TEECheck: Securing Intra-Vehicular Communication Using Trusted Execution

Tanmaya Mishra
tanmayam@vt.edu
Virginia Tech
Arlington, Virginia

Thidapat Chantem
tchantem@vt.edu
Virginia Tech
Arlington, Virginia

Ryan Gerdes
rgerdes@vt.edu
Virginia Tech
Arlington, Virginia

ABSTRACT

Modern vehicles have a large number of advanced driver assistance systems (e.g., adaptive cruise control and automatic lane keeping) that depend on the timely availability of data exchanged through the CAN bus from a variety of ECUs and sensors. Considering the large amount of data on the CAN bus, CAN has become a lucrative target for malicious parties wishing to take control of a vehicle. Specifically, an attacker may compromise an ECU to gain access to the bus. It is, thus, imperative that the CAN bus is secured from disruption by compromised ECUs to ensure its correct and timely operation. We design a new system architecture for ECUs that leverages trusted execution environments and propose TEECheck, an on-device message vetting mechanism to proactively contain masquerade and denial-of-service attacks, as well as eliminate information leakage. The uniqueness of our approach is that it require neither authentication at the receiver end adds any overhead if there is no need for CAN bus access. Experiments show that our technique has very low and predictable overhead, regardless of the workload.

CCS CONCEPTS

• **Computer systems organization → Real-time operating systems**; • **Security and privacy → Embedded systems security**; **Hardware-based security protocols**.

ACM Reference Format:
Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. 2020. TEECheck: Securing Intra-Vehicular Communication Using Trusted Execution. In *28th International Conference on Real-Time Networks and Systems (RTNS 2020), June 9–10, 2020, Paris, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3394810.3394822

1 INTRODUCTION

Today's vehicles are complex machines. While they still have the same basic design of the internal combustion engine and transmission as vehicles from decades ago, they now have sophisticated, highly automated features such as advanced fuel injection systems, hybrid drivetrains, traction control, adaptive cruise control, and automatic lane-keeping, all of which are supported by sensor data and processing units. In addition, vehicles have become even more like regular computing systems, with (1) remote software updates that improve performance, or (2) having a certain degree of autonomy, allowing it to drive itself for short distances. To support these functionalities, vehicles must now generate, process and act upon a large amount of information to make driving safer, more comfortable, and more efficient.

Vehicular control systems are distributed throughout the vehicle, with some located physically close to the sensors and actuators with which they interact. While there have been proposals to consolidate the various components into a centralized system [3, 33] that controls every aspect of the vehicle, most vehicles still utilize separate electronic control units (ECUs) that are dedicated to specific functions. Modern vehicles have upwards of 100 different ECUs and this number is constantly increasing as vehicle manufacturers add functionality. While one ECU may control the engine, others may control the vehicle entertainment system, dashboard information, brakes, fuel system, etc. Advanced driver assistance mechanisms act upon information, in real-time, from many of these ECUs simultaneously. For example, certain luxury modern vehicles have crosswind stabilization which adjusts the vehicle braking characteristics under strong crosswinds. To do so, information from sensors measuring wind speed, steering position and characteristics (steering column ECU), and vehicle speed (engine control unit), among others, is processed in real-time to provide safe braking assistance.

To achieve timely sharing of vehicle runtime information between ECUs while reducing the size, weight, and power constraints (SWaP) and manufacturing cost, modern cars utilize a shared bus system for inter-ECU communications. In this paper, we consider the CAN bus, an industry standard protocol for intra-vehicular networks. There is a large body of work in both academia and industry, along with the millions of cars that utilize it, which show that CAN is an efficient and robust communication network. While the older CAN protocol may not be sufficient for the autonomous vehicles of the near future, a newer variant (CAN-FD [17]) has been designed to increase the longevity of the protocol. However, with the increasing amount of data being shared over the bus, it becomes critical to develop techniques to not only maintain bandwidth availability but also the real-time nature of message transmission.

In addition to efficiency and timeliness, security has become an important consideration. As vehicles increasingly make decisions autonomously to ensure passenger comfort and safety, it becomes imperative that their operations are not disrupted. Due to the critical role of the CAN bus in facilitating and maintaining safe and reliable vehicle operation, CAN may draw heavy interest from malicious actors who wish to take control of a vehicle or change operational characteristics to make it unsafe for its riders. In fact, prior

RTNS 2020, June 9–10, 2020, Paris, France
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7593-1/20/06…$15.00
https://doi.org/10.1145/3394810.3394822

work [8, 18] has already shown various attacks that can take place on a vehicle, allowing malicious parties to perform actions such as applying brakes and causing a crash. For instance, newer vehicles have additional communication interfaces such as WiFi, Bluetooth and cellular connectivity to connect to external servers for software updates or for passenger convenience. These are usually provided by ECUs that are often directly connected to the vehicle's CAN bus. Such external interfaces may be vulnerable to exploitation and become a gateway to access the vehicle. For example, it has been shown that the Bluetooth stack in the car infotainment unit has vulnerabilities that when exploited, allow attackers to run arbitrary code on an ECU [8]. The attacker could then use this compromised ECU to perform masquerade attacks [6] where the attacker poses as a legitimate entity and sends out spoofed messages that could affect and/or control critical parts of the vehicle, e.g., sending messages to cut off fuel supply to the engine, or launch denial-of-service (DoS) attacks by flooding the bus with garbage messages. This is possible since CAN is a broadcast bus without message authentication. If such attacks are carried out in real-world scenarios, such as in high-speed traffic, it could have catastrophic consequences.

To maintain predictable and timely operation on the CAN bus, we believe that the best form of defense would be at the source itself. That is, an attacker that *fails to utilize* the bus is less effective than one which is given access to the bus and may have the ability to disrupt message transmissions. This paper builds upon this core idea and proposes a lightweight technique to secure the CAN bus from attacks such that the compromised ECU cannot engage the bus more than the original (uncompromised) ECU.

To do so, we propose that ECUs utilize trusted execution environment (TEE)-capable processors which allow code compartmentalization, making it possible to verify CAN message source and destination within an ECU itself instead of at the receiver ECU. TEEs are isolated execution environments designed to run trusted software and are supported by processor architecture extensions which include strict access control policies to processor components, peripherals, data, and address buses. TEE implementations such as Intel SGX [11] and ARM TrustZone [4] can currently be found in commercially available hardware and have been used in a variety of security-critical applications, such as Samsung Pay [5]. In addition, TEE is a known entity, as there exists a large body of work [27] which studies the security benefits and pitfalls of TEEs. Although our approach requires changes to the critical hardware components inside a vehicle, we believe it to be an especially effective one, both in terms of performance and security. In fact, our approach incurs *no* increase in CAN bus bandwidth consumption and we observe substantial performance improvements over currently available approaches while running on much slower (12 MHz clock instead of 100+ MHz clock for typical ECUs) hardware. The shift towards using CAN-FD over CAN in recent vehicles [2] shows that the automotive industry is willing to utilize newer technologies when significant advantages are demonstrated. This paper has the following major contributions:

(1) We propose a TEE-based ECU system architecture to separate message generation and consumption from CAN transmission and reception.
(2) Based on our new system architecture, we present **TEECheck**, an intermediary CAN bus interface which achieves efficient and trustworthy vetting of message origin and frequency before message transmission, and of request origin before received message data is disbursed. In particular, TEECheck:
   (a) Detects and prevents an attacker from *masquerading* as another legitimate message source such as other tasks on the same *or* different ECU.
   (b) Provides a *proactive* and *on-ECU* mechanism to mitigate DoS attacks on the CAN bus. Our technique is the first to utilize TEE to contain the aforementioned attacks to the compromised ECU and does *not* require a receiver ECU to validate whether a message is from a legitimate party.
   (c) Provides an *on-device* mechanism to prevent an attacker from *any* access to messages not intended for it, i.e., prevents snooping. To the best of our knowledge, there are no other techniques designed for the CAN bus that prevents the attacker from accessing the CAN message frames meant for other endpoints.
(3) We experimentally show that the overhead associated with our approach, which is incurred *only* when a task requires access to the CAN bus, is fairly negligible and quite predictable, making our approach suitable for resource-constrained devices running real-time applications. Specifically, the overhead typically takes 477 us from message generation to message transmission, and 480 us for message data reception, on a 12 MHz processor which would translate to 50 µs on a 100+ MHz processor.

