Cacao, a CAN-Bus Simulation Platform for Secured Vehicular Communication

Olivier Cros(✉), Alexandre Thiroux, and Gabriel Chênevert

JUNIA, Department of Computer Science and Mathematics, 41 Boulevard Vauban, 59000 Lille, France
{olivier.cros,gabriel.chenevert}@yncrea.fr, athiroux@ssl247.fr

Abstract. In its native version, the Controller Area Network (CAN) bus protocol used in most personal vehicles does not use any encryption nor message authentication mechanism. In order to test solutions dedicated to signing messages and protecting CAN infrastructures from external attacks, we built CAn enCryption simulAtion mOdule (Cacao). It is a CAN bus simulation platform dedicated to simulate a real CAN network. The following work presents this tool and the signature solution we did integrate in it to implement various vulnerabilities protection among the CAN bus.

Keywords: CAN bus · Simulation · Signature · Framework

1 Introduction

1.1 About CAN Bus

Embedded networks are concerned by cybersecurity and protection issues. In industrial domains such as automotive or spacecraft, the CAN bus is a well-known network backbone for all internal communications. It interconnects critical and less critical subsystems, sensors and actuators in order to perform the in-vehicle communication.

Recently, the CAN bus was built in a completely closed approach, relying on the fact that a vehicle was not meant to be connect to the outside. Thus, it was built considering that each node in the CAN bus was reliable in terms of security. The CAN bus, in its standard version, does not integrate any encryption nor signature functions. Some solutions, for spacecraft for example, were to avoid these problems by defining additional specific upper layers. But that leaves the CAN standard version (used in most cars) unprotected.

But, with the emergence of Vehicle to Infrastructure (V2I), Vehicle to Everything (V2X) and similar infrastructures, cars have become more and more connected: to other cars, to tolls, to grid management, etc. These interconnections represent a wide set of potential attack vectors from the outside, targetting the vehicle critical functions through CAN bus misuse or corruption. This can imply erroneous commands, data leaks and even unwanted remote control of the vehicle, all vulnerabilities which should be corrected.

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2021
Published by Springer Nature Switzerland AG 2021. All Rights Reserved
L. Foschini and M. El Kamili (Eds.): ADHOCNETS 2020, LNICST 345, pp. 213–224, 2021.
https://doi.org/10.1007/978-3-030-67369-7_16

1.2 Related Work

The CAN bus protocol was designed in 1983 (and 1991 for the second version [8]) as a common network backbone for embedded networks, particularly in the context of automotive and spacecraft [15]. At this time, the purpose of such a bus was to offer a common network standard for all internal communications inside a vehicle, no matter the type or number of sensors and actuators it has.

Authentication solutions for embedded networks and Cyber Physical Systems (CPS) are not new. Moreover, integrating cryptography in embedded network architectures is not an issue specific to CAN. Various approaches based on elliptic curves were already proposed for networks such as Zigbee [11] but appeared costful in terms of energy consumption and computation time. Concerning the CAN bus, both solutions MaCAN [9] and CaCAN [12] propose to encrypt data transmission inside CAN and provide key rotation solutions. Such solutions rely on CAN additional abstraction layers or extensions of the CAN protocol like CAN+ [18,21], CANAerospace [16] or either CAN-FD [10]. These solutions are fully reliable but are based on CAN additional layers, making them impossible to integrate in a simple CAN architecture without any further support for additional protocol.

Moreover, CANAuth [19] proposed a solution of frame hash inside strict CAN, based on tiny encryption [13]. This solution proposes a solution to compute a hash based on a private key (MAC algorithm). In a similar work [5] another signature protocol for CAN bus based on CANAuth was proposed. We show here its integration in Cacao.

The market of CAN bus simulation is pretty mature, and several libraries already exists such as Python-CAN [3] or CaringCaribou [2]. However, these libraries are built upon direct CAN communication whereas Cacao is architectured as a standalone CAN bus simulator which can perform virtual, real or hybrid simulation of CAN traffic. The Cacao framework is built as an additional abstraction layer of the cantools [4] library. In the following work, we intend to show and test its additional functionalities on a real CAN bus implementation.

