussed in [8] and [9]. Another possible disadvantage in implementing IDS on Android devices is that Android smartphones are not directly connected to the CAN bus and wireless communication may induce additional delays (fortunately, these delays may not be so significant but this depends on the interface used for data collection, as we show later in the experiments). Using smartphones may also turn into an advantage from a security perspective, since the smartphone is not directly linked to the CAN bus and wireless connectivity to the bus may be implemented in a read-only fashion. Thus, a compromised phone will not be able to cause attacks on the bus.

TABLE 1. Brief comparison between Android devices and in-vehicle ECUs (technical characteristics refer to devices in our experiments).

	Pros	Cons
Android devices	- high memory resources: 2-16 GB RAM - high clock speeds 1-2 GHz - more cores (4-8) - internet/cloud connectivity	- easier to compromise by malicious apps - real-time behavior harder to predict: less accurate CAN timestamps - costly calls through JNI - possible issues with serial/CAN communication
In-vehicle ECUs	- harder to compromise - real-time behavior - locally available on each ECU	- lower clock speeds: 200-400 MHz - less cores (1-3) - lower memory resources: 1-16 MB RAM



(i) IDS on Android devices

(ii) IDS on in-vehicle controller

FIGURE 2. The two addressed scenarios for intrusion detection.

2) *IDS deployed on in-vehicle controllers*. This is the basic setup depicted in Figure 2 (ii) in which the IDS is deployed in the usual way on one (or several) ECUs inside the vehicle. The main drawback of this approach is that in-vehicle controllers may not have extensive computational resources, nor the communication capabilities or outside connectivity, e.g., to garner cloud-based support. On the positive side, in-vehicle controllers should be harder to compromise and will exhibit a much more controlled real-time behavior. Ideally, the IDS should be deployed on each in-vehicle controller but this is rather debatable due to obvious computational and memory limitations as we discuss in the experimental section. Clearly, in-vehicle networks are heterogeneous and we cannot expect all devices to cope with such demands.

In the light of these scenarios, our work tries to bring a clearer image on the advantages and disadvantages for each of the approaches, i.e., from the more rigid, less corruptible, in-vehicle ECUs to the more flexible, perhaps less secure, Android platforms that may benefit from remote connectivity and increased computational power. A summary of the brief comparison between in-vehicle ECUs and Android units is given in Table 1. As a collateral contribution, though not necessary the main focus of our work, we also evaluate the efficiency of several machine-learning algorithms in detecting intrusions. Table 2 summarizes the binary classifiers that we use in our current work, the achieved performances will be presented in the next sections. Our contributions are fourfold:

TABLE 2. Analyzed binary classifiers.

Abbreviation	Python scikit-learn Function
LR	LogisticRegression
LDA	LinearDiscriminantAnalysis
kNN	KNeighborsClassifier
NB	GaussianNB
SVM	LinearSVC
MLP	MLPClassifier
CART	DecisionTreeClassifier
AB	AdaBoostClassifier
GB	GradientBoostingClassifier
BC	BaggingClassifier
ET	ExtraTreesClassifier
RFC	RandomForestClassifier

1) we design a two-stage IDS in which message arrival time is used in the first stage to detect replay and DoS attacks, then machine-learning algorithms are employed in the second stage to detect frame manipulations caused by fuzzing attacks,
2) we provide specific architectures for two possible deployments, showing the integration of the IDS both on Android devices (with the use of JNI) as well as on embedded development boards (in an AUTOSAR compliant architecture),
3) we collect CAN bus data from real-world vehicles, including CAN-FD data from an ADAS system, augment it with adversarial actions and evaluate twelve classifiers, out of which four are deployed in our experimental setup,
4) we provide computational results regarding the offline and online IDS performance on Android devices, cloud VMs and three representative automotive-grade microcontrollers, as well as memory requirements on the latter due to the stringent constraints on such platforms.

The rest of the paper is organized as follows. In Section II we provide some background on CAN buses and discuss related work. Section III presents the utilized in-vehicle

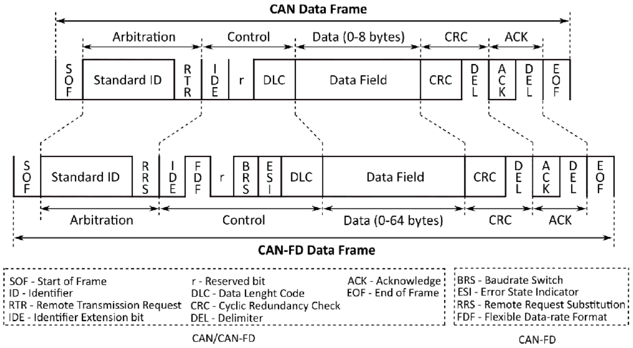


FIGURE 3. CAN and CAN-FD data frame format.

traces, the devices that we used in our experiments and the adversary model. In Section IV we present our experimental testbed. Section V places the binary classifiers in the previously outlined setups and evaluates their performances. Finally, Section VI holds the conclusion of our work and section VI contains the list of acronyms.

II. BACKGROUND AND RELATED WORK

In this section we discuss some background on CAN buses and then we proceed to a brief survey on existing related work for intrusion detection on CAN.

A. BRIEF INTRO ON CAN BUSES

The CAN protocol specification was standardized by International Organization for Standardization (ISO), which released the ISO 11898 standard. Data link layer and physical signalling are part of ISO 11898-1 document [10] while the ISO 11898-2 document [11] is dedicated for high-speed medium access unit. Physically, the CAN bus is designed as a two-wire (CAN-High, CAN-Low) bus connected by two 120 Ohm resistors at the end. The CAN-High and CAN-Low lines carry two complementary signals thus employing differential signaling. The structure of the central communication element for CAN and its extension proposed by BOSCH, i.e., the CAN and CAN-FD (CAN with Flexible Data-Rate) frame is depicted in Figure 3.

In what follows, we detail the most important parts of the frame structure and highlight the main differences between CAN and CAN-FD. In both cases, a dominant SOF bit and a recessive EOF bit marks the beginning and the end of the frame. The identifier of the frame (11 bits for standard format or 29 bits for extended format) along the RTR bit for CAN or RRS for CAN-FD establishes the arbitration field, which assures the collision avoidance mechanism (lower IDs values have a higher priority). We pay attention on the standard format since our CAN collected traffic does not contain any frame in the extended format.

CAN provides bit rates of up to 1 Mbit/s and encloses up to 8 bytes in the data field while the CAN-FD enables faster communication speeds, it usually employs from 2 to 5 Mbit/s, but there are transceivers that support up to 8 Mbit/s, while the data field carries of up to 64 bytes. Another relevant part from the control field is the DLC since it reveals the number of bytes that are carried by the CAN or CAN-FD frame. A 15-bit CRC is employed for verifying the correctness of the frame content. Finally, the correct reception of the frame is enabled by the receiver which overwrites a dominant value in the ACK slot. The most recent step in CAN evolution is CAN-XL which enables payloads up to 2048 bytes and communication speeds of up to 10 Mbit/s remaining compatible with CAN-FD for mixed networks.

B. RELATED WORK

In the recent years, an extremely large number of attacks were reported, e.g. [12] and [13], indicating that the current security mechanisms deployed by vehicle manufacturers are often not appropriate.

Surveys on in-vehicle network attacks and countermeasures can be found in several works, e.g., [14], [15], and [16]. Many solutions were considered, ranging from the use of cryptographic security up to physical layer protection [5] and the industry was not slow in responding with security standards that are part of the AUTomotive Open System ARchitecture (AUTOSAR). AUTOSAR defined the Secure Onboard Communication concept [17] which makes use of Message Authentication Codes (MAC) and freshness values to ensure the integrity and authenticity of the CAN messages. A complementary layer for cryptographic security is the use of an IDS which monitors the CAN network for malicious traffic. Recently, IDS design is among the most commonly debated topic and generates a considerable interest for the research community. In this respect, several relevant works were proposed, which we now discuss. However, most of these studies focus only on the detection accuracy and do not take into account the computational constraints which are crucial in the context of automotive embedded platforms - these constraints are the main focus in our work. In what follows we survey more than twenty-five papers related to the development of in-vehicle IDS, but only a small amount of them, namely [18], [19], [20], [21] and [22] are using embedded development boards. Also, a comparison between in-vehicle controllers and Android units that are now common in cars is missing from related works.

In [18] an IDS based on remote frames is presented. The authors measure the time-interval between request frames (also known as remote frames) and response frames (also known as data frames) and show how adversarial frames cause offset variations that do not occur in a free attack scenario. The use of Bloom filters was explored in [19] in order to detect malicious activity on the CAN bus. The proposed detection technique is based on a training stage that examines the message periodicity in order to detect replay attacks and the content for data field in order to detect injections with random data. The authors show that the real-time classification is time-memory efficient and obtain good detection results. A graph-based IDS that models the CAN traffic is considered in [23]. Other lines of works employ entropy characteristics [24], [25] in order to distinguish between normal or abnormal CAN bus activity. Other approaches include the use of Markov Model [26], decision trees [27] or finite-state automatons [28]. A number of studies found that hardware measurements can be used for intrusion detection. This accounts for the use of voltage thresholds [29], clock-skews [20] or signal characteristics [30].

Significant attention is also given to machine learning based approaches. A hierarchical taxonomy on these methodologies can be found in [31]. The authors from [32] provide a comparative view regarding the use of machine learning approaches for CAN IDSs. For example, in [33], the authors evaluate the performance of the K-Nearest Neighbour and Support Vector Machine algorithms against Denial of Service (DoS) and fuzzy attacks. The results obtained by them exhibit a good detection accuracy of over 90%. A potential weakness of the work in [33] is not considering CAN messages frequency in the training phase (note that CAN bus traffic is always periodic) which leads to the inability of detecting replay attacks. In [34], a deep neural network is employed experiments are performed on CAN traffic generated with a software tool, i.e. OCTANE [35] but not on real-world datasets. A recurrent neural network is employed in [36] where the authors prove the efficiency of the proposed method in detecting malicious frames on the CAN bus. The idea to convert the CAN frames into images in order to build a neural network based IDS was explored by [37]. A specification-based IDS using supervised learning and CAN timing is presented in [38]. The authors of [39] proposed a self-supervised method for intrusion detection which relies on the use of noised pseudo normal data. The detection system uses two deep-learning models, one is used to generate pseudo normal traffic data and the other one is used to detect anomalies. To detect variant attacks, the authors of [40] proposed an intrusion detection system based on the domain adversarial training of neural networks. An intrusion prevention system that detects and discards attack frames on CAN is presented in [21]. The proposed mechanism was implemented and validated on a Raspberry Pi, using the one-class support vector machine and the isolation forest algorithms for intrusion detection. The authors of [22] present a method to detect DoS attacks using the similarity of sliding windows. This method improves prior approaches that detect DoS attacks based on the entropy in a sliding window. For a broader image, recent surveys on intrusion detection mechanisms for vehicular networks can be found in [41] and [42].