While we consider CAN 2.0 as our target application in this paper, our mechanism can be easily adapted to any broadcast communication mechanism, such as I2C or SPI, with minimal changes. In terms of TEE implementation, we selected the ARM TrustZone for Cortex-M (ARMv8-M architecture with Security Extensions) [36] which, although fairly recently introduced, has a number of commercial-off-the-shelf microcontroller implementations available that can be used in modern ECUs today as drop-in replacements.

2 RELATED WORK

CAN bus security has become an important research area in the past several years. Considering the safety-critical nature of the systems where CAN is utilized, e.g., automobiles, this is not surprising. CAN bus hardening approaches are spread across the communication stack layers. At the physical layer, intrusion detection systems (IDS), have been introduced. These schemes are variants of clock-skew and voltage-based fingerprinting which help to detect and, in some cases, identify attacker ECUs. While such techniques can detect an attacker within a few frames [9, 10, 35] or single frame [13], they still, in general, require a specialized monitoring node for detection and are *reactive* in nature. That is, they only detect when the attacker has managed to engage the bus. Our aim is to limit the attacker to the compromised ECU itself.

Network-level authentication techniques for CAN bus have also been widely studied. LeiA [26] and vatiCAN [25] are AUTOSAR [14] compliant authentication schemes which utilize some form of MAC based authentication at the receiver end. To compensate for the increased overhead, others such as CANAuth [32] and LiBrA-CAN [15]

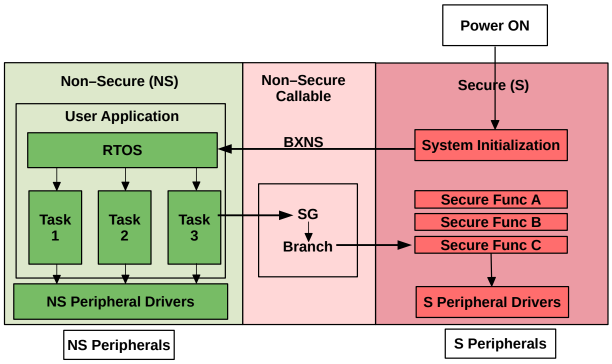


Figure 1: ARMv8-M microcontroller power ON code flow

are based on variants of CAN+ [37], a backward-compatible variant of CAN capable of higher data rates, or CAN-FD-Sec [7] based on the CAN-FD [17]. Unlike CAN+, CAN-FD is expected to be the next-generation replacement for CAN 2.0. All these works require receiver-end authentication, which incurs unnecessary bus overhead since illegitimate messages must still be transmitted before they can be authenticated. In addition, these work cannot prevent DoS attacks since receiver-end authentication cannot stop a transmitter from sending out a message. In contrast, we limit the frequency of message transmission of the compromised ECU. Further, we aim to be CAN variant agnostic. Our other goal is to propose a lightweight hardware based approach which requires minimal changes to the ECU software. In contrast, Berg et al. [7] considers separating the infotainment system from the rest of the vehicle by implementing secure gateways, which requires significant software addition and can incur large time overhead.

The concept of trust has been used to fortify CAN bus communication in prior work. VeCure [34] uses the concept of trust groups where ECUs handling critical operations are kept in the higher trust group, authenticating each other using a MAC based scheme. The work by Gui et al. work [16] is more similar to ours in that it utilizes hardware trusted platform module (TPM) for establishing a root of trust. Perhaps the closest work, in spirit, to ours is VulCAN [31] where the authors utilize trusted execution. Their technique builds upon LeiA and vatiCAN by utilizing a trusted computing base (TCB, in their case, Sancus [24]) inside which they generate their MACs. VulCAN takes much longer, i.e, around 2 ms for the entire authentication sequence. In addition, addressing DoS attacks are outside the scope of all of these techniques since they depend on the receiver for attack detection.

3 PRELIMINARIES

We now provide an overview of the underlying technologies of our system. Specifically, we consider the ARM TrustZone for ARMv8-M based microcontrollers and the CAN protocol. Due to the page limit, we only provide the details that are relevant to our work. For more details, readers are referred to existing publications [12, 36].

3.1 ARM TrustZone for Cortex-M

ARM TrustZone for Cortex-M [36] (based on ARMv8-M architecture) is a variant of the TrustZone technology first introduced in ARM's Cortex-A processors. ARM TrustZone is a set of processor architecture extensions which allow creating TEEs via software. It divides the processor execution into two domains, **secure** and **non-secure**. Code running in the secure domain has access to information from both domains while code running in the non-secure domain has access to information only from the non-secure domain. While TrustZone for Cortex-A is complex and has significant overheads [23], TrustZone for Cortex-M is designed to be very lightweight. To reduce the overhead for a low-powered microcontroller to switch between the two states, TrustZone for ARMv8-M utilizes a near static memory-mapped mechanism for delineating the domains. The TrustZone divides the memory space such that certain addresses are made available only to the secure domain. This is facilitated through a hard-wired controller logic called the implementation defined attribution unit (IDAU). The IDAU creates a striated memory partitioning scheme such that it is easy to identify to which domain an address belongs. Specifically, if the 29th bit of the memory address is 0, it is a secure domain address. Additionally, certain sections of the non-secure domain can be upgraded to the secure domain through software using the security attribution unit (SAU). Depending on the implementation, peripherals are memory-mapped into both secure and/or non-secure memory locations. The non-secure peripheral locations are enabled via a peripheral access controller (PAC) or security control unit (SCU). The SAU, PAC, and SCU are themselves mapped to secure locations by the IDAU, making it impossible for non-secure code to access or modify them, unless TrustZone is broken.

The secure code memory location is further divided into secure (S) and non-secure Callable (NSC) locations. While the former cannot be accessed by any code running in the non-secure domain (or it would cause the system to generate a hard fault likely requiring human intervention), the NSC locations provide an intermediary jump point where the secure gateway (SG) instruction is kept which switches the processor mode to secure when executed. All calls into the secure side have the interface function defined in the NSC. From an execution point of view, both domains have a *Thread* and *Handler* mode for regular and interrupt code executions, respectively. If an interrupt is generated from the non-secure side and the secure code is currently executing, all information is pushed to the secure stack and registers are cleared before the switch to the non-secure interrupt handler. The same set of steps happen when the situation is reversed. Based on our experiments (Section 7), the fact that both domains have the same execution flow and capabilities in ARMv8-M allows for consistently low interrupt latency regardless of the domain from which the interrupt originates (4 us overhead for switching in our case).

