Hardware-assisted Runtime In-vehicle ECU Firmware Self-attestation and Self-repair

Josh Dafoe, Job Siy, Niusen Chen, and Bo Chen*

Department of Computer Science, Michigan Technological University, Michigan, United States
bchen@mtu.edu

Abstract. Modern vehicles are largely controlled by many embedded computers, known as Electronic Control Units (ECUs). The increased use of ECUs has brought many in-vehicle security concerns. Specifically, injection of malware into ECUs poses a significant risk to vehicle operation. Indeed, many ECU malware injection attacks have been performed, and much work has been introduced towards mitigating these vulnerabilities. A main defense is for ECUs to perform a self-attestation over their firmware state. However, most current self-attestation solutions do not enable runtime checking due to their high computational cost. Additionally, existing solutions mostly do not incorporate any ECU self-repairing in coordination with the attestation mechanisms.
In this work, we have designed FSAVER, a highly efficient self-attestation and self-repair framework for in-vehicle ECUs. For the self-attestation, we adapt highly efficient spot-checking techniques, so that the firmware can be checked periodically at runtime. To perform these attestations, we rely on the TEE already equipped within each ECU. For self-repair, we take advantage of the isolated flash memory controller (FMC) in the storage device. Specifically, we coordinate it with the update mechanism and self-attestations to guarantee that the latest benign firmware version can always be restored. To realize this while malware is running, a special mechanism has been carefully developed to notify the FMC of the malicious presence.

Keywords: Autonomous Vehicles, Firmware Attestation, Self Repair, Trusted Execution Environment, Flash Memory Controller, Flash Translation Layer

1 Introduction

Modern vehicles increasingly contain many specialized computers, mostly known as Electronic Control Units (ECUs). These embedded real-time ECUs perform specialized tasks, many of which are safety critical vehicle operations such as controlling the brakes, throttle, or steering. Vehicles today can contain up to 150 ECUs [17]. With the advent of autonomous vehicles, these ECUs are gaining more responsibilities and so require more sophisticated software.

* Corresponding author.

To facilitate the transmission of control and communication signals between ECUs, the original Controller Area Network (CAN) protocol was introduced in 1986. Additionally, to provide users with diagnostic information and the ability to connect to the CAN bus, an OBD-II port has been introduced. This has been mandated on all new vehicles in the US since 2009 [51]. The control and communication signals sent over the CAN are essential information, used by many ECUs to initiate their safety-critical behavior. For example, the airbag control module receives information from deceleration sensors, yaw rate sensors, and seat occupancy sensors, all of which need to communicate properly in order to activate the airbags [49]. Upon airbag activation, the airbag control module communicates with the engine control module to cut off fuel supply which prevents fire [47]. If *any* one of the mentioned ECUs are compromised with malware, then the *entire* airbag deployment system may fail during a collision, or even be initiated during normal vehicle operation.

To introduce such malware to an ECU, an attacker would need to perform ECU reprogramming. Usually, reprogramming of ECUs is performed over the CAN bus. To do this, a diagnostic tool connected to the bus will initiate a reprogramming mode. During this mode, new software is sent to the target ECU. A widely deployed protocol which enables such a reprogramming mode is Unified Diagnostic Services (UDS). Before initiating the reprogramming mode, (and so allowing arbitrary manipulation of the ECU code) UDS performs an authentication of the diagnostic tool. This allows anyone with a valid diagnostic tool to enter programming mode [29]. This is the only authentication that UDS performs. Consequently, anyone who can pass this authentication can typically write arbitrary code to any in-vehicle ECU. Unfortunately, this authentication can usually be passed even without access to a valid diagnostic tool, enabling malware to be written directly to a target ECU [19, 29, 31]. Therefore, any compromised device connected to the CAN bus is able to reprogram any target ECU. These compromised devices include mobile phone apps, OBD-II dongles, or even other ECUs [7, 15, 28, 53].

To mitigate such attacks, one main strategy has been to employ a passive defense [50] solution which aims to detect malicious *behavior*, then perform some isolation and restoration. Towards accurate detection of malicious behavior, many intrusion detection systems have been implemented [16, 18, 20, 57]. These systems aim to identify malicious messages being sent over the CAN bus by a compromised device (e.g. an ECU, OBD-II dongle, etc), and localize that device [9, 37, 56]. Upon this detection and localization, [30] proposes several potential mitigation measures by placing the target ECU in a safety mode, or disabling the attack messages. However, they do not provide explicit methods to achieve this, and have no evaluation of their method. To provide such explicit restoration means, [23] proposes enabling a roll back to the original firmware before resetting the ECU, which allows a restoration of the compromised ECUs.

Another approach to detecting malware on an ECU has been to explicitly check the firmware contents, rather than performing detection based on adversarial CAN messages. This approach has several advantages. For example, ECU

malware does not always cause malicious behavior on the CAN bus, meaning that such malware may go undetected by CAN-based intrusion detection methods. However, checking of the firmware contents will immediately reveal any malicious code injections. One such firmware content based detection mechanism is to perform a self-attestation, where the entire firmware content is usually checked against a signature provided by the OEM [13, 35, 38, 41, 42, 52]. A few of these self-attestation solutions check this signature *only* immediately after writing it to the flash memory [35, 38, 52]. However, performing this check only once may not be sufficient [25, 38, 41]. To resolve this, several solutions rely on isolated software (i.e. bootloader, hardware security modules, etc) which performs an attestation upon each reboot before loading the firmware into memory [13, 25, 41, 42]. However, there is still need for a good *runtime* self-attestation solution [25, 38]. A challenge in ECU self attestation schemes has been the strict real time requirement that ECUs have [8, 25], making it difficult to implement a runtime self-attestation. In particular, since ECUs are usually low power embedded systems, the operations required to verify a cryptographic signature over the entire firmware are very time consuming. Towards providing such an efficient runtime self-attestation scheme, Kaster et al. [25] propose organizing the firmware into d blocks with b cells per block, then "slicing" the blocks so that one cell from each block is in a slice. Each slice can be checked individually. This allows a probabilistic self-attestation which is much more efficient than other designs. However, this solution still requires expensive cryptographic operations (CBC-MAC) for the attestation, and requires the use of additional secure hardware which is expensive.

In addition to the need for an efficient runtime self-attestation scheme which is adapted to the in-vehicle real time requirements, there is a need for quickly restoring the old firmware upon malware detection [8, 35, 38]. Mansor, et al [35] proposes enabling such restoration by *temporarily* storing old firmware on a "central communication unit". However, their design allows only one firmware backup at a time, so that restoration is possible only after a single initial attestation. When performing runtime self-attestation, a persistently available restoration mechanism is desirable. Dafoe et al [23] adapted a firmware rollback mechanism which was built into the flash memory controller software. This allows an automatic implicit backup of the old firmware version, taking advantage of the out-of-place updates nature of NAND flash memory, and the isolation of the flash memory controller. However, this rollback mechanism was adapted for a intrusion detection approach rather than a self-attestation.

In this work, we introduce FSAVER, which is a **F**irmware **S**elf **A**ttestation and in-**V**ehicle **E**CU **R**epair design. FSAVER is highly scalable (as demonstrated in Section 6), and conforms to the real-time requirement of a runtime attestation. We achieve this design by utilizing existing hardware components to develop two functions that will run on each ECU: self-attestation, and self-repair.

Self-attestation. In order to regularly check the firmware contents for malware, we need a trusted entity to perform regular attestations (i.e. a trusted *auditor*). Typically, ECUs are equipped with ARM Cortex-A or Cortex-M [43–45, 55]

CPUs which have TrustZone capabilities [32,33]. The TrustZone is a hardware-level security feature built into the processor which enables a trusted execution environment (TEE, introduced in Section 2.3) isolated from the normal insecure execution environment. Even if the ECU firmware is compromised, the execution running in the TEE secure world remains uncompromised. Thus, we can establish the TEE secure world within each ECU as the trusted auditor. As mentioned above, typical in-vehicle self-attestation solutions [13,35,38,41,42,52] will hash the entire firmware image to check its integrity against a signature provided by the OEM. This solution is usually viable when checking a *small* firmware image during boot (i.e. not runtime). However performing runtime attestation in this way over large firmware images will compromise the real-time requirement for in-vehicle ECUs [8,25]. In cloud storage applications, however, a highly efficient data checking solution known as spot checking has been used for some time [5,24] which challenges a small subset of the data periodically (Introduced in Section 2.1). Therefore, we can adapt the spot checking technique to the in-vehicle scenario, using the TEE as the auditor.