III. EXPERIMENTAL TRACES, DEVICES AND ADVERSARY MODEL

In this section we describe the in-vehicle traces and experimental devices that we use in our evaluation. Also, we discuss the adversarial behavior that our intrusion detection system accounts for.

A. COLLECTED IN-VEHICLE TRACES

In our analysis we used two real-world datasets collected by us and a data-set from [1] which we use as a reference.

FIGURE 4. Dacia Duster SUV from our experiments.

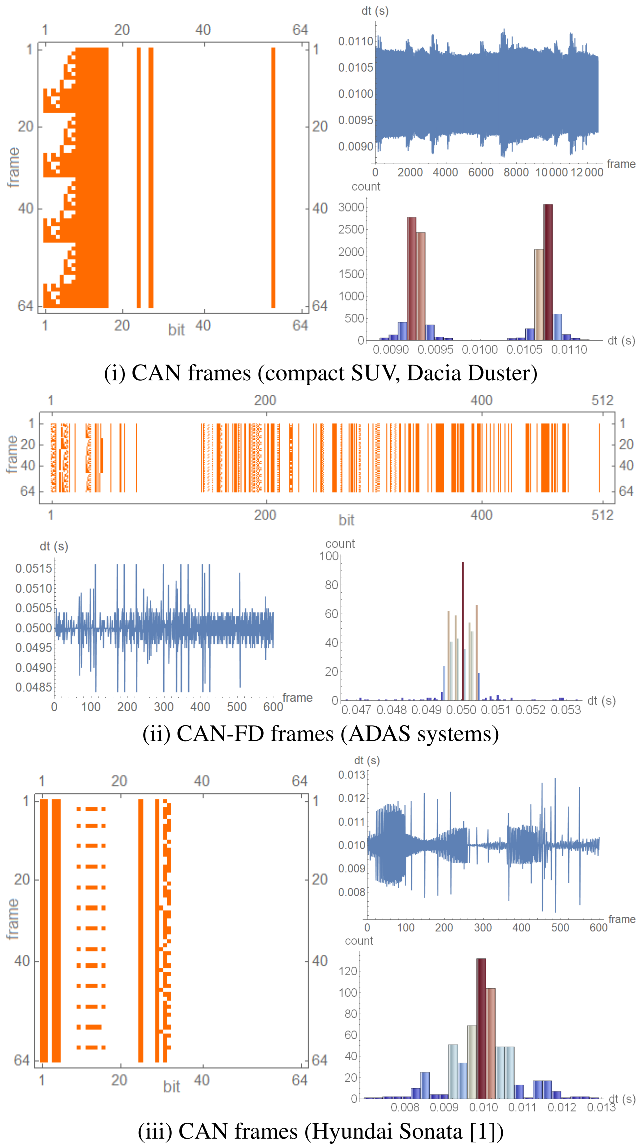
The collection of the CAN dataset from the cars was performed using a Vector VN1630 USB-to-CAN interface. We have implemented a Windows application using Vector XL Driver Library to interface with the VN1630 hardware. For the first CAN trace, we connected the VN1630 to the Dacia Duster in-vehicle OBD port and extracted the CAN bus traffic. Our second dataset was extracted directly from a private CAN bus on which automotive radar ECUs were connected, i.e., ADAS systems (Advanced Driver-Assistance Systems). Using data from such a system is relevant since future autonomous vehicles will directly depend on it, not to mention the increased help these system have to offer to regular drivers.

The CAN logging procedure is graphically depicted in Figure 9 (i). Several details on these datasets are summarized as follows:

1) the first data set comes from a *Dacia Duster* (Figure 4) which is a compact sport utility vehicle (SUV) which we see as representative for mid-range cars. The collected data is more limited in terms of the number of IDs, only 12 IDs are visible on the OBD port, but it is almost identical to the rest in terms of delays and entropy.
2) the second dataset comes from a high throughput CAN-FD network that accommodates *ADAS systems*, e.g., vehicle radars used to detect vehicles and pedestrians. This type of traffic is representative for mid to high-end cars that posses modern equipment needed for complex tasks such as autonomous driving. This dataset is more complex containing more than 80 IDs and frames of up to 512 bits. The communication layer is the newer CAN-FD.

We also use the dataset from [1] which was recorded in a *Hyundai Sonata* and we keep it as a reference to compare our results with existing works. The trace contains 27 IDs making it more similar to our first dataset and less complex than the second.

A few words on the traces based on the depictions from Figure 5 are necessary. This figure depicts some statistics for one ID in each trace. We notice that in all traces the content of the datafield shows clear patterns which would make it



(i) CAN frames (compact SUV, Dacia Duster)

(ii) CAN-FD frames (ADAS systems)

(iii) CAN frames (Hyundai Sonata [1])

FIGURE 5. Example for the values of the data-field (left), cycle time (right-up) and histogram distribution of the cycle time (right-down) for an ID collected in the compact SUV Dacia Duster (i), ADAS systems (ii) and the Hyundai Sonata from [1] (iii).

easy to detect certain attacks, e.g., randomized injection. For the first trace, there is also a limited set of identifiers which show more randomized patterns. The shortest cycle time for the IDs is at 10 ms in all traces. Interestingly, in the first trace the ID at 10 ms has a bimodal distribution of the arrival time. The variations however are generally of 1-2 ms at most in all traces. The CAN-FD trace, while carrying larger data payloads, does not exhibit more variability than the regular CAN traces. This suggests that the same mechanism for detecting intrusions will hold for all traces.

B. DEVICES FROM OUR SETUP

The first category of devices that we used in our setup comprises the Android-based devices. We used a PNI A8020 head unit whose production started in 2017 and a more recent one, Erisin ES8791V, released in 2019. Due to their high usage and capabilities, smartphones were also considered in our setup. Consequently, we chose to work with a Samsung A6, a Samsung S8 and a Samsung Note10+. In addition to smartphones, we also included a tablet in our work, namely the Samsung Galaxy Tab S7.

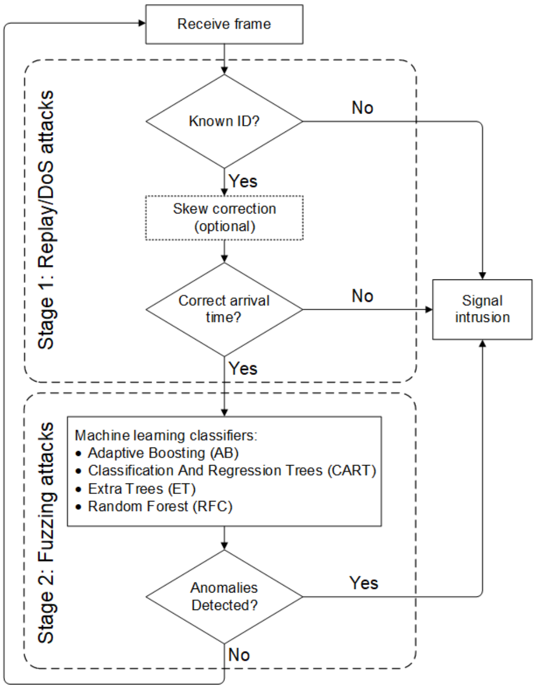


FIGURE 6. The two stage intrusion detection algorithm in our work.

The second category of devices that we worked with consists of automotive-grade microcontrollers. We used from Infineon two devices from the Aurix 32-bit microcontrollers family which are meant especially for automotive and industrial applications. The first microcontroller is a TC224, belonging to the 1st generation of AURIX, while the other microcontroller is a Tricore TC397, which is part of the 2nd generation of AURIX. From the low-end sector, we chose an S12XEP microcontroller which is part of S12XE family that provides 16-bit arhitecture microcontrollers having Hybrid Electric Vehicle (HEV), Tire Pressure Monitoring Systems (TPMS) or Motorcycle Engine Control Unit (ECU) as target applications in the automotive sector. All devices that we used in our experiments and their specifications are listed in Table 3.

C. ADVERSARY MODEL

The following three types of attacks have been commonly considered against CAN nodes. *Fuzzing attacks* in which an attacker modifies the data-field of the genuine CAN frames and transmits the malicious frames on the bus. The injected data field is filled with random values. *Replay attacks* in which genuine CAN frames are intercepted by an attacker and retransmitted on the bus at a later time. In this scenario,

TABLE 3. In-vehicle devices used in our evaluation.

	Device	Prod. Year	Android	CPU	Memory	Connectivity
Android devices	Head Unit PNI A8020	2017	7.1	Quad-core 1.63 GHz Cortex A7	8 GB, 1 GB RAM	WiFi, Bluetooth, USB
	Head Unit Erisin ES8791V	2019	10.0	Rockchips PX5 1512 MHz Cortex A53	64 GB, 4 GB RAM	WiFi, Bluetooth, USB
	Samsung A6	2018	8.0	Octa-core 1.6 GHz Cortex-A53	32 GB, 3 GB RAM	WiFi, Bluetooth, NFC, USB
	Samsung S8	2017	7.0	Octa-core (4x2.3 GHz Mongoose M2 & 4x1.7 GHz Cortex-A53)	64 GB, 4 GB RAM	WiFi, Bluetooth, NFC, USB
	Samsung Note10+	2017	9.0	Octa-core (2x2.73 GHz Mongoose M4 2x2.4 GHz Cortex-A75 & 4x1.9 GHz Cortex-A55)	256 GB, 12 GB RAM	WiFi, Bluetooth, NFC, USB
	Samsung Galaxy Tab S7	2020	8.0	Octa-core (1x3.09 GHz Kryo 585 & 3x2.42 GHz Kryo 585 & 4x1.8 GHz Kryo 585)	256 GB, 8 GB RAM	WiFi, Bluetooth, USB
ECUs	Infineon Tricore TC224	2015	N/A	Single-core 133 MHz TriCore	1 MB, 96 KB RAM	CAN 2.0, CAN-FD, Flexray, Ethernet, etc.
	Infineon Tricore TC397	2018	N/A	Hexa-core 300 MHz TriCore	16 MB, 2528 KB RAM	CAN 2.0, CAN-FD, Flexray, Ethernet, etc.
	S12XE	2006	N/A	Single-core 50 MHz S12X	1 MB, 64 KB RAM	CAN 2.0, LIN, SPI

as malicious frames and genuine frames are identical, the only visible aspect on the bus is an increased frequency of frames with the corresponding IDs which eventually leads to a different inter-frame delay for the respective ID. And finally, *flooding attacks* in which CAN frames with low valued IDs (that are not part of the dataset) and random data are injected on the CAN bus, causing a Denial of Service (DoS).