Figure 1 shows the flow of code execution once an ARMv8-M controller is powered on. Code execution for ARMv8-M processors begins in the secure domain, which then branches into the non-secure domain. The bulk of the application code is written to run in the non-secure domain, including a real-time operating system (RTOS), task code and peripheral drivers in our case[1]. When

[1] While it is possible to run all the code inside the secure domain (within space limitations), it is notoriously difficult to produce bug-free code on a larger scale [30]. An attacker with knowledge of vulnerabilities in the secure code could compromise the entire system since the secure code has access to the entire memory space.

required, the application task code makes calls to the secure code via the intermediary functions present in the NSC.

It should be noted that the secure and non-secure domains are orthogonal to the regular processor privilege levels. Within each domain, the processor still executes under the traditional privilege model, where interrupts and the RTOS may run in privileged processor execution mode while task code may run under unprivileged processor execution mode. Further, there may be a shared or separate memory protection unit (MPU) for the secure and non-secure domains. An MPU is accessible from the privileged execution mode and enforces fine-grained access rights to certain memory locations for privileged and unprivileged code. Privileged code can access memory locations not specified in the MPU table, while access from unprivileged code would generate a fault. It must be noted the SAU has a very similar operation to that of the traditional microcontroller MPU but they are separate entities that can work together. Specifically, the SAU is used to augment the partitioning of the memory space into secure and non-secure domains over and above the fixed partitioning scheme provided by the IDAU while the MPU works within this partitioned memory space to provide different access right to different pieces of code. For example, an RTOS can load task-specific access rights before context switching to a task. Our solution uses a combination of the MPU and careful partitioning of resources using the TrustZone.

3.2 Controller Area Network (CAN)

CAN is a protocol originally designed for communication between different vehicle components but has been applied to other areas such as industrial automation. It is a serial communication protocol designed to broadcast small messages over a shared bus. CAN uses a multi-master communication paradigm where nodes compete, on a per-message basis, to send messages on the bus and it is upto each node to accept or ignore the messages.

Currently, vehicles utilize the CAN 2.0 version which supports bitrates up to 1 Mbits/sec, although there has been ongoing effort to introduce the newer CAN-FD (CAN flexible data rate) [17] variant which allows larger message data (64 bytes instead of 8 bytes) and higher bitrates of up to 5 Mbits/sec. The CAN 2.0b standard frame format is shown in Figure 2. CAN supports different types of frame formats, including a DATA frame which contains the actual payload for communication between nodes, a REMOTE frame which is sent from a node requesting data from another node, an ERROR frame which is sent from a node when it detects an error on the bus, and an OVERLOAD frame to provide an additional delay between successive DATA or REMOTE frames. While we concern ourselves, in this work, with the DATA frame format since it contains the actual data being transmitted on the bus, our solution can be extended to consider REMOTE frames with minimal modifications.

Bus contention is resolved using the arbitration field. DATA frames may have different arbitration field lengths–11 bits for the *Standard Frame* (shown in Figure 2), and 29 bits for the *Extended Frame* formats (extended frame format is only valid for DATA or REMOTE frames). These arbitration bits constitute the message IDENTIFIER and is used by receiver nodes to identify if the message pertains to them. All nodes which wish to transmit sense the bus for any ongoing transmission and back off when they detect one. When

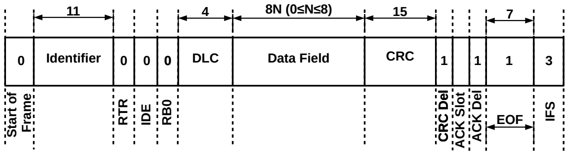


Figure 2: CAN 2.0b standard frame format

the nodes sense that the bus is free to use, they send a start of frame (SOF). All contentions are resolved as the nodes send the arbitration bits. If a node senses a dominant bit (logic 0) being transmitted as it is transmitting a recessive bit (logic 1), it loses arbitration and stops transmission. Receiver CAN controllers utilize message identifier masks for filtering messages sent out on the bus. The bitmasks are applied to message identifiers as they are made available on the bus. If there is no match, the CAN controller stops listening on the bus until it detects a SOF on the bus. These bitmasks are programmed and kept in the CAN controller memory for comparison when a message IDENTIFIER is received.

4 SYSTEM MODEL AND PROBLEM STATEMENT

We now discuss the real-time task model, threat model, and formally define the problem in this section.

4.1 Real-Time Task Model

We model all ECU tasks as periodic real-time tasks, each of which is described by a worst-case execution time (WCET) and a period. The period of these tasks can be lower-bounded based upon the time it takes to generate a message for transmission. Without a loss of generality, we assume that each task in the ECU is responsible for generating CAN messages with *non-overlapping* identifiers. That is, each task has a set of identifiers associated with it and only it. However, this is not a limitation of our approach and multiple tasks can share the same identifiers if so required. Further, certain ECUs may contain *emergency* tasks. For example, for airbag deployment, the airbag control module, arguably one of the most time-sensitive ECUs in a vehicle, may generate an emergency hard sporadic task to send a message to cut-off fuel to the engine to prevent a fire in the event of a crash. We consider such a task as having a very high priority, and which would be able to preempt any currently running periodic ECU task. We assume that tasks are scheduled using a priority-based round-robin scheduling algorithm where a higher-priority task always preempts a lower-priority task and processor time is equally divided between tasks of equal priority. Such a policy is selected for ease of implementation and predictability. In fact, ARM's commercial RTOS, Keil RTX5 [21], which we utilize in our experiments, uses this policy.

4.2 Threat Model

We consider a threat model where the attacker has *remote* access to a vehicle ECU. For detecting physical intrusions, such as an attacker attaching a malicious ECU to the bus, authentication must be done at the network level or at the receiver, which has already has been addressed by prior work [7, 15, 25, 26, 34] and is beyond the scope of this work. We consider that the attacker has taken advantage of external network interfaces made available by certain ECUs on the vehicle (such as WiFi and Bluetooth network interfaces created by a vehicle's infotainment unit) and has managed to compromise task(s) on the ECU. We also assume that the attacker operates *only* within the non-secure domain, and an attacker-controlled task can still make calls to the API made available in the NSC region. We assume two attacker privilege cases:

(1) Base case - The attacker takes control of task(s) (for example, through return-oriented programming [29]) on an ECU, is able to execute arbitrary code under this context, and can generate messages with identifiers meant for other tasks on the same, or different, ECU. The attacker, however, is unable to escalate its privilege level to match that under which the RTOS is executing. We believe that an attacker can be restricted to this privilege level since our system utilizes strict memory access guards using a hardware MPU when running any unprivileged non-secure code which limits its ability to force the RTOS to grant it a higher privilege.
(2) Advanced case - This is where the attacker has taken control of the RTOS and is able to run arbitrary privileged non-secure code. We believe our work is the first to provide some security guarantees on the compromised ECU even when the attacker has control of the RTOS and task code.