Self-repair. Once compromised firmware is detected, the next step is to quickly restore it to the latest benign firmware. Our insight is that the ECU firmware is typically stored on a flash memory medium [11,21] managed by an isolated Flash Memory Controller (FMC, introduced in Section 2.2). This isolation ensures that even if the ECU firmware is compromised, the FMC flash memory management software remains intact. Our key observation is that the software running on the FMC will usually perform out-of-place updates, conforming to the unique hardware nature of flash memory. This implies that any updates to the ECU firmware will only invalidate rather than delete the original code. Therefore, by manipulating the garbage collection strategy implemented by the FMC, this invalidated code can always remain on the flash memory medium. Then, we can efficiently restore the latest benign firmware by simply rolling back the invalidated code. To perform this repair, the FMC would need to be aware of the firmware corruption. Therefore, we develop a secure notification mechanism between the TEE and FMC.

2 Background

2.1 Remote Data Integrity Checking (RDIC)

Remote data integrity checking enables a *client* to check the integrity of data outsourced to any *storage provider*. Such RDIC schemes were originally introduced primarily for cloud storage applications, and the primary RDIC schemes are Provable Data Possession, (PDP) [4,5] or Proof of Retrievability (PoR) [24,46]. The essential idea is that rather than checking the entire data, the data are viewed as a collection of blocks, with a small random subset periodically selected for integrity checks. This approach is known as spot checking, and it is able to detect data corruptions with an arbitrarily high probability for a given amount of corruption over the entire data [4,5]. To enable such spot checking, verification tags are computed over each block. These tags may be constructed

in either a privately or publicly verifiable manner [5]. In the privately verifiable schemes, the same private key which generated the tags is used in data integrity verification. In the publicly verifiable schemes, the verification relies on a public key associated with the private key originally used in tag generation. In general, the publicly verifiable schemes are much more expensive in terms of the computations performed for both generating and verifying the integrity proof due to the use of asymmetric cryptographic primitives. After generating the tags, both these and the data are outsourced to the storage provider. The "setup phase" in an RDIC scheme involves both generation of the tags, and distribution of the tags and data. After the setup phase, the "verification phase" (in our adaptation, we call this the "attestation phase") starts, during which a client can issue challenges to the server, requesting a proof that a random subset of the data are stored correctly. In response, the storage provider will use the challenged data and tags to compute a proof. Importantly, the size of this proof is independent of the number of blocks challenged, significantly improving the efficiency of verification, and enabling the ability to aggregate proofs over any number of arbitrarily selected blocks. The proof is returned to the client, who can check its validity based on the maintained keys.

2.2 Flash Memory

Typically, in-vehicle computers are equipped with NAND flash memory to store their firmware image [11, 21]. This is used in vehicles due to its very high I/O throughput, which is necessary due to the real-time requirements. However, NAND flash has some unique hardware properties. NAND flash is organized into many contiguous storage chunks known as blocks, each containing a certain number of pages. The read and program (write) operations always occur over pages, while the erasure operation is over entire blocks. In order to perform a program operation, the encompassing block data should be erased. However, each block can only sustain a finite number of program/erase cycles before it becomes unreliable. Therefore, to manage this unique hardware nature, it is more economical to perform writes to a new page rather than the original data location. This is known as an *out-of-place updates* strategy. Upon a programming operation, this strategy results in the data for a constant logical location to end up in a different physical location. To manage this, the NAND flash also performs *address translation* by maintaining a dynamic mapping between the logical and physical locations. Another mechanism for relocating data is known as *wear leveling* which regularly swaps blocks to evenly distribute programming/erasure operations. Upon moving to a new physical location, the previous physical page is marked as invalid. Once a block is full of invalid pages, it can be removed, and so the *garbage collection* mechanism will eventually perform an erasure over it. These sophisticated NAND flash management operations require a firmware layer on top of the memory hardware. This firmware layer is usually either a flash translation layer or a flash file system. This firmware layer is implemented on a dedicated embedded processor known as the Flash Memory Controller (FMC). Essentially, the FMC is isolated from the host computer, such that if the host

OS is compromised, any software running in the FMC will remain intact. Additionally, the FMC presents a very limited read/write interface to the host OS, and so has a very limited attack surface.

2.3 Trusted Execution Environment (TEE)

Many processors today are equipped with a Trusted Execution Environment (TEE) such as Intel SGX, AMD SEV/SME/TSME, and ARM TrustZone. The TEE allows a sensitive application to run in a secure memory area, where the application code and data can be isolated at the hardware level. The secure memory area and execution state are known as the secure world, while all other memory and computation is known as the untrusted world. Typically an operating system is running in the untrusted world so we refer to this as the host OS. Typically a TEE enabled application has two components, a trusted application (TApp) running in the secure world, and an untrusted application (UApp) running on the Host OS. Therefore, the TApp component is protected even if the host OS is compromised. Additionally, there is an interface for UApp to call predefined functions on the TApp. Further, TEE enables a mechanism called sealing, where any data which needs to be persistently stored, such as keys, can be securely stored on the host device. Typically, in-vehicle ECUs use embedded systems processors, many of which are ARM based. Specifically, many ECUs [43–45, 55] are equipped with ARM Cortex-A or Cortex-M CPUs, which have TrustZone capabilities enabled [32, 33].

2.4 In-Vehicle Network Architecture

Today's in-vehicle electronics systems are typically organized into a domain based architecture [2], in which the Electronic Control Units (ECUs) are organized into *functional* domains, which are collections of functionally related ECUs. The ECUs in each domain are connected to each other via a shared CAN FD bus. CAN FD was introduced in 2012 as an extension of the original CAN. CAN is a protocol for communications between many nodes connected by two wires, where each message is broadcast to all other connected nodes. Compared to the original CAN protocol, CAN FD supports 64 byte messages in each data frame, and an increased throughput up to 5MB/s (CAN is capped at 1MB/s). Due to the broadcast nature of the CAN FD bus, all ECUs in a domain can freely broadcast messages to all other ECUs in their domain. At the "head" of each domain is a dedicated *gateway* unit [22]. Essentially, gateways are special ECUs which, rather than containing low-power embedded processors, are equipped with much more powerful processors. Gateways act as an intermediary between domains, providing the ability for inter-domain communications [22]. Additionally, gateways are equipped with capabilities to communicate with external networks for functions such as over-the-air updates. As mentioned in Section 1, an OBD-II port has been mandated in the US since 2009 [51], providing direct external physical access to the CAN FD bus. Due to the domain-based architecture, however, when communicating with a particular ECU through the

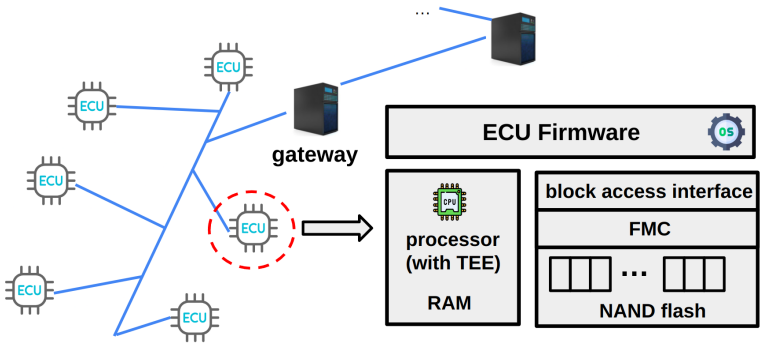

Fig. 1. System Model.

