Secure Automotive On-Board Protocols: A Case of Over-the-air (OTA) Firmware Updates

Muhammad Sabir Idrees[1], Hendrik Schweppe[1], Yves Roudier[1], Marko Wolf[2], Dirk Scheuermann[3], and Olaf Henniger[3]

EURECOM[1], Escrypt GmbH[2], Fraunhofer-SIT[3]
{muhammad-sabir.idrees,hendrik.schweppe,yves.roudier}@eurecom.fr
marko.wolf@escrypt.com,{dirk.scheuermann,olaf.henniger}@sit.fraunhofer.de

Abstract. *The software running on electronic devices is regularly updated, these days. A vehicle consists of many such devices, but is operated in a completely different manner than consumer devices. Update operations are safety critical in the automotive domain. Thus, they demand for a very well secured process. We propose an on-board security architecture which facilitates such update processes by combining hardware and software modules. In this paper, we present a protocol to show how this security architecture is employed in order to achieve secure firmware updates for automotive control units.*

Keywords: Security Protocols; Security Architectures; Secure firmware updates over the air; Software functionality

1 Introduction

Current research activities in vehicular on-board IT architectures basically follow two key trends: unification of network communication and centralization of functionality. Recent on-board IT architectures comprise a very heterogeneous landscape of communication network technologies, e.g., CAN, LIN, FlexRay and MOST. Internet Protocol (IP) based communication is currently being researched as a technology for unifying the overall interconnection of ECUs in future on-board communication systems [17]. In addition, there is a shift towards multipurpose ECUs and usage of flash memory technology in the microcontrollers. Besides these trends in the design of automotive on-board IT architectures, new external communication interfaces, fixed and wireless are becoming an integral part of on-board architectures. One key factor for this development is the integration of future e-Safety applications based on V2X communications (external communications of vehicles, e.g. with other vehicles - V2V, or with the infrastructure - V2I) [10,3] that have been identified as one promising measure for increasing the efficiency and quality of operational performance of all vehicles and corresponding intelligent transportation systems.
Firmware updates are crucial for the automotive domain, in which recalls are a very costly activity and thus should be avoided where possible. The practicability

of remotely updating devices has been shown by Google for their Android telephones. With this, they have a powerful tool to immediately react on discovered security flaws in very short time [21]. In the automotive domain, update intervals are calculated in quarters of a year and not quarters of a day right now. This paradigm is about to change and security mechanisms within the car provide the necessary building blocks. With the arising "always-connected" infrastructure, it will be possible to perform over-the-air (OTA) diagnosis and OTA firmware updates (see Fig. 1), for example. This will provide several advantages over hardwired access, such as saving time by faster firmware updates, which improves the efficiency of the system by installing firmware updates as soon as they are released by the car manufacturer. However, adding new in-vehicle services not only facilitates novel applications, but also imposes stringent requirements on security, performance, reliability, and flexibility. As discussed in [11], in-vehicle components need not only to be extremely reliable and defect free, but also resistant to the exploitation of vulnerabilities. Although on-board bus systems are not physically accessible (apart from via diagnostic interfaces), this provides only a limited degree of security for vehicles that are in wireless communication with other vehicles and devices (e.g., consumer devices connected to the vehicle).

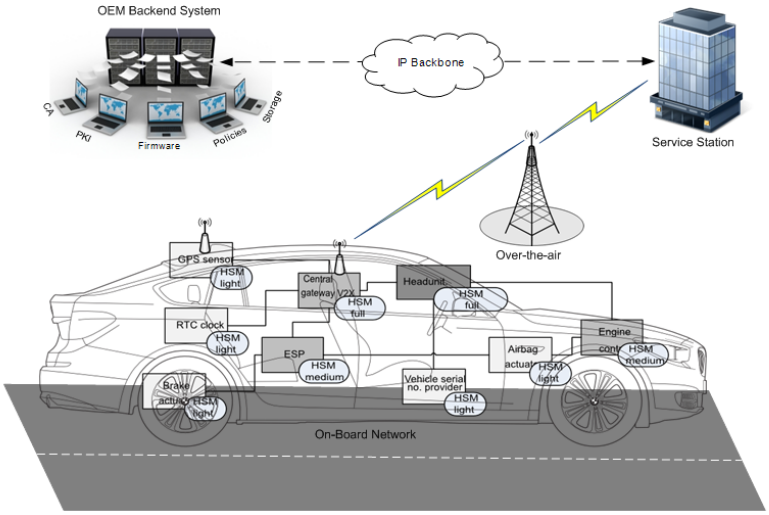

Fig. 1. Over-the-Air Firmware Updates

As seen in [11,18], attacks on the in-vehicle network have serious consequences for the driver. If an attacker can install malicious firmware, he can virtually control the functionality of the vehicle and perform arbitrary actions on the in-vehicle network [14]. Furthermore, since the ECU itself is an untrusted environment, there exist challenges in how to securely perform cryptographic operations (i.e., encryption/decryption, key storage). Thus, it does not make much sense, if the verifier software runs from the same flash as the software to be verified. In this paper we present a generic firmware update protocol, that can be used for both, hardwired and remote firmware flashing. The protocol has especially been designed with respect to the above mentioned functional and

non-functional requirements. Our approach is based on hard security services. The hardware security measures are required in order to raise the security level of specific security services, e.g. secure storage of security credentials. We present a hardware security module design, which protects most critical parts of the architecture during firmware updates, such as secure key storage, secure operation of cryptographic algorithms, etc.