From these three types of attack, the most involving for the machine learning algorithms is the fuzzing attack since it requires analysis of the complete frame. As for DoS and replay attacks, these may be detected by a simple inspection of the arrival time and frame rate on the bus. Notably, in most cars the bus is kept at around 50% busload or less and all frames have fixed periodicity. When the frame rate exceeds the expected threshold a DoS or replay attacks can be signaled without the need of more expensive machine learning algorithms. This is suggested in Figure 6 which presents the two-stage intrusion detection mechanism that we envision. The first stage simply checks for known IDs and the correctness of the arrival time, possibly by performing some additional skew corrections to avoid synchronization issues, and only then the second stage enters to detect anomalies based on the machine learning classifiers. Consequently, we use the CAN IDs and timestamps as features in the first stage and the CAN IDs and data fields as features for the binary classifiers employed in the second stage. Figure 7 suggests the feature extraction from a CAN frame and the allocation of the features to the two-stage IDS.

In addition to our own datasets, we also executed our algorithms on datasets from related work [1] to validate them and have a common denominator with other related works, e.g. [36] and [43]. In this way, a more accurate comparison of the results is possible. Although essentially the same types of attacks that we previously mentioned are evaluated, there are small differences in how they are implemented or named. For this, we first clarify how the attacks used in [1] differ. Han et. al. focused on three types of attacks: flooding, fuzzy, and malfunction. The first type of attack consists of injecting messages with ID 0 × 00, which based on the CAN specification, is the ID with the highest priority. The consequence of this attack is a DoS, i.e., the malicious ECU will occupy the resources allocated to the CAN bus, limiting the communication among the other ECUs. The data field of the injected messages is always set to zero. Departing from [1], in our work, the flooding attack consists of frames with IDs whose values are less than the genuine ID with the lowest value from the dataset and the data field is filled with random values. The effect is similar, although the attack is more difficult to detect and more realistic since, with our adversary model, a DoS is not caused by ID 0 × 00 alone. The second type of attack, i.e. fuzzy attack, consists of sending frames with random IDs and data. This type of attack will be much easier to detect than ours since most of the random IDs will not be part of the legitimate trace (for this, machine learning algorithms are not needed since unknown IDs are easy to detect by a look-up-table). Since this attack will be immediately detected by filtering, we do not reproduce it in our dataset as it will be trivial to detect by the first stage of the intrusion detecting mechanism which checks that IDs belong to genuine ECUs. Note that in real-world scenarios, the IDs are indeed known by manufacturers at the time of designing the in-vehicle

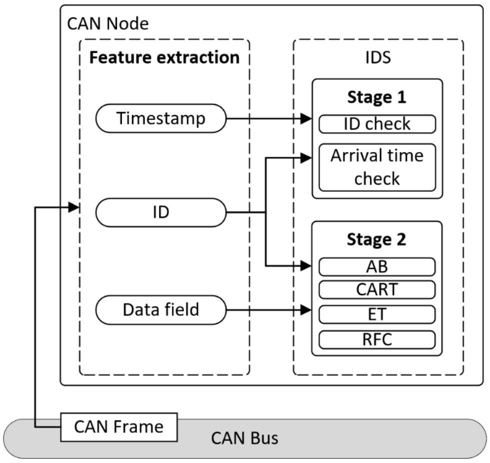


FIGURE 7. Feature extraction for the two-stage IDS.

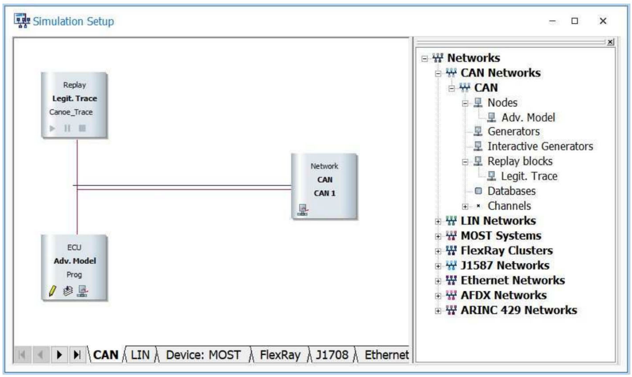


FIGURE 8. **CANoe simulation setup.**

components. The third and last attack, i.e. malfunction attack, is described by the authors of [1] as the attack in which a malicious ECU injects frames with IDs, which are part of the ID list for the respective network, and random data. This type of attack is similar with our fuzzing attack.

IV. SETUP FOR THE EXPERIMENTS

In this section we present the setup that we use for the synthetic analysis (off-line) of the attack traces as well as for the on-line analysis with physical devices plugged to the CAN bus to perform the attack detection in real-time.

A. SETUP FOR OFF-LINE ANALYSIS

To make the attacks realistic, we use the CANoe environment to mount attacks on the frames from the genuine datasets. This working scenario has also been considered by other works, e.g. [44]. For this, we configured inside the environment a CAN node to replay the genuine dataset and another node to inject malicious frames. While the first node is a predefined replay block, for the second node we implemented the logic of the previously mentioned three attacks in CAPL (Communication Access Programming Language). Our CANoe simulation setup is depicted in Figure 8 and the attacks injection procedure in Figure 9 (ii). The resulting traces, which include both genuine and attack frames, were used as inputs for the IDS algorithms in the training and testing phase. The existing datasets from [1] are taken for comparison in the format provided by the authors and was not run by us inside the CANoe environment.

B. SETUP FOR ON-LINE ANALYSIS

For the on-line analysis we aimed to connect the Android units to the CAN bus in order to monitor the CAN bus traffic. We investigated two options to achieve this with the Android head unit and with Android smartphones respectively.

We first used an USB to CAN adapter to connect the Android head unit to the CAN bus. The USB to CAN adapter[1] is commercialized by Seeed Technology and supports both CAN 2.0A and CAN 2.0B with baudrates ranging from 5 kbit/s to 1 Mbit/s. A software application is available for Windows and Linux which may be used to work with the adapter. In addition, a document that describes the UART protocol and the way in which the device can be configured and controlled is available on Github.[2] Therefore, as our target was to use it on the Android head unit, we implemented our own control code in Android Studio. To enable the UART communication on Android, we have used a library also hosted on Github [45].

As a second option, we used a Raspberry Pi module to wirelessly route the CAN messages to the Android head unit. This is the scenario in which the Raspberry Pi is connected to the CAN bus using the OBD port and forwards all the CAN messages from the bus to the head unit or smartphone via WiFi. The Raspberry Pi device does not feature an embedded CAN transceiver, therefore we needed to use an external one. We chose to work with MCP2518FD click board[3] from MikroElektronika which provides a complete CAN and CAN-FD solution. The board is equipped with the MCP2518FD CAN controller, which has SPI interface, and the ATA6563 transceiver. Both integrated circuits are produced by Microchip. The MCP2518FD click board ensures CAN communication speeds up to 5 Mbps and can run in one of the followings operating modes: normal CAN 2.0, normal CAN FD, restricted operation, sleep, listen only, internal and external loop back modes and configuration. The CAN click board is connected to the Raspberry Pi via the Pi 3 Click shield, which is designed by MikroElektronika to support a wide range of click boards.

For both scenarios we used the CANoe environment and a VN1630 hardware to replay the attack traces on the CAN bus. The replay procedure is ilustrated in Figure 9 (iii). The frames were monitored, processed and classified in genuine or attack frames by the Android smartphone in one scenario or by the head unit in the other scenario. The results are discussed in the next section.

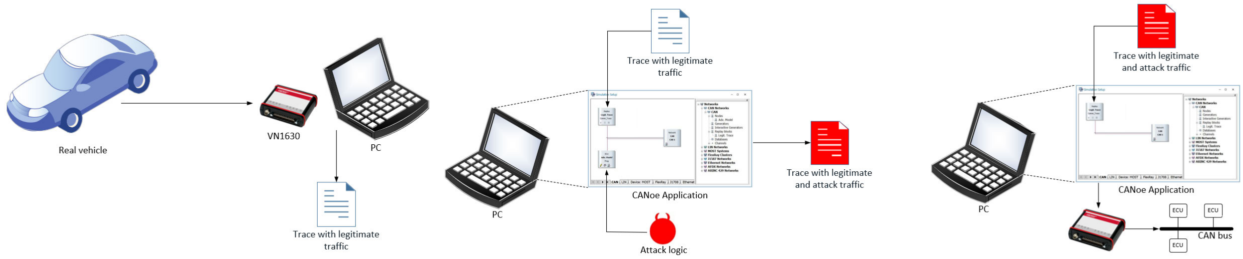
Our experimental setup with all components that we used to deploy the two scenarios is presented in Figure 10. We used a Mastech power supply for the PNI head unit, a laptop to run CANoe Application, a VN1630 hardware to connect the laptop to the CAN bus, a CAN decoder to enable the head unit to communicate on the CAN bus, a Raspberry Pi to route the CAN traffic from the CAN bus to the smarthone and eventually the Samsung A6 and PNI head unit which ran the IDS procedures.

The four classifiers that we later used in our on-line analysis, i.e. AB, CART, ET and RFC, were trained in Python and the generated code was converted to C code using sklearn-porter library [46], so that it becomes easy to adapt and use for Android devices and microcontrollers. On the Android devices, we used the Native Development Kit (NDK) which allows developers to use C and C++ code with Android applications. Therefore, we compiled the C code of

[1] https://www.seeedstudio.com/USB-CAN-Analyzer-p-2888.html

[2] https://github.com/SeeedDocument/USB-CAN-Analyzer

[3] https://www.mikroe.com/mcp2518fd-click


Real vehicle
VN1630
PC
Trace with legitimate traffic
Trace with legitimate traffic
PC
CANoe Application
Trace with legitimate and attack traffic
Attack logic
Trace with legitimate and attack traffic
PC
CANoe Application
ECU
ECU
ECU
CAN bus

(i) traffic collection (ii) attack frames injection (iii) trace replay

FIGURE 9. The three stages employed in our testing: (i) traffic collection, (ii) attack frames injection and (iii) trace replay.

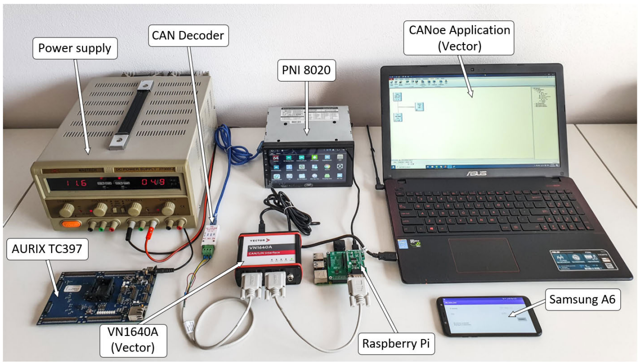


FIGURE 10. Experimental setup for the intrusion detection systems (IDS).

the classifiers into a native library which was then included in our Android Application Package (APK). The connection between the Java code and C code is made using the Java Native Interface (JNI) framework. We built an Android application which receives CAN messages via WiFi (from Raspberry Pi) or USB (from CAN decoder) and using JNI calls accesses the four machine learning (ML) algorithms and classfies the received CAN messages into genuine or attack frames. Figure 11 depicts the classic Android architecture extended with the modules that we developed (highlighted with yellow).

