applied sciences

MDPI

Article

Prevention of Controller Area Network (CAN) Attacks on Electric Autonomous Vehicles

Salah Adly [1,2,*], Ahmed Moro [2], Sherif Hammad [1] and Shady A. Maged [1]

[1] Mechatronics Engineering Department, Faculty of Engineering, Ain Shams University, Cairo 11517, Egypt; sherif.hammad@eng.asu.edu.eg (S.H.); shady.maged@eng.asu.edu.eg (S.A.M.)
[2] Siemens Digital Industries Software, Integrated Electrical Systems Segment, Cairo 11835, Egypt; ahmed.moro@siemens.com
* Correspondence: salah.adly.ext@siemens.com

Abstract: The importance of vehicle security has increased in recent years in the automotive field, drawing the attention of both the industry and academia. This is due to the rise in cybersecurity threats caused by (1) the increase in vehicle connectivity schemes, such as the Internet of Things, vehicle-to-x communication, and over-the-air updates, and (2) the increased impact of such threats because of the added functionalities that are controlled by vehicle software. These causes and threats are further amplified in autonomous vehicles, which are generally equipped with more electronic control units (ECUs) that are connected through controller area networks (CANs). Due to the holistic nature of CANs, attacks on the networks can affect the functionality of all vehicle ECUs and the whole system. This can lead to a breach of privacy, denial of services, alteration of vehicle performance, and exposure to safety threats. Although cryptographic encryption and authentication algorithms and intrusion detection systems (IDS) are currently being used to detect and prevent CAN bus attacks, they have certain limitations. Therefore, this study proposed a mitigation scheme that can detect and prevent such attacks at the ECU level, which could address the limitations of existing algorithms. This study proposed the usage of a secure boot scheme to detect and prevent the execution of malicious codes, as the presence of one or more ECUs with a malicious code is the root cause of most CAN bus attacks. Secure boot schemes apply cryptographic data integrity algorithms to ensure that only authentic and untampered software can run on the vehicle's ECUs. The selection of an appropriate cryptographic algorithm is important because it affects the secure boot schemes' security level and performance. Therefore, this study also tested and compared the performance of the proposed secure boot scheme with five different data security algorithms implemented using the hardware security module (HSM) of the TC399 32-bit AURIX™ TriCore™ microcontroller through an electric autonomous vehicle's control unit. The tests showed that the two most favorable schemes with the selected hardware are the secure boot scheme with the cipher-based message authentication code (CMAC), because it possesses the highest performance with an execution rate of 26.07 (ms/MB), and the secure boot scheme with the elliptic curve digital signature algorithm (ECDSA), because it provides a higher security level with an acceptable compromise in speed. This study also introduced and tested a novel variation of the ECDSA algorithm based on the CMAC algorithm, which was found to have a 19% performance gain over the standard ECDSA-based secure boot scheme.

Keywords: controller area network; CAN bus; denial of service; automotive safety; automotive cybersecurity; secure boot; hardware security module; HSM; over-the-air update; data integrity algorithms; illegal tunning; autonomous vehicles; intrusion detection systems; IDS

Citation: Adly, S.; Moro, A.; Hammad, S.; Maged, S.A. Prevention of Controller Area Network (CAN) Attacks on Electric Autonomous Vehicles. *Appl. Sci.* **2023**, *13*, 9374. https://doi.org/10.3390/app13169374

Academic Editors: Yuejiao Gong, Qiang Yang and Ting Huang

Received: 19 July 2023
Revised: 12 August 2023
Accepted: 13 August 2023
Published: 18 August 2023

1. Introduction

The functionality of modern vehicles has increased in recent years, with vehicle manufacturers adding more features to enhance vehicles' comfort, safety, efficiency, performance, and autonomy. The realization of these features is achieved through (1) increasing vehicles'

connectivity to the outside world via schemes such as the Internet of Things, vehicle-to-x communication, and over-the-air updates, and (2) adding more software components and more complex software, which are deployed to a larger number of electronic control units (ECUs). These ECUs are connected through various communication networks including controller area networks (CANs). This increase in external connectivity and code complexity leads to increased chances of cybersecurity attacks because of the enlargement of the attack surface caused by the expansion of possible avenues of attacks, as well as an increase in such attacks' impact, which is due to vehicles' increasing reliance on software. This effect can also be magnified due to an increase in the internal connectivity of the implemented ECUs through the CAN bus. This further expands the attack surface as an attack on a single ECU can be transferred to other ECUs that communicate with the attacked ECU. Hence, a malicious code injection attack or a code reprogramming attack on a single ECU can affect all ECUs connected to the CAN, which, depending on the vehicle architecture, could be the whole system.

A CAN bus was introduced to vehicles to enable real-time communication between the vehicles' ECUs to enhance the functionalities, performance, comfort, and safety of the vehicles. Although the CAN bus is designed for safe communication, it is not designed for secure communication, which has led to various vulnerabilities, such as the deficiency in network segmentation, the lack of device authentication, and the absence of data encryption [1]. Such vulnerabilities can be exploited by attackers, who have access to the network, to gain knowledge of private information, alter the vehicle's behavior, impersonate an ECU, and perform a denial-of-service (DoS) attack. An attacker can gain access to an ECU by adding a malicious code to an existing ECU that performs the attack, or by introducing a malicious ECU to the network. Between these two paths of attacks, the path of adding a malicious code to an existing ECU is more dangerous because it can be performed remotely without the need for physical access to the vehicle, and it can be scaled to other vehicles that contain the same vulnerability.

Existing mitigation schemes against CAN bus attacks rely on cryptographic algorithms for encryption and authentication, or on machine learning and deep learning algorithms implemented in intrusion detection systems (IDS). Although both of these types of algorithms are capable of preventing or detecting CAN bus attacks, each has its limitations. Cryptographic encryption and authentication algorithms introduce latency in communication, rely on the secrecy and strength of the used key, may require changes to all CAN nodes, and may alter CAN bus behavior. IDS systems lack mitigation actions, are not able to detect all attacks, introduce communication latency, rely on data training, and may not detect minor deviation attacks. Because of the mentioned limitations, this paper suggests the usage of a mitigation scheme that detects and mitigates attacks at the ECU level rather than at the network level. The proposed secure boot scheme is able to detect and stop attacks at an early stage by detecting the presence of a malicious code in an ECU that is connected to the CAN bus and preventing the execution of that malicious code. This prevents the manipulation of the CAN bus that may have resulted from the malicious code.

The contributions of this manuscript are as follows: (1) It presents a secure boot scheme to eradicate the root cause of most CAN bus attacks by detecting and mitigating the presence of malicious code in all vehicle ECUs that are connected to the bus. (2) It implements and tests five different variants of the suggested secure boot scheme, using the TC399 32-bit AURIX™ TriCore™ microcontroller via the usage of a productized ready-to-deploy software. The five tested variants are the hash variant, hash-based message authentication code (HMAC) variant, cipher-based message authentication code (CMAC) variant, elliptic curve digital signature algorithm (ECDSA) variant, and RSA signature scheme with appendix–probabilistic signature scheme (RSASSA PSS) variant. (3) It compares the results of the tested variants and concludes that the most appropriate variants to be used are the CMAC variant, as it is the fastest, and the ECDSA variant, as it provides higher security than the CMAC variant with an acceptable speed downgrade. (4) It introduces a new secure boot variant that is based on the novel ECDSA-CMAC data integrity algorithm,

which results in a 19% increase in performance over the traditional hash-based ECDSA variant. Based on our knowledge of the latest published papers that are publicly available, the usage of ECDSA-CMAC as a data integrity algorithm and in the secure boot process is a novel approach, which this manuscript proposes to improve the performance of the ECDSA algorithm in the secure booting of systems with faster CMAC calculation than hash calculation.

This paper is structured as follows: In the following section, a brief description of possible CAN bus attacks is presented. In Section 3, the possible mitigation mechanisms for CAN bus attacks are identified and compared. An overview of the secure boot process and different secure boot strategies is provided in Section 4. An introduction to the five tested validation algorithms and the ECDSA-CMAC algorithm is given in Section 5. In Section 6, the selected hardware is described. The tested software and the tests results are outlined in Section 7, while the conclusions are drawn in Section 8.

2. CAN Bus Attacks

Although modern vehicles are reliant on CAN buses in internal ECU communication, CAN buses possess multiple security vulnerabilities, such as the deficiency in network segmentation, the lack of device authentication, and the absence of data encryption. These vulnerabilities can be exposed by attackers to gain access to confidential data, alter the vehicle functionality, and possibly endanger the targeted vehicle's driver and passengers.

Attackers target vehicle CAN buses through attacks such as sniffing attacks, fuzzing attacks, frame falsifying attacks, injection attacks, DoS attacks, and ECU impersonation [1]. Some of the mentioned attacks, if performed successfully on a vehicle, can unlock other attacks on other vehicles with the same CAN architecture. For example, an attacker can use sniffing attacks to gain access to CAN bus confidential messages' specifications. The attacker then can use those specifications to manipulate and generate confidential messages [2], unlocking other attacks such as frame falsifying attacks and injection attacks. Another example is attackers using fuzzing attacks to discover possible vulnerabilities in the CAN system that are not yet known to the manufacturer, leading to zero-day attacks [3]. The most common CAN attacks are summarized in Table 1.

Table 1. CAN bus attacks, the vulnerabilities that each attack exploits, and whether a successful attempt of an attack on a vehicle could unlock the same attack or other attacks on other vehicles.

CAN Bus Attack	Exploited Vulnerabilities	Unlock Attacks on Other Vehicles
Sniffing attacks	Absence of segmentation Absence of data encryption	Yes
Fuzzing attacks	Absence of segmentation Absence of authentication	Yes
Frame falsifying attacks	Absence of segmentation Absence of authentication	No
Injection attacks	Absence of segmentation Absence of authentication Absence of data encryption	Yes
DoS attacks	Absence of segmentation	No
ECU impersonation	Absence of segmentation Absence of authentication Absence of data encryption	Yes

Attackers could gain access to the CAN bus through two attack paths. The first attack path, as shown in Figure 1, is carried out by injecting a malicious code into an ECU that is connected to the CAN bus, which the attackers can use to sniff on the network and to inject their attacks. The second attack path, as shown in Figure 2, is carried out by connecting a malicious external ECU to the network and using that malicious ECU to perform the

attacks. Between these attack paths, the first path imposes more threat because it is scalable to other vehicles, and it does not require physical access to the vehicle as malicious code injection could occur remotely [4,5].