2 Towards Secure On-Board Electronics Network Architecture: Hardware Security Module

The concept of a security architecture encompasses various technical approaches where security is introduced at different level of abstraction and based on different mechanisms: i.e., software and hardware based solutions. We believe that the combination of software and hardware based security solutions provides the measures for meeting discussed requirements for on-board network architecture. However, depending on the risk level it should be analyzed, for which use cases a security level using pure software security mechanisms is sufficient. Based on the security level identified in [18], we emphasize to employ hardware security services during firmware updates. The hardware security services, we call these services the hardware security module (HSM), is primarily used as a root of trust for integrity measurement and responsible for performing all cryptography applications including symmetric/asymmetric encryption/decryption, symmetric integrity checking, digital signature generation/verification, and generation of random numbers.

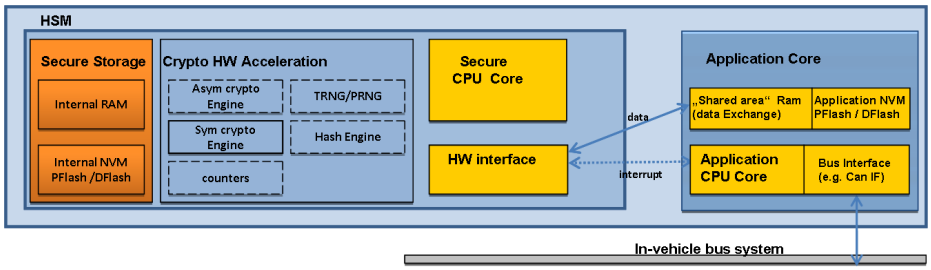

Fig. 2. HSM Architecture Overview

Considering constraints of current vehicular on-board networks and the trend towards more centralization of functions, very flexible and scalable on-board hardware security module is required. The design of the HSM needs to consider the different available resources on sensors and actuators, ECUs and bus systems. We have decided to create three different variants of our HSM: The full, the medium and the light hardware security module. The simplest hardware security module is designed for sensors/actuators (see Figure 1). On ECU level, a more complex architecture can be applied (i.e., medium HSM), which e.g. provides services for managing keys for the on-board system and to protect the ECU itself. In order to satisfy the performance requirements for signing and verifying messages for V2I communications (i.e., OTA firmware updates), a very efficient

asymmetric cryptographic engine is required. Thus, the full HSM architecture is applied in this case, which provides the maximum level of functionality, security, and performance. Table 1 presents security features of the different HSM variants and comparison with other existing HSM approaches. These variants offer different levels of security functionality. The components of the HSM are divided into mandatory and optional components because, depending on the use case, different security requirements have to be fulfilled. The optional components are represented in figure 2 with dashed lines. Furthermore, It is compliant to the Secure Hardware Extension (SHE) specification proposed by the automotive HIS consortium [13]. All HSM modules (full to light) are able to understand and process most SHE commands [4] accordingly.

Table 1. Components of different HSMs.

Security Features	**Hardware Security Module - HSM**			**SHE**	**TPM**	**Smart card**
	Full	**Medium**	**Light**			
Boot Integrity protection	Auth & Secure	Auth & Secure	Auth & Secure	Secure	Auth	None
HW crypto algorithms (incl. key generation)	ECDSA, ECDH, AES/MAC, WHIRLPOOL / RSA, SHA1 HMAC	RSA, SHA1, AES/ MAC	AES/ MAC	AES/ MAC	RSA, SHA1 / HMAC	ECC, RSA, AES, 3DES, MAC, SHA x..
HW crypto acceleration	ECC, AES, WHIRLPOOL	AES	AES	AES	None	None
Internal CPU	Programmable	Programmable	None	None	Preset	Programmable
RNG	TRNG	TRNG	PRNG w/ext. seed	PRNG w/ext. seed	TRNG	TRNG
Counter	16x64bit	16x64bit	None	None	4x32bit	None
Internal NVM	Yes	Yes	Optional	Yes	Indirect (Via SRK)	Yes
Internal Clock	Yes w/ext UTC Sync	Yes w/ext UTC Sync	Yes w/ext UTC Sync	No	No	No
Parallel Access	Multiple sessions	Multiple sessions	Multiple sessions	No	Multiple sessions	No
Tamper Protection	Indirect (passive, part of ASIC)	Indirect (passive, part of ASIC)	Indirect (passive, part of ASIC)	Indirect (passive, part of ASIC)	Yes (mfr.dep.)	Yes (active, up to EAL5)

Another aspect associated with the hardware security module is: How is the HSM integrated with the microcontroller? There are different possibilities for the realization of an HSM such as, `i)` HSM in the same chip as the CPU but with a state machine, and `ii)` HSM in the same chip as the CPU but with a programmable secure core. However, each having their advantages and disadvantages. In our proposed solution, a programmable CPU core is inside the same chip as the main microcontroller to perform cryptography operations (see Figure 2). Note that when software-based cryptography implementation is used, it can be easily modified (possibly not a highly efficient solution) but changing a state machine requires that hardware to be redesigned and is very expensive. It is necessary that the HSM be in the same chip as the application CPU and contains a microprocessor, to protect it from physical tampering.