2 The Cacao Simulator

2.1 Introduction

Cacao is the combination of a physical CAN bus simulation platform and a development framework dedicated to CAN bus analysis. The purpose of the Cacao platform is to be able to test and analyse various cryptographic and signature algorithms through CAN protocol. It is dedicated to network simulation where each sensor or actuator from an actual vehicle is represented as an independent node generating random (or configurable) data and sending it on the CAN bus. Each node is represented by a set of software modules, and all nodes in a networks are linked among them by a common CAN bus. Cacao is basically oriented for vehicle internal communications but can be extended to all contexts of use of the CAN bus protocol.

2.2 Hardware

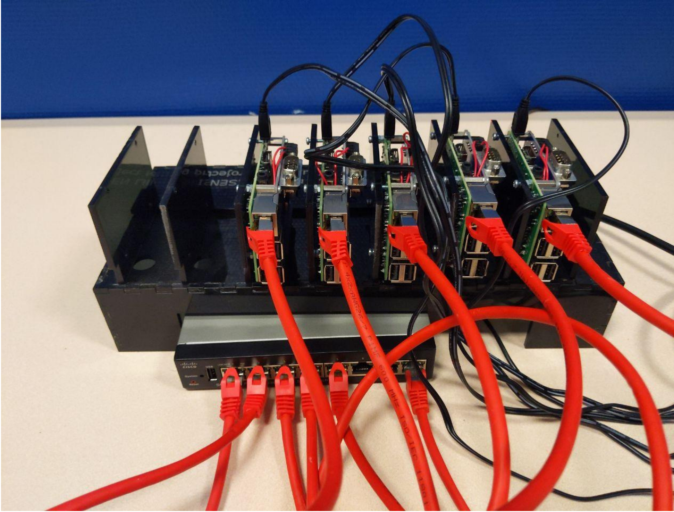

Fig. 1. Cacao simulation tool

Physically, Cacao consists in a Raspberry Pi (RPi) based backbone with additional CAN Shields. These shields are interconnected through two CAN interfaces (high rate and low rate). The whole rack is connected to a monitoring computer through Ethernet. Cacao is built on a modular approach. We can easily interconnect any additional RPi to the structure just by connecting it to the CAN bus and ethernet monitoring network. In its native version, Cacao is composed of 5 different RPi, which was considered sufficient for our simulation purposes.

Each RPi is shielded by a PiCAN 2 shield [7] which offers a high and a low rate bus connection (compliant to CAN standards [15]). All shields are linked among themselves through both buses (high rate and low rate). The complete tool bus is pictured on Fig. 1. As we can see, each Raspberry Pi is connected by Ethernet (through a dedicated switch) to a monitoring computer (see Fig. 3).

Each RPi emulates a dedicated set of nodes, all independantly connected to the CAN bus. Each subnetwork is connected to the CAN bus through the corresponding PiCAN 2 shield. We performed various experimentations on Raspberry Pi computation power (see Subsect. 4.1) to determine that we can, in its default configuration and considering the computation potential of the RPi, further simulate 30 nodes per RPi. It means that, in the regular 5-cards configuration of Cacao, we can emulate up to 150 different nodes, which corresponds to the number of nodes found in small vehicles [17,20].

2.3 Architecture

Cacao modelling is based on the concept of nodes. A node is a network end-point, able to send and receive messages. Each node behaves as an emission and reception entity, able to generate, send, receive and analyze its own messages. Nodes in Cacao are based on a structure of independant threads. As a result, each node is represented as a duo of threads, one for transmission and one for reception. Then, each individual node is connected to a common CAN bus.

The software part of Cacao is based on a CAN-dedicated library written in Python [1] and based on an open-source library named cantools [4]. Whereas cantools provides the basic interfaces to interconnect and send messages to the CAN bus, the threading and monitoring layer is integrated in Cacao framework directly. The software architecture of Cacao is detailed in Fig. 2.