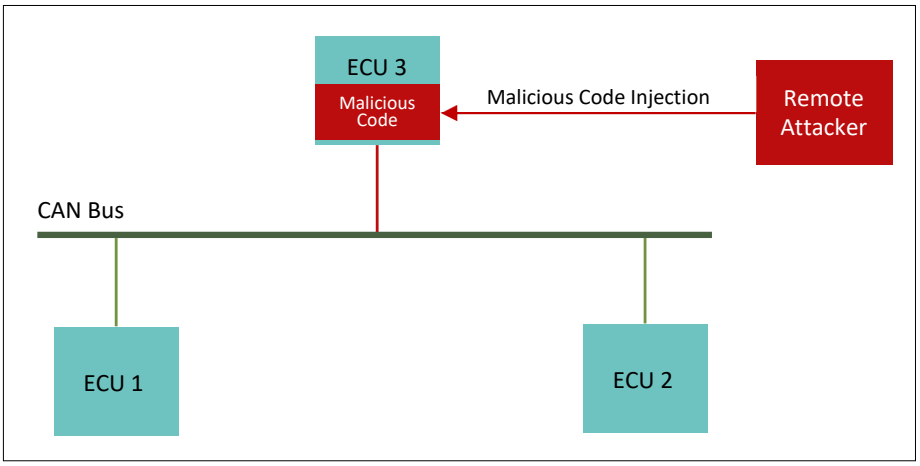

Figure 1. CAN attack path through malicious code injection (the first attack path).

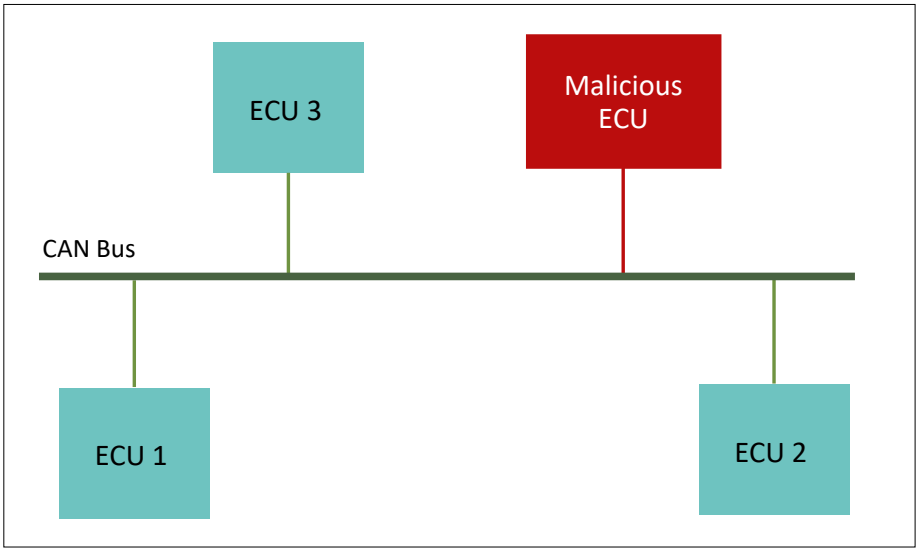

Figure 2. CAN attack path through adding a malicious ECU (the second attack path).

As an example of the first attack path, the authors of [6] injected a code that monitors vehicle behavior, and if a specified condition occurs, it sends a CAN message containing an open window command to open the window. In [4], a personal computer (PC) attached to a powertrain network was used to demonstrate the danger of interfering with the CAN bus. The attached PC, which is an example of an added malicious ECU in the second attack path, interfaced with the CAN bus to falsify the presence of the airbag system when it was removed.

The literature contains various other attacks targeting the CAN bus. In [7], experiments were conducted to demonstrate that malicious access to the CAN bus can lead to the attackers taking control over critical components of the targeted vehicle, such as the brakes. Ref. [8] performed a DoS attack by occupying the CAN bus with high-priority messages sent at high frequencies. Ref. [9] used a spoofing attack to impersonate a targeted ECU and caused an error that caused the targeted ECU to be disconnected from the network. In [10], the authors were able to use the Wi-Fi hotspot and the cellular network interface of the 2014 Jeep Cherokee to remotely access its CAN bus.

3. CAN Bus Attack Mitigation Schemes

Detecting and preventing CAN bus attacks has been a field of interest in academia, leading to the development of several detection and prevention schemes. These detection and prevention schemes are classified into cryptography-based encryption and authentication schemes, and intrusion detection systems.

Cryptography-based schemes ensure CAN messages' confidentiality by applying cryptographic encryption algorithms and ensure these messages' integrity and authenticity by using message authentication codes (MACs) and digital signatures [11]. Ref. [12] proposed the usage of a 128-bit advanced encryption standard (AES-128) to encrypt CAN messages at the sender node and then decrypt the messages at the receiver node. This protects the network against sniffing attacks, as an eavesdropper will gain access to the encrypted messages without knowledge of the used key. Hence, the attacker will not be able to decipher the content of the messages. The usage of encryption algorithms also prevents injection attacks and ECU impersonation, as an attacker who does not have access to the used key cannot send targeted falsified messages to the receiver nodes, who expect the received messages to be encrypted.

Although encryption algorithms ensure the confidentiality of transmitted messages, they do not guarantee the authenticity of these messages' sources. Therefore, encryption algorithms cannot be used to protect against attacks such as fuzzing attacks. Also, encryption algorithms cannot protect against injection attacks and ECU impersonation if an attacker has access to the encryption key, which could be achieved by injecting a malicious code into an ECU that has knowledge of the encryption key. This limitation is addressed through the usage of message authentication codes and digital signatures that can be used to verify the authenticity of the messages' sources. The authors of [13] suggested replacing the cyclic redundancy check (CRC) field in the CAN frame with cipher-block chaining (CBC) MAC, which is generated by the sender node and verified at the receiver node using a key that is only shared between the two nodes.

Although cryptographic encryption and authenticity schemes provide preventive algorithms to CAN bus attacks, they possess multiple limitations. Firstly, cryptographic encryption algorithms require a change to all CAN bus nodes because all nodes should be able to decrypt and encrypt messages. Secondly, the usage of cryptographic algorithms increases message latency because of the extra computations performed for each message. This delay can be decreased through the usage of cryptographic hardware accelerators but at a large cost of implementation. Thirdly, some cryptographic algorithms, such as MACs and digital signatures, require changes to the CAN bus behavior to include the added verification tags. Finally, the strength of cryptographic algorithms relies on the secrecy of the used key, which is vulnerable because it is shared between the system's ECUs.

Intrusion detection systems use machine learning and deep learning algorithms to detect the presence of malicious messages in the CAN bus. Intrusion detection systems are classified into signature-based IDSs and anomaly-based IDSs. Signature-based IDSs use a database of CAN bus attacks to extract the signature of each attack and use these signatures to detect malicious behavior in the bus. Although signature-based IDSs have a high prediction rate of attacks, they require a continuous update of the database to be able to detect new attacks [14]. Anomaly-based IDSs use learning models to monitor the CAN traffic behavior and the CAN physical characteristics and to identify abnormal behavior or characteristics [2]. Anomaly-based IDSs monitor traffic behavior such as message frequency, message payload, and message identifiers, and monitor physical characteristics such as voltage and signal profiles.

The study and development of CAN IDSs has been extensively investigated in the literature. The authors of [15] developed a system that identifies the ECUs connected to the CAN bus and detects if a malicious ECU has been added to the bus. The developed system connects a monitoring unit that observes the signal profile of the network and sends an alarm when a malicious ECU is detected. The developed detection system is useful in detecting CAN bus attacks performed through the attack path of adding a malicious

ECU, but it cannot detect attacks carried out through injecting a malicious code into an existing ECU.

A novel intrusion detection system called CANintelliIDS was developed to identify CAN bus single intrusion and mixed intrusion attacks [16]. CANintelliIDS uses a combination of attention-based gated recurrent unit (GRU) and convolutional neural network (CNN) to analyze the timing behavior of the CAN bus traffic. CANintelliIDS is able to detect DoS attacks, fuzzing attacks, and ECU impersonation attacks.

A combination of CNN- and LSTM-based deep learning models was used to develop a novel intrusion detection system, called NovelADS [17], which is capable of detecting four types of attacks: fuzzing attacks, RPM spoofing attacks, gear spoofing attacks, and DoS attacks. Since NovelADS is trained using unsupervised learning, it can effectively identify zero-day attacks.

The authors of [18] developed a multiple-observation hidden Markov model (HMM) that is able to detect the anomaly of a single frame rather than over a time period. The multi-observation HMM model observes the timing characteristics of the CAN bus and the messages' identifier (ID) sequence to calculate the existence possibility of a CAN frame. This approach is proven to have a better frame-by-frame anomaly detection performance than the other approaches studied in the literature in detecting fuzzing attacks, DoS attacks, replay attacks, and ECU impersonation.

A novel intrusion detection system that is based on attention mechanisms and a developed autoencoder is proven to be able to detect CAN bus attacks in real time and to outperform traditional machine learning algorithms [19]. The attention mechanism and autoencoder for intrusion detection (AMAEID) algorithm uses the autoencoder to decode a CAN message into a binary format and to detect the hidden feature representation. Then, the attention mechanism with the aid of a single-layer connected network outputs a prediction of whether the message is normal or abnormal.

To overcome communication latency and possible privacy breaches that may occur in IDS that use centralized learning approaches, the usage of federated-learning approaches has been further investigated by the literature. The research conducted in [20] developed a federated long short-term memory (LSTM)-based intrusion detection system. The federated LSTM is able to detect attacks such as DoS, drop, replay, and spoofing attacks, with an accuracy of over 90%. The usage of federated LSTM overcomes the limitations of deep neural network (DNN) algorithms, such as the need for a centralized cloud server model training which can lead to privacy data leak. The proposed federated LSTM algorithm monitors messages' identifier sequence, predicts the upcoming message's ID, and detects if an abnormal message is sent through the bus. Another example of an intrusion detection system that uses federated learning is the ImageFed privacy-preserving intrusion detection system [21], which uses a federated CNN model to detect attacks such as fuzzing, gear spoofing, RPM spoofing, and DoS attacks.

Current state-of-the-art intrusion detection systems can detect abnormal message frames with a high frequency, but there exist some limitations to current CAN bus IDSs. Firstly, most IDSs do not provide mitigation or prevention schemes and are limited to just detecting and reporting abnormal frames. Secondly, current IDSs cannot detect all CAN bus attacks, such as sniffing attacks which are not detected nor prevented by IDSs. Thirdly, some IDSs fail to detect CAN messages with minor deviation and abnormality [19]. Fourthly, IDS systems have detection latency because of the complexity of the underlying machine learning and deep learning algorithms. Finally, IDSs rely on data training, which generally requires a large data set, or they may be ineffective.

Unlike the techniques discussed in the literature, this paper proposes a CAN bus attack prevention scheme to be implemented at the ECU level rather than at the CAN bus level. This paper suggests the usage of secure boot process on all ECUs that are connected to the network; this is to prevent any CAN bus attacks that may result due to the presence of malicious software on a connected ECU. The secure boot should detect an attack at its

root level and apply the necessary countermeasures to prevent the attack from continuing to the CAN bus.