3 Implementing Security Primitives

In this section we briefly review the hardware interfaces that covers the invocation specification of cryptographic hardware security blocks, higher-level security functionality and necessary security management functionality (e.g., key import/export, signature, and message authentication code) that are required during OTA firmware update. However, more detail about HSM functional calls/descriptions can be found in [8].

Signature: This function is used for demonstrating the authenticity and integrity of a message. A valid signature gives a recipient reason to believe that the message was created by a known sender, and that it was not altered in transit. For signature generation, a signature generation scheme $sig(m)_{S_k}$ takes as input a key k, and message m, outputs a signature $\widehat{\sigma}$; we write $sig(\mathrm{m})_{S_k} = \{\widehat{\sigma}\}_{S_k}$. Where k is the security parameter, outputs a pair of keys (S_k;V_k). S_k is the signing key, which is kept secret, and V_k is the verification key which is made public. We also assume that a time stamp (UTC Time) is generated, using `HSM_UTC_TIME(seconds_since_1970, mseconds_fraction)` function, and then also covered by the signature calculation, and write $\overrightarrow{m} = (\mathtt{m + Ts})$ to denote the message and a time stamp whose signature is $\widehat{\sigma}$. For the signature verification, $ver\_sig(\overrightarrow{m}\ , \widehat{\sigma})_{V_k} \rightarrow \alpha$ function is defined, takes as input the signature $\widehat{\sigma}$, the signature verification public key part V_k, and outputs the answer α which is either succeed (signature is valid) or fail (signature is invalid). As a precondition, the V_k must be loaded and enabled for verify.

Message Authentication Code – MAC: This function is used to protect both a message's data integrity as well as its authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content. For generating a MAC as well as the message itself, the notation $MAC\,(m)_{M_k} = \{\widehat{m}\}_{M_k}$ is used, so that it produces the message itself plus the cryptographic authentication code based on M_k and m. Here, M_k refers to a cryptographic key for MAC generation and m to the message to be authenticated. In the same way as for signatures, the use of the time stamp $\overrightarrow{m} = (\mathtt{m + Ts})$ is covered by the MAC calculation. For the verification of a MAC, the notation $ver\_MAC(\overrightarrow{m}, \widehat{m})_{M_k}$ is used. Based on the M_k, it is verified whether $\widehat{m}$ corresponds to the message $\overrightarrow{m}$.

Key Creation: This security building block is used for the creation of a key on a hardware module, using HSM `Create_Random_Key` function. During creation all properties of the key are determined and fixed. This includes the cryptographic algorithm to be used, the use and further property use flags indicating what actions may be done with this key (i.e., sign and verify) as well as the authorization data needed for key usage. The `use_flag` parameter indicates the operations that may be performed with the key. In particular, the following flags are present:

- `sign|verify`: Key can be used to generate or/and verify digital signatures or H/MACs of any data.
- `encrypt|decrypt`: Key can be used to encrypt or/and decrypt any data.
- `secureboot`: Can be used to create/verify secure boot references.

- `keycreation`: Can be used for creation of new keys, e.g. via key derivation functions (symmetric) or DH key agreement (asymmetric).
- `utcsync`: Can be used for synchronizing internal tick counter to UTC.
- `transport`: Can be used to protect transports of keys (i.e., migrate, swapping, move) between locations, according to individual `trnsp_flag` [7].

Only the `use_flag` may explicitly be set by the creator whereas further property flags are set inherently. Once created, the key properties are unchangeable. As output, the function delivers a key handle for later usage of the key.

Key Export: With this function, keys stored on HSM module are transported to other HSM modules or another external trusted party. During transport, the key is encrypted ($\epsilon\,(k)_{T_k}$) with a special transport key (T_k) that may be symmetric or asymmetric. In addition, the authenticity of the key is protected by a key authenticity code which consists in a MAC or a signature appended to the encrypted key. The key authenticity code can be explicit symmetric key enabled with `use_flag = verify` or an implicit symmetric key derived from a symmetric transport key or an implicit asymmetric key enabled with `use_flag = verify`. The usage of this key authenticity code is mandatory. As output, the function delivers the encrypted key together with its authentication code. As an important precondition, the specified transport key must be loaded and enabled to be used for transport. Furthermore, the transport flag of the key to be exported must be appropriately marked according to the type of module managing the transport key.

Key Import: This function is used for importing keys into the HSM module that was moved to another HSM module or other external trusted party. In this way, the function provides the opposite to the previously described export function. The key may be imported either into the non-volatile memory or into the main memory (RAM) of the HSM module. In the same way as for Key_Export, the use of the key authenticity code is mandatory. As output, the function delivers a key handle to reference the key for later usage. As a precondition, the transport key must be loaded and enabled before. In addition, the authentication code verification key must be loaded if the key is protected by a signature.

Key Master – KM: We introduce a new functional entity, which we call the $KeyMaster$. As there exist multiple variants of the HSM, that support different cryptographic keys (symmetric/asymmetric), we had to take this into account for key distribution. The KM is a central element in the establishment of a session between entities. It holds public key (P_k) and pre-shared keys (Psk) of the individual ECUs, which are used as transport keys, to establish a secure session. This functional entity reside on a dedicated ECU or be integrated into another ECU. There may be more than one KM node in a vehicle and may be replicated in different locations.