OBD-II port (e.g. during reprogramming), all messages are sent through a gateway before being forwarded to the target ECU. Recently, a zonal architecture has been introduced, where rather than being organized by functional domains, ECUs will be organized by geographical zones. In this model, the ECUs in each zone still share a CAN FD bus, and at the head of each zone is a zonal gateway. In this paper, when we refer to a "domain", it may be applied to either domains or zones, depending on the vehicle architecture.

3 Models and Assumptions

3.1 System Model

We consider a typical vehicle consisting of multiple ECUs connected to a CAN FD bus (later in the paper, we refer to this simply as CAN). The ECUs are separated into a few domains, where each ECU in a domain may communicate freely within that domain. Without loss of generality, our solution focuses mainly on interactions within a single domain. Our prototypical domain contains one gateway unit, which acts as an intermediary between domains. The typical ECUs are low power embedded systems, while the gateway contains far more powerful processing capabilities. Each ECU is equipped with a processor, RAM, and flash memory. The flash memory is managed by a flash memory controller (`FMC`) which runs software isolated from the host OS (Section 2.2). This software performs out-of-place updates to manage the unique hardware nature of flash memory. Additionally, the processor is equipped with a TEE on which a secure world is enabled. In the gateway, this can be on TrustZone, SGX, or SME/TSME, depending on the gateway CPU, while in the other ECUs, this will usually be TrustZone. In the secure world, we run a trusted application (`TApp`), which can directly communicate only with an untrusted application, (`UApp`) running in the normal world. `UApp` has kernel level privileges, which is typical in real-time operating systems [36]. Thus, `UApp` can directly access the read/write

interface provided by the FMC. Additionally, UApp can send communication messages over the CAN FD bus (via communication with the CAN controller). Note that a valid firmware update will always pass through the gateway before ECU reprogramming (Section 2.4).

In the gateway, TApp is equipped with a certificate from the OEM which can be established securely during vehicle manufacturing. Further, we assume that a valid firmware image is always equipped with a signature signed by the OEM private key, and that an adversary will not obtain this key, an assumption which is standard in self-attestation schemes [35, 41, 42, 52]. The existence of a shared secret key (K_d) between TApp in all ECUs in a domain is also assumed. Within each ECU, there is a key (K_s) and counter (γ) shared between TApp and FMC. The length of all keys is κ. All the shared secrets can be established securely during manufacturing.

3.2 Adversarial Model

The firmware of any in-vehicle computer may be compromised and misbehave. Specifically, an adversary may cause an ECU to enter into reprogramming mode via any means of connecting to the in-vehicle network (e.g. OTA-updates, diagnostic tools, remote OBD-II connection, etc.). During this time, the code of UApp which is stored in the flash memory will be modified. This may result in malicious behaviour such as disruptions to ECU operation or sending malicious CAN messages (*Adversary I*). Additionally, the malware is in between TApp and FMC, and so can arbitrarily manipulate the communications between them (*Adversary II*). The Adversary II may perform any of the following attacks which manipulate the communications between TApp and FMC: 1) Manipulation of message content during transmission. 2) Generating arbitrary message responses on the fly (without the knowledge of any secrets). 3) Attempt to imitate one of the trusted components (a spoofing attack). 4) Delay the communications. 5) Block the communications completely (a DoS attack). We do not consider an adversary which will shut down the TEE completely. Both Adversary I and II are computationally bounded.

3.3 Assumptions

Our design relies on a few assumptions: 1) the TEE is secure (Section 2.3), a common assumption for any TEE-based applications [12]. 2) The FMC is secure (Section 2.2). This assumption is also reasonable as the FMC is isolated from the OS by the storage hardware [54]. 3) The initial firmware on a given ECU is benign. 4) If there is no malware in the flash memory, then there is no malware in the RAM. This implies that if there is malware in RAM, then necessarily there is malware in the flash memory (this is the contrapositive). Note that the converse statement is not necessarily true. An implication of this assumption is that misbehaviour will always indicate malware is present in flash memory.

4 FSAVER

4.1 Design Overview

To ensure that the in-vehicle ECUs are always malware-free, we take an approach of explicitly checking the firmware contents. To do this, we use a self-attestation. Our self-attestation design improves on existing solutions by incorporating an efficient runtime checking by using a spot checking based RDIC scheme (Section 2.1). In particular, we employ `UApp` as the storage provider, and `TApp` as the client analogue, performing regular checks over the committed firmware "data". To enable an efficient RDIC scheme in the in-vehicle scenario, we carefully adapt the setup and attestation phases.

In addition to checking the firmware contents and so detecting malicious code, ensuring that the ECUs are malware-free requires some active response. To enable this, we use self-repair mechanisms within `FMC`. The first step towards self repairing is for `FMC` to become aware of a malicious presence through notification. Next, `FMC` will rollback to the latest benign firmware, which is enabled by ensuring that the old firmware data are maintained and locations are known. Next, a reboot is initiated so that the malware is replaced in memory by the benign firmware.

4.2 Self-attestation

Setup phase. As established above (Section 1), a typical ECU update strategy is to initiate a reprogramming mode by authenticating some flashing device [29] over CAN via a protocol like UDS[1]. When the flashing device is a physical diagnostic device, it is attached to the OBD-II port. With the rise of over-the-air updates [27], updates can now be sent directly to a gateway, which then acts as the flashing device, performing authentication with the target ECU.

Once authenticated, the flashing device can send arbitrary code to the ECU. This code may be malicious, so it is essential to check whether it was provided by the OEM. Towards achieving this, the most natural solution which incorporates RDIC is to use a publicly verifiable RDIC scheme. The OEM would generate publicly verifiable tags, and each `TApp` would store an OEM certificate to verify RDIC proofs. However, using a publicly verifiable scheme is much less efficient than a privately verifiable scheme (Section 2.1). Consequently, our design adapts a privately verifiable scheme. In order to do this, the key used in tag generation must also be used during attestation. However, if the OEM generates these tags (as in the publicly verifiable case), then its private key must be shared with all the ECUs. This would involve complex key management and potential security vulnerabilities. FSAVER addresses these challenges by leveraging the gateway generate tags before the firmware reaches the target ECU.

We observe first that the benign new firmware contents will always pass through the gateway before it is received by the target ECU (Section 2.4). In

[1] FSAVER does not modify UDS.

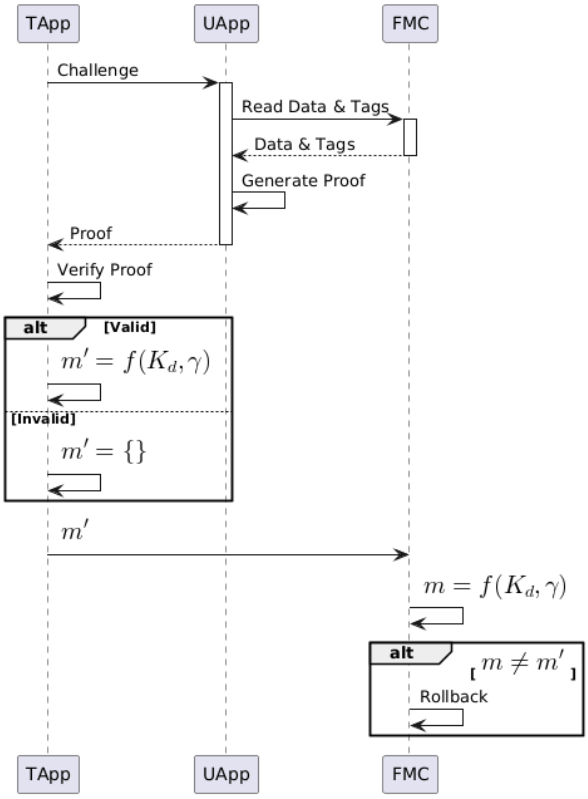

Fig. 2. A Sequence diagram for *attestation*, *notification* and *rollback*. Valid indicates no malware detected while invalid indicates malware. Note that the diagram uses "alt" (alternative) fragments to show different paths of execution based on conditions.