The proposed scheme can mitigate the limitations of existing state-of-the-art cryptographic encryption and authentication algorithms, and intrusion detection systems. Because the secure boot scheme is implemented during the boot process only, it does not require changes in CAN bus behavior and does not add latency to CAN bus communication, as the only delay introduced is at the startup of the ECUs. While the secure boot scheme may use a secret key, the vulnerability associated with using secret keys is not the same as that of cryptographic encryption and authentication because the secure boot key is not shared between ECUs. Because secure boot schemes prevent the execution of malicious codes, secure boot is able to prevent all CAN bus attacks that occur through the path of injecting a malicious code into an existing ECU. Additionally, secure boot has the added benefit of mitigating the risk of executing a malicious code in a vehicle's ECUs even if it does not affect the CAN bus. Finally, as secure boot algorithms are built on strong cryptographic functions, secure boot is able to detect minor changes in the code. A comparison between the existing state-of-the-art CAN attack prevention schemes and the proposed secure boot scheme is summarized in Table 2.

Table 2. Comparison between the CAN bus attack mitigation schemes.

Scheme	Methodology	Merits	Limitations
Cryptographic algorithms	Usage of cryptographic encryption to guarantee the confidentially of CAN messages, and of cryptographic MACs and digital signatures to guarantee the authenticity of the sender.	• Guarantee message confidently. • Guarantee message authenticity. • Stop sniffing, injection, ECU impersonation, and frame falsifying attacks.	• Change to all CAN nodes. • Increased latency. • Change to the CAN bus behavior. • Reliance on key secrecy.
Intrusion detection systems	Usage of machine learning and deep learning algorithms to detect anomalies in CAN messages.	• Detect fuzzing, injection, ECU impersonation, DoS, and frame falsifying attacks. • Able to learn from new attacks. • Able to predict new attacks.	• Lack of mitigating actions. • Not all attacks are detected. • Minor deviation may not be detected. • Increased latency. • Reliance on data training.
Proposed secure boot scheme	Usage of cryptographic integrity algorithms to detect the presence of a malicious code and perform the necessary mitigation actions to prevent the code from affecting the CAN bus.	• Prevents all attacks that are due to malicious code injection. • Added benefit of mitigating malicious code risks that do not affect the CAN bus.	• Change to all CAN nodes. • Cannot detect attacks due to added malicious ECUs.

The proposed secure boot scheme's limitations are that it cannot detect attacks that involve adding a malicious ECU to the CAN bus and it has a high cost of implementation. The high implementation cost is because all ECUs of the system have to incorporate the secure boot algorithm for the scheme to be effective, as indicated in Figure 3.

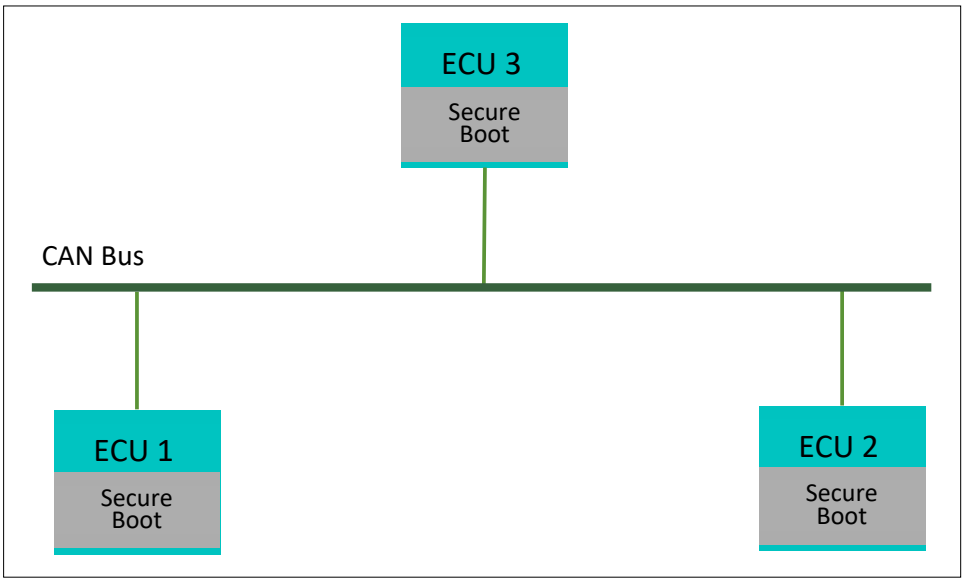

Figure 3. CAN bus with deployed secure boot scheme.

4. Secure Boot Process

A secure boot process is a process that starts each boot cycle of an ECU to verify the integrity of the software stored in the ECU's non-volatile memory before permitting the software to execute. The secure boot process uses cryptographic data integrity algorithms to ensure the integrity of the software by calculating an integrity value for the software and comparing the integrity value to a stored reference value.

The number of electronic control units in modern vehicles exceeds 100 ECUs [22]. Therefore, performing a secure boot process in vehicles' ECUs is crucial to ensure that tampered software is not allowed to execute. Executing tampered software poses the risk of altering the performance of vehicles, breaching the privacy and confidentiality of vehicle owners, and imposing safety threats to passengers and pedestrians. A malicious code impacts not only the targeted ECU but also other ECUs connected through the vehicle networks like the CAN bus. Performing a secure boot process prevents a malicious code from performing CAN bus attacks by not allowing the malicious code to execute.

Although the secure boot concept was recently introduced in modern vehicles, it has been long used in classic computer systems. Ref. [23], which is one of the first research studies in secure booting, suggested the usage of a physically secure coprocessor to perform a set of security protocols including secure booting. The implemented security protocols aim to protect against the security threats that were imposed at the time. The concept of using physically secure coprocessors to secure boot the application processor is currently realized through the usage of secure hardware modules such as trusted platform modules (TPMs) [24], secure hardware extensions (SHEs), and hardware security modules (HSMs) [25]. This study used the HSM of the TC399 32-bit AURIX™ TriCore™ microcontroller to perform the secure boot process.

Currently, many vehicle manufacturers are opting to include secure boot as a requirement for their ECUs to prevent illegal tuning and to protect against risks associated with executing tampered software even once. Executing unauthentic software may permanently change the ECU functionality [26], which may threaten both drivers' and pedestrians' lives [27].

A secure boot process could be implemented using various strategies, most of which are built on the concept of chain of trust. In strategies that use a chain of trust, the secure boot is composed of multiple boot stages, where each boot stage validates the succeeding boot stage and gives it trust if it is authentic [28], and if all boot stages are authentic, the whole software is considered authentic. A chain of trust could be viewed as if the initial

boot stage (BS_0) secure boots the following boot stage (BS_1), which in turn secure boots the following boot stage, and so on, until all boot stages are secure booted. Although it is more common that all boot stages use the same secure boot strategy, some chain-of-trust systems include boot stages that use different secure boot strategies such as different software integrity algorithms [29].

As shown in Figure 4, each boot stage is formed of three components. The measurement component is responsible for measuring the software integrity value and validating this integrity value against the reference value. The storage component is responsible for storing the values needed by the measurement component, such as the reference value and the cryptographic key. The storage component is also responsible for preserving the confidentiality of cryptographic private keys [29]. Finally, the reporting component is responsible for applying the necessary countermeasures if the secure boot of the succeeding boot stage fails. All three components of each boot stage should be validated by the preceding boot stage.

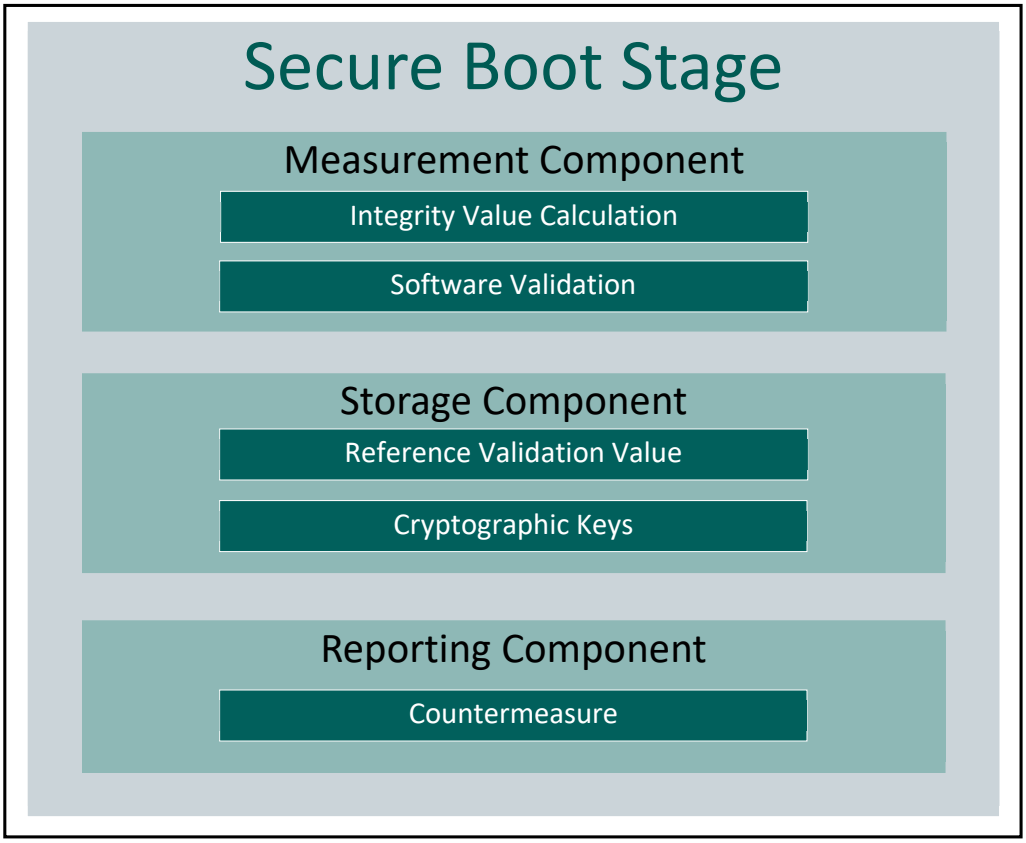

Figure 4. The three components of a secure boot stage.

The initial boot stage of a chain of trust is called the root of trust (RoT), which is assumed to be authentic and trusted because its integrity is not validated in the secure boot process. Therefore, the RoT should be protected through hardware mechanisms such as making the RoT one-time programmable (OTP) or deploying the RoT to secure hardware modules such as HSMs and TPMs. As shown in Figure 5, the RoT is used to validate the succeeding stages and it is inherently trusted. Therefore, an attack on the RoT can compromise the whole boot process [30]. The three components of the RoT are called the root of trust for measurement, the root of trust for storage, and the root of trust for reporting.