With small adaptations, the C code was usable for all microcontrollers, with few exceptions where the compiled code of some algorithms did not fit into available memory of devices. From our perspective, in classic AUTOSAR ECUs, the machine learning algorithms should be developed as AUTOSAR software components, being part of the Application Layer. The AUTOSAR community already released some specifications regarding vehicle onboard IDS [47], [48]. According to the AUTOSAR requirements [48], an onboard IDS consists of Security Sensors, Security Event Memory (Sem), Intrusion Detection System Manager (IdsM) and Intrusion Detection System Reporter (IdsR). Briefly, security sensors are modules used to detect security events which are then reported to IdsM. The IdsM is a module which manages the received security events by passing them through a filter chain. If the events pass all filters, they will be classified as Qualified Security Events (QEVs). These events can be locally stored in the Security Event Memory or transmitted to the IdsR which collects the QEVs from multiple ECUs and can provide the data to Security Operation Centers for further processing. Currently there is no specification for IdsR provided by AUTOSAR. In our case, we consider that our intrusion detection mechanism should be categorized as an advance security sensor and its deployment should be done on the application layer of the AUTOSAR architecture. This is suggested in Figure 12. The ML module would receive CAN frames from the communication (COM) stack and would report security events to IdsM if intrusions are detected.

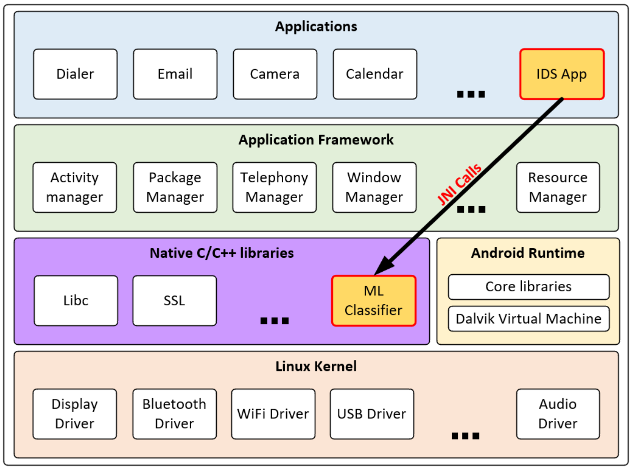


FIGURE 11. Android architecture including IDS modules.

V. EXPERIMENTAL RESULTS

In this section we discuss experimental results both from the off-line and on-line analysis. We also focus on computational and memory requirements and particularly highlight the importance of delays.

A. OFF-LINE EVALUATION

In order to compare the performance of the binary classifier candidates and decide which of them is suitable to be embedded in a vehicular CAN bus IDS, we used regular metrics for machine-learning algorithms.

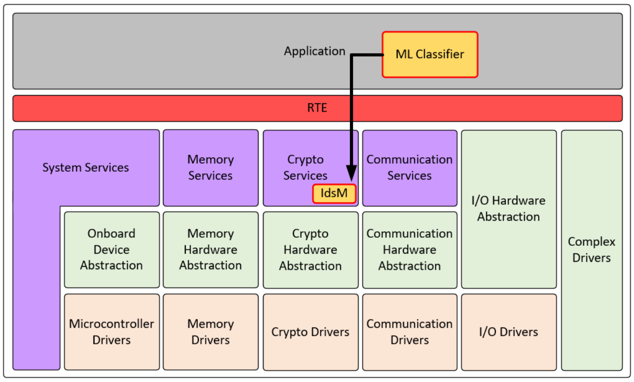


FIGURE 12. **AUTOSAR architecture including CAN IDS modules.**

Each CAN frame that is classified in genuine or intrusion frame by the machine learning algorithms is categorized into one of the following four groups based on the correctness of the classification:

- TP – "true positive", when an intrusion frame is correctly classified as intrusion;
- FP – "false positive", when a genuine frame is incorrectly classified as intrusion;
- TN – "true negative", when a genuine frames is correctly classified as genuine;
- FN – "false negative", when an intrusion frame is incorrectly classified as genuine;

Some of the most common performance metrics used in machine learning classification are the accuracy, precision, recall and specificity. The first one, also called positive predictive value, is the fraction of intrusion frames correctly classified as intrusions among all the frames reported as intrusions:

$$precision = \frac{TP}{TP + FP}$$

The recall is defined as the overall number of true intrusion frames divided by the overall number of frames classified as intrusions:

$$recall = \frac{TP}{TP + FN}$$

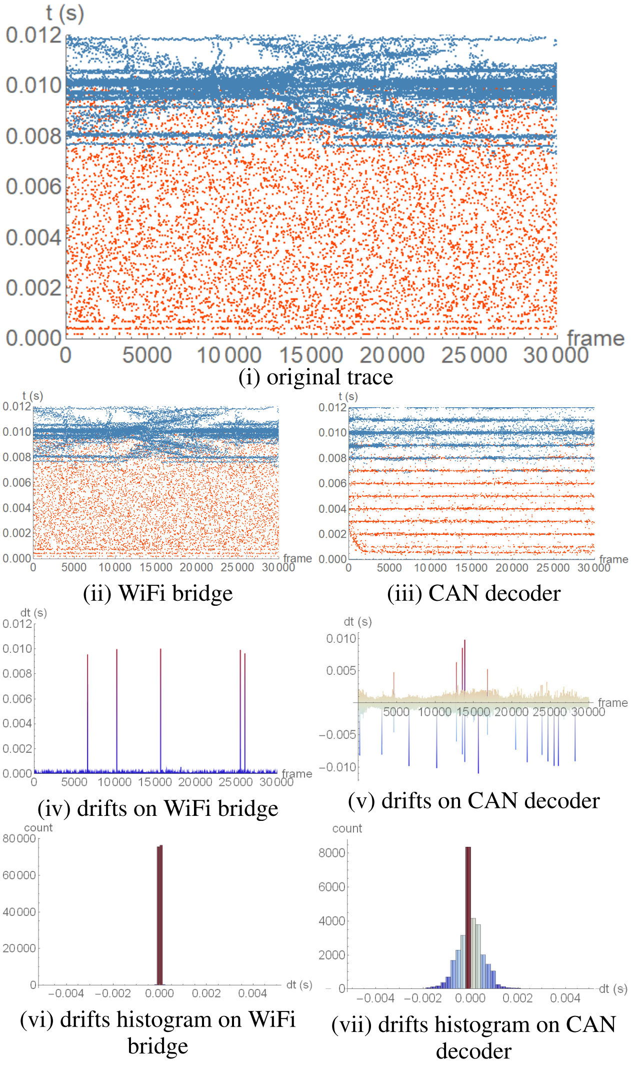
Specificity, also called true negative rate, indicates the proportion of genuine frames that are correctly reported as genuine. Therefore, the definition of specificity is formalized as:

$$specificity = \frac{TN}{TN + FP}$$

The accuracy score is a metric that defines the ratio of correctly classified frames to the total number of frames processed by the machine learning algorithms:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

We first ran our machine learning algorithms on the Survival Analysis datasets from [1]. These datasets were logged from three different vehicles, i.e. Hyundai YF Sonata, KIA Soul, and CHEVROLET Spark. Then, the authors of [1] created for each vehicle three different traces, each of them containing one of the three attacks that they defined in their work, i.e. flooding, fuzzy and malfunction attack. In our offline analysis, we evaluated the datasets which contained the fuzzing and malfunction attacks on the Hyundai Sonata CAN traffic. Both datasets contain approximately 60 seconds of CAN traffic. We trained the algorithms on the CAN frames from the first half of the datasets (≈30 seconds) while the second half of the datasets was used for the evaluation phase.



(i) original trace

(ii) WiFi bridge

(iii) CAN decoder

(iv) drifts on WiFi bridge

(v) drifts on CAN decoder

(vi) drifts histogram on WiFi bridge

(vii) drifts histogram on CAN decoder

FIGURE 13. **Frame cycle time as recorded on: (i) original attack trace, (ii) WiFi bridge, (iii) CAN decoder, (iv) drifts on WiFi bridge, (v) drifts on CAN decoder, (vi) drifts histogram on WiFi bridge and (vii) drifts histogram on CAN decoder.**

The results are presented in Table 4 and Table 5. The performance of the algorithms was almost perfect in detecting fuzzing attacks. The recall was the only metric whose value was 0.99 for half of the classifiers, while the other metrics values were 1.00 for all classifiers. In case of malfunction attacks, the results decreased a bit, especially in precision.

TABLE 4. Survival analysis dataset (HYUNDAI YF Sonata) - Fuzzing attack.

Algorithm	Accuracy	Precision	Recall	Specificity
LR	1.00	1.00	0.99	1.00
LDA	1.00	1.00	0.99	1.00
KNN	1.00	1.00	0.99	1.00
NB	1.00	1.00	0.99	1.00
SVM	1.00	1.00	0.99	1.00
MLP	1.00	1.00	1.00	1.00
CART	1.00	1.00	1.00	1.00
AB	1.00	1.00	0.99	1.00
GB	1.00	1.00	1.00	1.00
BC	1.00	1.00	1.00	1.00
ET	1.00	1.00	1.00	1.00
RFC	1.00	1.00	1.00	1.00

TABLE 5. Survival analysis dataset (HYUNDAI YF Sonata) - Malfunction attack.

Algorithm	Accuracy	Precision	Recall	Specificity
LR	0.98	0.92	1.00	0.98
LDA	0.97	0.87	1.00	0.97
KNN	0.98	0.92	1.00	0.98
NB	0.98	0.92	1.00	0.98
SVM	0.98	0.89	1.00	0.97
MLP	0.98	0.92	1.00	0.98
CART	0.98	0.92	1.00	0.98
AB	0.98	0.92	1.00	0.98
GB	0.98	0.92	1.00	0.98
BC	0.98	0.92	1.00	0.98
ET	0.98	0.92	1.00	0.98
RFC	0.98	0.92	1.00	0.98

TABLE 6. Duster dataset - Fuzzing attack.

Algorithm	Accuracy	Precision	Recall	Specificity
LR	0.98	0.95	0.94	0.99
LDA	0.95	0.89	0.86	0.97
KNN	0.99	0.96	1.00	0.99
NB	0.96	0.90	0.93	0.97
SVM	0.98	0.96	0.92	0.99
MLP	0.99	0.95	1.00	0.99
CART	0.99	0.96	1.00	0.99
AB	0.99	0.96	1.00	0.99
GB	0.89	0.64	1.00	0.86
BC	0.89	0.64	1.00	0.86
ET	0.99	0.96	1.00	0.99
RFC	0.99	0.96	1.00	0.99

TABLE 7. ADAS systems dataset - Fuzzing attack.