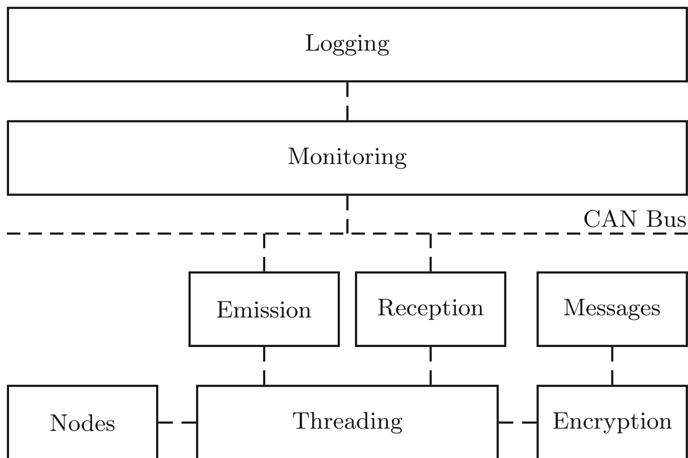

Fig. 2. Cacao framework modules

The framework is built around an object-oriented structure split into 4 different modules:

- The node module, dedicated transmission and reception (node simulation).
- The message module made for message and data management.
- The threading module allowing nodes to connect send and receive messages to and from the bus independantly.
- The monitoring module, dedicated to logging and data analysis.

This module-oriented structure integrates genericity in the tool, permitting to define additional tools as complementary functions for each module. For example, the encryption and signature functions such as the ones presented in [5] are defined as an upper layer of the message module.

In the bus, the Ethernet network is dedicated to maintenance and monitoring purposes. All the logs are centralized and managed by an administrating computer (connected to each RPi through Ethernet). All the nodes configuration and management are performed through a set of JSON configuration files (network and subnetwork size, traffic, etc.). This network architecture is detailed in Fig. 3.

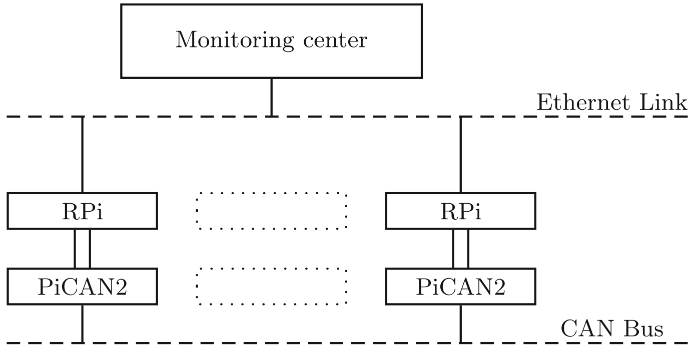

Fig. 3. Cacao architecture

2.4 Attack Model

When Cacao was designed, the primary goal was to test and improve signature solutions in the CAN bus. We detail below the attack model we target with these solutions.

Error management in the CAN bus is detailed in Fig. 4. It is based on 2 internal error counters integrated in each node. Each node has its own counters, one for transmission errors ($t_{c,n}$) and one for reception errors ($r_{c,n}$). Given a node n, when there is a difference between a message sent by n and what is on the bus (due to a message with higher priority), n will increase its internal transmission counter $t_{c,n}$ by 8 and send on the bus a message to warn that it got an error.

Once its transmission counter $t_{c,n}$ went beyond a threshold of 128, a node n is put on passive error state, meaning that it cannot warn all the other nodes anymore by sending error messages. This mechanism is further detailed in the litterature [5,8,14].