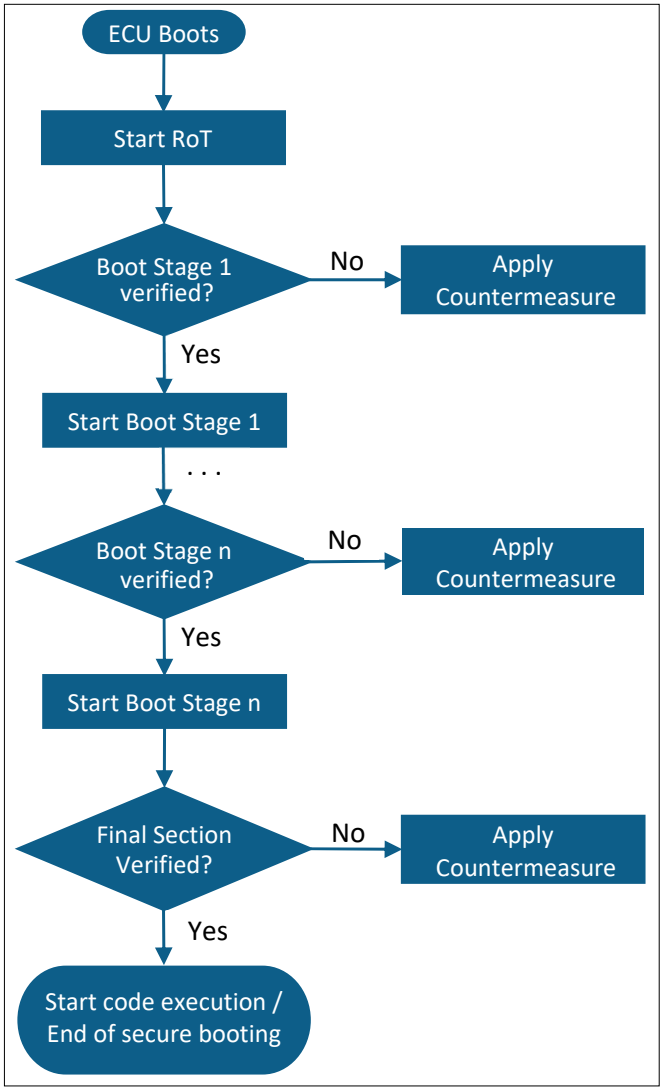

Figure 5. Flowchart of the secure boot's chain of trust.

Secure boot strategies are distinguished according to four parameters: the integrity algorithm used to validate the software, the software sections that will be validated, the selected execution mode, and the countermeasures that will be applied if the secure boot fails. The four parameters should be carefully selected to not compromise the system and to meet the ECU timing requirement. Typically, ECUs are required to be able to receive CAN messages within 50–100 ms after booting up [31,32].

The integrity algorithm in the system is the algorithm applied to the software data to calculate the software validation value, which is compared to the stored reference value to detect whether the software has been tampered with. The integrity algorithm used in the secure boot process is a massive factor in the effectiveness of the process because it affects the security level and the speed of the secure boot process. This study compared five validation algorithms in terms of speed and security level. The five algorithms are implemented using the HSM of the TC399 32-bit AURIX™ TriCore™ microcontroller used in an electric autonomous vehicle's control unit. The five tested validation algorithms are the secure hash algorithm (SHA) 256, the HMAC algorithm, the CMAC algorithm, the ECDSA algorithm, and the RSASSA-PSS algorithm.

Also, to benefit from the high speed of the CMAC algorithm on the selected hardware and the added security of the ECDSA algorithm, this study introduced and tested a novel variation of the ECDSA algorithm. The CMAC-based ECDSA algorithm uses the CMAC

algorithm to calculate the message image instead of the standard hash-based functions that are used in the traditional ECDSA algorithm.

Although it is not recommended to execute software sections that have not been validated, some secure boot strategies choose not to secure boot low-priority software sections. These strategies opt to restrict the authority and accessibility of these sections instead of validating them to speed up the secure boot process. Executing unauthenticated software sections should be investigated carefully to ensure that attacks on these sections will not compromise the vehicle's security. Furthermore, the CAN bus should only be accessed by authentic software sections to prevent malicious access to the bus, which can jeopardize the full network.

The execution mode of the secure boot process determines the different secure boot stages and identifies whether these boot stages are validated sequentially, in parallel, or in a combination of sequential validation and parallel validation. The sequential execution mode is when a boot stage is only allowed to execute after it has been validated and trusted, while the parallel execution mode is when a boot stage could be executed before its validation has been completed. In parallel execution, a boot stage is released and then it is validated parallel to its execution. A boot stage executing in a parallel mode should be restricted so that it is not able to interfere with its validation process. The advantage of the sequential mode over the parallel mode is that it offers a higher security level [33] because software sections are not allowed to execute prior to their validation. But in some cases, to meet the timing requirements, the parallel mode is selected to speed up the secure boot process.

Countermeasures are the mitigation actions applied by the secure boot process when the validation of a boot stage fails. Countermeasures may consist of one or more of the following: (1) bricking the ECU by not allowing it to execute further code; (2) performing a safe fallback procedure for critical ECUs and then bricking the ECU; (3) restricting the accessibility and authority of the unauthentic software section, which can include blocking access of the unauthentic software section to the CAN bus; (4) blocking the execution of the unauthentic software section; (5) locking the cryptographic secrets and preventing these secrets from being read or modified; and (6) sending a notification indicating that one or more software sections have failed validation.

Although the usage of secure boot processes in automotive ECUs is not excessively reviewed in the literature, various research papers have addressed the subject. Ref. [32] implemented a secure boot process of an engine management system using a combination of sequential secure boot and parallel secure boot and using a CMAC as the validation algorithm. The secure boot process was executed via the HSM module of the Infineon TC27x 32-bit AURIX™ TriCore™ microcontroller. In [34], a secure boot process was implemented on a trusted platform module using an HMAC validation scheme. Although [33] suggested the usage of cyclic redundancy check as a validation scheme, it is not recommended to use non-cryptographic schemes because they could be easily bypassed by attackers [29].

With the advancement of quantum computers and the risk they impose on classic cryptographic schemes, more research is being conducted on using post-quantum digital signature schemes in secure boot processes. Examples of such schemes include Dilithium digital signature scheme and eXtended Merkle signature scheme (XMSS) [35,36].

With the increased need for fast authentication protocols, more novel authentication protocols are being introduced in the literature. Ref. [37] proposed an ultra-fast authentication protocol that uses extended chaotic maps to authenticate the communication between electric vehicles and charging stations. Although such algorithms are not intended for secure booting, their merits to secure boot schemes should be further investigated.

5. Data Integrity Algorithms

Proper selection of the data integrity algorithm used in the secure boot process is crucial because data integrity algorithms affect the performance and security of the secure boot process and may render the secure boot process ineffective if they are improperly

selected or implemented. Therefore, only state-of-the-art and tested integrity algorithms should be used. Although each boot stage could use a different integrity algorithm, the general case is that all boot stages use the same algorithm. In this section, each of the five tested validation algorithms is briefly described and the novel ECDSA-CMAC algorithm is introduced.

5.1. SHA 256 Hash Function

Hash functions are one-way functions that take a variable length input and result in a fixed length output called a hash value or a digest value. Strong hash functions should have the properties of collision resistance, preimage resistance, and second preimage resistance [38]. Collision resistance means that it is computationally infeasible to have two different hash inputs that output the same hash value. Preimage resistance means that knowing a specific hash value, it is computationally infeasible to find an input hash that outputs that specific hash value. Preimage resistance gives hash functions their one-way characteristic. Second preimage resistance means that knowing a specified hash input, it is computationally infeasible to find a second hash input that outputs the same hash value.

In a secure boot process, hash functions are applied to the software to be validated to output the validation value; therefore, only approved strong hash functions should be used. This is because collision resistance guarantees that any change in the software will result in a different hash value, which will fail the validation process. Also, second preimage resistance guarantees that an attacker will not be able to find malicious software that generates a similar hash value to the authentic software, even if the attacker has knowledge of the authentic software. And lastly, preimage resistance guarantees that even if the reference validation value is not confidential, the attacker will not be able to produce malicious software that will bypass the validation.

SHA 256 [39] is a hash function from the SHA-2 family of hash algorithms, which are approved by the National Institute of Standards and Technology (NIST) [38]. SHA 256 outputs a 256-bit hash value and has 128-bit collision resistance strength, and 256-bit preimage resistance strength and second preimage resistance according to Equation (1):

$$\text{second preimage resistance} = 256 - \mathrm{L(M)} \tag{1}$$

where L(M) is defined as

$$\mathrm{L(M)} = \mathrm{Log_2}\left(\frac{\mathrm{Len(M)}}{512}\right) \tag{2}$$

and Len(M) is the length of the software in bits [38].

Since SHA 256 does not require a cryptographic key, it is recommended to use a unique ECU identifier in the hash computation. This makes the reference validation value ECU-specific and prevents attacks on a single ECU from scaling to similar ECUs that execute the same software [29].

Aside from being used for integrity checks, hash functions are the building blocks of multiple cryptographic algorithms, such as HMAC, ECDSA, and RSASSA-PSS. Hash functions are also used in pseudo-random number generators (PRNGs), in the construction of block ciphers, in key derivation algorithms, and in digital time-stamping algorithms [40].

5.2. HMAC SHA 256 Algorithm

HMAC algorithms are message authentication code algorithms that are based on hash functions. MAC algorithms use symmetric cryptographic keys to generate a digest value for the software data that ensures both the software's integrity and authenticity. HMAC performs multiple hashing and xor operations on the software code, the secret key, and predefined constants to generate a unique digest value. This digest value would change with any modifications of the software or the secret key [41].

HMAC algorithms inherit their strength from the underlying hash algorithm's strength plus the strength provided by the HMAC structure [42]. Therefore, only HMAC algorithms

that use NIST-approved hash functions should be used in secure boot processes. Also, because HMAC functions use variable-length cryptographic keys, the used key derivation and key management mechanisms highly affect the algorithm strength. Therefore, it is recommended to derive a unique HMAC key for each ECU [29] and to only use HMAC keys with equal lengths to the output length [42] because keys with a smaller length than the output length decrease the algorithm strength, and keys with a larger length than the output length do not significantly increase the algorithm strength [43].

The HMAC SHA 256 algorithm uses the SHA 256 hash algorithm to calculate a 256-bit output MAC to be used as a validation value in the secure boot process. In general, the calculation speed of the SHA 256 and the HMAC SHA 256 algorithms are comparable.

HMAC algorithms are used in systems that require integrity and authenticity checks, such as secure booting and secure flashing algorithms. Also, HMAC is widely used in vehicles' secure onboard communication (SecOC) by appending an HMAC value to the transmitted message to be checked by the receiver for integrity and authenticity. One of the main applications of HMAC is the HMAC-based key derivation algorithm (HKDF) [44], which is used at the start of a communication session to derive a variable-length key from a secret input, a random value, and an optional identifier for the key. This key is then used throughout the communication session.