Algorithm	Accuracy	Precision	Recall	Specificity
LR	0.94	0.89	0.90	0.96
LDA	0.96	0.96	0.89	0.99
KNN	0.97	0.97	0.93	0.99
NB	0.77	0.55	0.81	0.76
SVM	0.94	0.89	0.89	0.96
MLP	0.98	0.96	0.97	0.99
CART	0.90	0.73	0.97	0.87
AB	0.98	0.95	0.97	0.98
GB	0.96	0.90	0.97	0.96
BC	0.89	0.72	0.97	0.87
ET	0.97	0.91	0.97	0.97
RFC	0.97	0.92	0.97	0.97

However, the overall performance is still pretty good, i.e. accuracy and specificity between 0.97 and 0.98, precision of 0.92 for the most classifiers, while the recall was perfect for all algorithms. When compared to the results obtained in [36] on the Hyundai Sonata datasets, we achieved similar scores in terms of accuracy and recall (with differences of at most 0.01) for fuzzing attacks. In case of malfunction attacks, we achieved the same recall score, i.e. 1.00, but a lower accuracy (i.e. with 0.02 lower in case of the most algorithms and with 0.03 lower in case of LDA) compared to the results obtained in [36]. Next, we ran our algorithms on the datasets that we collected in our work. We only considered the fuzzing attack for the off-line analysis, since machine learning algorithms are part of stage 2 of our IDS. The other attacks, flooding and replay, are handled in the 1st stage of our IDS and were considered in the on-line evaluation that will be presented later. From the Duster dataset, we used the first 20% of the frames (i.e. the first 60 seconds from the trace) for training the classifiers and the rest of the frames (≈240 seconds) were considered in the evaluation phase. In case of the ADAS Systems dataset, we trained the algorithms on the first half of the traffic (≈150 seconds) while the second half of the traffic was included in the evaluation set. The performance results for Duster dataset are listed in Table 6. The accuracy is more than 0.95 for all classifiers, except GB and BC, which have an accuracy of 0.89. These two algorithms performed poor also in terms of precision, with a score of 0.64, but they also ranked last in terms of specificity. The rest of the ten algorithms ranged between 0.89 and 0.96 in precision and between 0.97 and 0.99 in specificity. The last dataset that we assessed was the ADAS Systems dataset which contains CAN-FD traffic. The results are presented in Table 7. Perhaps not surprising, as this dataset is the most complex from the ones that we evaluated, the classifiers recorded the lowest performance results on this trace. Except for the NB, which did not performed well on this dataset, for the rest of the algorithms the accuracy varied between 0.89 and 0.98, precision between 0.72 and 0.97 and specificity between 0.87 and 0.99.

The results from the off-line analysis prove better than the ones obtained in the on-line analysis and this is due to the fact that in the on-line evaluation variations of the timestamps are possible due to frame overlaps on the bus. This points out that the off-line analysis presented in most papers may provide more optimistic results compared to the real-world evaluation.

B. ON-LINE EVALUATION

One specific problem in the on-line evaluation is that the devices which we used for recording CAN bus traffic, have their own imperfections which influenced the performance of the IDS. We note that the timestamps of the frames may have slight variations according to the device. In particular, the Raspberry Pi that we used over the WiFi bridge performed excellent, offering almost identical timestamps to that from the VN1630. The CAN decoder however did not perform very well, giving poor accuracy for the recorded timestamps.

Figure 13 first shows frame cycle time as recorded on (i) original attack trace, (ii) WiFi bridge, (iii) CAN decoder,

TABLE 8. Duster dataset - Fuzzing and replay (WiFi dridge).

Att. type	Algorithm	Accuracy	TN	FP	FN	TP	Precision	Recall	Specificity
fuzzing	AB	0.89	93757	15520	9	27274	0.64	1.00	0.86
	CART	0.89	93757	15520	27	27256	0.64	1.00	0.86
	ET	0.89	93757	15520	0	27283	0.64	1.00	0.86
	RFC	0.89	93757	15520	1	27282	0.64	1.00	0.86
replay	-	0.78	93757	15520	14939	12344	0.44	0.45	0.86

TABLE 9. ADAS systems dataset - Fuzzing and replay (WiFi bridge).

Att. type	Algorithm	Accuracy	TN	FP	FN	TP	Precision	Recall	Specificity
fuzzing	AB	0.97	152634	4628	1386	37994	0.89	0.96	0.97
	CART	0.88	135256	22006	1066	38314	0.64	0.97	0.86
	ET	0.96	150328	6934	1193	38187	0.85	0.97	0.96
	RFC	0.96	150480	6782	1205	38175	0.85	0.97	0.96
replay	-	0.90	154404	2858	16844	22536	0.89	0.57	0.98

then it depicts (iv) drifts on WiFi bridge, (v) drifts on CAN decoder, (vi) drifts histogram on WiFi bridge and (vii) drifts histogram on CAN decoder. The depiction is for a frame with a cycle time of 10ms, in part (i) of the figure the legitimate frames are depicted in blue and attack frames are in red (this is a fuzzing attack where attack frames with random content arrive at random time interval). It is obvious that the WiFi bridge records almost identical timestamps compared to the original attack trace. There are only several drifts of 10 ms when the classification algorithm confused one legitimate frame with an attack frame and thus the legitimate frame is missing in that time slot. For the CAN decoder the timestamps are no longer accurate, the histogram in part (vii) of the figure shows that drifts of up to 2 ms are common. These drifts of around 20% of the 10 ms frame cycle time may lead to false positives in case of legitimate frames. This suggests that the CAN decoder with the employed Android drivers from our experiments is not a very good tool for implementing an IDS, a reason for which we used a Raspberry Pi as a WiFi bridge between the CAN bus and the Android devices. As shown in Figure 13 (ii) the delays over WiFi bridge are nearly identical to the original trace.

We ran the on-line evaluation using the first IDS scenario with WiFi Bridge and a smartphone, using both the Duster and ADAS systems datasets. For this, we connected the Samsung A6 and the Raspberry Pi over WiFi. To ensure security, a WPA2 (WiFi Protected Access II) connection was used, which encrypts all packets with AES (Advanced Encryption Standard). The delays caused by the WiFi network, i.e., by encrypting the traffic and retransmitting it, were too small to affect the real-time performance. We evaluated four algorithms, i.e., AB, CART, ET and RFC, on two types of attacks (fuzzing and replay). We chose to work with these four classifiers, since it was at hand to generate C code for them using the sklearn-porter library. In the on-line evaluation, we ran both stages of the proposed intrusion detection algorithm, described in section III-C. Since the IDS has two stages, we have to redefine the true and false positives as $TP = TP_1 + TP_2$ and $FP = FP_1 + FP_2$ respectively since a frame will be classified as an intrusion if it is marked so by any of the two IDS stages. The true and false negatives will be the true and false negatives that pass the second stage which means that none of the two stages reported them as intrusions thus $TN = TN_2$ and $FN = FN_2$ respectively.

The results obtained on the Duster dataset are presented in Table 8 and the results on the ADAS systems dataset in Table 9. In case of Duster datasets, the detection performance of the four algorithms was almost identical, i.e. accuracy of 0.89, precision of 0.64, recall of 1 and specificity of 0.86 in detecting fuzzing attacks. The detection of replay and flooding attacks is made only in stage 1, so it's independent of what algorithms are used in stage 2. Flooding attacks are trivial to detect with the proposed approach since the legitimate IDs of the network are known (this is always the case in the automotive industry). For the replay attacks, the IDS performed a score of 0.78 in terms of accuracy, 0.44 and 0.45 in terms of precision and recall, and 0.86 in terms of specificity. The performance results are better in case of the ADAS systems datasets, the accuracy ranges from 0.88 to 0.97, precision from 0.64 to 0.89, recall from 0.96 to 0.97 and specificity from 0.86 to 0.97. The replay attacks were detected with an accuracy of 0.90, a precision of 0.89, a recall of 0.57 and a specificity of 0.98.

C. COMPUTATIONAL RESULTS

In addition to the detection performance evaluation, we also assessed the proposed IDS mechanism in terms of runtime speed and memory requirements on several Android devices and three automotive-grade microcontrollers. It is well known that controllers employed nowadays as automotive ECUs have limited computational power and memory. On the other hand, ECUs communicate in real time inside the in-vehicle network, so the IDS algorithms have to be very efficient in terms of execution speed. Computational time and memory requirements are the main challenges in adopting IDS solutions in the automotive world.

The first stage of our proposed IDS mechanism, which simply evaluates the arrival time and frame rate, is of no concerns in terms of execution speed or memory consumption. Therefore we focus our evaluation on the four selected machine

TABLE 10. Infotainment Units - computational time for IDS algorithms [μs] - multiple JNI calls.

Device	Algorithm	Duster (CAN)	ADAS systems (CAN-FD)
PNI A8020	AB	102.87	107.12
	CART	11.47	17.93
	ET	16.07	35.77
	RFC	14.39	29.78
Erisin ES8791V	AB	78.87	84.01
	CART	6.20	9.22
	ET	8.95	16.04
	RFC	7.90	13.97

TABLE 11. Infotainment Units - computational time for IDS algorithms [μs] - one JNI call.

Device	Algorithm	Duster (CAN)	ADAS systems (CAN-FD)
PNI A8020	AB	85.04	85.89
	CART	0.59	2.12
	ET	3.82	10.67
	RFC	2.21	7.90
Erisin ES8791V	AB	71.67	70.47
	CART	0.47	1.55
	ET	2.22	6.57
	RFC	1.63	4.97

TABLE 12. Android devices - computational time for IDS algorithms [μs] - multiple JNI calls.

Device	Algorithm	Duster (CAN)	ADAS systems (CAN-FD)
Samsung A6	AB	60.25	62.69
	CART	5.76	8.02
	ET	7.97	13.56
	RFC	7.09	11.99
Samsung S8	AB	42.40	44.19
	CART	5.44	5.96
	ET	6.72	12.90
	RFC	7.10	10.90
Samsung Note10+	AB	15.07	16.07
	CART	1.42	2.06
	ET	1.97	3.92
	RFC	1.76	3.25
Samsung Tab S7	AB	7.98	8.58
	CART	0.77	1.27
	ET	1.24	2.38
	RFC	1.02	1.94

TABLE 13. Android devices - computational time for IDS algorithms [μs] - one JNI call.