Counter: As a further instrument to control the behavior of hardware security modules, the possible use of counters is introduced as an additional security building block. The concrete, central task of a counter is freshness enforcement to prevent different kinds of replay attacks. For handling these counters, the following HSM functions are provided: `Create_counter`; `Read_Counter`;

`Increment_Counter`; and `Delete_Counter`. As input data, access authorization data needs to be provided which is later necessary to create, increment or delete the counter.

4 Secure Firmware Update Protocol

In order to provide secure firmware updates, we listed the most desired security properties (*Authentication, Integrity, Freshness*, and *Confidentiality*) to be able to prevent threats, as identified in [18]. The specified protocol provide means to satisfy these security requirements. The protocol shows how modules of HSM are used and how they interact in order to ensure secure firmware is downloaded and installed securely in the vehicle.

4.1 OTA Update Requirements

Before sketching the protocol, we describe some additional constraints/requirements responsible for secure firmware updates:

Hardware Security Module in the Diagnostic Tool: As stated in [18], there are numerous scenarios, where an attacker attacks on the diagnostic tool (DT) such as, the attacker injects bogus authority keys into the ECU, through DT, which compromise the overall security of the vehicular on-board architecture. In particular, this means that the DT stores challenges and public strings for key recovery (i.e., ECU unlock key) and is therefore responsible for the security of the subsystem. Therefore, this information needs to be stored securely on the DT-side. An additional advantage of HSM is the resistance against physical tampering of the DT. Any damage to the HSM changes the behavior and therefore prevents the extraction of secret key material.

Use of a secure evolution of Common Transport Protocol (CTP): Beside a secure V2X communication, a secure on-board communication is also important due to the large number of interconnected ECUs inside the vehicle involved in the process of firmware update. Therefore, secure transport protocols are needed for the exchange of on-board messages. In the on-board bus systems used, a specific restriction lies in the limited size of data packets. For the CAN bus, for example, this means that only eight bytes of payload may be transmitted at a time. For this purpose, secure common transport protocols (S-CTP) [7], extension of common transport protocol defined in [2], are applied to diagnosis jobs, where typically larger data chunks need to be transmitted.

4.2 Protocol Description

Remote Diagnosis: In the over-the-air firmware update scenario, a service station using a diagnostic tool (DT) connect remotely, using V2I (Vehicle-to-Infrastructure) infrastructure, to assess the state of the vehicle (see Figure 1). To know which version is installed, a diagnosis of the vehicle is required to have all necessary information such as ECU type, firmware version, and date of last update. An employee of the station using the diagnosis tool establishes a secure connection with the vehicle, at the ECU level, in order to determine the current state of the vehicle. To do so, DT creates a session key M_{sk} (exportable), by sending a HSM command `Create_Random_key` and specifies set of allowed

key properties such as, `use_flag = sign|verify, encrypt|decrypt`. It then `export` the M_k using P_k_ccu (Public key of the central communication unit - CCU) as a transport key (T_k) and transmit it to the vehicle. Here, the central communication unit is the first receiving entity in the vehicle, responsible for receiving and distributing V2X messages to the in-vehicle network.
In the vehicle, the CCU, equipped with the Full HSM module and act as a KM node, receives the connection request. The authorization for the connection is verified in the CCU. The message $\overrightarrow{m}$ is checked for freshness, integrity and the service station is authenticated. If the check succeeds, CCU-KM import the key in the HSM. It then export the received $Exported\_M_k$ with the corresponding P_k_ecu or Psk, depending on the ECU type, and distribute to the target ECU in order to enable end-to-end communication. This message includes all information that is necessary to deliver this message to the correct ECU. On the ECU side, ECU verifies integrity, authenticity and authorization of CCU/DT based on the policy whether DT is allowed to deliver a message or not. If this is true, and message is fresh, ECU import the M_k in the HSM. Once M_k key have been imported, acknowledgment is sent back to the DT (see Algorithm 1). After this acknowledgment frame, the DT sends, depending on the option chosen by the employee of the service station, requests to read out diagnosis information (State/Log information) from the ECU it wants to check.

Algorithm 1 Remote Diagnosis

Require: Signature verification Key V_k of DT, CCU, ECU are pre-loaded
Ensure: Establishing a fresh and authentic session between DT and ECU based on a symmetric session key M_k, where CCU-KM acts as a Key Master Node.

- DT $\rightleftarrows$ HSM: M_k-handle:= *create_random_key* (*use_flag* = sign| verify, encrypt| decrypt, export, *use_authorization_data*)
  DAT: $Exported\_M_k$:= *key_export*(T_k-handle = <P_k_ccu-handle>, kh=<M_k-handle>)