5.3. AES-128 CMAC

CMAC algorithms are MAC generation algorithms that use a sequence of symmetric key block cipher processes with xor operations to output an MAC value that is used as a validation value [45]. The strength of CMAC algorithms lies in the strength of the underlying block cipher algorithm and the appropriate key management mechanisms [46]. Therefore, it is recommended to generate a CMAC key according to the latest specifications and to have a unique CMAC key for each ECU. AES-128 CMAC utilizes the NIST-approved AES block cipher algorithm using a 128-bit key size to output a 128-bit MAC value, which is a secure MAC length against guessing attacks [45].

Typically, CMAC algorithms and HMAC algorithms can be used interchangeably. However, CMAC algorithms are preferred when strong block cipher algorithms are supported by the underlying hardware and strong hash algorithms are not, or when the hardware offers a faster block cipher accelerator that consumes less time than the provided hash accelerators. Also, CMAC algorithms are used in symmetric key diversification algorithms that are used in access control systems [47].

5.4. ECDSA–P25–SHA256

ECDSA is an asymmetric digital signature generation and verification algorithm. Digital signature algorithms use a pair of public and private keys to ensure data authenticity and integrity. A digital signature is generated by using the private key to encrypt a message digest code, which is usually typically created by hashing the data to be signed. The signature can then be verified by decrypting the data using the public key, extracting the message digest code from the decrypted signature, recalculating the data message digest code, and comparing the two digest codes. ECDSA generates and verifies digital signatures by using elliptic curve mathematics, modular arithmetic, a hash function, and a random number [48,49]. The random number, which is generated during the signature generation phase, ensures that a different signature is produced each time for the same data and key.

ECDSA–P256 utilizes the P256 elliptic curve over prime fields [48] to generate a 512-bit signature using a 256-bit private key. The signature can then be verified using a 512-bit public key. Typically, ECDSA and other asymmetric digital signature schemes provide a greater level of security for secure boot processes compared to symmetric MAC algorithms and hash functions. However, this comes at the cost of processing speed, as asymmetric schemes consume more time than their symmetric counterparts.

Digital signatures are the digital correspondence of handwritten signatures. Therefore, the ECDSA algorithm is used in online banking services to avoid identity forging and

guarantee non-repudiation. ECDSA is also used in a variety of different applications, such as communication of wireless sensor networks, authentication of smart cards, handling of cryptocurrencies such as bitcoins, and the validation process of radio frequency identifiers [50].

5.5. RSASSA–PSS 2048–SHA256

RSASSA–PSS is a digital signature generation and verification scheme that uses the RSA to encrypt a digested message, which is resulted from multiple hashing of the data along with a random salt value. The result of the RSA encryption process is a digital signature, which can be verified by decrypting the signature, extracting the hash of the data stored in the signature, and comparing it to the calculated data hash [51]. The use of random salt in the signature generation gives the scheme its probabilistic characteristic.

RSASSA–PSS 2048 uses a 2048-bit modulus to output a 2048-bit signature. Although the length of the key pair in RSA 2048 can vary, the most commonly used lengths are 4096 bits for the private key and 2072 bits for the public key. Key size is the main disadvantage of RSASSA–PSS because algorithms with a smaller key size, such as RSA 256 and RSA 512, do not provide adequate security and should not be used. Despite using a modulus size eight times larger than the modulus size used in the ECDSA P256 algorithm, RSA 2048 still provides less security strength [52].

As RSASSA–PSS is a digital signature scheme, it can be used in the same applications as the ECDSA algorithm but at the expense of increased key size. One of the most common applications of RSASSA–PSS in the automotive field is in secure flashing to authenticate the update code. In secure flashing schemes, the code to be updated is signed using RSASSA–PSS, and the generated signature is appended to the code. The code with the appended signature is then sent to the ECU bootloader, which verifies the signature and flashes the code if the signature is valid.

5.6. ECDSA–P256–AES-128 CMAC

The ECDSA–CMAC algorithm is an ECDSA algorithm that generates its message digest using the CMAC algorithm instead of the standard hash function. The ECDSA–CMAC algorithm is explained in Figure 6, where d is the ECDSA private key, Q is the ECDSA public key, G is the selected curve base point, n is the selected curve prime number, k_{CMAC} is the CMAC private key, and m is the software to be signed. The output signature is the combination of the calculated values r and s, which are integers that range between 1 and $(n - 1)$. Any signature value out of range or non-integer is rejected by the signature verification process.

The ECDSA-CMAC algorithm cannot be used as a digital signature generation and verification algorithm because it requires a private CMAC key in the verification process. This contradicts the objective of digital signature algorithms, which is to enable other parties to verify a specific party's signature without the need for confidential information. This contradiction cannot be solved by making the CMAC key public because it will make the CMAC algorithm more vulnerable. But in the case of secure booting, the ECDSA-CMAC algorithm can be used since secure boot processes require data integrity only and do not involve other parties' authentication. This means that the generated signature is verified by the same party, which is the measurement component of the secure boot process. So, the CMAC key can remain private and is not required to be shared with other parties.

The ECDSA—P256—AES-128 CMAC algorithm uses a 256-bit private ECDSA key, a 128-bit private CMAC key, and a 512-bit public ECDSA key to generate a 512-bit signature. To the best of our knowledge, the usage of CMAC as the underlying algorithm for the ECDSA signature scheme is a unique approach that has not been investigated for applications other than those mentioned in this manuscript.

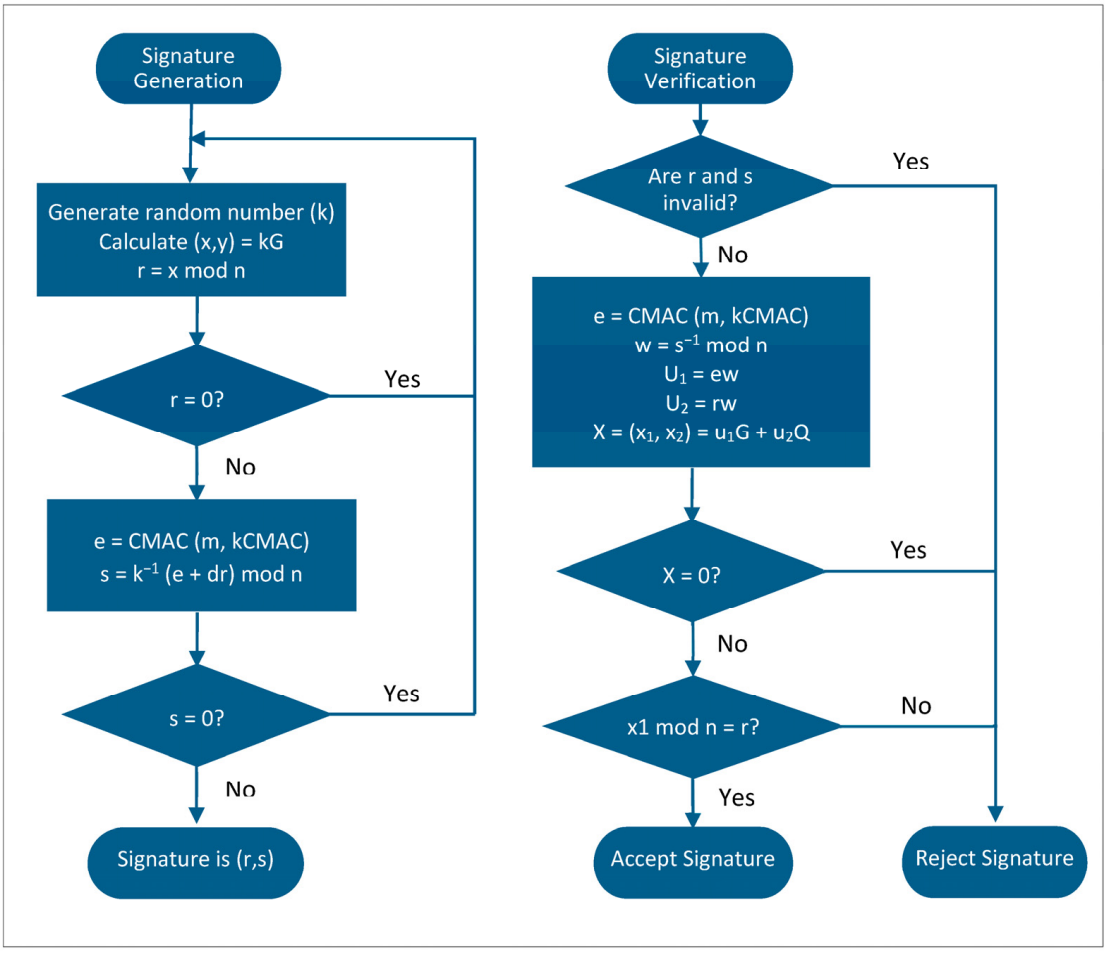

Figure 6. ECDSA-CMAC algorithm flowchart.

6. TC399 32-Bit AURIX™ TriCore™ Microcontroller

The AURIX TC399 is a member of the TC3xx family, which is designed by AURIX specifically for automotive ECUs. The TC3xx family is designed for safety-critical applications with the addition of a separate HSM core for security-critical applications. In addition to the HSM core, the AURIX TC399 contains six TriCore processors called host cores, which are used to execute the ECU application code. Each of the six cores can operate with frequencies up to 300 MHz, has its separate SRAM, and has its separate flash memory that can be used for storing both code and data. Although the six cores can access each other's memory, this access can be restricted by memory protection units (MPUs) because each core has its designated MPU. The MPUs can be configured to protect their cores' memory and registers from being manipulated by other cores.

The AURIX TC399 HSM processor is based on the ARM™ Cortex™ M3 architecture and can operate with frequencies up to 100 MHz. The HSM core was built while applying the EVITA classification [53] to provide the ECU with security schemes and fast cryptographic primitive calculation. The HSM core was designed to be isolated from the application and the host cores to protect the HSM from being compromised due to a compromise in the application code. This is achieved by restricting all communication from the host cores to the HSM core through a special bridge module. This bridge module consists of multiple special function registers that are used to synchronize between the host cores and the HSM core and to transfer data between the two sides. The bridge module also enables each side to send interrupt signals to the other side, which can be enabled to invoke an interrupt service routine. The AURIX TC399 memory is implemented such that the HSM core can have access over the whole memory which cannot be restricted by the MPUs, while

the host cores cannot modify the HSM memory if the HSM memory is configured as being HSM exclusive. This isolation of the HSM enables it to be used for the secure storage of cryptographic secrets and keys, which prevents these secrets and keys from being modified by unauthenticated sources. To further protect the memory from being modified, code memory sections can be configured to be one-time programmable, which ensures that the configured code memory sections are hardware protected against modifications.