4.3 Problem Statement

We aim to design a lightweight, predictable defense mechanism for securing the CAN bus that achieves the following:

(1) **P1: Prevent Masquerade Attacks.** The attacker, who has infiltrated an ECU's task, tries to masquerade as another task running in the same ECU *or* in another legitimate ECU on the network, e.g., a compromised dashboard entertainment unit may send valid messages to control the engine RPM or apply the brakes. We aim to ensure that a compromised task can, at the most, only send out messages it was designed to generate and transmit, without controlling the actual message content.
(2) **P2: Prevent DoS attacks.** The attacker tries to launch a DoS attack by continuously sending messages. We aim to ensure that under no circumstance can a compromised task exceed the max rate at which it was designed to send out messages.
(3) **P3: Prevent Snooping.** The attacker tries to read messages intended for other tasks or other ECUs. We aim to remove any control the attacker has over the actual transmission or reception of a message. The attacker should only be able to read messages that were pre-destined for the compromised task. The attacker should also have no mechanism to know *when* another task or ECU sends a message on the bus.
(4) **P4: Ensure low latency to allow real-time operation.** Specifically, we aim to ensure that our approach has the smallest, yet predictable, effect on a task's worst-case execution time and that the task structure does not change.

Our proposed technique aims to address each of these problems for the *base case* (Section 4.2), and **P2** and **P4** for the *advanced case.* We also partially address **P1** and **P3** for the *advanced case.*

5 SYSTEM DESIGN AND OVERVIEW

Traditionally, all application code, including peripheral access, executes in unprivileged processor execution mode. Figure 3a showcases how such a traditional ECU system might look like. In addition, there is a supervisory code running in privileged execution mode, such as an RTOS, and application code runs in RTOS managed tasks. The RTOS provides scheduling capabilities and task stack management for context switching.

To support TEECheck, modifications must be made to the traditional system setup (Figure 3b). Specifically, we consider a TrustZone equipped microcontroller. The RTOS and application code remain in the non-secure domain. We do not need to change the task code structure, other than replacing the CAN driver and transmission code with calls to utilize TEECheck. The task code calls TEECheck's `request_for_transmission` (Section 6.1) and `request_for_reception` (Section 6.2) NSC functions, for transmission and accessing received messages, respectively. It must be noted that moving the entire RTOS into the secure region defeats the purpose of using TrustZone, since the secure domain code has unrestricted access to the memory space, only the smallest amount of code should be kept in the secure domain, to reduce the possibility of a vulnerability and, hence, attack surface. Further, by keeping only TEECheck and the CAN controller driver in the secure domain, we force every call to the CAN peripheral to utilize TEECheck.

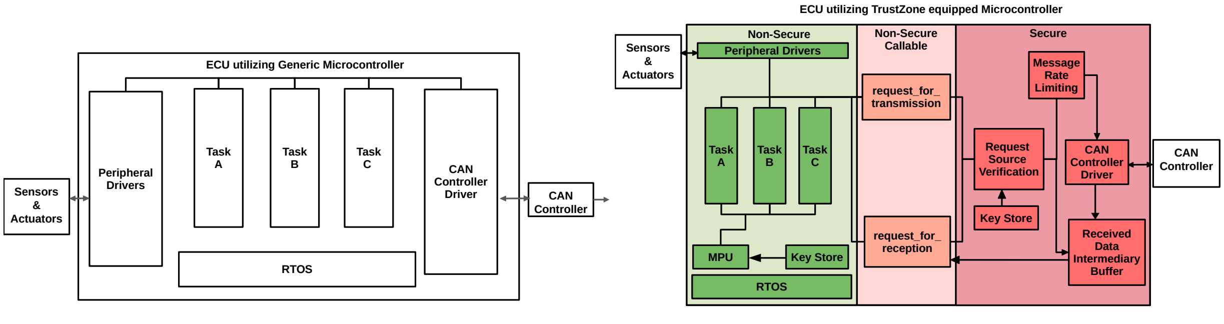
The RTOS is augmented with access to the non-secure domain's MPU which is loaded with task-specific access rights on every context switch. All peripherals (except CAN) necessary for task functionality are partitioned to the non-secure domain. The CAN controller driver must be paritioned into the secure domain such that it is exclusively accessible to TEECheck. All application code must utilize TEECheck's NSC functions (`request_for_transmission` and `request_for_reception`) to access the CAN bus, as discussed above. While a system timer is required by the RTOS for scheduling tasks, the secure domain requires one more timer. Considering our experimental testbed's (Section 7.1) microcontroller has four other timers, we believe this is a reasonable requirement.

It must be noted that the RTOS must be augmented with functionality to manage the per-task secure stack. Since we consider that every task generates CAN messages which are then transferred to and transmitted from the secure domain, stack management must be present for secure domain function calls to allow for safe context switching during task scheduling. Fortunately, most commercial RTOS that we surveyed for our experimental setup, which advertise official support for TrustZone, already provide an extensible mechanism for the RTOS to manage each task's secure stack.

6 TEECHECK : A TEE BASED CAN MESSAGE CHECKER

We now present our TEE based defense mechanism that leverages the new system design (Section 5) to address the problems stated in Section 4.2. TEECheck is built on two components:

(1) **Transmission:** TEECheck uses a two-stage pipeline, one stage for message source verification and another for message frequency enforcement.
(2) **Reception:** TEECheck does not allow ECUs direct access to any messages which pass the CAN controller message



(a)

(b)

Figure 3: (a) Regular ECU system and (b) ECU system utilized for TEECheck

filtering stage and which are made available for reading. TEECheck verifies the identity of the requesting task before it forwards the message from the reception buffer.

Please note that we address **P4** of our problem statement (Section 4.3), by designing TEECheck to work as a set of sequential function calls with no waiting to (1) reduce the impact on a task's WCET and (2) keep the task's structure unchanged.

6.1 Transmission

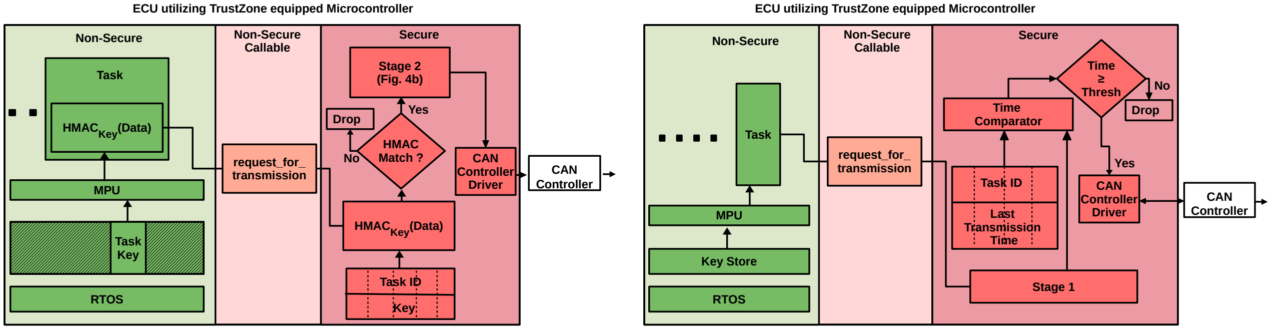
While problem **P3** of our problem statement (Section 4.3) is addressed by the reception scheme, the transmission scheme is designed to solve problems **P1** and **P2**. The transmission scheme leverages our proposed system (Section 5) to prevent an attacker from masquerading as another task or ECU, as well as from overwhelming the bus (launch a DoS attack). Stage 1 details our message source verification scheme. Stage 2 provides details on limiting the rate of messages that are sent out to the CAN controller.