When targetting a given node for an attack, it is possible for an attacker to listen to the bus and send messages at the exact same time that the targetted node. Following this approach, the transmission error counter of the target will tend to increase. According to Fig. 4, once the transmission error counter goes beyond 255, a node will put itself in bus off mode, rendering it unable to send nor receive any message from the bus. In this situation, the node is considered,

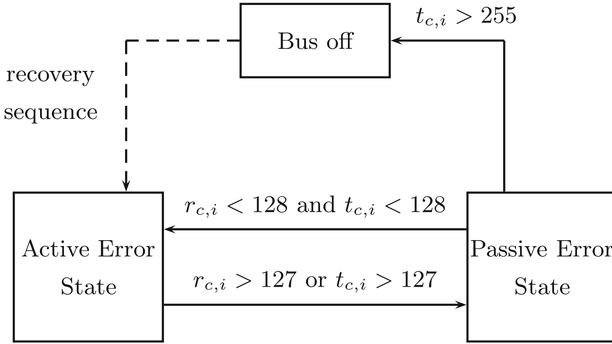

Fig. 4. Error management in CAN bus

from the point of view of the network, as dysfunctional. No further command can be sent nor received from this node, which makes it out of order for the vehicle. We can imagine attack scenarios targetting mechanics or safety-related sensors with this attack, putting the brakes or the airbag out of order in emergency situations.

3 Signature Modules

3.1 Signature Among CAN Bus

In [5], a signature solution for CAN bus was proposed, allowing to sign all messages in the network by reusing part of the CAN frames data and header to store the signature data. We summarize this process below, mostly the parts related to its integration in Cacao.

A CAN frame is composed of two different parts (see Fig. 5): a header to store the message identifier, and a payload to store the message content itself. In the CAN 2.0B version of, the number of bits (29) used to store the message identifier is higher than the number of different required ID in a regular CAN bus vehicle (around 1000) [17,20]. As a result, part of these bits (16) can be used to store signature data instead of message ID.

Algorithmically speaking, the signature of each message is computed with the SHA-256 algorithm. Each node stores local keys (see below, Sect. 3.2) and uses one specific local key (depending on the nature of the message) to compute a signature. The signature S of a given message will then depend on the local key K, the message data D and a random value r (unique for each message). The obtained signature is then truncated on 3 bytes to be integrated in the CAN frame (2 bytes in the headers, 1 byte in the payload). We keep the first three bytes of it [6]. The signature is then computed as follows:

$$S = SHA256(K \parallel D \parallel r)[0:3] \tag{1}$$

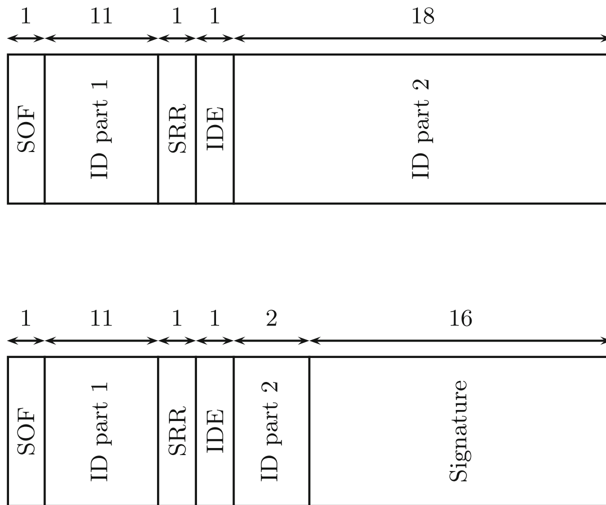

Fig. 5. CAN 2.0B protocol frames (without signature/with signature)

3.2 Group Keys

In a regular vehicle, each set of functions is associated to a given message id in the bus. For example, we have a dedicated ID for wipers speed increase, one for the decrease, one id for each steering wheel direction, a set of ids for gps tracking, etc. Each id concerns a group of sensors: each message from the steering wheel is only dedicated to be processed by specific nodes in the network. As a result, each message m_i concerns a static group of nodes determined by the network designer.

Each message can be sent to a specific set of nodes, which can be the entire bus or any subset of nodes. For example, the message corresponding to brake command will tend to concern mainly the brake pedal and the brake mechanism in the wheels. We reuse this to define groups. Each subset of nodes having, at least, one message in common is a group. If there is different messages identifiers for the same subset, then these messages will be in the same group.