The HSM core performs fast cryptographic primitives through its designated hardware accelerators. The AURIX TC399 HSM contains hardware accelerators such as a true random number generator (TRNG); an AES 128 encryption/decryption device; a hash module that performs the message-digest algorithm (MD-5); SHA-1, SHA-224, and SHA-256 hash functions; and a public key cryptography (PKC) module that support asymmetric cryptographic operations that use key sizes up to 256 bits.

As indicated in Figure 7, the AURIX TC399 starts the host core 0 upon booting, and if the HSM core is configured to be enabled, the host core 0 starts the HSM core and waits for the HSM to release it. When the HSM is configured to be enabled, the AURIX TC399 microcontroller detects any attempts to bypass the execution of the HSM core code and locks the microcontroller if an attempt is detected.

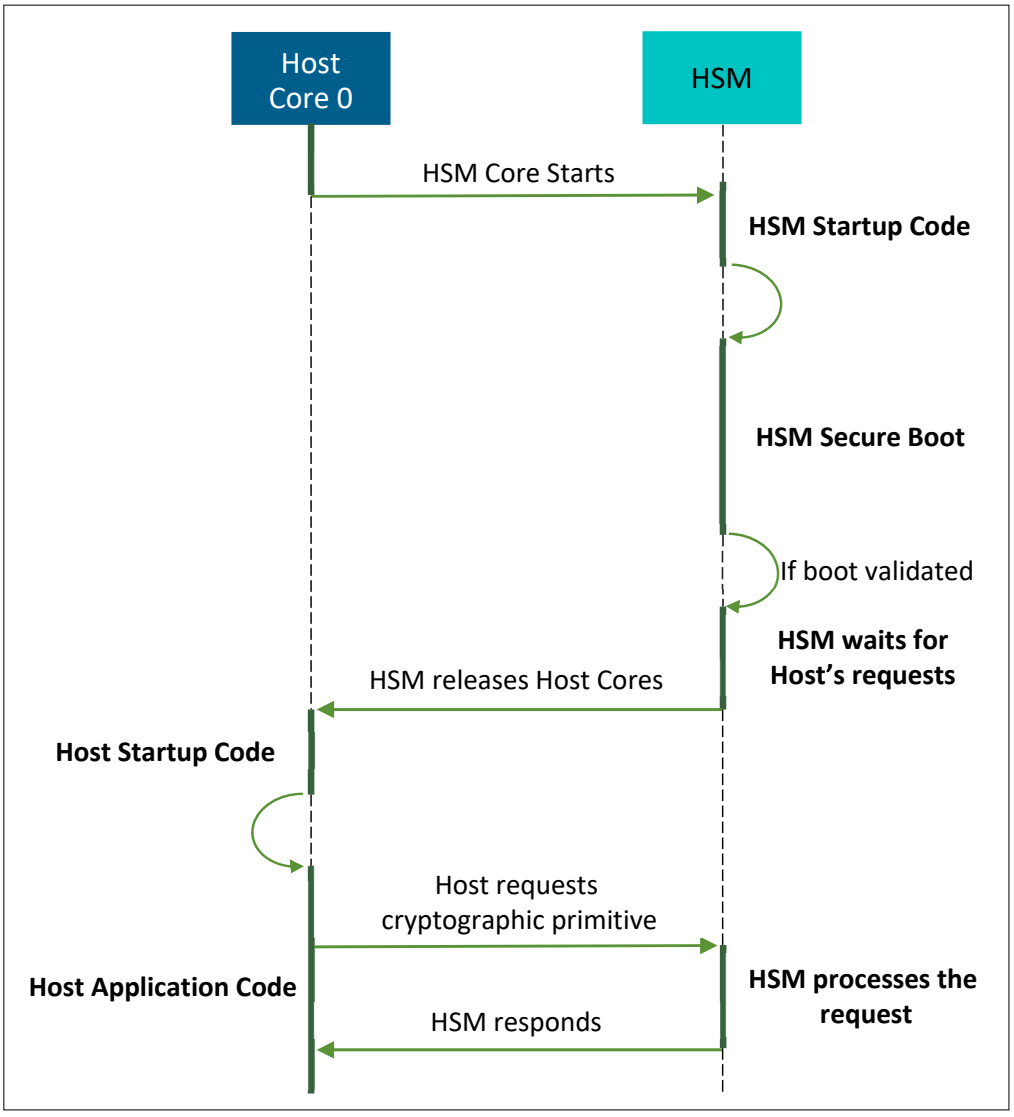

Figure 7. AURIX TC399 boot sequence.

All the prementioned characteristics make the HSM of the AURIX TC399 a suitable root of trust for the secure boot process. Because it is the first software section to be executed by an ECU, and it possesses the hardware accelerators that speed up the measurement of the validation value. Also, the HSM has a hardware-secured storage area to store the reference validation value and the cryptographic keys. The HSM also has the ability to brick the ECU if needed by preventing the release of the host cores and trapping the code executed by the HSM core. Finally, as an extra layer of security, the software responsible for secure booting could be set to be one-time programmable.

7. Tests and Results

A secure boot process of an electric autonomous vehicle's ECU was developed to test the performance of the six prementioned data integrity algorithms using the HSM module of the AURIX TC399. The integrity algorithms selected are the five most common validation algorithms used in secure booting in both the industry and academia, along with the newly introduced CMAC-based ECDSA algorithm. The CMAC-based ECDSA algorithm is a novel integrity algorithm, which was developed to combine the security advantages of the ECDSA and the performance advantages of the CMAC algorithms. The AURIX TC399 target was selected because it is one of the widely used microcontroller targets in the automotive industry.

The used secure boot strategy was a sequential-mode strategy that has a single boot stage. This boot stage is executed by a one-time programmable HSM code that performs the secure boot process to the HSM firmware code and the host application code. The HSM firmware code is responsible for providing different cryptographic primitives to the host core upon the requests of the host application code. The host application code is an AUTOSAR-based code that is executed on the host core 0 and is responsible for executing the functionality needed from the ECU. The host application code defers cryptographic operations to the HSM firmware code, which performs the operations by interfacing with the hardware accelerators and then communicating the result to the host core. Upon startup, the OTP secure boot code executes and validates the HSM firmware code and the host application code. If both codes are valid, the OTP secure boot code releases the host core, allowing it to start executing, and gives control of the HSM core to the HSM firmware code. If one or both validations fail, the OTP secure boot code bricks the ECU and waits for the next reset to retry the validation process. This prevents the execution of any unauthorized code, which could gain access to the CAN bus and jeopardize the whole network. The full secure boot sequence is presented in Figure 8.

Five test cases were performed to test the effectiveness of the proposed secure boot scheme in detecting malicious code injection. The first test case involved running the secure boot process using a valid HSM firmware code and a valid host application code. The second test case was performed after injecting a malicious code into the HSM firmware code, while the third test case was carried out by injecting a malicious code into the host application code. The fourth and fifth test cases were conducted by injecting a single-byte malicious code into the HSM firmware code and the host application code, respectively. The second and third test cases were meant to test major malicious code injection attacks, while the fourth and fifth test cases were meant to test minor malicious code injection attacks.

In the first test case, the software validation process succeeded, and the ECU execution continued normally. In the second to fifth test cases, the secure boot process was able to detect the presence of the malicious code, which caused the validation process to fail. This prevented the malicious code from being executed. The results of the five test cases are summarized in Table 3.

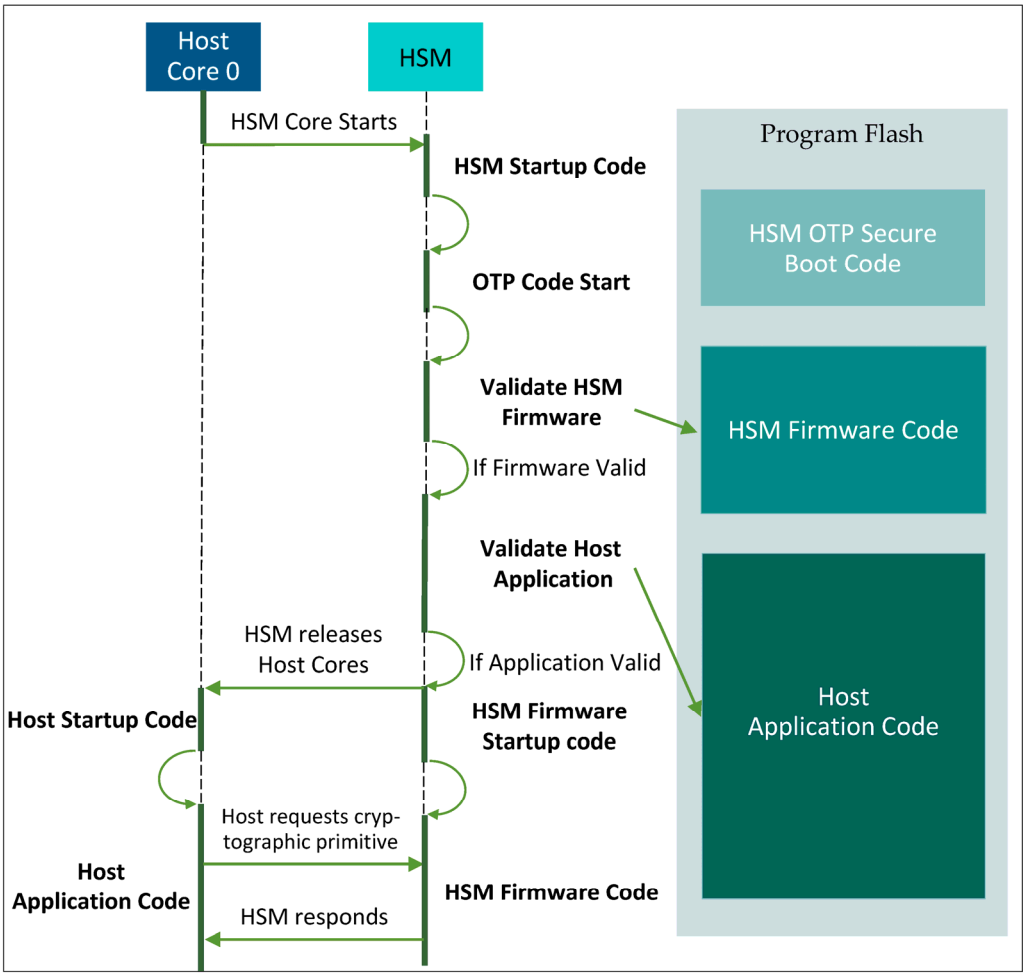

Figure 8. The implemented secure boot strategy.

Table 3. The results of testing the effectiveness of the proposed secure boot scheme.