FSAVER, as in the existing solutions [13,35,38,41,42,52], the OEM will initially provide a signature over the new firmware. As the new firmware is passed through the gateway, `TApp` will verify the signature using its OEM certificate. In this way, the gateway `TApp` checks whether the received firmware is from the OEM. Next, the privately verifiable tags must be generated. The gateway `TApp`, having verified the firmware source, will generate these tags using K_d, the private key which is shared between all `TApp` in the domain. Next, the firmware, along with the generated tags, are written to the target ECU. Since the tags were generated using K_d, which is shared between `TApp` in all domain ECUs, the target ECU `TApp` is able to perform a self-attestation (as described below).

Sometimes, the target ECU may be the gateway itself. In this case, we observe that there are multiple gateways which communicate with one another via their network interfaces. Therefore, another connected gateway can perform the role typically designated to the domain gateway (i.e. considering the gateways together as a domain).

Attestation phase. Once the new firmware and privately verifiable tags are stored in the target ECU, the attestation phase will begin. We establish an epoch of duration T_{epoch} so that within each epoch, one spot checking self-attestation will occur. To perform such a attestation, `TApp` will act as the client in an RDIC scheme, randomly selecting which blocks will be challenged. This challenge will be generated close to the beginning of the epoch, as it is a very quick operation, and can typically be scheduled at any time. This quick challenge allows a duration close to T_{epoch} for the remainder of the attestation phase. This consists of `UApp` receiving the challenge and, acting as the storage provider, generating a proof that the challenged blocks are stored properly. Upon receiving the proof, `TApp` will perform verification. The verification results will either be valid or invalid, indicating that the firmware is benign or malicious respectively. This approach provides a wide time window within each epoch to perform proof generation and verification, allowing these processes to be flexibly scheduled without compromising the real-time requirements of the system.

4.3 Self-repair

Notification. Upon malware entering the ECU, `FMC`, which performs the self-repair, should be aware as soon as possible. To become aware of this, we have designed a notification checking mechanism which is activated each epoch in the `FMC`. To receive a notification, we rely on the key length κ, a security parameter l, as well as the key (K_s) and counter (γ) which is shared between `TApp` and `FMC`.

First, we define a pseudo random function (PRF) as $f : \{0,1\}^{\kappa} \times \{0,1\}^{*} \rightarrow \{0,1\}^{l}$. Upon computing self-attestation results, `TApp` will compute $m = f(K_s, \gamma)$ then increment γ. Next, if the attestation results are *valid*, `TApp` will write m to the flash storage at a location agreed upon with `FMC` (we call this location the "notification address"). If the results are *invalid*, `TApp` will write an arbitrary, unrelated value to the same location.

To receive this notification, `FMC` will have a built in hardware timer which triggers a timer interrupt at the end of each epoch. To handle this interrupt, `FMC` will also compute $m = f(K_s, \gamma)$ then increment γ. `FMC` then reads the value m' from the notification address. If m and m' match, then there is no malware present. Otherwise, `FMC` will know malware is present and so can initiate self-repair.

Rollback. Once `FMC` is aware of a malicious presence, `FMC` should be able to rollback to the previous benign firmware state. To enable this rollback, there are two challenges: 1) The old firmware should be present on the flash storage device, and 2) this data should be easily located.

Essentially, solving *challenge 1* requires modifying the garbage collection policy, which typically would erase the old firmware data soon after it is invalidated (Section 2.2). Our solution is to coordinate our garbage collection strategy in time with ECU reprogramming, so that we can ensure any activated rollback will always be *from* malware *to* the latest benign firmware. In FSAVER, garbage collection is disabled when the original firmware is invalidated, after a new firmware

image is written. This preserves the old firmware for rollback. Garbage collection is re-enabled for the invalidated firmware only when the new firmware is invalidated and replaced by another update, allowing old firmware versions to be replaced in sequence.

Before GC is re-enabled on the invalidated firmware, however, it is crucial to ensure that it does not erase the latest benign firmware. To do this, we need to determine whether the active firmware or the invalidated firmware (i.e., the previous version) is the latest benign version before proceeding with reprogramming. This is verified through a self-attestation over the active firmware. There are two possible self-attestation results:

1. **Valid Result:** If the self-attestation confirms that the current firmware is benign, garbage collection is enabled before reprogramming. This allows the old firmware data to be reclaimed.
2. **Invalid Result:** If the self-attestation indicates that the active firmware is compromised, a self-repair is initiated to restore the previous firmware, which is ensured to be the latest benign version (Section 5.2). After restoring the firmware, garbage collection is enabled to remove the identified malware before reprogramming.

In both scenarios, the latest benign firmware is overwritten during subsequent reprogramming. This ensures that the firmware to be restored upon activating rollback is always the latest verified benign version. A proof of this is given is Section 5.2.

Solving *challenge 2* requires extending the address translation policy, which upon writing the new firmware would typically *replace* the old logical to physical mapping with the new one (Section 2.2). In order to restore the old firmware, the old mapping simply needs to be maintained. To ensure this, we reserve some space as a backup mapping table.

Recall that prior to reprogramming, we ensured (in solving challenge 1) that the active firmware is the latest benign version (i.e., the version to which we want to roll back). Thus we can simply save the mapping table in this state prior to reprogramming to maintain its location. Since the corresponding data is maintained as established in challenge 1, these mappings will continue to point to the correct firmware. Therefore, upon restoring this mapping table, the latest benign firmware will also be restored.

Reboot. Upon rolling back to the latest benign firmware in the flash storage, malware will still be contained in the ECU RAM and be running on the CPU. To remove the malware from RAM and run the benign firmware again, the target ECU should simply reboot. Fortunately, this reboot can easily be initiated by `TApp`. However, in order to always initiate this reboot, `TApp` should always be aware of when rollback is completed by `FMC`. When self-attestation verification returns a invalid result, `TApp` can always be assured that rollback will be performed. Thus, in this case, `TApp` can initiate a reboot after the epoch expires, since by this time, `FMC` would have checked the results and performed rollback. However, there is a case where self-attestation verification returns a valid result, yet there is still malware present, and rollback is subsequently performed by `FMC`

(see Section 5) while TApp is unaware. To make TApp aware of this, TApp will always read from the notification address at then end of each epoch. TApp will then check whether m', the value read, matches m, the previously generated shared random value. If it does match, then the special case did not occur. Contrarily, if it does not match, then this case did occur and a reboot should be performed. A safety-focused discussion on when this reboot should be performed is given in Section 7.2.

5 Security Analysis

5.1 Self-attestation

Setup. This is the phase during which an adversary would write malicious firmware to the target ECU. The aim would be to perform this writing so that malware is not detected by the self-attestation. To perform such an attack, tags would need to be generated so that verification of a self-attestation proof using K_d would always return valid results. We show this to be infeasible through the two possible attacker strategies:

1. The attacker obtains the OEM private key (with key length κ) and uses this to sign the malware. Subsequently, the attacker will enter reprogramming mode with the target ECU and transmit the malware through the gateway. Then, the gateway will verify the signature and generate tags using K_d. Thus, if the adversary can obtain the OEM private key, then this attack can be successfully performed. Assuming that the OEM will properly secure this key, the best strategy is for the attacker to randomly guess it with probability $\frac{1}{2^\kappa}$.
2. The attacker obtains the key K_d (with key length κ) and uses this to generate tags for the malware. Subsequently, the attacker will enter reprogramming mode either 1) directly with the target ECU, bypassing the gateway which is typically involved in the ECU reprogramming, or 2) through a compromised gateway, which will forward malware to the ECU regardless of signature verification results. Thus, if the adversary can obtain K_d *and* bypass the gateway, then this attack can be successfully performed. Since the TEE sealing mechanism (Section 2.3) will secure the storage of K_d on the target ECU, so that it is inaccessible to the adversary, the best strategy is for the attacker to randomly guess K_d with probability $\frac{1}{2^\kappa}$.