In order to centralize and clarify the key management inside the simulator, a dedicated key management Application Programmable Interface (API) is integrated in the framework. The API integrates a mapping of every node in the network and its corresponding groups. Whenever a node or a message requires a specific key either to hash a message or to verify a hash, it makes a direct call.

In the section below, we call a message any data frame with a specified ID and a dedicated function. For example, hitting the brake will tend to emit a given message m_i, and hitting it several times will tend to send the same message m_i several times. Inside the CAN, there is no identification of the nodes. Only the messages are identified, and their id is transmitted in each message [8]. Each node is then configured with a set of masks and filters in order for the node to

receive only the messages it is supposed to manage. Each node can be configured to receive a specific set of messages corresponding to some ids. We propose to reuse this configuration to define message groups. This mechanism filter is further detailed in [8].

Given that each message m_i is associated with a group of nodes $\mathcal{G}_i$, we propose to define an encryption key K_i for each group of nodes. This way, each node from group $\mathcal{G}_i$ will be able to sign all messages concerning specifically $\mathcal{G}_i$. Each set of messages in the same group will be associated with the same key K_i, and only the nodes concerned by a group $\mathcal{G}_i$ will get to know and be able to sign all messages with ID m_i.

For each group $\mathcal{G}_i$, we define a group key $K_{\mathcal{G}_i}$. Using this approach, we can guarantee that, even if a node is corrupted, it will be likely to corrupt only the subsets of nodes which share a common group with it instead of all of the network like it is the case in standard CAN protocol.

3.3 Generating Keys and Keymapping

Each group key $K_{\mathcal{G}_i}$ is stored locally in each node of $\mathcal{G}_i$ and is never meant to be shared among the CAN bus. Each key is pontentially unique and is generated as follows: As we can see in Fig. 6, each key is an ASCII string. This is based on the approach that the keys in the network should be manipulated by potential external operators like passwords, for maintenance purposes. In a future version of Cacao, we can imagine group keys generation by *AES* key generation algorithms.

```
i = randomint(8, 12)
key = ""
for j in range(0,i):
    key = key + randomchar()
return key
```

Fig. 6. Group key generation algorithm in Cacao

Moreover, based on a key rotation mechanism detailed in [5], the group keys are frequently changed and randomized. Doing this prevents a potential attacker to corrupt a node just by copying its internal memory. It also prevents the network designer to have to manually generate keys for each group of nodes when performing maintenance tasks. Basically, the keys are randomly generated at start.

4 Experimentations

During the simulations, we generated CAN bus traffic over 150 independant nodes. The attackers were represented by a various set of external nodes (between

1 and 5) to perform a more efficient bruteforce. Each attacking node is in charge of forging random signatures with a constant set of data. We then compute the number of triggered errors dued to these messages.

4.1 Subnetwork Size Computation

In the Cacao platform, the nodes are grouped under different RPi cards. Each card represents a subset of nodes of a given size, all the cards among themselves representing the global CAN bus network. Following this approach, we operated a benchmarking process to test which number of nodes was adequate for each card. Considering the number of cores and the scheduling of all threads linked to the nodes, if the number of nodes goes beyond a certain limit, the nodes will not be fully independant because all the threads will tend to block each other due to scheduling and performance constraints. The results of this benchmark are shown in Fig. 7.

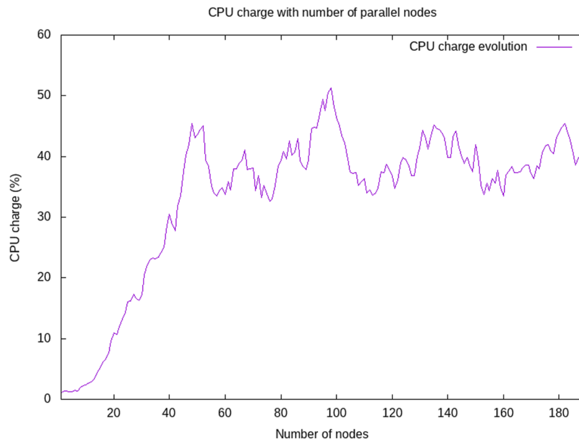

Fig. 7. CPU charge with the number of nodes

The Fig. 7 shows that, starting around 50 nodes per card, the CPU load represented by the Cacao framework will tend to converge. It means that, starting around 50 nodes per RPi, there is no more independance between the nodes and the different threads tend to block each other. In order to lower the risk and to anticipate an error margin, the default configuration of Cacao operates with 30 nodes per card.