HSM Firmware Code	Host Application Code	Secure Boot Validation Result
Valid	Valid	Pass
Major Malicious Code	Valid	Fail
Valid	Major Malicious Code	Fail
Minor Malicious Code	Valid	Fail
Valid	Minor Malicious Code	Fail

The OTP secure boot code interfaces with the HSM hardware accelerators to perform the integrity algorithms on the intended software. Six different variants of the secure boot code were implemented on the AURIX TC399 HSM using a productized ready-for-development code. Each variant implements the secure boot process using one of the algorithms discussed in Section 5. The performance of the six variants was tested using the HSM timer module to measure the time each variant took to finish the validation of 147,455 bytes of code. The reason for using a productized ready-for-development code was to ensure that the resulting timing characteristics are approximate to the performance of a deployed code in an industrial vehicle.

The hash secure boot variant and the HMAC secure boot variant rely on the HSM hash accelerator to perform the SHA 256 calculations, while the CMAC variant uses the HSM AES hardware accelerator. The ECDSA variant uses the hardware TRNG to generate the random number required by the algorithm, and the hardware hash module to generate the code image. The code image is then signed or verified by the hardware PKC module. The

ECDSA-CMAC variant is similar to the ECDSA variant in implementation, except that the ECDSA-CMAC variant uses the HSM AES hardware accelerator rather than the hardware hash module to generate the code image. The RSA-PSS variant also uses the TRNG in random number generation and the hardware hash module in code image generation. But because the PKC module supports modulus sizes up to 256 bits and the RSASSA–PSS 2048 uses a 2048-bit modulus, the RSA-PSS variant uses the Mbed TLS Library (Version 2.28) [54] to perform the 2048-bit modular exponentiation operations needed in the RSA encryption and decryption operations. The RSA-PSS variant uses a 2048-bit private exponent and a 24-bit public exponent.

As shown in Table 4, the CMAC variant is the fastest, with almost double the speed of the SHA 256 and the HMAC variants; this is because the AES hardware accelerator in the TC399 is more optimized and, hence, faster than the hash module. The speed of the SHA 256 and the HMAC SHA 256 variants is very similar, as the HMAC algorithm uses the code image generated by the SHA 256 variant and then performs multiple lightweight operations. These lightweight operations are the reason behind the 1.1 (ms/MB) difference between the two variants.

Table 4. The results of secure booting a total of 147,455 bytes of code using different data integrity algorithms that are implemented using the HSM module of the AURIX TC399 microcontroller, with a 100 MHz clock speed.

Data Integrity Algorithm	Secure Boot Time (ms)	Secure Boot Execution Rate (ms/MB)
SHA 256	7.35	52.27
HMAC SHA 256	7.5	53.36
AES 128 CMAC	**3.67**	**26.07**
ECDSA P256 SHA 256	**23.58**	**167.67**
RSASSA–PSS 2048 SHA 256	167.44	1190.8
ECDSA P256 AES 128 CMAC	**19.81**	**140.83**

The performance of the HMAC variant is followed by the performance of the ECDSA variant, which is slower because of the costly elliptic curve operations that are performed by the PKC module. The performance cost of these elliptic curve operations is the cause of the 115.4 (ms/MB) difference between the ECDSA variant and the SHA 256 variant. The slowest variant is the RSA-PSS variant, which was expected to be significantly slower because its modular exponentiation operations are implemented through the software code. Also, as modular exponentiation operations are proportional to the algorithm modulus size, the large 2048-bit modulus of the RSASSA-PSS algorithm further contributes to the slow performance of the algorithm.

The usage of the AES 128 CMAC to generate the code image when using the ECDSA-CMAC variant, instead of the SHA 256 variant, improved the performance of the ECDSA algorithm by **19%**. This is because of the faster AES hardware accelerator of the AURIX TC399 HSM.

The tests confirm that all six variants possess a computational time complexity of O(n), which means the time taken to perform the secure boot process is linearly proportional to the size of the software validated.

Table 5 compares the performance results presented in this paper to the results available in the related literature. It was found that the developed AES CMAC algorithm has a 35% performance gain over the AES CMAC algorithm presented in [32], and a 76% performance gain over the work presented in [55]. Additionally, the developed SHA 256 algorithm is faster by 81% than the SHA 256 algorithm developed in [55].

Table 5. Comparing the implementations of the AES CMAC algorithms and SHA 256 algorithms.

Data Integrity Algorithm	Used Hardware	Secure Boot Execution Rate (ms/MB)
AES 128 CMAC [32]	HSM of AURIX TC27x	40
AES 128 CMAC [55]	HSM of AURIX TC377TP	111.13
AES 128 CMAC (This paper)	HSM of AURIX TC399	26.07
SHA 256 [55]	HSM of AURIX TC377TP	275.71
SHA 256 (this paper)	HSM of AURIX TC399	52.27

8. Conclusions

With the increase in connectivity of autonomous vehicles and their reliance on software for functionality, as well as the rise in inner connectivity between a vehicle's ECUs through the CAN bus, the importance of cybersecurity schemes to counter CAN bus attacks has significantly increased. Due to the limitations of current cryptographic encryption and authentication schemes, and intrusion detection systems, this study proposed a secure boot scheme to prevent CAN bus attacks at the ECU level. The proposed secure boot scheme implements a secure boot process in all ECUs that are connected to the CAN bus to detect and mitigate the presence of unauthorized modification of the ECUs' software. This prevents the access of an ECU with an injected malicious code to the CAN bus. The proposed secure boot scheme is capable of detecting and mitigating major and minor malicious code injection attacks to both host cores and the HSM core. Hence, the proposed secure boot scheme is able to prevent all CAN bus attacks that may result from a malicious code.

This study compared various secure boot strategies, outlining the advantages and disadvantages of each. Additionally, this study analyzed the performance of six data integrity schemes on the secure boot process of the TC399 32-bit AURIX™ TriCore™ microcontroller. The test results indicate that the recommended data integrity algorithms for secure boot processes of the AURIX TC399 are (1) the AES 128 CMAC algorithm implemented using the HSM AES encryption/decryption device because it is the fastest algorithm with an execution rate of 26.07 (ms/MB), and it provides comparable strength to the HMAC or SHA 256 algorithms; (2) the ECDSA P256 SHA 256, which uses the HSM true number generator, hash module, and PKC module, because it provides higher security level than the CMAC algorithm without a large speed cost, such as the speed cost of the RSASSA PSS SHA 256; and (3) the ECDSA P256 AES 128 CMAC, which is a novel approach that utilizes the AES 128 CMAC instead of the SHA 256 in code image generation. This approach combines the strength advantage of the ECDSA algorithm and the speed advantage of the CMAC algorithm, resulting in a 19% performance gain over to the standard hash-based ECDSA.

The limitations of the proposed secure boot scheme are that it cannot detect the presence of a malicious ECU added to the network. As a result, it cannot mitigate attacks that result from this path. Also, the secure boot scheme has a large cost of implementation because all ECUs connected to the CAN bus must perform a secure boot process for the scheme to be effective.

The planned future work aims to address the limitations of the secure boot scheme by exploring detection systems that can identify and mitigate malicious ECUs that are connected to the network. Sucha detection system could be integrated with the secure boot scheme to protect vehicles from the two possible attack paths.

Author Contributions: Conceptualization, S.A., A.M. and S.A.M.; methodology, S.A. and A.M.; software, S.A.; validation, S.A.; formal analysis, S.A.; investigation, S.A. and A.M.; writing—original draft preparation, S.A.; writing—review and editing, S.A., A.M., S.A.M. and S.H.; supervision, A.M., S.A.M. and S.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The raw data used in this study are not publicly available.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Currie, R. *Developments in Car Hacking*; Tech. Rep., Dec.; SANS Institute: Bethesda, MD, USA, 2015; pp. 1–33.
2. Aliwa, E.; Rana, O.; Perera, C.; Burnap, P. Cyberattacks and Countermeasures for In-Vehicle Networks. *ACM Comput. Surv.* **2021**, *54*, 1–37. [CrossRef]
3. Khan, J. Vehicle network security testing. In Proceedings of the 2017 3rd IEEE International Conference on Sensing, Signal Processing and Security, ICSSS 2017, Chennai, India, 4–5 May 2017. [CrossRef]
4. Hoppe, T.; Kiltz, S.; Dittmann, J. Security threats to automotive CAN Networks—Practical examples and selected short-term countermeasures. In *Lecture Notes in Computer Science, Proceedings of the International Conference on Computer Safety, Reliability, and Security, Newcastle upon Tyne, UK, 22–25 September 2008*; Springer: Berlin/Heidelberg, Germany, 2008. [CrossRef]
5. Parameswaran, S.; Wolf, T. Embedded systems security-an overview. *Des. Autom. Embed. Syst.* **2008**, *12*, 173–183. [CrossRef]
6. Hoppe, T.; Kiltz, S.; Lang, A.; Dittmann, J. Exemplary automotive attack scenarios: Trojan horses for Electronic Throttle Control System (ETC) and replay attacks on the power window system. In Proceedings of the VDI/VW Gemeinschaftstagung Automotive Security, Wolfsburg, Germany, 27–28 November 2007; VDI-Verlag: Wolfsburg, Germany, 2007; pp. 165–183, ISBN 978-3-18-092016-0.
7. Checkoway, S.; McCoy, D.; Kantor, B.; Anderson, D.; Shacham, H.; Savage, S. Comprehensive experimental analyses of automotive attack surfaces. In Proceedings of the 20th USENIX Security Symposium, San Francisco, CA, USA, 8–12 August 2011.
8. Huang, T.; Zhou, J.; Wang, Y.; Cheng, A. On the security of in-vehicle hybrid network: Status and challenges. In *Lecture Notes in Computer Science, Proceedings of the International Conference on Information Security Practice and Experience, Melbourne, VIC, Australia, 13–15 December 2017*; Springer: Cham, Switzerland, 2017. [CrossRef]
9. Iehira, K.; Inoue, H.; Ishida, K. Spoofing attack using bus-off attacks against a specific ECU of the CAN bus. In Proceedings of the CCNC 2018—2018 15th IEEE Annual Consumer Communications and Networking Conference, Las Vegas, NV, USA, 12–15 January 2018. [CrossRef]
10. Miller, C.; Valasek, C. Remote Exploitation of an Unaltered Passenger Vehicle. *Black Hat USA* **2015**, *2015*, 1–91.
11. Pham, M.; Xiong, K. A survey on security attacks and defense techniques for connected and autonomous vehicles. *Comput. Secur.* **2021**, *109*, 102269. [CrossRef]
12. Siddiqui, A.S.; Plusquellic, Y.G.J.; Saqib, F. Secure communication over CANBus. In Proceedings of the 2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS), Boston, MA, USA, 6–9 August 2017; pp. 1264–1267. [CrossRef]
13. Nilsson, D.K.; Larson, U.E.; Jonsson, E. Efficient in-vehicle delayed data authentication based on compound message authentication codes. In Proceedings of the IEEE Vehicular Technology Conference, Marina Bay, Singapore, 11–14 May 2008. [CrossRef]
14. SyedNavaz, A.S.; Sangeetha, V.; Prabhadevi, C. Entropy based Anomaly Detection System to Prevent DDoS Attacks in Cloud. *Int. J. Comput. Appl. (0975-8887)* **2013**, *62*, 42–47. [CrossRef]
15. Choi, W.; Jo, H.J.; Woo, S.; Chun, J.Y.; Park, J.; Lee, D.H. Identifying ECUs Using Inimitable Characteristics of Signals in Controller Area Networks. *IEEE Trans. Veh. Technol.* **2018**, *67*, 4757–4770. [CrossRef]
16. Javed, A.R.; Rehman, S.U.; Khan, M.U.; Alazab, M.; Reddy, T.G. CANintelliIDS: Detecting In-Vehicle Intrusion Attacks on a Controller Area Network Using CNN and Attention-Based GRU. *IEEE Trans. Netw. Sci. Eng.* **2021**, *8*, 1456–1466. [CrossRef]
17. Agrawal, K.; Alladi, T.; Agrawal, A.; Chamola, V.; Benslimane, A. NovelADS: A Novel Anomaly Detection System for Intra-Vehicular Networks. *IEEE Trans. Intell. Transp. Syst.* **2022**, *23*, 22596–22606. [CrossRef]
18. Dong, C.; Wu, H.; Li, Q. Multiple Observation HMM-Based CAN Bus Intrusion Detection System for In-Vehicle Network. *IEEE Access* **2023**, *11*, 35639–35648. [CrossRef]
19. Wei, P.; Wang, B.; Dai, X.; Li, L.; He, F. A novel intrusion detection model for the CAN bus packet of in-vehicle network based on attention mechanism and autoencoder. *Digit. Commun. Netw.* **2023**, *9*, 14–21. [CrossRef]
20. Yu, T.; Hua, G.; Wang, H.; Yang, J.; Hu, J. Federated-LSTM based Network Intrusion Detection Method for Intelligent Connected Vehicles. In Proceedings of the IEEE International Conference on Communications, Seoul, Republic of Korea, 16–20 May 2022. [CrossRef]
21. Taslimasa, H.; Dadkhah, S.; Neto, E.C.P.; Xiong, P.; Iqbal, S.; Ray, S.; Ghorbani, A.A. ImageFed: Practical Privacy Preserving Intrusion Detection System for In-Vehicle CAN Bus Protocol. In Proceedings of the 2023 IEEE 9th International Conference on Big Data Security on Cloud, IEEE International Conference on High Performance and Smart Computing, and IEEE International Conference on Intelligent Data and Security, BigDataSecurity-HPSC-IDS 2023, New York, NY, USA, 6–8 May 2023; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2023; pp. 122–129. [CrossRef]
22. Zhang, H.; Pan, Y.; Lu, Z.; Wang, J.; Liu, Z. A Cyber Security Evaluation Framework for In-Vehicle Electrical Control Units. *IEEE Access* **2021**, *9*, 149690–149706. [CrossRef]