6.1.1 Stage 1: Source Verification. Stage 1 implements a source verification scheme to eliminate the possibility of a masquerade attack. It utilizes the strict partitioning of the TrustZone to verify the source of a message transmission request. The original system shown in Figure 3a, while simple, is flawed from a security perspective. Any misbehaving task can generate messages on behalf of other tasks or even other ECUs in the network. Further, currently available RTOS such as Keil RTX5 (which we use in our experiments in Section 7) do not mediate access to device drivers like general purpose OS such as Linux. Rather, the RTOS provides memory management and scheduling capabilities while device access and manipulation (such as access to the CAN controller) is accomplished within the task code. Since the tasks have direct access to the CAN controller, they can simply queue spoofed messages for transmission. However, the TrustZone provides a strong access control mechanism for peripherals. Shifting peripheral access into the TrustZone while keeping the tasks in the non-secure domain provides us with the opportunity to control data entry into and out of the secure domain. Under the assumptions of our threat model, the TrustZone interface is the last point at which the attacker still has control over the data. We, thus, propose building a task verification stage at the TrustZone interface before admitting data for transmission on the CAN bus. Every task requesting for transmission must first pass this verification stage on a message-by-message basis or else the message is immediately dropped.

We now discuss the implementation of the verification stage. As noted in the system overview presented in Section 5, the RTOS, which controls the MPU, runs as the privileged code while the tasks run in unprivileged mode. Any access to memory locations that are not explicitly marked as accessible by a non-secure task leads to a hard fault generation (a high priority interrupt originating in hardware that halts system execution and requires human intervention to reset the entire system and/or perform cleanup before continuing).

There are two possible avenues of approach to verify a task. One mechanism is by querying the RTOS which task is requesting for CAN access and the other is to build a non-intrusive scheme on top of the RTOS. While utilizing the former approach can be very lightweight, it increases our dependence on the RTOS, enforcing the requirement that the RTOS must not be compromised. However, if an attacker operates under the *advanced case* assumption, such a system is bound to fail immediately. Further, querying the RTOS requires a dialogue between secure domain code and RTOS, which requires RTOS modifications. For automotive systems, this would require expensive re-verification of RTOS functionality and safety. Instead we propose a verification stage between task code and secure domain while keeping the RTOS code untouched. A limitation of the proposed stage 1 is that it still requires certain RTOS guarantees which causes it to partially fail (explained later) when considering an attacker operating under the *advanced case*. However, our proposed approach works well for a *base case* attacker, and being largely RTOS independent, provides a platform for future work to address all problems for the *advanced case* while still providing performance improvements over related work.

A detailed overview of the steps in the verification stage is presented in Figure 4a. We verify the origin of each request by computing and then verifying an HMAC on the message data. HMAC, or hash-based message authentication codes [20], are cryptographic algorithms that take an arbitrary length input and produce a fixed-length output by utilizing a secret key. Only entities possessing the key can generate the same output from a given input, assuming that the HMAC is well designed. We utilize HMACs to verify that a message originates from the correct task. Applying our mechanism to authenticating REMOTE frames, the resultant mechanism would



(a) Stage 1 **(b) Stage 2**

Figure 4: (a) Source verification using HMAC (b) Rate limiting messages based on per-task last transmission time

require a minor modification such that it follows the same authentication steps as that in the reception scheme which we detail in Section 6.2, where a counter is utilized instead of the message data.

Authentication is done by keeping copies of a table of keys, one key per task, in both secure and non-secure domains. The key table consists of the task identifier and its associated key. An MPU is utilized to restrict a task to access only its designated table entry. We load the MPU during every task context switch with the task-specific address masks such that the task can (i) read its task-specific key-table entry, (ii) read and execute its code section (iii) read and write to its stack in RAM, (iv) call relevant RTOS API, and (v) call TEECheck. Note that, loading the MPU only takes a few clock cycles. The address mask is applied to every memory operation and takes a single cycle due to the MPU being wired-logic hardware. [2]

The key table can be generated and stored in flash during ECU deployment. Alternatively, a small procedure can be added during RTOS initialization to regenerate the keys and store a copy in both key tables. An example mechanism of key regeneration could be where the RTOS requests the secure domain code for a fresh set of keys. The secure domain code generates the new keys using a random number generator, saves it to the secure domain table, and forwards it to the RTOS for storage on the non-secure side. Considering the *base case* assumptions, regenerating the keys during RTOS initialization ensures that the MPU is able to hide them before any non-secure task is allowed to run. The task utilizes its key to generate an HMAC based on the data that it needs to transmit. It then calls the `request_for_transmission` function and passes the message pointer, the generated HMAC tag pointer, and the task identifier. Once code execution enters the secure domain, the control is no longer in the hand of the attacker. We read the value of the generated HMAC, look up the key copy from the secure domain key table based on the advertised task identifier, and regenerate the HMAC based on the data to be transmitted in the secure domain. Since we target the CAN bus, the data size is assumed to be 8 bytes. While we do not have any specific requirements regarding the HMAC algorithm, utilizing lightweight algorithms (such as Chaskey [22]) built for small data sizes is advised. Once the task identifier is validated, a CAN message identifier is assigned based on it. In case of the *advanced case* there is only a partial failure of verification stage as the key tables contain identifiers only for tasks meant to run on the ECU. Even if an attacker controls the RTOS, they cannot send out a message which should originate from a different ECU. While we cannot control the actual data, short of recomputing the data in the secure domain, we limit the attacker to only the compromised task and its related message.

6.1.2 *Stage 2: Rate Limiting.*

We introduce stage 2 of the transmission scheme where we rate limit each task's message transmission to address **P2**. Since this stage is entirely within the secure domain, it works equally well for both threat model cases. The overview of this stage is presented in Figure 4b. While utilizing the HMAC scheme in stage 1 prevents a task from sending out false messages on behalf of another task, the attacker could continuously send out valid HMACs to the secure domain to pass stage 1 and force the secure domain to send messages out onto the bus, keeping the bus as busy as possible. While there have been techniques presented in prior work to detect the occurrence of a DoS attack and shut down the offending ECU, these mechanisms are *reactive* in nature and can still disrupt the CAN bus, albeit for short periods of time. We wish to prevent a DoS attack *proactively* before the bus is affected.

Fortunately, since messages are usually generated in a periodic manner inside a vehicle, a system designer is aware of the maximum frequency at which a legitimate task generates messages. We utilize this knowledge to create a per-task rate limiter. Our rate limiting scheme is similar to the more sophisticated mechanism employed in the per-core queue in Carousel [28], to reduce space and computation time. We utilize the hardware timer partitioned to the secure domain to create periodic time *ticks*. The timer counts to the desired period and generates an interrupt. The interrupt handler records the number of ticks. Utilizing interrupts allows for asynchronous operation. While deciding the period value is left to the designer, an example would be to set it to the greatest common divisor of all message periods. For tasks that generate messages of varying frequencies, the worst-case frequency can be used and finding an optimal rate limiter is left for future work. Along with the key table copy, the tick value at the time when the last message was accepted for transmission for the relevant task identifier is also recorded. Stage 2 checks the current tick value, the previous transmission acceptance tick value and the maximum frequency of the task. If the frequency is higher than the maximum allowed, the message is dropped. Else, it is forwarded for transmission.

[2] Loading the MPU is the only guarantee required of the RTOS and we aim to remove this dependency in future work.