Attestation. Periodically, TApp will challenge a random subset of the firmware blocks. For each attestation, the probability of successfully selecting "corrupted" blocks to challenge is at least $P_m = 1 - (1 - \beta)^c$ [5], where c is the number of blocks being challenged, and β is the proportion of blocks containing at least one bit of corruption. For α attestations, this probability is $P_{\text{detect}} = 1 - (1 - P_m)^\alpha$. As an example, when $c = 100$, $\beta = .01$, and $\alpha = 5$, $P_{\text{detect}} = .993$ [2].

[2] Here, there is a design trade-off between the number of blocks and the number of audits performed to detect a given β proportion of malware. For smaller c, it will

Upon challenging "corrupted" blocks, several adversarial responses by UApp are possible. Specifically, five different methods may be used by the Adversary II, as outlined in Section 3:

1. For method 1, UApp may attempt to modify the challenged data so that it can pass the challenge. However, UApp does not have access to K_d, required to generate valid tags on the fly. Additionally, UApp does *not* have immediate access to the previous firmware content or tags, which may be used to pass an attestation. Therefore, the best strategy is to guess K_d with probability $\frac{1}{2^\kappa}$, then generate tags for the malware. This guessing is infeasible for a computationally bounded adversary, especially within duration T_{epoch}.
2. For method 2, UApp will attempt to generate some data or tags which will result in computing a valid proof. This is again not feasible for a computationally bounded adversary, which cannot access K_d.
3. For method 3 (spoofing attack), there is no sensible attack.
4. For method 4 and 5, a response will be delayed greater than duration T_{epoch}. This will result in TApp generating a invalid verification result.

Due to the infeasible nature of performing these attacks within the capabilities of the Adversary II, challenging "corrupted" blocks will always result in a invalid verification result within TApp. In all these cases, TApp is assured that there is malware present, since misbehaviour will always indicate malware is present in flash memory (Assumption 4 in Section 3).

5.2 Self-repair

Notification. When the timer interrupt is triggered in FMC, it will compute $m = f(K_s, \gamma)$ and compare this with the value m' contained at the notification address. If m does not match m', then FMC will automatically trigger rollback. The security goal is that upon a invalid verification result from TApp, rollback will always occur. Upon a valid verification result, we ensure that rollback is not performed *unless malware is present in the ECU*. There are three cases to consider:

Case 1: Invalid verification result, with malware present. In this case, TApp will not reveal the value m to UApp. We consider the Adversary II, which may employ five different methods, as outlined in Section 3. Using method 1, 2, or 3, UApp would need to generate m. The best strategy is to guess both K_s and γ, then compute m using f. The probability of correctly guessing K_s, however is $\frac{1}{2^\kappa}$, which is negligibly small for sufficiently large κ. Using methods 4 and 5, the timer interrupt in FMC would trigger before any value is written to the notification address. Therefore the value m' which is read will not match m.

Case 2: Valid verification result, with malware present. In this case, TApp will reveal the value m to an adversarial UApp. We again consider the

take more attestations (a higher α) to detect malware, but each attestation will be more efficient, so that the real-time requirement is met.

Adversary II. If the adversary does not manipulate the writing of m to the notification address, then `FMC` will not perform rollback. However, `UApp` does not always know whether `TApp` computes valid or invalid verification results, and so may still manipulate the communications between `TApp` and `FMC`. Using methods 1, 2, and 3, the value m' written to the notification address will very likely be different than the value m provided by `TApp`. Therefore, `FMC` will perform rollback. Using methods 4 or 5, the timer interrupt in `FMC` would again trigger before any value is written to the notification address. Therefore the value m' which is read will not match m, and rollback will be triggered.

Case 3: Valid verification result, no malware present. In this case there is no adversary present, so the correct notification $m' = m = f(K_s, \gamma)$, generated by `TApp`, will always be written to the correct location. Therefore, a rollback will never be triggered in this case.

Rollback. To demonstrate the security of our firmware rollback mechanism, we prove the following theorem:

Theorem 1. *For any firmware version P_{n+1}, if P_{n+1} is identified as malicious, then the rollback mechanism will restore the system to P_n, which is the latest benign version prior to P_{n+1}.*

Proof. We have established in the above analysis that under our adversarial model, if P_{n+1} is identified as malicious, then the rollback mechanism will be triggered. First, we show that the last active firmware (malicious or benign) P_n can always be restored to. That is, that through our garbage collection management and address translation management, the data and mappings for the previous firmware version P_n can always be restored (i.e the data and mappings for P_n are always present when P_{n+1} is active).

We can prove this by induction. The initial firmware update is from P_0 to P_1. Our garbage collection polity is that before this reprogramming GC is enabled. However, there is no previous firmware version to reclaim, and so nothing occurs. GC is disabled for any new writes, and so P_0 is maintained. Additionally, by copying the mapping table prior to the update, the locations for the firmware P_0 are maintained. Therefore, P_0 can always be restored while P_1 is active.

Now, suppose that for P_k such that $1 < k \leq n$, the firmware can be rolled back to P_{k-1}. Prior to updating from firmware version P_n to P_{n+1}, GC is enabled on firmware version P_{n-1}, and so it may be reclaimed. Again, as GC is disabled for new writes, the data for P_n is maintained when P_{n+1} is written. Additionally, as the mapping table for P_n is saved prior to writing P_{n+1}, the locations can be restored.

Now, we have to show that P_n, the prior firmware version, which is rolled back to, is always the latest benign version. This proof is by induction. By our assumption 3 (Section 3), the initial firmware version P_0 is always benign. Now, assume that for any firmware version P_k, where $0 < k \leq n$, that if P_k is identified as malicious, then P_{k-1} is the latest benign version prior to P_k. Now, suppose that the firmware version P_{n+1} is identified as malicious and so rollback is initiated. Note that before writing P_{n+1} to the flash memory, a self-attestation was performed over P_n. If the verification results were invalid, then

P_{n-1} was restored to replace P_n, which, by the inductive hypothesis, is the latest benign firmware. If the results were valid, then P_n was already the latest benign firmware. Therefore, in both cases, P_n, at the time of rollback, is the latest benign firmware version. □

Reboot. Essentially, in order to effectively perform the rebooting after rollback, this will be triggered by `TApp`. Since we have established that rollback will occur only when there is malware present, it is simply required that `TApp` is always aware when rollback is performed. There are two cases when rollback may be performed:

1. **When a self-attestation verification returns invalid results:** In this case, `TApp` is aware of the invalid results because it is directly performing the self-attestation verification. `TApp` can initiate a reboot after the epoch expires, knowing that rollback has occurred.
2. **When a self-attestation verification returns valid results:** In this case, rollback can still occur due to other issues, such as message manipulation by an adversary (Adversary II) which is not *yet* detected via spot-checking. In this scenario, recall that `TApp` checks the value m' read from the notification address at the end of each epoch. If m' differs from the previously generated value m, it indicates that `FMC` has detected a problem and performed rollback. Therefore, `TApp` can still initiate a reboot even when the verification results were valid, but rollback has been triggered due to external manipulation.

This demonstrates that FSAVER ensures that `TApp` is aware of all rollback scenarios and can initiate a reboot as needed. For a discussion on the timing and implications of this reboot, refer to Section 7.2.