1. **DT** $\rightarrow$ **CCU-KM:** $\left\{(Exported\_M_k, Ts)\,,\ \{\widehat{\sigma}\}_{S_k\_dt}\right\}$
   - CCU-KM $\rightleftarrows$ HSM: M_k-handle := *key_import* (T_k-handle =<S_k_ccu-handle>, kh= <M_k-handle>)
   - CCU-KM: $\rightleftarrows$ HSM:*Exported_*M_k := *key_export*(T_k-handle = <P_k_ecu-handle | Psk-handle>, kh=<M_k-handle>)
2. **CCU-KM** $\rightarrow$ **ECU:** $\left\{(Exported\_M_k, Ts)\,,\ \{\widehat{\sigma}\}_{S_k\_ccu}\right\}$
   - ECU $\rightleftarrows$ HSM: M_k-handle := *key_import* (T_k-handle = <S_k_ecu-handle| Psk-handle>, kh= <M_k-handle>)
3. **CCU-KM** $\leftarrow$ **ECU:** $\left\{(ACK, Ts)\,,\ \{\widehat{\sigma}\}_{S_k\_ecu}\right\}$
4. **DT** $\leftarrow$ **CCU-KM:** $\left\{(ACK, Ts)\,,\ \{\widehat{\sigma}\}_{S_k\_ccu}\right\}$

Discussion - Advance Notification: Due to legal reasons and to allow for flexible deployment, we consider that service station sends an advance notification of possible firmware updates, if the type is the expected one. This advance notification is intended to help customers plan for effective deployment of updates, and includes information about the number of new updates being released. These updates still need to be `Approved` for install before downloading. The customer receives this information on the vehicle HMI and can decide about possible deployment (i.e., `Install, Decline, Decide later`). Only updates that have the approval status `Install` will be downloaded to the vehicle. If firmware update request is approved for install, it requires that vehicle is stopped and have access to the V2I infrastructure for receiving firmware data. Disabling

ECU while vehicle is running may cause some serious problems, depending on the function ECU is responsible for. Thus, we consider some additional checks, provided by the on-board system, to ensure that vehicle is stopped and have access to the infrastruture, before switching the ECU into the `re-programming` mode. Furthermore, we assume that V2I infrastructure is available through out the firmware update process.

ECU Reprogramming Mode: If the type is the expected one, the diagnostic tool forces the ECU to switch from an application mode into a reprogramming mode by requesting a seed (`Na`). This seed is required to calculate an ECU specific key value to unlock the ECU for reprogramming. ECU verifies desired security properties. If it is true, ECU sends a HSM command `SecM_Generate(seed)` to generate a seed. It then encrypt the seed $\epsilon\left(Na\right)_{M_k}$ for confidentiality enforcement, calculate a MAC on $\overrightarrow{m} = \left(\epsilon\left(Na\right)_{M_k} + Ts\right)$ and transmit it to the DT. At the same time, the ECU sends a HSM command `SecM_ComputeKey(Na, SecM_key)` to compute the key on the HSM with the aid of the generated `Na`. As output, the function delivers a $SecM_{key}$ key handle, we write $SecM_{key} = S_{mk}$, that is used to `unlock` the ECU.

Algorithm 2 ECU Reprogramming Mode

Require: DT and ECU have established a fresh and authentic connection based on a M_k. Vehicle is stopped and have access to infrastructure
Ensure: Authentic and confidential exchange of ECU unlock key.

1. **DT** $\rightarrow$ **ECU:** $\left\{\left(request\_seed,\, Ts\right),\ \{\widehat{m}\}_{M_k}\right\}$
   - ECU: $\rightleftarrows$ HSM:: SecM_Generate($seed$)
2. **DT** $\leftarrow$ **ECU:** $\left\{\left(\epsilon\left(Na\right)_{M_k},\, Ts\right),\ \{\widehat{m}\}_{M_k}\right\}$
   - ECU: $\rightleftarrows$ HSM:: S_{mk}:= SecM_ComputeKey(seed, $SecM\_key$)
   - DT: $\rightleftarrows$ HSM:: S_{mk}:= SecM_ComputeKey(seed, $SecM\_key$)
   - DAT$\rightleftarrows$ HSM: $Exported\_S_{mk}$:= key_export(T_k-handle = <M_k-handle>, kh=<S_{mk}-handle>)
3. **DT** $\rightarrow$ **ECU:** $\left\{\left(Exported\_S_{mk},\, Ts\right),\ \{\widehat{m}\}_{M_k}\right\}$
   - ECU: $\rightleftarrows$ HSM:: $SecM\_CompareKey\left(key, seed\right)$
4. **DT** $\leftarrow$ **ECU:** $\left\{\left(ACK,\, Ts\right),\ \{\widehat{m}\}_{M_k}\right\}$

In the diagnostic tool, verifies $\{\widehat{m}\}_{M_k}$, decrypt the received seed $\left(\epsilon^{-1}\left(Na\right)_{M_k}\right)$ and compute the S_{mk} with the aid of the received seed (`Na`). Once the S_{mk} key value is computed, it is exported, using M_k as a transport key, and transmitted to the target ECU. The ECU verifies the $\{\widehat{m}\}_{M_k}$ and compares the received S_{mk} key with the self-generated S_{mk}. If the two values are identical, the ECU switched into `unlock` state (from application mode to the reprogramming mode) and send an `ACK` message to the DT (see Algorithm 2). This message is sent after the ECU is switched into the `unlock` state to make sure the switch has been performed. The information whether a re-programming request has been received or not shall be stored in non-volatile memory, e.g. EEPROM. Since switching from the application to the reprogramming mode shall be done via a hardware reset, all contents of volatile memory will be lost [13]. If the comparison failed, the flashloader [13] hold the ECU in `locked` state. Only in the unlock state ECU `reprogramming` is possible.