23. Tyger, J.D.; Yee, B. *Dyad: A System for Using Physically Secure Coprocessors*; Technical Report CMU-CS-91-140R; School of Computer Science, Carnegie Mellon University: Pittsburgh, PA, USA, 1991.
24. Profentzas, C.; Gunes, M.; Nikolakopoulos, Y.; Landsiedel, O.; Almgren, M. Performance of secure boot in embedded systems. In Proceedings of the 15th Annual International Conference on Distributed Computing in Sensor Systems, DCOSS 2019, Santorini Island, Greece, 29–31 May 2019. [CrossRef]
25. Groll, A.; Holle, J.; Wolf, M.; Wollinger, T. Next generation of automotive security: Secure hardware and secure open platforms. In Proceedings of the 17th ITS World Congress, Busan, Republic of Korea, 25–29 October 2010.
26. Nasahl, P.; Timmers, N. *Attacking AUTOSAR using Software and Hardware Attacks*; Escar: St. Johns, MI, USA, 2019.
27. Lee, K.S. Considerations for Cyber Security Implementation in Autonomous Vehicle Systems. In Proceedings of the International Conference on Control, Automation and Systems, Jeju, Republic of Korea, 12–15 October 2021. [CrossRef]
28. Li, Y.; Zhang, Y.; Li, P.; Guo, P. An efficient trusted chain model for real-time embedded systems. In Proceedings of the 2015 11th International Conference on Computational Intelligence and Security, CIS 2015, Shenzhen, China, 19–20 December 2015; Institute of Electrical and Electronics Engineers Inc.: Piscataway, NJ, USA, 2016; pp. 428–432. [CrossRef]
29. Sanwald, S.; Kaneti, L.; Stöttinger, M.; Böhner, M. Secure Boot Revisited: Challenges for Secure Implementations in the Automotive Domain. *SAE Int. J. Transp. Cyber. Priv.* **2019**, *2*, 69–81. [CrossRef]
30. Löhr, H.; Sadeghi, A.R.; Winandy, M. Patterns for secure boot and secure storage in computer systems. In Proceedings of the ARES 2010—5th International Conference on Availability, Reliability, and Security, Krakow, Poland, 15–18 February 2010. [CrossRef]
31. Matheus, K.; Konigseder, T. *Automotive Ethernet*; Cambridge University Press: Cambridge, UK, 2017.
32. Kim, D.; Shin, E.; Park, J.S.; Lee, K.; Gui, K.C.; Scheibert, K. *Secure Boot Implementation for Hard Real-Time Powertrain System*; SAE Technical Paper; SAE: Warrendale, PA, USA, 2017. [CrossRef]
33. Mihalache, A.; Bedoucha, F. Need to Take Cybersecurity into Account in Secure ECUs. [Online]. 2020. Available online: https://hal.archives-ouvertes.fr/hal-03331273 (accessed on 11 February 2023).
34. Gui, Y.; Siddiqui, A.S.; Saqib, F. Hardware Based Root of Trust for Electronic Control Units. In Proceedings of the Conference Proceedings—IEEE SOUTHEASTCON, SoutheastCon 2018, St. Petersburg, FL, USA, 19–22 April 2018. [CrossRef]
35. Kumar, V.B.Y.; Gupta, N.; Chattopadhyay, A.; Kasper, M.; Kraus, C.; Niederhagen, R. Post-Quantum Secure Boot. In Proceedings of the 2020 Design, Automation and Test in Europe Conference and Exhibition, DATE 2020, Grenoble, France, 9–13 March 2020. [CrossRef]
36. Marzougui, S.; Krämer, J. Post-quantum cryptography in embedded systems. In Proceedings of the ACM International Conference Proceeding Series, Canterbury, UK, 26–29 August 2019. [CrossRef]
37. Wang, W.; Han, Z.; Alazab, M.; Gadekallu, T.R.; Zhou, X.; Su, C. Ultra Super Fast Authentication Protocol for Electric Vehicle Charging Using Extended Chaotic Maps. *IEEE Trans. Ind. Appl.* **2022**, *58*, 5616–5623. [CrossRef]
38. Dworkin, M. Hash Functions. Available online: https://csrc.nist.gov/projects/hash-functions (accessed on 19 December 2022).
39. *FIPS PUB 180-4*; Secure Hash Standard. National Institute of Standards and Technology: Gaithersburg, MA, USA, 2015.
40. Sobti, R.; Geetha, G. Cryptographic Hash Functions: A Review. *Int. J. Comput. Sci. Issues* **2012**, *9*, 461–479.
41. *FIPS PUB 198-1*; The Keyed-Hash Message Authentication Code. National Institute of Standards and Technology: Gaithersburg, MA, USA, 2008.
42. Kelly, S.; Frankel, S.; Networks, A. Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec. *Angew. Chem. Int. Ed.* **2007**, *6*, 951–952.
43. Krawczyk, H.; Bellare, M.; Canetti, R. HMAC: Keyed-Hashing for Message Authentication; RFC 2104, February 1997. Available online: https://www.rfc-editor.org/info/rfc2104 (accessed on 12 March 2023). [CrossRef]
44. Akevren, S.; Krawczyk, D.H.; Eronen, P. HMAC-Based Extract-and-Expand Key Derivation Function (HKDF). Internet Engineering Task Force, IETF, RFC 5869. May 2010. Available online: https://rfc-editor.org/rfc/rfc5869.txt (accessed on 21 July 2023).
45. Dworkin, M. Recommendation for Block Cipher Modes of Operation. National Institute of Standards and Technology Special Publication 800-38A 2001 ED, vol. X, no. December 2005. Available online: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38a.pdf (accessed on 12 March 2023).
46. Song, J.H.; Poovendran, R.; Lee, J.I.T. RFC 4493: The AES-CMAC Algorithm. Network Working Group. 2006. Available online: https://tools.ietf.org/html/rfc4493 (accessed on 12 March 2023).
47. Semiconductors, N.X. AN10922 Symmetric key diversifications. July 2019. Available online: https://www.nxp.com/docs/en/application-note/AN10922.pdf (accessed on 21 July 2023).
48. Gallagher, P.D.; Romine, C. FIPS PUB 186-4 Digital Signature Standard (DSS). In *Encyclopedia of Cryptography and Security*; The Information Technology Laboratory, National Institute of Standards and Technology: Gaithersburg, MD, USA, 2013. Available online: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf (accessed on 12 March 2023).
49. Pornin, T. RFC 6979: Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). RFC. August 2013. Available online: https://datatracker.ietf.org/doc/html/rfc6979 (accessed on 21 July 2023).
50. Al-Zubaidie, M.; Zhang, Z.; Zhang, J. Efficient and secure ECDSA algorithm and its applications: A survey. *Int. J. Commun. Netw. Inf. Secur.* **2019**, *11*, 7–35. [CrossRef]
51. Moriarty, K.; Kaliski, B.; Jonsson, J.; Rusch, A. PKCS #1: RSA Cryptography Specifications, Version 2.2. RFC 8017. November 2016. Available online: https://www.rfc-editor.org/rfc/rfc8017 (accessed on 12 March 2023).

52. Barker, E.; Barker, W.; Burr, W.; Polk, W.; Smid, M. *Recommendation for Key Management SP 800-57 Part 1: General Revision 3*; NIST Special Publication: Gaithersburg, MD, USA, 2007; Volume 800. [CrossRef]
53. Pott, C.; Jungklass, P.; Csejka, D.J.; Eisenbarth, T.; Siebert, M. Firmware Security Module. *J. Hardw. Syst. Secur.* **2021**, *5*, 103–113. [CrossRef]
54. Rodgman, D. Mbed TLS 2.28.0. Available online: https://github.com/Mbed-TLS/mbedtls/releases/tag/v2.28.0 (accessed on 17 July 2022).
55. Reier, L. Security Concept and Evaluation of an Off-Highway Electronic Control Unit. Master's Thesis, Technische Universität Wien, Vienna, Austria, 2020.