6 Implementation and Evaluation

Implementation. We have implemented the self-attestation component (including setup and attestation phases) in real-world hardware. As an ECU, we used a Raspberry Pi 3B+ [1] (With 1.4GHz 64-bit quad-core ARM Cortex-A53 CPU, and 1GB LPDDR2 SDRAM) with TEE enabled. We implemented `UApp`, which will receive the challenge from `TApp` and compute an RDIC proof, under the host OS Raspbian Stretch. `TApp`, which will generate the challenge and verify the received RDIC proof, was implemented by porting OP-TEE [39] (Open Portable Trusted Execution Environment) to the Raspberry Pi via the Raspbian-TEE open source project [6]. For our RDIC scheme, we used the privately verifiable scheme from Compact PoR [46]. Additionally, we have implemented a rollback mechanism on the USB header development prototype board LPC-H3131 [34] (with ARM9 32-bit ARM926EJ-S, 180Mhz, 32MB of SDRAM, and 512MB NAND flash). We have ported [48] the open source NAND flash manager OpenNFM [10] to the LPC-H3131, and verified the feasibility of a rollback based self-repair mechanism.

Procedure	Component	Throughput (KiB/s)
Setup (one time)	Gateway `TApp`	22.04
Challenge Generation	ECU `TApp`	363.64
Proof Generation	ECU `UApp`	28.52
Proof Verification	ECU `TApp`	15.43
Rollback (Mapping Restoration)	ECU `FMC`	1724

Table 1. Average throughput (KiB/s) for setup, attestation, and rollback.

Evaluation. To evaluate the efficiency of the self-attestation component in both `TApp` and `UApp`, we timed the setup and attestation phases. For evaluating self-repair, we timed the rollback phase. We present our results in Table 1 as throughput rates. This table demonstrates the amount of data which can be attested, and mapping content which can be rolled back in given time.

For the setup phase, we timed the generation of privately verifiable tags in the gateway `TApp`. Note that this is a "one-time" operation, occurring only when a new firmware update is received. Additionally, while our implementation uses the Raspberry Pi 3B+, gateway units typically contain much more powerful processing capabilities. For the attestation phase, we have timed three distinct phases: 1) The challenge generation (`TApp`), 2) the proof generation (`UApp`), and 3) the proof verification (`TApp`). As anticipated (Section 4), we observe that the challenge generation throughput is significantly higher than the other procedures. This is because the computation is limited to generating a few random numbers. Additionally, we observe that the proof verification is slower than the proof generation despite the proof verification being a more lightweight computation. We suspect this is because the proof generation is run on the host OS, while verification is run in the TrustZone secure world. The TrustZone TEE is generally much slower than the host OS, and the Raspberry Pi is not optimized for the use of TrustZone. In general, our attestation results demonstrate that a small spot checking based self-attestation would be quickly performed, since the amount of data being checked is small.

For the rollback phase of self-repair, we have timed the restoration of the backup mapping table. The observed mapping restoration throughput of 1724KiB/s results in an associated *data* restoration rate of 836MiB/s. This is very quick, considering that the typical ECU firmware size is much less than 20MB [14].

These results also effectively demonstrate the scalability of FSAVER. Specifically, we have demonstrated above (Section 5) that by using the spot checking-technique, checking a small constant number of blocks will have the same probability to detect a given proportion of firmware corruption, regardless of the overall firmware size. Therefore, the attestation cost can be constant for *any* firmware size. In contrast, the rollback phase scales linearly with the firmware size. However, by only restoring *mappings*, we scale the *data* restoration speeds significantly, enabling us to restore any reasonably sized ECU firmware very quickly.

7 Discussion

7.1 Limitations

During the *setup* phase for self-attestation, FSAVER first requires the gateway to compute a signature over a hash of the entire firmware image. However, doing this requires the full firmware image to be contained in gateway memory at one time. Due to memory constraints, this may not always be feasible. To mitigate this, this authentication may use a hash chain rather than a single hash over the complete data. As each block is processed, its tags can immediately be generated and stored in gateway memory while the block's contents are sent to the target ECU. Only if the authentication passes will the complete set of tags be released.

Another limitation of FSAVER is that the adversary may find some way to shut down the TEE, and thus disable `TApp`. If this occurs, `FMC` will still be aware (since an invalid m' will be present at the notification address), but `TApp` will not be able to perform the *reboot* step upon self-repair. Thus, the only way to clear the ECU memory and restore functionality would be to manually reboot the ECU. This is why we cannot consider the adversary which can shut down the TEE (Section 3).

It may be beneficial to notify either other ECUs or the driver of failed attestations prior to repair (Section 7.2), so that they can respond and be aware of the full vehicle state. However, under our solution, the Adversary II may disrupt any communication between `TApp` and the CAN bus. Therefore it is in general not feasible to perform this communication *before* self-repair is completed.

Due to these limitations inherent to most self-attestation approaches, we may in the future investigate a decentralized firmware attestation solution [26] which can establish external awareness of attestation results, and is resilient to the adversary which can shut down the TEE.

7.2 Real-time vs Delayed Rebooting

While FSAVER technically enables almost immediate ECU restoration, it may be unsafe to do this during vehicle operation. This is because rebooting the ECU may cause undefined behaviour while driving, as usually the vehicle is idle when ECUs are powered on. This undefined behaviour may cause safety issues such as crashing the vehicle or stopping it in the middle of the road. Due to these safety concerns, Andréasson et al. [3] has suggested a comprehensive strategy: 1) always notifying the user upon malware detection, so that *in certain cases,* they can stop the vehicle while repair proceeds. 2) Upon detecting malware in non safety-critical ECUs, immediately performing repair while still informing the driver. 3) For safety-critical ECUs, wait until the vehicle is completely stopped before performing repair. In this case, the user would initiate repair when he/she feels it is safe. We believe this is a reasonable approach, though as noted above, an inherent limitation of self-attestation schemes is that they cannot notify other ECUs of the detected malware under the Adversary II. This makes it infeasible to reliably notify the user of malware detection. In the future, we may investigate a solution which enables such external communication.

8 Related Works

8.1 In-vehicle Firmware Self-Attestation

Secure Firmware Flashing. In 2008, Weimerscirch [52] developed a secure software flashing strategy for in-vehicle ECUs. In their strategy, the bootloader will receive a certificate, then the new firmware will be flashed. During flashing, a hash is incrementally computed over each block and a signature provided by the OEM is checked. Nilsson et al. [38] introduce a distinction between control and functional systems within an ECU. The control system will manage the flashing procedure and will check that the firmware contents were provided by a trusted "portal". This attestation is accomplished via computing a hash chain with an incorporated challenge value, which is checked against a provided "verification code".

Secure Boot. Towards checking the firmware contents after the initial flashing, Gui et al. [13] developed a hardware based root of trust for ECUs, which includes secure boot component. This secure boot component largely relies on the maintenance of "golden measurements" of the trusted code. The ECU firmware is checked against these golden measurements before booting to ensure that the trusted code will run. Indeed, such secure boot solutions, where *upon each boot* a bootloader which will check the entire ECU firmware using cryptographic signatures is in widespread use by the automotive industry today [40, 42].

Runtime Self-attestation. The above solutions [13, 35, 40, 42, 52] all rely on checking a digest over the entire firmware image, and so are not efficient enough to meet the in-vehicle real time requirements for a runtime attestation [8, 25]. However, an efficient runtime attestation solution is desired [13, 52]. In 2023, Kaster et al. introduced sliced secure boot [25], designed for in-vehicle ECUs. This solution uses a Hardware Security Module (HSM) to compute "re-usable fingerprints" over the firmware image, after the download has been authenticated. Using these "fingerprints", the HSM can challenge "slices" of the firmware image at runtime, using a Cipher Block Chaining Message Authentication Code (CBC-MAC) for authentication. Unlike the bootloader, the HSM is always available at runtime, allowing for runtime checking. The efficiency of their solution relies on a similar technique as spot-checking; by checking one "slice" at a time, the verification can be made much more efficient. Compared to FSAVER, sliced secure boot introduces the HSM as an added hardware component, while our solution uses the TEE, an existing trusted hardware component within ECUs native CPU (Section 2.3). Additionally, by using a publicly verifiable spot-checking RDIC based attestation (Section 2.1), FSAVER replaces the expensive cryptographic operations used by a CBC-MAC with simple linear combinations [46]. Further, sliced secure boot provides no explicit ECU restoration mechanism.