Firmware Encryption Key Exchange: In this phase we are considering two possible scenarios of exchanging firmware encryption keys: `i) on-line`

`solution` and `ii) off-line solution`. In the on-line solution: service station has access to an online infrastructure of the manufacturer, it can request the firmware and as well as firmware encryption key – (SSK). The SSK is a stakeholder symmetric key pair [7], externally created, with `use_flag=decrypt`, key for stakeholder individual usage e.g., software update. Whereas, in the case of off-line, firmware is encrypted with pre-installed SSK.
Considering current trends and advancements in the automotive industry, on-line solutions provide more reliability, flexibility and will eventually increase the security of the on-board network. Sharing firmware encryption key only with specific ECUs makes an on-line solution more robust and generic from off-line approaches, where all vehicles share unique symmetric keys, that are pre-installed in the vehicles. Moreover, considering security levels [18], we argue to specify a key validity period (short term or long term keys) of the SSK, for an individual ECU. Both have its advantages and disadvantages. We suggest to use short term keys for firmware encryption. Short terms keys will expire after a short amount of time and thus, we see no need for instant revocation, if keys are compromised. This has the advantage that OEMs do not have to go through another key migration (installing new keys) process, if keys are compromised. Thus, in the following section we present the on-line solution.

Algorithm 3 Firmware Encryption Key Exchange

Require: On-line access to OEM server and PKI infrastructure
Ensure: Authentic and confidential firmware encryption key exchange between OEM and ECU

1. **DT** $\rightarrow$ **OEM:** $\{(request\_firmware\_encryption\_key,\, Ts)\,,\ \{\widehat{\sigma}\}_{Sk\_dt}\}$
   - OEM:$Exported\_SSK$:=key_export(T_k-handle=<P_k_ecu-handle|Psk-handle>, kh=<SSK-handle>)
2. **DT** $\leftarrow$ **OEM:** $\{(Exported\_SSK,\, Ts)\,,\ \{\widehat{\sigma}\}_{Sk\_oem}\}$
3. **DT** $\rightarrow$ **ECU:** $\left\{(Exported\_SSK,\, Ts)\,,\ \{\widehat{m}\}_{M_k}\right\}$
   - ECU: $\rightleftarrows$ HSM:: SSK-handle :=key_import (T_k-handle =<S_k_ecu-handle|Psk-handle>, kh= <SSK-handle>
4. **DT** $\leftarrow$ **ECU:** $\left\{(ACK,\, Ts)\,,\ \{\widehat{m}\}_{M_k}\right\}$

After successful reprogramming access at the ECU level, the DT sends a request (`request_firmware_encryption_key`), to the OEM server to get the firmware encryption key (see Algorithm 3). This request includes information about the ECU (i.e., ECU type, ECU identification number, firmware version, etc.). In the OEM side, verifies the authenticity and integrity of the received message. If this is true, OEM server retrieve the Pk_ecu from the Public Key Infrastructure (PKI), (possibly) maintained by an individual OEM, and export SSK using P_k_ecu as a T_k. This is only feasible if ECU is equipped with full HSM. In the case of the medium or light HSM-ECUs, the pre-shared key Psk will be used as a T_k. The OEM server export the SSK and send a signed message to the DT. As the SSK key blob is encrypted with the ECU key. It is not possible for the DT to retrieve the firmware encryption key. Next, the DT transmits the received firmware encryption key to the ECU. The ECU imports the SSK in the HSM using `key_import` function. The `key_import` function provides the assurance to the ECU that the key is generated by the OEM, by verify the authentication code send along with the encrypted key, and can only

be decrypted by the specific ECU key. After importing SSK in the ECU-HSM, ECU sends an `acknowledgment` about successful import of the SSK key.

Firmware Download: Once the SSK is successfully imported in the ECU-HSM, the DT sends the received signed and encrypted firmware ($\mathbb{F}_{rm}$) along with its ECU Configuration Register (ECR) reference: $sig\left(\mathbb{F}_{rm}, \mathbb{E}_{cr}, Ts\right) \rightarrow \widehat{\sigma_{\mathbb{F}_{rm}}} \overset{\epsilon nc}{\rightarrow} \epsilon\left(\widehat{\sigma_{\mathbb{F}_{rm}}}\right)_{ssk}$, from manufacturer, to the Random-Access Memory (RAM) of the ECU. Following HSM `use_flag` approach, where multiple key-properties may be set, only the OEM server can sign and encrypt the firmware, whereas the receiving ECU can decrypt and verify the received firmware, using the same key material. The encrypted firmware is downloaded block by block (logical block). Each of those blocks is divided into segments, which are a set of bytes containing a start address and a length. The start address and the length of each segment is sent to the HSM during the segment initialization. For one block, a download request is sent from the DT to the HSM. HSM initializes the decryption service and sent an answer to the DT. The download then starts segment by segment. After sending last firmware segment, DT sends a `transfer_exit` message to the ECU (see Algorithm 4).