Device	Algorithm	Duster (CAN)	ADAS systems (CAN-FD)
Samsung A6	AB	53.12	54.57
	CART	0.37	1.27
	ET	1.78	5.09
	RFC	1.31	3.90
Samsung S8	AB	37.91	39.39
	CART	0.36	1.25
	ET	1.84	6.71
	RFC	1.15	4.92
Samsung Note10+	AB	13.91	13.91
	CART	0.09	0.30
	ET	0.35	1.35
	RFC	0.27	0.99
Samsung Tab S7	AB	7.03	6.90
	CART	0.07	0.29
	ET	0.34	0.94
	RFC	0.26	0.8

learning algorithms. We ran the algorithms on six Android devices, i.e. two Android based head units, three smartphones and one tablet. For these devices we evaluated the execution speed, since all our devices are equipped with at least 8 GB of ROM memory, so there are no problems regarding the needed memory to employ machine learning algorithms. The results for the head units are listed in Tables 10 and 11. The results for each classifier contains the average time in microseconds that is needed to classify a CAN frame (Duster trace) and a CAN-FD frame (ADAS Systems trace). In order to compute the average time, we ran each classifier on one thousand messages that include both genuine and attack messages. As explained in section IV-B, within our Android application we had to perform JNI calls in order to access the ML algorithms. Obviously, each JNI call requires an additional execution time. Therefore, each message which is received in the application layer of the Android architecture and has to be classified would require a JNI call. The results from Table 10 reflect this situation. We performed a JNI call for each of the one thousand evaluated messages. On the head units, CART, ET and RFC are executed between 6.20 and 16.07 μs when classifying regular CAN frames and between 9.22 and 35.77 μs when classifying CAN-FD frames. It seems that for AB the generated code for CAN-FD frames is very similar to the regular CAN frames, consequently the execution time varies by a few micro-seconds. The AB classifier requires up to 107.12 μs to be executed on the PNI head unit and up to 84.01 μs to be executed on the Erisin head unit, which proves to be faster. We consider that the communication procedure (via WiFi or USB) may also be implemented on the native level, enabling the CAN messages to be received directly at this level, and finally avoid multiple JNI calls. This approach would decrease the overall execution time per message. With this in mind, we also measured the execution time with only one JNI call. For this, we hardcoded the evaluated messages in a C file which we compiled with the application so that we can ran the algorithms on all messages at the native layer. Consequently, we reduced the numbers of JNI calls to one. These results are presented in Table 11. With only one JNI call, the time decreases significantly for all algorithms on both head units. On Duster dataset, the required execution time of CART, ET and RFC ranges between 0.47 μs and 3.82 μs and between 1.55 μs and 10.67 μs in case of the ADAS Systems dataset. AB is executed in less than 86 μs on the PNI head unit and in less than 72 μs on the Erisin head unit.

We further did the same evaluations on the Android smartphones and tablet. The results are presented in Tables 12 and 13. Table 12 contains the execution times evaluated with multiple JNI calls, while Table 13 lists the execution time results with one JNI call. The smartphones and the tablet prove to be somewhat faster than the head units. According to the results, the fastest algorithm is CART, which required an execution time in the ranges of 0.77 μs (on Samsung Galaxy Tab S7) to 5.76 μs (on Samsung A6) when classifying frames from the Duster dataset. As expected, the time

TABLE 14. Azure virtual machines.

VM Size	OS	Family	CPU	vCPUs	Memory
Standard B2ms	Linux (ubuntu 18.04)	General purpose	Intel Xeon Platinum 8171M 2.10 GHz	2	8 GB RAM
Standard F2s_v2		Compute optimized	Intel Xeon Platinum 8272CL 2.59 GHz	2	4 GB RAM
Standard B2ms	Windows Server	General purpose	Intel Xeon Platinum 8171M 2.10 GHz	2	8 GB RAM
Standard F2s_v2		Compute optimized	Intel Xeon Platinum 8272CL 2.59 GHz	2	4 GB RAM

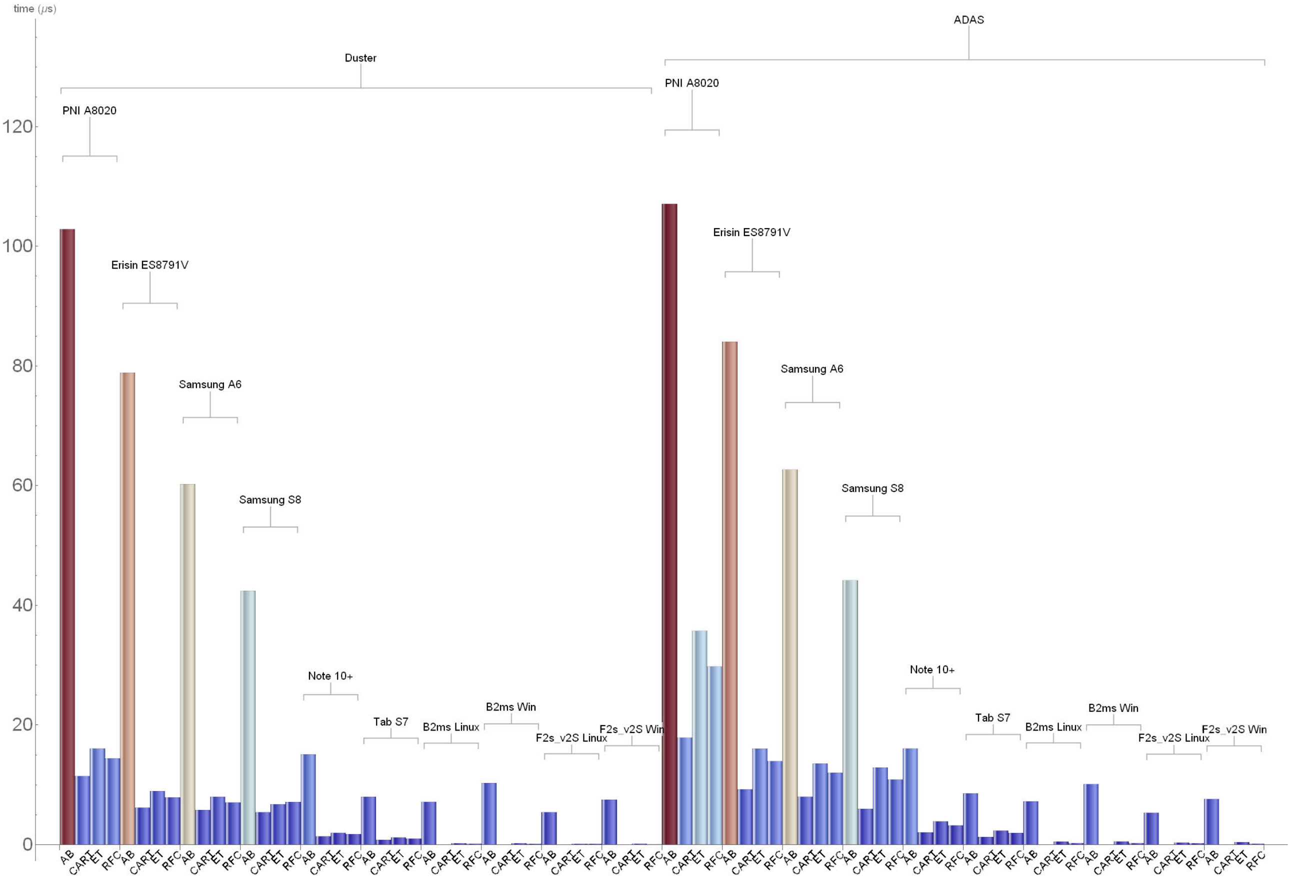


FIGURE 14. Computational results on Android (multiple JNI calls) and cloud VMs for the four ML classifiers.

is higher for the CAN-FD frames (ADAS Systems dataset), ranging from 1.27 μs (on Samsung Galaxy Tab S7) to 8.02 μs (on Samsung A6). The algorithm which requires the highest execution time is AB, which classifies CAN frames in ≈8 μs on the most powerful device and in ≈60 μs on the slowest device. The required execution time for ET and RFC lies between CART and AB, with values between 1.02 μs and 13.56 μs. With only one JNI call, CART, ET and RFC are executed in less than 2 μs by all devices in case of Duster dataset frames, and in less than 6.71 μs in case of the ADAS Systems dataset frames. AB is executed between 6.90 μs and 54.57 μs in case of both datasets.

As described in the introductory part, an important advantage of Android devices, which can be connected on the CAN bus, is that they can be also easily connect to cloud services. This opens road to deploy more complex IDS algorithms on cloud and take advantage of the high computational resources that the cloud servers are capable of. In order to get a clear picture of the computational capabilities of cloud solutions, we evaluated the four classifiers (i.e. AB, CART, ET and RFC) on cloud virtual machines (VMs). Therefore, we deployed four VMs using Microsoft Azure service.

TABLE 15. Cloud VMs - computational time for IDS algorithms [μs].

VM	OS	Algorithm	Duster (CAN)	ADAS systems (CAN-FD)
Standard B2ms	Linux	AB	7.16	7.20
		CART	0.02	0.03
		ET	0.20	0.49
		RFC	0.15	0.28
	Windows	AB	10.31	10.14
		CART	0.03	0.03
		ET	0.21	0.56
		RFC	0.12	0.27
Standard F2s_v2S	Linux	AB	5.43	5.37
		CART	0.02	0.02
		ET	0.14	0.37
		RFC	0.11	0.21
	Windows	AB	7.49	7.65
		CART	0.02	0.02
		ET	0.15	0.40
		RFC	0.09	0.19

TABLE 16. Automotive grade controllers - computational time and memory consumption for IDS algorithms.

Microcontroller	Algorithm	Duster (CAN) [μs]	Code flash [kB]	ADAS systems (CAN-FD) [μs]	Code flash [kB]
Tricore TC224	AB	5320	5.8417	5466	5.8144
	CART	0.59	0.7519	1.14	103.7226
	ET	5.60	10.5117	n/a	1573.6210
	RFC	3.20	3.8496	11.6	476.4375
Tricore TC397	AB	112.7475	4.9062	113.1530	4.9902
	CART	0.7075	1.3203	1.0321	136.6621
	ET	5.5111	16.5292	11.4667	2005.2207
	RFC	3.9546	6.5976	8.9753	634.6835
S12	AB	1570000	7.8300	1600000	7.8232
	CART	4300	2.0263	n/a	203.767
	ET	37900	28.07	n/a	code too large
	RFC	3880	6.5068	n/a	511.9375