4.2 Signature

Considering the signature solutions we integrated in the bus, performing an attack to put a node in bus-off state will now require enough messages to forge a

corrupted signature, i.e. to bruteforce the signature. We developed a tool which acts as an additional set of nodes connected to the CAN, but designed to send corrupted messages in it. Through the monitoring of the bus, we counter the number of corrupted signatures which were not detected by the nodes as false. The results are shown in Fig. 8.

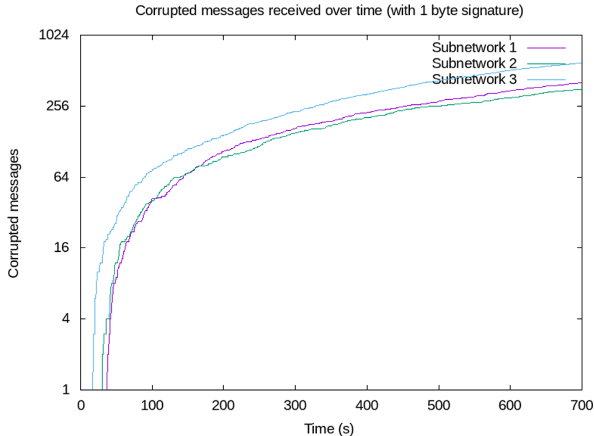

Fig. 8. Simulation of bruteforce on CACAO

In Fig. 8, we performed a simulation on 90 nodes splitted in 3 subnetworks of 30 nodes. We performed a random bruteforce attack, massively forging corrupted messages and sending them through the bus. We monitored the number of messages forged by the bruteforce tool (split on 10 threads). During the simulation, we observed the number of messages who had a valid 1-byte signature. During this duration, we also observed the number of corrupted messages who had a valid 3-bytes signature but did not observe any. This is due to the key rotation mechanism, which allowed to perform a sufficiently frequent rotation to avoid bruteforce in reasonable time (a few hours) of our signature model.

4.3 Attack Detection

In order to be able to detect a bruteforce attack on the CAN bus, we monitored the global bandwidth of the bus. These are the results shown in Fig. 9. We observed during a window of time of 80 s, with an attack during 20 s. As we can see, there is a strong increase of the used bandwidth between 50 and 70 s, which corresponds to the period during which we launched the bruteforce attack. As a conclusion, a solution to monitor the bandwidth of the CAN bus (coupled with signature implementation) could provide a potential solution to detect bruteforce attacks. Moreover, the needed time to detect an attack through bandwidth monitoring is far smaller than the needed time to forge a valid 3-bytes signature for a corrupted message. As a conclusion, our solution (tested on Cacao)

can provide a sufficiently reliable signature solution to represent an interesting potential of CAN bus communication protection.

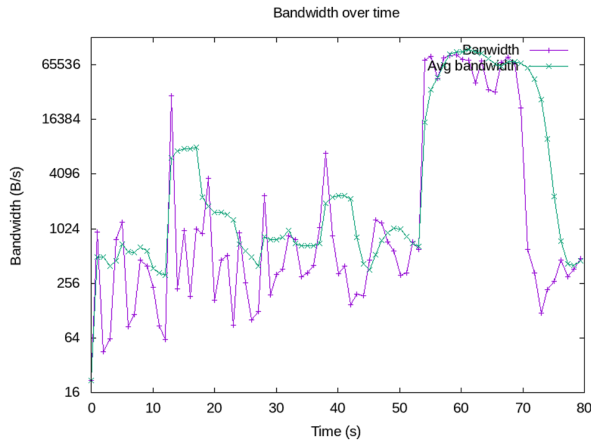

Fig. 9. Bandwidth monitoring

5 Conclusion and Perspectives

The Cacao project allows us to test various signature solutions among CAN and particularly the one developed in [5]. We conclude with the results of the benchmark that the signature protocol we provide implements a protection against the presented attacks in CAN bus. This way, we can prevent nodes to be put off the network by external attackers. It is a first step towards encryption in vehicular networks based on standard CAN and Cacao allowed us to test our solution on a real CAN infrastructure;