Algorithm 4 Firmware Download

Require: Signed and encrypted firmware from OEM
Ensure: Authentic, fresh and Confidential firmware downloaded in the ECU

1. **DT** $\rightarrow$ **ECU:** $\left\{\left(\epsilon\left(\widehat{\sigma_{\mathbb{F}_{rm}}}\right)_{SSK}, Ts\right), \{\widehat{m}\}_{M_k}\right\}$
   - ECU $\rightleftarrows$ HSM: SecM_InitDecryption($\epsilon\left(\widehat{\sigma_{\mathbb{F}_{rm}}}\right)_{ssk}$)
   - ECU $\rightleftarrows$ HSM:SecM_Decryption($\varepsilon^{-1}\left(\widehat{\sigma_{\mathbb{F}_{rm}}}\right)_{ssk}$)
2. **DT** $\rightarrow$ **ECU:** $\left\{\left(request\_transfer\_exit, Ts\right), \{\widehat{m}\}_{M_k}\right\}$
   - ECU $\rightleftarrows$ HSM:SecM_DeinitDecryption()
3. **DT** $\leftarrow$ **ECU:** $\left\{\left(ACK, Ts\right), \{\widehat{m}\}_{M_k}\right\}$

Firmware Installation and Verification: For an installation of the firmware, we consider standard firmware installation procedure defined in [13], where each logical block is erased and reprogrammed. However, before the flash driver can be used to re-program an ECU, its compatibility with the underlying hardware, the calling software environment and with prior versions of the firmware has to be checked. This compatibility check is performed by means of a version information stored in the HSM monotonic counters. The HSM `Read_Counter` function is used to read out the value of a counter. The counter is referenced by a $counter\_identifier$ previously increased, after every authentic and successful installation of the firmware. These monotonic counters are defined to perform such a checking of its current version against the new firmwares version in order to prevent the downgrading attack to older firmwares.
For the verification, we defined two step verification process: In the first step, before `re-programming`, ECU verifies the signature of the firmware data. This is verified by using the pre-installed Manufacturer Verification Key MVK. It proves that the software was indeed released from the OEM. In the second step: we construct a tiny trusted computing base (TCB) during the installation phase. We compute an ECR trusted chain at each step of the firmware installation. The ECR reference is needed to ascertain the integrity/authenticity of the firmware

data. An `Extend_ECR` function is defined to build the ECR trusted chain. This function is used for updating the ECR with a new hash value. The new value is provided as input and chained with the existing value stored in the ECR, using a hash update function. As output, the function delivers the updated ECR value. After a successful installation of the new firmware data, software consistence check is performed. The check for software dependencies shall be done by means of a callback routine provided by the ECU supplier. This check is done after reprogramming and before setting the new ECR reference. Next, the `Compare_ECR` function is called. This comparison can only be performed after all writing procedures for the logical block have been finished. This function allows the direct comparison of the current ECR with a reference ECR value received with the firmware. It is also possible that the ECR reference may be contained inside the firmware itself. In this case the flashloader shall call a routine provided by the ECU supplier to obtain the ECR reference. If the check succeed, HSM `Preset_ECR` function is called. This function is used to manage references to ECR values by ECR indices in the context of secure boot. After successfully setting the ECR value, HSM (`Increment_Counter`) function is called to increment the monotonic counter with the new value. In the last step, the actual hardware reset is executed, the flashloader delete (i.e. overwrite) the routines for erasing and/or programming the flash memory from the ECU's RAM [13]. Making sure those routines are not present on the ECU in application mode. After the reset the application is started.

Discussion - Error Handling: Each function of the HSM returns a status after its successful or unsuccessful execution. Some functions may deliver further function specific error codes. The value of the status shows the positive execution of the function or the reason for the failure. In the case of failure, the flash process must stop with an error code and the ECU enters the `locked` state.

5 Related Work

The past decade has seen a tremendous growth in the vehicular communication domain, yet no comprehensive security architecture solution has been defined that covers all aspects of on-board communication (data protection, secure communication, secure and tamper proof execution platform for applications). On the other hand, several projects, namely GST [5], C2C-CC [3], IEEE Wave [22] and SeVeCOM [20] have been concerned with inter-vehicular communication and have come up with security architectures for protecting V2X communications. These proposals essentially aim at communication specific security requirements in a host-based security architecture style, as attackers are assumed to be within a network where no security perimeter can be defined (ad-hoc communication). These proposals consider the car mostly as a single entity, communicating with other cars using secure protocols.

Mahmud et al. [12] present a security architecture and discuss secure firmware upload. There are, however, a number of prerequisites and assumptions (i.e., sending multiple copies to ensure firmware updates) in order to make secure firmware update. However, sending multiple copies is not realistic and impose several constraints on the infrastructure. This proposal does not consider auto-