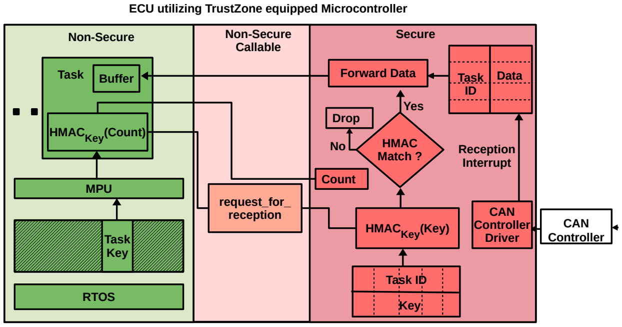


Figure 5: TEECheck Reception scheme

6.2 Reception

The reception scheme (Figure 5) addresses **P3** of our problem statement (Section 4.3). The reception scheme aims to allow tasks to access *only* those messages with identifiers that belong to them. This prevents a compromised task from snooping for messages to gather sensitive information about the status of the vehicle.

Messages are accepted by the CAN controller when they pass the CAN filtering as described in Section 3.2. The filter masks can only be changed by secure domain code since the CAN peripheral is memory-mapped exclusively into secure domain, preventing any non-secure domain code from modifying these masks. This ensures that only messages intended to be received by the ECU, are accepted by the controller. By itself, this limits the attacker to only messages that are to be received by the ECU with zero additional overhead, partially fulfilling the requirements of even the *advanced case*. All messages are handled and stored in a table along with the designated receiving task identifier *asynchronously* via a high priority interrupt handler. Older values are always overwritten, keeping the contents of the table fresh.

The reception scheme is similar to stage 1 of the transmission scheme. While it would be possible to simply share the task-specific key (made available by the MPU) with the secure domain to ascertain task identity, if an attacker manages to brute-force the correct victim task's key value, the attacker could access *every message* intended for the victim task. Instead, we utilize a per-task counter kept in the secure domain. Every task must first request its counter by supplying its task identifier, then generate the HMAC tag based on this counter and its specific key and then pass the generated tag back to the secure domain for verification using the `request_for_reception` TEECheck NSC function. Regardless of whether task verification passes or fails, the counter is always incremented to prevent an attacker from brute-forcing the correct HMAC tag for a constant input (counter value). Since differentiation between tasks on the ECU requires the MPU, this scheme will fail under the *advanced case* and will be addressed in future work.

Based on the verified task identifier, the corresponding message from the message table is copied to the designated non-secure location pointer passed to the `request_for_reception` after the location is verified as belonging exclusively to the task. For example, an HMAC could be generated from the task key and a nonce during the key refresh phase at system bootup. The task must then write that value to the pointer location passed to the secure domain, which verifies it before overwriting the location with message data.

Table 1: Single task running at highest frequency (RL - Rate Limiting, CAN - CAN controller transmission time)

Test	Avg (µs)	Med (µs)	Max (µs)	Min (µs)
HMAC-NS (Chaskey)	223	227	229	185
HMAC-S (Chaskey)	247	247	259	242
HMAC-S + RL	254	253	265	248
HMAC-S + RL + CAN	368	357	400	355

7 EXPERIMENTATION

We now provide details on our experimental testbed and results. We run different loads on our testbed to gauge the overhead and predictability of utilizing TEECheck. We also apply TEECheck to a well known automotive benchmark as a case study.

7.1 Experimental Setup

Our experimental testbed consists of a Nuvoton NuMaker PFM-M2351 development board [1]. It uses Nuvoton's M2351KIAAE implementation of ARM Cortex-M23, based on the ARMv8-M baseline architecture (the least powerful ARMV8-M variant) and has 512 KBytes and 96 KBytes of on-board flash memory and RAM, respectively. The microcontroller supports separate MPUs for secure and non-secure domains, an on-board CAN 2.0b peripheral, and four general purpose hardware timers. We use ARM's Keil RTX5 [21], an automotive functional safety compliant (ISO 26262) RTOS which provides a convenient API for real-time data logging. Our testbed's operating frequency is 12 MHz which allows easy collection of real-time runtime data via the debugger. While the ARM Cortex-R series provides similar operating frequencies as vehicle ECUs (> 100 MHz), we cannot use them as they do not yet support ARM TrustZone.

Since we are not concerned with network-level authentication techniques in this work, we configure the CAN controller to run in Loopback mode, where all transmitted messages are routed back to the controller's reception interface. We enable loading of the MPU in Keil RTX5 for every task context switch. For our experiments, we utilize the ISO standardized Chaskey [22] lightweight HMAC algorithm which is specifically designed for 32-bit microcontrollers and small data sizes. Please note that we use a software HMAC implementation to show a worst-case overhead for our approach. In the case where a hardware-accelerated HMAC is used, our overhead will further reduce since hardware-accelerated HMAC are known to take less than 100 cycles per computation. Each data point shown in the tables below is generated from 10 experiments due to space limitations for storing timestamps on board in real-time.

7.2 Results

We now conduct experiments considering different workloads.

7.2.1 Single task transmitting at highest frequency. The first workload consists of a single task running at the highest possible frequency (constantly looping). This is to simulate a situation where the processor never idles, e.g, an ECU that continuously gathers sensor data and sends it out to a central ECU. We present the results in Table 1. Our experimental data shows that when we utilize a secure domain call to generate the HMAC in the secure domain, the entire operation takes the same time as generating the HMAC in the non-secure domain plus the domain switch overhead. The rate

Table 2: Single task running at highest frequency with reception. TEECheck call overhead

Test	Avg (µs)	Med (µs)	Max (µs)	Min (µs)
Counter request	10	10	11	7
Reception	269	270	271	263
Transmission	373	357	404	356

limiting stage is very lightweight, only taking an additional 7 us on average. It should be noted that we do not consider reception for these set of results to remove the overhead that could be caused by the CAN reception interrupt. Finally, we also show that the call to the CAN controller for data transmission takes an additional 114 us for a total average of 368 us. Our TEECheck Transmission scheme, thus takes a total of 477 us (223 us + 368 us - 114 us) on average over the base case of providing access to the CAN controller directly to the task. On realistic ECU hardware, our overhead would scale down to 50µs making it extremely lightweight since ECUs send out messages at intervals of milliseconds or higher [19].

7.2.2 *Single task transmitting at the highest frequency with reception.*

We now enable the CAN controller loopback in the same setup as that used in Section 7.2.1 to test the reception overhead. While the interrupt handler runs asynchronously to store the received data in the intermediary data reception table, the TEECheck Reception scheme first checks the task identifier before copying the data to the task's allocated reception space. Results for the reception scheme are presented in Table 2. Requesting the counter value takes about 10 us. Results show a 1% difference in transmission overhead from that shown in Table 1. As expected, the results for the call to TEECheck's reception scheme are similar to the first stage of the transmission scheme with some additional overhead (15 us) that can be attributed to the additional data copy from the secure domain data reception table to the non-secure domain memory address.

While our transmission overhead is provided for 8 bytes, We also provide results for transmitting different sizes of data, from 1 to 7 bytes in a CAN frame in Table 3. Although the HMAC utilizes zero-padding to account for smaller message sizes, the data shows that the overhead for passing the data to the CAN controller is negligible, leading to nearly the same time taken for that when transmitting 8 bytes. Since the behavior of reception and transmission schemes and overhead are so similar, we concentrate on the (heavier) transmission scheme overhead for the rest of this section.