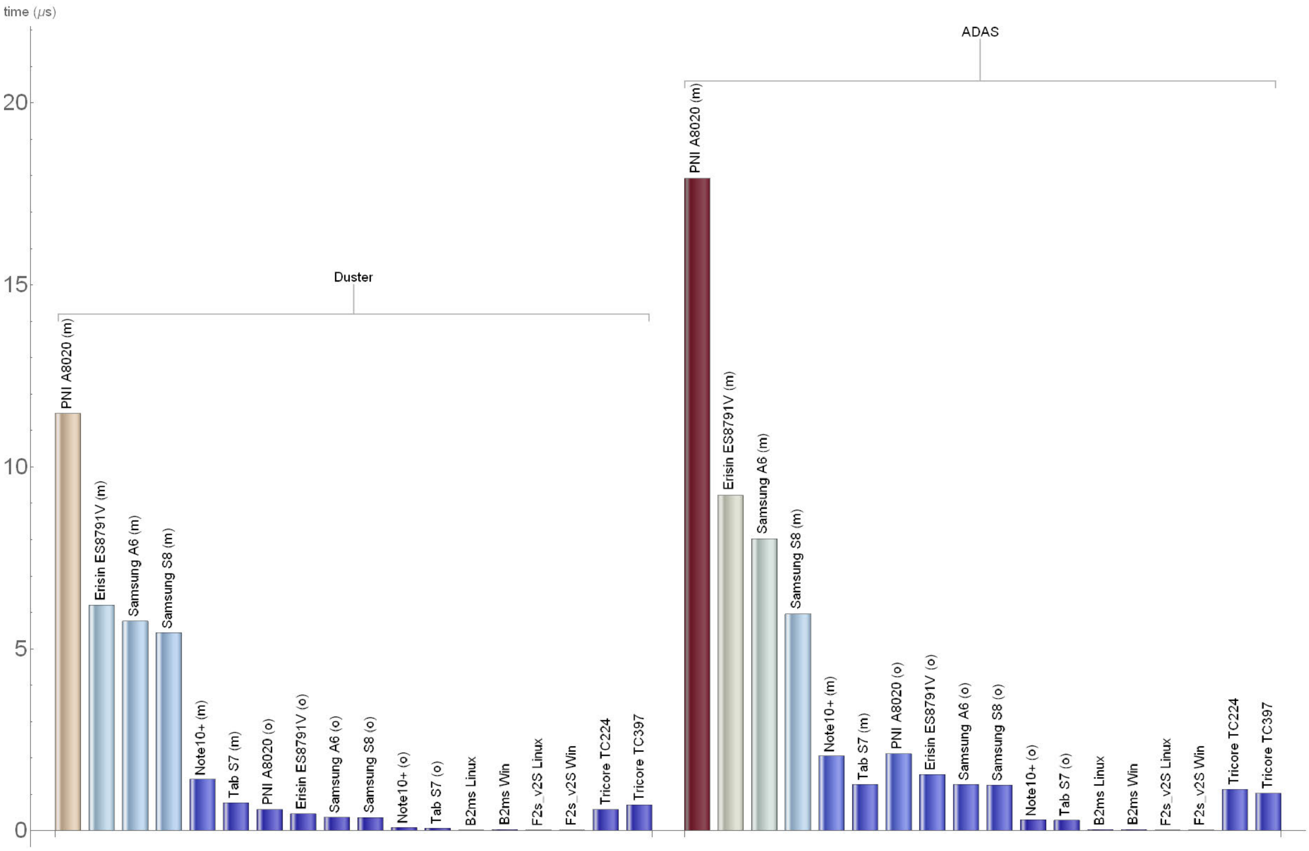


FIGURE 15. Computational results on Android (m-multiple/o-one JNI call(s)) and cloud VMs vs. ECUs for the CART classifier.

We created two VMs running Ubuntu and two Windows based VMs. For each operating system we chose a *general purpose* VM with 8 GB of RAM and a CPU running at 2.10 GHz and a *compute optimized* VM with 4 GB RAM and a CPU running at 2.59 GHz. Each VM features two virtual CPUs (vCPUs). The VMs specifications are listed in Table 14. The runtime measurements are represented in Table 15. In general, the cloud VMs seem to be the most performant devices, from the ones that we evaluated, in terms of execution speed. CART, ET and RFC classifiers are executed in less than 1 μs by all the four VMs while AB is executed between 5.37 μs and 10.31 μs, depending on the VMs configuration. It seems that VMs running Ubuntu are somewhat faster than the Windows based VMs. However, an important aspect that needs to be considered for cloud solutions is the data transmission time, which depending on various factors (e.g. location of the server, internet connection) can range from tens of milliseconds to hundreds of milliseconds or even more. For a better visualization, the computational results on the Android devices and cloud VMs are depicted as bar-charts in Figure 14.

Next, we evaluated the algorithms on the automotive-grade microcontrollers. In our experiments, we compiled the C code with the default compiler options for each microcontroller, which leaves room for optimization in terms of memory or execution speed, depending on the needs. For this class of devices, in addition to execution speed, we also evaluate the required code flash for each algorithm, since memory consumption is one of the most stringent limitations of the automotive microcontrollers. The results are listed in Table 16. Regarding memory consumption, the situation looks good for the algorithms that were trained on the Duster dataset. The necessary free memory ranges up to 2.0263 kB in case of CART, up to 7.83 kB in case of AB, up to 6.5976 kB in case of RFC and up to 28.07 kB in case of ET. These values should be acceptable for deploying IDS algorithms on automotive ECUs. However, the situation gets more complicated with the code that was generated for the ADAS Systems dataset. Except AB, the required available memory increased a lot for the other three classifiers. ET could be loaded only on the Tricore TC397 memory. CART and RFC could be compiled and linked by both the Infineon devices TC224 and TC397. However, even if they fit in the memory, the requirements are not very convenient, at least in case of RFC which requires 476.4375 kB of code flash memory on TC224, i.e., already more than 40% of the entire available memory of this microcontroller. We were not able to include and assess CART or RFC on S12 as the compiler that we used for S12 has a 64 kB code limitation.

From the execution point of view, most of the results for the Infineon microcontrollers are comparable with the Android devices. CART, ET and RFC algorithms trained on Duster dataset are executed between 0.59 μs and 5.60 μs. In order to classify CAN-FD frames, CART and RFC requires between 1.0321 μs and 11.6 μs on the two Infineon devices. The ET algorithm which was trained on the ADAS Systems dataset could be successfully evaluated only on the Infineon TC397. It requires a bit over 2000 kB of code flash memory and it's executed in less than 11.5 μs. Based on our results, the S12 microcontroller requires a few seconds to execute the machine learning algorithms which is way too much for the IDS requirements. This indicates that it's not possible to deploy such an IDS mechanism on microcontrollers with low CPU operating frequencies. Based on the results that we obtained, CART seems to be the most convenient classifier to be deployed in terms of execution speed and required memory (only applicable for microcontrollers). In Figure 15 we depicted the execution speed of CART classifier on the Android devices and on the two Infineon microcontrollers.

VI. CONCLUSION

In this work we made a comparative analysis between two implementation options for deploying an intrusion detection system on the CAN bus: the use of an in-vehicle ECU and the use of Android head units connected via a CAN decoder or of an Android device connected to a WiFi bridge. The Android devices do outperform in-vehicle ECUs, but not by such a high margin when using one of the most powerful in-vehicle controllers available on the market, i.e., an Infineon TC397. However, this happens only if one can avoid expensive API calls over the JNI interface and if the code is run at the native level on the ARM processor of the Android unit. This will depend on the number of JNI calls that are time consuming, i.e., when multiple calls are used, the high-end controllers will outperform low-end Android devices. This implementation detail may significantly reduce the capability of such devices. For example, when performing multiple calls from Java to the C/C++ code of the classifier, the Android head unit proved to be slower than the fastest microcontroller. Also, we notice that the CAN decoder that was linked through the serial interface to the Android Unit is not reliable enough for recording the timestamps which further impedes the detection rates of the IDS. Nonetheless, the same CAN decoder was unable to cope with the frame rate from the bus and there was a consistent frame loss. Finally, the WiFi bridge performed very well giving almost identical results in terms of timestamps compared to industry standard VN1630. This suggests this option as a reliable one for implementing an IDS inside vehicles. The flexibility offered by implementing an IDS on Android devices, which may take advantage of high CPU and memory resources as well as cloud support, may open road for the deployment of more advanced IDS in future cars.

LIST OF ACRONYMS

AB	Adaptive Boosting.
ACK	Acknowledge.
ADAS	Advanced Driver-Assistance Systems.
AES	Advanced Encryption Standard.
API	Application Programming Interface.
APK	Android Application Package.
AUTOSAR	AUTomotive Open System ARchitecture.
BC	Bagging Classifier.
CAN	Controller Area Networks.
CAPL	Communication Access Programming Language.
CART	Classification And Regression Tree.
COM	Communication.
CPU	Central Processing Unit.
CRC	Cyclic Redundancy Check.
DLC	Data Length Code.
DoS	Denial of Service.
ECU	Electronic Control Units.
EOF	End of Frame.
ET	Extra Tree.
FN	False Negative.
FP	False Positive.
GB	Gradient Boosting.
HEV	Hybrid Electric Vehicle.
IDS	Intrusion Detection Systems.
IdsM	Intrusion Detection System Manager.
IdsR	Intrusion Detection System Reporter.
IFS	Interframe Space.
ISO	International Organization for Standardization.
JNI	Java Native Interface.
KNN	K-Nearest Neighbors.

LDA	Linear Discriminant Analysis.
LR	Logistic Regression.
MAC	Message Authentication Codes.
ML	Machine Learning.
MLP	Multi-Layer Perceptron Network.
NB	Gaussian Naive Bayes.
NDK	Native Development Kit.
OBD	On-Board Diagnostic.
QEVs	Qualified Security Events.
RFC	Random Forest.
ROM	Read-only Memory.
RRS	Remote Request Substitution.
RTR	Remote Transmission Request.
Sem	Security event memory.
SOF	Start of Frame.
SPI	Serial Peripheral Interface.
SUV	Sport Utility Vehicle.
SVM	Support Vector Machine.
TN	True Negative.
TP	True Positive.
TPMS	Tire Pressure Monitoring Systems.
UART	Universal Asynchronous Receiver/Transmitter.
VM	Virtual Machine.
WPA2	WiFi Protected Access II.

REFERENCES

[1] M. L. Han, B. I. Kwak, and H. K. Kim, "Anomaly intrusion detection method for vehicular networks based on survival analysis," *Veh. Commun.*, vol. 14, pp. 52–63, Oct. 2018.

[2] S. Checkoway, "Comprehensive experimental analyses of automotive attack surfaces," in *Proc. USENIX Secur. Symp.*, San Francisco, CA, USA, 2011, pp. 77–92.

[3] C. Miller and C. Valasek, "A survey of remote automotive attack surfaces," *Black Hat USA*, p. 90, Aug. 2014. [Online]. Available: https://ioactive.com/pdfs/IOActive_Remote_Attack_Surfaces.pdf

[4] J. Petit and S. E. Shladover, "Potential cyberattacks on automated vehicles," *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 2, pp. 546–556, Apr. 2015.

[5] B. Groza and P. Murvay, "Security solutions for the controller area network: Bringing authentication to in-vehicle networks," *IEEE Veh. Technol. Mag.*, vol. 13, no. 1, pp. 40–47, Mar. 2018.

[6] Y. Lee, S. Woo, J. Lee, Y. Song, H. Moon, and D. H. Lee, "Enhanced Android app-repackaging attack on in-vehicle network," *Wireless Commun. Mobile Comput.*, vol. 2019, pp. 1–13, Feb. 2019.

[7] G. Costantino and I. Matteucci, "CANDY CREAM—Hacking infotainment Android systems to command instrument cluster via can data frame," in *Proc. IEEE Int. Conf. Comput. Sci. Eng. (CSE), IEEE Int. Conf. Embedded Ubiquitous Comput. (EUC)*, Aug. 2019, pp. 476–481.

[8] M. Ruef, "Car hacking—Analysis of the Mercedes connected vehicle API," Scip AG, Zürich, Switzerland, Tech. Rep., Apr. 2018.