8.2 In-Vehicle ECU Restoration

In cyber-physical systems such as vehicles, a quick repair mechanism is desirable [8, 35, 38], which the above solutions do not provide. In 2015, Mansor et

al. [35] developed a mechanism which, in addition to verifying an OEM signature upon writing new firmware to the ECU, enabled temporarily storing the old firmware version on a "central communication unit". This only allows restoration immediately after flashing and is not adaptable to a secure boot or runtime attestation solution. Towards providing explicit ECU restoration mechanisms in conjunction with CAN intrusion detection [16, 18, 20, 57] mechanisms, Kwon et al. (2018) introduced a design to reconfigure ECUs and limit their behaviour upon detection of malware [30]. The solution was to send a CAN message to a suspected ECU which initiates a generic "safe mode" and reboot. Additionally, all benign ECUs would ignore certain messages sent from the ECU to cause harm. In 2023, Dafoe et al. extended this idea by relying on the unique nature of flash storage to coordinate a repair between the TrustZone and flash firmware [23]. Specifically, upon detection of a malicious ECU via intrusion detection, a CAN message would be sent to the target ECU and firmware rollback would be initiated. Similar to FSAVER, this firmware rollback was enabled by altering the garbage collection strategy within the flash memory. Different from [23], FSAVER does not rely on CAN based intrusion detection, which can be unreliable, and usually assumes a trusted detection ECU. FSAVER is based upon a highly efficient runtime self-attestation, which is far more reliable. Additionally, [23] fails to protect against the Adversary II and so cannot guarantee successful ECU restoration.

9 Conclusion

In this work, we have designed FSAVER, a self-attestation and self-repair scheme for the in-vehicle scenario which is highly efficient, adapting to the real-time requirement of any cyber-physical system. Due to the increased efficiency, we are able to perform a runtime attestation using the TEE as a trusted auditor. For self-repair, we enable immediate rollback to the latest benign firmware version by incorporating the isolated flash memory controller into our design. We have implemented a prototype of the self-attestation and rollback mechanisms, and have effectively demonstrated its viability.

Acknowledgments

This work was supported by US National Science Foundation under grant number 2225424-CNS and 2043022-DGE.

References

1. Raspberry pi 3 model b+. https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/
2. Aberl, P.: How a zone architecture paves the way to a fully software-defined vehicle (2023), https://api.semanticscholar.org/CorpusID:260712942

3. Andréasson, E., Lyesnukhin, I.: Device attestation for in-vehicle network (2022)
4. Ateniese, G., Burns, R., Curtmola, R., Herring, J., Khan, O., Kissner, L., Peterson, Z., Song, D.: Remote data checking using provable data possession. ACM Transactions on Information and System Security (TISSEC) **14**(1), 12 (2011)
5. Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., Song, D.: Provable data possession at untrusted stores. Cryptology ePrint Archive, Paper 2007/202 (2007), https://eprint.iacr.org/2007/202, https://eprint.iacr.org/2007/202
6. benhaz1024: Raspbian with op-tee support. https://github.com/benhaz1024/raspbian-tee, accessed: 2024-07-15
7. Bielawski, R., Gaynier, R., Ma, D., Lauzon, S., Weimerskirch, A.: Cybersecurity of firmware updates. Technical Report DOT HS 812 807, University of Michigan. Transportation Research Institute and University of Michigan, Dearborn and Volkswagen Group of America (Herndon, VA) (10 2020). https://doi.org/10.21949/1530213, corporate contributor: United States. Department of Transportation. National Highway Traffic Safety Administration. Office of Vehicle Safety Research
8. Cárdenas, A.A., Amin, S., Sinopoli, B., Giani, A., Perrig, A., Sastry, S.: Challenges for securing cyber physical systems (2009), https://api.semanticscholar.org/CorpusID:13643850
9. 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 Transactions on Vehicular Technology **67**(6), 4757–4770 (2018). https://doi.org/10.1109/TVT.2018.2810232
10. Code, G.: Opennfm. https://code.google.com/p/opennfm/
11. Electronic Products: Memory use in automotive (2024), https://www.electronicproducts.com/memory-use-in-automotive/, accessed: 2024-06-23
12. Guan, L., Jia, S., Chen, B., Zhang, F., Luo, B., Lin, J., Liu, P., Xing, X., Xia, L.: Supporting transparent snapshot for bare-metal malware analysis on mobile devices. In: Proceedings of the 33rd Annual Computer Security Applications Conference. pp. 339–349 (2017)
13. Gui, Y., Siddiqui, A.S., Saqib, F.: Hardware based root of trust for electronic control units. In: SoutheastCon 2018. pp. 1–7 (2018). https://doi.org/10.1109/SECON.2018.8479266
14. Gupta, S.: The role of phase-change memory in automotive ota firmware upgrades. Embedded.com (September 2023), https://www.embedded.com/the-role-of-phase-change-memory-in-automotive-ota-firmware-upgrades/, accessed on September 8, 2024
15. Hackenberg, R., Weiss, N., Renner, S., Pozzobon, E.: Extending vehicle attack surface through smart devices (09 2017)
16. Hamada, Y., Inoue, M., Ueda, H., Miyashita, Y., Hata, Y.: Anomaly-based intrusion detection using the density estimation of reception cycle periods for in-vehicle networks. SAE International Journal of Transportation Cybersecurity and Privacy **1** (05 2018). https://doi.org/10.4271/11-01-01-0003
17. Hammerschmidt, C.: Number of automotive ecus continues to rise. eeNews Europe (2019), https://www.eenewseurope.com/en/number-of-automotive-ecus-continues-to-rise/, accessed: 2024-06-20
18. Han, M.L., Kwak, B.I., Kim, H.K.: Anomaly intrusion detection method for vehicular networks based on survival analysis. Vehicular Communications **14**, 52–63 (2018). https://doi.org/https://doi.org/10.1016/j.vehcom.2018.09.004, https://www.sciencedirect.com/science/article/pii/S2214209618301189

19. den Herrewegen, V., Garcia: Beneath the bonnet: A breakdown of diagnostic security. In: Lopez, Zhou, Soriano (eds.) Computer Security. pp. 305–324. Springer International Publishing, Cham (2018)
20. Hoppe, T., Kiltz, S., Dittmann, J.: Applying intrusion detection to automotive it-early insights and remaining challenges. Journal of Information Assurance and Security (JIAS) **4**, 226–235 (01 2009)
21. Inc., P.: Autonomous vehicle data storage (2024), https://premioinc.com/pages/autonomous-vehicle-data-storage, accessed: 2024-06-23
22. Instruments, T.: Processing the advantages of zone architecture in automotive. Tech. rep. (April 2023), https://www.ti.com/document-viewer/lit/html/SSZT211#:~:text=A%20zone%20architecture%20organizes%20the,module%20to%20manage%20network%20traffic, accessed: 2024-07-04
23. Josh, D., Harsh, S., Niusen, C., Bo, C.: Enabling real-time restoration of compromised ecu firmware in connected and autonomous vehicles. In: Yu, C., Chung-Wei, L., Bo, C., Qi, Z. (eds.) Security and Privacy in Cyber-Physical Systems and Smart Vehicles. pp. 15–33. Springer Nature Switzerland, Cham (2024)
24. Juels, A., Kaliski, B.S.: Pors: proofs of retrievability for large files. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. p. 584–597. CCS '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1315245.1315317, https://doi.org/10.1145/1315245.1315317
25. Kaster, R., Ma, D., Behl, A., Bakalarczyk, B.: Sliced secure boot. In: 19^{th} escar Europe : The World's Leading Automotive Cyber Security Conference (Konferenzveröffentlichung). sampled secure boot with re-usable fingerprints (2021). https://doi.org/10.13154/294-8354
26. Khodari, M., Rawat, A., Asplund, M., Gurtov, A.: Decentralized firmware attestation for in-vehicle networks. In: Proceedings of the 5th on Cyber-Physical System Security Workshop. p. 47–56. CPSS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3327961.3329529, https://doi.org/10.1145/3327961.3329529
27. Kim, B., Park, S.: Ecu software updating scenario using ota technology through mobile communication network. In: 2018 IEEE 3rd International Conference on Communication and Information Systems (ICCIS). pp. 67–72 (2018). https://doi.org/10.1109/ICOMIS.2018.8645019
28. Klinedinst, D.: On board diagnostics: Risks and vulnerabilities of the connected vehicle. Carnegie Mellon University, Software Engineering Institute's Insights (blog) (Aug 2016), https://insights.sei.cmu.edu/blog/board-diagnostics-risks-and-vulnerabilities-connected-vehicle/, accessed: 2024-Jun-21
29. Kulandaivel, S., Jain, S., Guajardo, J., Sekar, V.: Candid: A stealthy stepping-stone attack to bypass authentication on ecus. ACM J. Auton. Transport. Syst. (apr 2024). https://doi.org/10.1145/3657645, https://doi.org/10.1145/3657645, just Accepted
30. Kwon, H., Lee, S., Choi, J., Chung, B.h.: Mitigation mechanism against in-vehicle network intrusion by reconfiguring ecu and disabling attack packet. In: 2018 International Conference on Information Technology (InCIT). pp. 1–5 (2018). https://doi.org/10.23919/INCIT.2018.8584882
31. Lauser, T., Krauß, C.: Formal security analysis of vehicle diagnostic protocols. In: Proceedings of the 18th International Conference on Availability, Reliability and Security. ARES '23, Association for Computing Machinery, New York, NY,