7.2.3 *3 tasks - 1 legitimate, 1 attacker and 1 idle.*

We now simulate ECU with two tasks, one of which has been compromised. A third idle task runs when both of the other two tasks are not ready to run. A priority-based round-robin scheduling mechanism is used which provides equal time to tasks of the same priority and preempts a currently running task if a task with higher priority arrives. All 3 tasks have the same priority. The legitimate task's period is set to 1ms and is always allowed to send a CAN frame. The attacker task does not follow any periodicity and sends out messages whenever it is provided with CPU time. The rate limiting for the attacker task is kept in such a manner that the messages from that task are dropped on every alternate call. Results show that the average, median, maximum and minimum TEECheck call overhead of the legitimate task are **369**µs, **364**µs, **405**µs and **362**µs, respectively. This shows that the average time for calling TEECheck for secure domain processing from the legitimate task remains the same, with a slight increase in the minimum time to account for the overhead due to task switching. In addition, the overhead is negligible even when the attacker has the same priority as a legitimate task and utilizes as much CPU time as possible. Due to the limitations of the Keil RTX5 scheduler, we do not show the results of a higher-priority attacker task since the scheduler would always allow the higher-priority task to run if it is ready to run, impeding the operation of the lower-priority, legitimate task. Modifying the scheduler to deal with such situations is out of the scope of our work. However, regardless of task priority, no task can launch a DoS attack on the CAN bus due to our rate limiting.

Table 3: Transmitting different message sizes

No. of bytes	Avg (µs)	Med (µs)	Max (µs)	Min (µs)
1	368	256	298	351
2	368	356	398	351
3	369	357	399	352
4	370	357	400	353
5	366	358	398	352
6	367	358	398	354
7	367	358	398	354

7.2.4 *4 tasks - 1 emergency, 1 legitimate, 1 attacker and 1 idle.*

We extend the system setup for the experiment in Section 7.2.3 with an emergency task. We consider a sporadic emergency task, with a minimum inter-arrival time of 3 ms, so as to allow easy gathering of the event data while ensuring that the legitimate task meets its deadlines. Results show that the average, median, maximum and minimum TEECheck call overhead of the emergency task are **374**µs, **370**µs, **408**µs and **367**µs, respectively. There is a slight increase in the overhead by 5 µs from the previous cases. This is to be expected as the emergency task is activated frequently when the other 2 tasks have already entered the secure domain, requiring tear-down of both secure and non-secure domain stacks before switching to the emergency task.

7.2.5 *Real world automotive benchmarks.*

We now augment a well know real-world automotive benchmark [19] with TEECheck. We sort the 9 tasks in a non-increasing order of periods (ranging from 1s to 1ms) and augment each task with a transmission request to TEECheck. Table 4 provides our results. Here Augmented Tasks 1 is where only the task with 1 second period calls the TEECheck NSC function and the values are the execution times for that task, Augmented Task 2 is where both 1 s and 0.5 s tasks call TEECheck and the values are presented for the 0.5 s task. Augmented Tasks 9 is when all tasks have a transmission request and we report the execution times for the task with the shortest period. This experiment confirms that TEECheck incurs a predictable overhead on the highest frequency augmented task's execution time even when scheduled in a round-robin (with same priority) fashion in the presence of other tasks calling TEECheck. When disregarding TEECheck's and CAN controller overheads (477µs and 144µs), the values presented here exceed the actual task execution time by only 30-90 µs. This is likely because TEECheck overhead brings task execution time at-par with the period of the higher frequency tasks, which interfere with another task's execution every time they become ready to run, and due to interrupts from the CAN controller. On realistic ECU hardware, TEECheck's overhead for

Table 4: Automotive benchmark with increasing number of tasks with TEECheck

Augmented Tasks	Avg (µs)	Median (µs)	Max (µs)	Min (µs)
1	677	692	858	618
2	660	623	776	587
3	1087	1093	1280	1017
4	756	742	895	701
5	978	1007	1092	895
6	1010	1024	1131	912
7	766	766	960	690
8	680	654	874	648
9	666	646	827	615

this benchmark is <60 µs. Please note that we present pessimistic results where every task calls TEECheck as we do not know which tasks, in reality, require CAN access.

8 ANALYSES

We now provide security and real-time analyses.

8.1 Real-Time Analysis

We first analyze the real-time properties of our approach in relation to **P4** in our problem statement. Considering our new system architecture presented in Section 5, TEECheck simply acts as sequential function calls without additional buffers or other mechanisms that could change the nature of the code flow to affect the task model. As such, TEECheck can simply be modeled as a *constant-time overhead* that must be added to a task's worst-case execution time. Note that this overhead is only applicable to tasks which interact with the CAN bus. The overhead is constant since HMAC generation time on a given length of input data (8 bytes for CAN) is constant and the rate limiting stage is an arithmetic filter that always compares two values: the previous message transmission time and the current time. Our experimental results based on different system setup scenarios, presented in Section 7, verify our analysis. They show almost constant time overhead with minimal variation regardless of system load. The slight variance in data is due to the interrupts being fired by the CAN controller when it receives the looped-back messages. Our approach also shows negligible variation in the case of emergency (sporadic) tasks such as airbag deployment.

8.2 Security Analysis

We now evaluate each stage of the transmission and reception schemes for their effectiveness in addressing our problem statement detailed in Section 4.3. Stages 1 and 2 of the transmission scheme are designed to address the problems **P1** and **P2**, respectively. The efficacy of stage 1 is based on the assumption of HMAC unforgeability. An attacker could either brute force every HMAC tag value until it matches a valid HMAC tag for the data that it wishes to send, or generate key possibilities for creating the HMAC valid for the data that it wishes to send. Brute-forcing an HMAC algorithm with an n-bit key takes, on average, 2^{n-1} HMAC calculations. Considering an HMAC algorithm (such as Chaskey which we utilized as a part of our experimental setup) which requires a 128 bit key and generates a 128 bit output tag, the number of tries, on average, for each of the two types of attack would be 2^{127}. While the second attack mechanism is much slower, since the attacker needs to generate the HMAC for every possible key, with a total overhead of about 470 us (223 us for HMAC tag generation in non-secure domain and 247 us for the HMAC tag generation with call to the secure domain), the first attack mechanism would take roughly half the time since the attacker could simply use a counter and pass its value to the secure domain as an HMAC instead of running the HMAC algorithm in the non-secure domain. However, in either case, the verification HMAC generation in the secure domain code cannot be bypassed since accessing the CAN controller must go through TEECheck which always verifies the HMAC first. As such, this makes the attack practically impossible.

Further, the attacker is limited to a short window for guessing the correct HMAC. This is due to the rate limiting stage 2. Since the time-sharing between ECU tasks is enforced by the RTOS, it is guaranteed that the victim task will get to execute. Considering a very basic setup scheduled under the round-robin execution policy, where all tasks have equal priority and context switch is enforced after the same amount of time τ for every task, and considering the time difference δ between messages for the victim task running at highest frequency (that is, a task which is constantly looping over the HMAC generation and call to the `request_for_transmission` NSC function), the victim task (in a system with n tasks) will send out messages every *Msg_Send* which is computed as:

$$Msg\_Send = \delta + (n-1) \cdot \tau \tag{1}$$

While an attacker may have prior knowledge of this value, the exact time of the previous transmission during vehicle operation by the victim task cannot be guessed easily, especially because our system prevents tasks from snooping the bus. The attacker must guess during *Msg_Send* for successful transmission else even a correct guess will be blocked by stage 2. An intelligent attacker could try to observe execution time differences to detect which stage caused transmission failure. This would require a high resolution timer and can be mitigated by partitioning unused timers to the secure domain. Due to the vastly different execution times of the stages, it is recommended to keep stage 1 and stage 2 in their current positions to prevent an attacker from using the system tick timer (if timer frequency is sufficiently high) to differentiate between stages.