Thus, the perspectives of this work includes improvements to Cacao,mainly by increasing the degree of configuration available to the user but also by proposing a more advanced solution of data managing and monitoring, for example through visualization tools.

References

1. Can bus simulation framework. https://github.com/DoctorSauerkraut/canbus. Accessed 29 Sept 2020
2. Caring caribou security exploration tool. https://github.com/CaringCaribou/caringcaribou. Accessed 29 Sept 020
3. Python can bus simulation library. https://github.com/hardbyte/python-can. Accessed 29 Sept 2020
4. Python cantools. https://github.com/eerimoq/cantools. Accessed 29 Sept 2020

5. Cros, O., Chênevert, G.: Hashing-based authentication for can bus and application to denial-of-service protection. In: Cyber Security in Networking Conference CSNet 2019. IEEE (2019)
6. Dang, Q.: Recommendation for applications using approved hash algorithms. Technical report (2012)
7. Electronics, S.P.: Pican 2 datasheet. Technical report (2016)
8. GmbH, R.B.: Can specification version 2.0. Technical report (1991)
9. Hartkopp, O., Schilling, R.M.: Message authenticated can. In: Escar Conference, Berlin, Germany (2012)
10. Hartwich, F., et al.: Can with flexible data-rate. In: Proceedings of the ICC, pp. 1–9, Citeseer (2012)
11. Hoceini, O., Afifi, H., Aoudjit, R.: Authentication based elliptic curves digital signature for ZigBee networks. In: Bouzefrane, S., Banerjee, S., Sailhan, F., Boumerdassi, S., Renault, E. (eds.) MSPN 2017. LNCS, vol. 10566, pp. 63–73. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67807-8_5
12. Kurachi, R., Matsubara, Y., Takada, H., Adachi, N., Miyashita, Y., Horihata, S.: CaCAN-centralized authentication system in can (controller area network). In: 14th International Conference on Embedded Security in Cars (ESCAR 2014) (2014)
13. Jukl, M., Čupera, J.: Using of tiny encryption algorithm in can-bus communication. Res. Agric. Eng. **62**(2), 50–55 (2016)
14. Microchip: Dspic30f family reference manual, section 23. Technical report (2003)
15. Plummer, C., Roos, P., Stagnaro, L.: Can bus as a spacecraft onboard bus. In: DASIA 2003-Data Systems in Aerospace, vol. 532 (2003)
16. Ren, L.P., Zhou, J.: Canaerospace-upper layer protocol for can and its design application. Measur. Control Technol. **2**, 59–61 (2008)
17. Stence, R.W.: Digital by-wire replaces mechanical systems in cars. Technical report, SAE Technical Paper (2004)
18. Stock, M.: Interface specification for airborn can applications v1.7. Technical report (2006)
19. Van Herrewege, A., Singelee, D., Verbauwhede, I.: Canauth-a simple, backward compatible broadcast authentication protocol for can bus. In: ECRYPT Workshop on Lightweight Cryptography, vol. 2011 (2011)
20. Wang, Q., Sawhney, S.: Vecure: a practical security framework to protect the can bus of vehicles. In: 2014 International Conference on the Internet of Things (IOT), pp. 13–18. IEEE (2014)
21. Ziermann, T., Wildermann, S., Teich, J.: Can+: a new backward-compatible controller area network (CAN) protocol with up to 16× higher data rates. In: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 1088–1093. European Design and Automation Association (2009)