USA (2023). https://doi.org/10.1145/3600160.3600184, https://doi.org/10.1145/3600160.3600184
32. Ltd., A.: Trustzone for cortex-a. https://www.arm.com/technologies/trustzone-for-cortex-a (2024), accessed: 2024-06-23
33. Ltd., A.: Trustzone for cortex-m. https://www.arm.com/technologies/trustzone-for-cortex-m (2024), accessed: 2024-06-23
34. Ltd., O.: Lpc-h3131. https://www.olimex.com/Products/ARM/NXP/LPC-H3131/, accessed: June 30, 2023
35. Mansor, H., Markantonakis, K., Akram, R.N., Mayes, K.: Don't brick your car: Firmware confidentiality and rollback for vehicles. In: 2015 10th International Conference on Availability, Reliability and Security. pp. 139–148 (2015). https://doi.org/10.1109/ARES.2015.58
36. u/Head Measurement1200: I am coming from developing in linux. i was wondering if my analogy is right that bare-metal programming is like operating in 'kernel' mode the whole time? https://www.reddit.com/r/embedded/comments/ug3kau/i_am_coming_from_developing_in_linux_i_was/ (2022), accessed: 2024-07-14
37. Murvay, P.S., Groza, B.: Source identification using signal characteristics in controller area networks. IEEE Signal Processing Letters **21**(4), 395–399 (2014). https://doi.org/10.1109/LSP.2014.2304139
38. Nilsson, D.K., Sun, L., Nakajima, T.: A framework for self-verification of firmware updates over the air in vehicle ecus. In: 2008 IEEE Globecom Workshops. pp. 1–5 (2008). https://doi.org/10.1109/GLOCOMW.2008.ECP.56
39. OP-TEE: Op-tee documentation. https://optee.readthedocs.io/en/latest/general/about.html, accessed: July 14, 2024
40. Ring, M., Frkat, D., Schmiedecker, M.: Cybersecurity evaluation of automotive e/e architectures. In: ACM Computer Science In Cars Symposium (CSCS 2018). vol. 92 (2018)
41. Sanwald, S., Kaneti, L., Stöttinger, M., Böhner, M.: Secure boot revisited: Challenges for secure implementations in the automotive domain. SAE International Journal of Transportation Cybersecurity and Privacy **2**(2), 69–81 (2019). https://doi.org/10.4271/11-02-02-0008, also appears in SAE International Journal of Transportation Cybersecurity and Privacy-V128-11EJ
42. Schrotter, M.: Understanding and designing an automotive-like secure bootloader. Regensburg Applied Research Conference (2020)
43. Semiconductors, N.: S32g3 vehicle networking reference design. https://www.nxp.com/design/designs/s32g3-vehicle-networking-reference-design:S32G-VNP-RDB3 (2024), accessed: 2024-06-23
44. Semiconductors, N.: S32k3 automotive telematics box (t-box) reference design board. https://www.nxp.com/design/designs/s32k3-automotive-telematics-box-t-box-reference-design-board:S32K3-T-BOX (2024), accessed: 2024-06-23
45. Semiconductors, N.: S32z and s32e real-time processors. https://www.nxp.com/products/processors-and-microcontrollers/s32-automotive-platform/s32z-and-s32e-real-time-processors:S32Z-E-REAL-TIME-PROCESSORS (2024), accessed: 2024-06-23
46. Shacham, H., Waters, B.: Compact proofs of retrievability. vol. 26, pp. 90–107 (12 2008). https://doi.org/10.1007/978-3-540-89255-7_7
47. Solutions, B.M.: Airbag control unit (2024), https://www.bosch-mobility.com/en/solutions/control-units/airbag-control-unit/, accessed: 2024-06-20

48. Tankasala, D., Chen, N., Chen, B.: Creating a testbed for flash memory research via lpc-h3131 and opennfm-linux version. Tech. rep., Technical report, Department of Computer Science, Michigan Tech (2022)
49. Team-BHP: Technically understanding airbag systems & srs (2017), https://www.team-bhp.com/forum/road-safety/189259-technically-understanding-airbag-systems-srs.html, accessed: 2024-06-20
50. Thing, V.L., Wu, J.: Autonomous vehicle security: A taxonomy of attacks and defences. In: 2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). pp. 164–170 (2016). https://doi.org/10.1109/iThings-GreenCom-CPSCom-SmartData.2016.52
51. U.S. Environmental Protection Agency: Final Rule for Control of Air Pollution from New Motor Vehicles and New Motor Vehicle Engines. EPA Regulation: Emissions from Vehicles and Engines (2009), Available at: https://www.epa.gov/regulations-emissions-vehicles-and-engines/final-rule-control-air-pollution-new-motor-vehicles-and
52. Weimerskirch, A.: Secure software flashing. SAE International Journal of Passenger Cars - Electronic and Electrical Systems **2**(1), 83–86 (2009). https://doi.org/10.4271/2009-01-0272, https://doi.org/10.4271/2009-01-0272, also in: SAE International Journal of Passenger Cars - Electronic and Electrical Systems-V118-7, SAE International Journal of Passenger Cars - Electronic and Electrical Systems-V118-7EJ
53. Wen, H., Chen, Q.A., Lin, Z.: Plug-N-Pwned: Comprehensive vulnerability analysis of OBD-II dongles as a new Over-the-Air attack surface in automotive IoT. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 949–965. USENIX Association (Aug 2020), https://www.usenix.org/conference/usenixsecurity20/presentation/wen
54. Xie, W., Chen, N., Chen, B.: Enabling accurate data recovery for mobile devices against malware attacks. In: International Conference on Security and Privacy in Communication Systems. pp. 431–449. Springer (2022)
55. Xilinx: Zynq ultrascale+ mpsoc zcu104 evaluation kit. https://www.xilinx.com/products/boards-and-kits/zcu104.html (2024), accessed: 2024-06-23
56. Yang, Y., Duan, Z., Tehranipoor, M.: Identify a spoofing attack on an in-vehicle can bus based on the deep features of an ecu fingerprint signal. Smart Cities **3**(1), 17–30 (2020). https://doi.org/10.3390/smartcities3010002, https://www.mdpi.com/2624-6511/3/1/2
57. Ying, X., Sagong, S.U., Clark, A., Bushnell, L., Poovendran, R.: Shape of the cloak: Formal analysis of clock skew-based intrusion detection system in controller area networks. CoRR **abs/1807.09432** (2018), http://arxiv.org/abs/1807.09432