The analysis for the reception scheme is similar to stage 1 of the transmission scheme. The reception scheme is aimed at solving problem **P3** by making it difficult for an attacker to snoop on messages intended for other recipients on the same ECU. Since the attacker has no control over the counter value, the attacker cannot practically perform a replay attack or brute-force the HMAC key, especially if a key refresh occurs on every system reboot.

9 CONCLUSION

We designed a new TEE based architecture for ECUs that effectively partitions the CAN controller from the rest of the ECU code. We presented TEECheck, an on-device TEE based defense mechanism to prevent masquerade attacks, DoS, and information leakage. We implemented our proposed architecture on an actual device, a TrustZone enabled ARM Cortex-M23 based Nuvoton M2351 and showed that our technique has very low overhead (maximum of 494 us and 500 us for transmission and reception respectively), and is very predictable showcasing negligible (around 1%) variance in overhead regardless variations in, and types of, system loads.

ACKNOWLEDGMENTS

This work was supported in part by NSF under grant numbers 1650540 and 1618979, and by the Commonwealth Cyber Initiative, an investment in the advancement of cyber R&D, innovation and workforce development in Virginia. For more information about CCI, visit cyberinitiative.org.

REFERENCES

[1] 2019. *NuMicro M2351 series – a TrustZone empowered micro-controller series focusing on iot security.*

[2] 2019. Renesas provides chips for Toyota. https://can-newsletter.org/engineering/applications/171114_17-4_renesas-provides-chip-for-toytas-self-driving-cars_renesas.

[3] Sherif Aly. 2017. *Consolidating AUTOSAR with complex operating systems (AUTOSAR on Linux).* Technical Report. SAE Technical Paper.

[4] ARM. 2009. Security technology building a secure system using TrustZone technology (white paper). *ARM Limited* (2009).

[5] Ahmad Atamli-Reineh, Ravishankar Borgaonkar, Ranjbar A Balisane, Giuseppe Petracca, and Andrew Martin. 2016. Analysis of trusted execution environment usage in Samsung KNOX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution.* ACM.

[6] Malek Ben Salem. 2012. *Towards effective masquerade attack detection.* Ph.D. Dissertation. Columbia University.

[7] Donghoon Chang. 2018. CAN-FD-Sec: Improving Security of CAN-FD Protocol. In *Security and Safety Interplay of Intelligent Software Systems: ESORICS 2018 International Workshops, ISSA 2018 and CSITS 2018, Barcelona, Spain, September 6–7, 2018, Revised Selected Papers.* Springer, 77.

[8] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. 2011. Comprehensive experimental analyses of automotive attack surfaces.. In *USENIX Security Symposium*, Vol. 4. San Francisco, 447–462.

[9] Kyong-Tak Cho and Kang G Shin. 2016. Fingerprinting electronic control units for vehicle intrusion detection. In *USENIX Security Symposium.* 911–927.

[10] Kyong-Tak Cho and Kang G Shin. 2017. Viden: Attacker identification on in-vehicle networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 1109–1123.

[11] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[12] Mohammad Farsi, Karl Ratcliff, and Manuel Barbosa. 1999. An overview of controller area network. *Computing & Control Engineering Journal* 10, 3 (1999), 113–120.

[13] Mahsa Foruhandeh, Yanmao Man, Ryan Gerdes, Ming Li, and Thidapat Chantem. 2019. SIMPLE: Single-Frame based Physical Layer Identification for Intrusion Detection and Prevention on In-Vehicle Networks. In *Annual Computer Security Applications Conference.*

[14] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. 2009. AUTOSAR–A Worldwide Standard is on the Road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, Vol. 62. 5.

[15] Bogdan Groza, Stefan Murvay, Anthony Van Herrewege, and Ingrid Verbauwhede. 2012. Libra-can: a lightweight broadcast authentication protocol for controller area networks. In *International Conference on Cryptology and Network Security.* Springer, 185–200.

[16] Yutian Gui, Ali Shuja Siddiqui, and Fareena Saqib. 2018. Hardware Based Root of Trust for Electronic Control Units. In *SoutheastCon 2018.* IEEE, 1–7.

[17] Florian Hartwich et al. 2012. CAN with flexible data-rate. In *Proc. iCC.* Citeseer, 1–9.

[18] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. 2010. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy.* IEEE, 447–462.

[19] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. 2015. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS).*

[20] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. 1997. HMAC: Keyed-hashing for message authentication. *RFC Editor* (1997).

[21] Arm Ltd. 2019. *Keil RTX5.*

[22] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. 2014. Chaskey: an efficient MAC algorithm for 32-bit microcontrollers. In *International Conference on Selected Areas in Cryptography.* Springer, 306–323.

[23] Anway Mukherjee, Tanmaya Mishra, Thidapat Chantem, Nathan Fisher, and Ryan M. Gerdes. 2019. Optimized trusted execution for hard real-time applications on COTS processors. In *RTNS '19.*

[24] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 2013. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium.* 479–498.

[25] Stefan Nürnberger and Christian Rossow. 2016. −vatican−vetted, authenticated can bus. In *International Conference on Cryptographic Hardware and Embedded Systems.* Springer, 106–124.

[26] Andreea-Ina Radu and Flavio D Garcia. 2016. LeiA: A lightweight authentication protocol for CAN. In *European Symposium on Research in Computer Security.* Springer, 283–300.

[27] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. IEEE, 57–64.

[28] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Carlo Contavalli, Amin Vahdat, et al. 2017. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* ACM, 404–417.

[29] Hovav Shacham et al. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86).. In *ACM conference on Computer and Communications Security.* 552–561.

[30] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP).* IEEE, 138–157.

[31] Jo Van Bulck, Jan Tobias Mühlberg, and Frank Piessens. 2017. VulCAN: Efficient component authentication and software isolation for automotive control networks. In *Proceedings of the 33rd Annual Computer Security Applications Conference.* ACM, 225–237.

[32] Anthony Van Herrewege, Dave Singelee, and Ingrid Verbauwhede. 2011. CANAuth-a simple, backward compatible broadcast authentication protocol for CAN bus. In *ECRYPT Workshop on Lightweight Cryptography*, Vol. 2011.

[33] Kizheppatt Vipin. 2018. CANNoC: An open-source NoC architecture for ECU consolidation. In *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS).* IEEE, 940–943.

[34] Qiyan Wang and Sanjay Sawhney. 2014. VeCure: A practical security framework to protect the CAN bus of vehicles. In *2014 International Conference on the Internet of Things (IOT).* IEEE, 13–18.

[35] Xuhang Ying, Giuseppe Bernieri, Mauro Conti, and Radha Poovendran. 2019. TACAN: Transmitter authentication through covert channels in controller area networks. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems.* ACM, 23–34.

[36] Joseph Yiu. 2015. ARMv8-M architecture technical overview. *ARM WHITE PAPER* (2015).

[37] Tobias Ziermann, Stefan Wildermann, and Jürgen Teich. 2009. CAN+: A new backward-compatible Controller Area Network (CAN) protocol with up to 16x higher data rates. In *Proceedings of the Conference on Design, Automation and Test in Europe.* European Design and Automation Association, 1088–1093.