[9] H. J. Jo, W. Choi, S. Y. Na, S. Woo, and D. H. Lee, "Vulnerabilities of Android OS-based telematics system," *Wireless Pers. Commun.*, vol. 92, no. 4, pp. 1511–1530, Feb. 2017.

[10] *Road Vehicles—Controller Area Network (CAN)—Part 1: Data Link Layer and Physical Signalling*, Standard ISO11898-1, 2nd edition, Dec. 2015.

[11] *Road Vehicles—Controller Area Network (CAN)—Part 2: High-Speed Medium Access Unit*, Standard ISO11898-2, 2nd edition, Dec. 2016.

[12] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *Proc. IEEE Symp. Secur. Privacy*, Dec. 2010, pp. 447–462.

[13] C. Miller and C. Valasek, "Adventures in automotive networks and control units," *Def Con*, vol. 21, pp. 260–264, Aug. 2013.

[14] J. Liu, S. Zhang, W. Sun, and Y. Shi, "In-vehicle network attacks and countermeasures: Challenges and future directions," *IEEE Netw.*, vol. 31, no. 5, pp. 50–58, May 2017.

[15] N. Khatri, R. Shrestha, and S. Y. Nam, "Security issues with in-vehicle networks, and enhanced countermeasures based on blockchain," *Electronics*, vol. 10, no. 8, p. 893, Apr. 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/8/893

[16] X. Sun, F. R. Yu, and P. Zhang, "A survey on cyber-security of connected and autonomous vehicles (CAVs)," *IEEE Trans. Intell. Transp. Syst.*, vol. 23, no. 7, pp. 6240–6259, Jul. 2022.

[17] *Specification of Secure Onboard Communication*, AUTOSAR, Munich, Germany, 2020.

[18] H. Lee, S. H. Jeong, and H. K. Kim, "OTIDS: A novel intrusion detection system for in-vehicle network by using remote frame," in *Proc. 15th Annu. Conf. Privacy, Secur. Trust (PST)*, Aug. 2017, pp. 57–66.

[19] B. Groza and P.-S. Murvay, "Efficient intrusion detection with bloom filtering in controller area networks," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 4, pp. 1037–1051, Apr. 2019.

[20] K.-T. Cho and K. G. Shin, "Fingerprinting electronic control units for vehicle intrusion detection," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 911–927.

[21] P. Freitas De Araujo-Filho, A. J. Pinheiro, G. Kaddoum, D. R. Campelo, and F. L. Soares, "An efficient intrusion prevention system for CAN: Hindering cyber-attacks with a low-cost platform," *IEEE Access*, vol. 9, pp. 166855–166869, 2021.

[22] S. Ohira, A. K. Desta, I. Arai, H. Inoue, and K. Fujikawa, "Normal and malicious sliding windows similarity analysis method for fast and accurate IDS against DoS attacks on in-vehicle networks," *IEEE Access*, vol. 8, pp. 42422–42435, 2020.

[23] R. Islam, R. U. D. Refat, S. M. Yerram, and H. Malik, "Graph-based intrusion detection system for controller area networks," *IEEE Trans. Intell. Transp. Syst.*, vol. 23, no. 3, pp. 1727–1736, Mar. 2020.

[24] M. Müter and N. Asaj, "Entropy-based anomaly detection for in-vehicle networks," in *Proc. IEEE Intell. Vehicles Symp. (IV)*, Jun. 2011, pp. 1110–1115.

[25] M. Marchetti, D. Stabili, A. Guido, and M. Colajanni, "Evaluation of anomaly detection for in-vehicle networks through information-theoretic algorithms," in *Proc. IEEE 2nd Int. Forum Res. Technol. Soc. Ind. Leveraging Better Tomorrow (RTSI)*, Sep. 2016, pp. 1–6.

[26] S. N. Narayanan, S. Mittal, and A. Joshi, "Obd_securealert: An anomaly detection system for vehicles," in *Proc. IEEE Int. Conf. Smart Comput. (SMARTCOMP)*, Jun. 2016, pp. 1–6.

[27] D. Tian, Y. Li, Y. Wang, X. Duan, C. Wang, W. Wang, R. Hui, and P. Guo, "An intrusion detection system based on machine learning for can-bus," in *Proc. Int. Conf. Ind. Netw. Intell. Syst.* Cham, Switzerland: Springer, 2017, pp. 285–294.

[28] I. Studnia, E. Alata, V. Nicomette, M. Kaâniche, and Y. Laarouchi, "A language-based intrusion detection approach for automotive embedded networks," *Int. J. Embedded Syst.*, vol. 10, no. 1, pp. 1–12, 2018.

[29] W. Choi, K. Joo, H. J. Jo, M. C. Park, and D. H. Lee, "VoltageIDS: Low-level communication characteristics for automotive intrusion detection system," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 8, pp. 2114–2129, Aug. 2018.

[30] P.-S. Murvay and B. Groza, "Source identification using signal characteristics in controller area networks," *IEEE Signal Process. Lett.*, vol. 21, no. 4, pp. 395–399, Apr. 2014.

[31] Y. Luo, Y. Xiao, L. Cheng, G. Peng, and D. D. Yao, "Deep learning-based anomaly detection in cyber-physical systems: Progress and opportunities," 2020, *arXiv:2003.13213*.

[32] T. Moulahi, S. Zidi, A. Alabdulatif, and M. Atiquzzaman, "Comparative performance evaluation of intrusion detection based on machine learning in in-vehicle controller area network bus," *IEEE Access*, vol. 9, pp. 99595–99605, 2021.

[33] A. Alshammari, M. A. Zohdy, D. Debnath, and G. Corser, "Classification approach for intrusion detection in vehicle systems," *Wireless Eng. Technol.*, vol. 9, no. 4, pp. 79–94, 2018.

[34] M.-J. Kang and J.-W. Kang, "Intrusion detection system using deep neural network for in-vehicle network security," *PLoS ONE*, vol. 11, no. 6, Jun. 2016, Art. no. e0155781.

[35] C. E. Everett and D. McCoy, "Octane (open car testbed and network experiments): Bringing cyber-physical security research to researchers and students," in *Proc. 6th Workshop Cyber Secur. Experimentation Test. USENIX*, 2013, pp. 1–8.

[36] M. D. Hossain, H. Inoue, H. Ochiai, D. Fall, and Y. Kadobayashi, "LSTM-based intrusion detection system for in-vehicle can bus communications," *IEEE Access*, vol. 8, pp. 185489–185502, 2020.

[37] M. Han, P. Cheng, and S. Ma, "CVNNs-IDS: Complex-valued neural network based in-vehicle intrusion detection system," in *Proc. Int. Conf. Secur. Privacy Digit. Economy*. Cham, Switzerland: Springer, 2020, pp. 263–277.

[38] H. Olufowobi, C. Young, J. Zambreno, and G. Bloom, "SAIDuCANT: Specification-based automotive intrusion detection using controller area network (CAN) timing," *IEEE Trans. Veh. Technol.*, vol. 69, no. 2, pp. 1484–1494, Feb. 2020.

[39] H. M. Song and H. K. Kim, "Self-supervised anomaly detection for in-vehicle network using noised pseudo normal data," *IEEE Trans. Veh. Technol.*, vol. 70, no. 2, pp. 1098–1108, Feb. 2021.

[40] J. Wei, Y. Chen, Y. Lai, Y. Wang, and Z. Zhang, "Domain adversarial neural network-based intrusion detection system for in-vehicle network variant attacks," *IEEE Commun. Lett.*, early access, Aug. 1, 2022, doi: 10.1109/LCOMM.2022.3195486.

[41] O. Y. Al-Jarrah, C. Maple, M. Dianati, D. Oxtoby, and A. Mouzakitis, "Intrusion detection systems for intra-vehicle networks: A review," *IEEE Access*, vol. 7, pp. 21266–21289, 2019.

[42] T. Limbasiya, K. Z. Teng, S. Chattopadhyay, and J. Zhou, "A systematic survey of attack detection and prevention in connected and autonomous vehicles," *Veh. Commun.*, vol. 37, Oct. 2022, Art. no. 100515. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214209622000626

[43] S. Park and J.-Y. Choi, "Hierarchical anomaly detection model for in-vehicle networks using machine learning algorithms," *Sensors*, vol. 20, no. 14, p. 3934, Jul. 2020.

[44] C. Jichici, B. Groza, and P.-S. Murvay, "Integrating adversary models and intrusion detection systems for in-vehicle networks in canoe," in *Innovative Security Solutions for Information Technology and Communications*, E. Simion and R. Géraud-Stewart, Eds. Cham, Switzerland: Springer, 2020, pp. 241–256.

[45] F. Herranz. *Usbserial*. Accessed: Jun. 28, 2022. [Online]. Available: https://github.com/felHR85/UsbSerial

[46] D. Morawiec. *Sklearn-Porter, Transpile Trained Scikit-Learn Estimators to C, Java, JavaScript and Others*. Accessed: Jun. 28, 2022. [Online]. Available: https://github.com/nok/sklearn-porter

[47] *Specification of Intrusion Detection System Protocol*, AUTOSAR, Munich, Germany, 2020.

[48] *Requirements on Intrusion Detection System*, AUTOSAR, Munich, Germany, 2020.

TUDOR ANDREICA received the B.Sc. and M.Sc. degrees from the Polithenica University of Timisoara, in 2016 and 2018, respectively, where he is currently pursuing the Ph.D. degree. He was a Research Student in the CSEAMAN and PRESENCE projects. Since 2015, he has been working as a Software Engineer at HELLA Romania, focusing on the security of various in-vehicle systems. His research interest includes automotive cybersecurity.

CHRISTIAN-DANIEL CURIAC received the B.Sc. and M.Sc. degrees in electrical and computer engineering from the Technical University of Munich (TUM), in June 2019 and June 2021, respectively. He is currently pursuing the Ph.D. degree with the Politehnica University of Timisoara. His research interests include cybersecurity, signal processing, and machine learning. He was a recipient of a DAAD Scholarship (2016–2021) to support his bachelor's and master's degree studies.

CAMIL JICHICI received the Dipl.Ing. and M.Sc. degrees from the Politehnica University of Timisoara (UPT), in 2016 and 2018, respectively, where he is currently pursuing the Ph.D. degree. He has been working as a Software Integrator in the automotive industry for Continental Corporation, Timişoara, since 2014. Since 2018, he has been working as a Young Researcher in the PRESENCE project. His research interests include security of in-vehicle components and networks.

BOGDAN GROZA (Member, IEEE) received the Dipl.Ing. and Ph.D. degrees from the Politehnica University of Timisoara (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 is currently a Professor at UPT. He has been actively involved inside UPT with the development of laboratories by Continental Automotive and Vector Informatik. Besides regular participation in national and international research projects in information security, he lead the CSEAMAN (2015–2017) and PRESENCE (2018–2020) projects, two national research programs dedicated to in-vehicle security and the interaction between vehicles and smartphones.