motive on-board networks, where domains are traditionally separated, due to functional and non-functional requirements. Kim et.al [9] presents remote progressive updates for flash-based networked embedded systems. In their solution a link-time technique is proposed which reduce the energy consumption during installation. However, no security aspects are mentioned in this proposal.
Nilsson et al. in [15,16] provide a lightweight protocol and verification for secure firmware updates over the air (SFOTA) in vehicles is proposed. In the SFOTA protocol, different properties are ensured during firmware update protocol (i.e., data integrity, data confidentiality, and data freshness). However, this approach also imposed several assumptions in order to ensure the secure software upload such as an authentication of the vehicle is not considered, keys are stored securely and single encryption key for each car. Furthermore, no specific execution platform requirements are put forward by this proposals. For instance, switching the ECU into reprogramming mode. In [14], key management issues are discussed while performing software updates. A rekeying protocol is defined in order to distribute keys with only specific nodes in the group. It also uses the multicast approach to update the software on a group of node. However, we consider that different firmware are installed on different ECUs, depending on the ECU functionalities, which makes multicast approach not useful. Furthermore, as mentioned above, this approach also does not consider execution platform requirements. It does not discuss about computation attacks, where, attacker can learn and modify the firmware, during installation phase or simply prevent to update the counter, for later replay attacks.
Hagai [19] present an approach that takes hardware into account by providing a secured runtime environment with a so-called Trust Zone on an ARM processor, the solutions of [1,6] are software based. The so called *tools* and *enablers*, which are low-level and application-level security functions in [1] also cover a number of on-board automotive use-cases, while leaving the essential link to the external communication domain uncovered. The most related approach with our work is Hersteller-Initiative Software – HIS [13]. The flashing process provides by HIS provides a good basis for the OEMs, but the recommended protocols does not provide all necessary security functionalities (i.e., freshness). Furthermore, this process only considers hardwired firmware updates and does not provide any information about which key is used for firmware encryption, in a very heterogeneous landscape of communication network technologies.

6 Conclusion

We have presented a firmware update protocol for a new security architecture that is deployed within the vehicle. We showed how a root of trust in hardware can sensibly be combined with software modules. These modules and primitives have been applied to show how firmware updates can be done securely and over-the-air, while respecting existing standards and infrastructure. In contrast to existing approaches, the protocols presented in this paper show the complete process, which involves service provider, vehicle infrastructure as well as manufacturer and workshop. By using secure in-vehicle communication and a trusted platform model, we show how to establish a secure end-to-end link between man-

ufacturer, workshop and vehicle. Despite the fact that a trusted platform model poses certain constraints, such as cryptographic keys bound to a boot configuration, we showed how the protocols presented take these constraints into account by updating platform reference registers used during the boot phase of an ECU.

Acknowledgments

This work has been carried out in the EVITA (E-safety Vehicle InTrusion protected Applications) project, funded by the European Commission within the Seventh Framework Programme, for research and technological development.

References

1. H. Bar-El. Intra-vehicle information security framework. In *Proceedings of the 7th escar Conference*, Düsseldorf, Germany, 2009.
2. M. Busse and M. Pleil. Data exchange concepts for gateways. Technical Report Deliverable D1.2-10, EASIS Project, 2006.
3. C2C-CC. Car2Car Communication Consortium. http://www.car-to-car.org/.
4. R. Escherich, I. Ledendecker, C. Schmal, B. Kuhls, C. Grothe, and F. Scharberth. SHE – Secure Hardware Extension - Functional Specification Version 1.1.
5. GST. Global systems for telematics, EU FP6 project. http://www.gst-forum.org/.
6. A. Hergenhan and G. Heiser. Operating Systems Technology for Converged ECUs. *Embedded Security in Cars*, 2008.
7. S. Idrees and et al. Secure On-Board Protocols Specification. Technical Report Deliverable D3.3, EVITA Project, 2010.
8. B. Weyl and et al. Secure On-board Architecture Specification. Technical Report Deliverable D3.2, EVITA Project, 2010.
9. J. Kim and Pai H Chou. Remote progressive firmware update for flash-based networked embedded systems. *ISLPED'09*, pages 407–412, 2009.
10. T. Kosch. Local Danger Warning based on Vehicle Ad-hoc Networks: Prototype and Simulation. In *WIT 2004*, pages 3–7, 2004.
11. K. Koscher and et.al. Experimental Security Analysis of a Modern Automobile. *In Proc. of the 31st IEEE Symposium on Security and Privacy*, May 2010.
12. S.M. Mahmud, S. Shanker, and I. Hossain. Secure software upload in an intelligent vehicle via wireless communication links. In *Proc. IEEE Intelligent Vehicles Symposium*, pages 588–593, 2005.
13. T. Miehling, P. Vondracek, M. Huber, H. Chodura, and G. Bauersachs. HIS flashloader specification version 1.1. Technical report, HIS Consortium, 2006.
14. D.K Nilsson and et.al. Key management and secure software updates in wireless process control environments. *WiSec 08*, 2008.
15. D.K. Nilsson and U.E. Larson. Secure Firmware Updates Over the Air in Intelligent Vehicles. In *Proc. ICC Workshops*, 2008.
16. D.K. Nilsson, Lei Sun, and T. Nakajima. A Framework for Self-Verification of Firmware Updates Over the Air in Vehicle ECUs. In *GLOBECOM Workshops 08*.
17. M. Rahmani and et.al. A novel network architecture for in-vehicle audio and video streams. In *IFIP – BcN*, 2007.
18. A. Ruddle and et al. Security Requirements for Automotive On-Board Networks based on Dark-side Scenarios. Technical Report Deliverable D2.3, EVITA Project.
19. Towards a secure automotive platform. White paper, secunet, 2009.
20. SeVeCOM. Secure Vehicle Communication. http://www.sevecom.org/.
21. A. Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, and Shlomi Dolev. Google Android: A State-of-the-Art Review of Security Mechanisms, 2009.
22. IEEE WAVE. Wireless Access in Vehicular Environments, IEEE standard 1609.2.