UniSUF: a unified software update framework for vehicles utilizing isolation techniques and trusted execution environments

UniSUF: A Unified Software Update Framework for Vehicles utilizing Isolation Techniques and Trusted Execution Environments

Kim Strandberg[1,2][0000−0003−0892−2600], Dennis Kengo Oka[3][0000−0003−2714−0882], and Tomas Olovsson[2][0000−0001−9548−819X]

[1] Volvo Car Corporation, Sweden kim.strandberg@volvocars.com
[2] Chalmers University of Technology, Sweden firstname.lastname@chalmers.se
[3] Synopsys, Japan dennis.kengo.oka@synopsys.com

Abstract. Today's vehicles depend more and more on software, and can contain over 100M lines of code controlling many safety-critical functions, such as steering and brakes. Increased complexity in software inherently increases the number of bugs affecting vehicle safety-critical functions. Consequently, software updates need to be applied regularly. Current research around vehicle software update solutions is lacking necessary details for a versatile, unified and secure approach that covers various update scenarios, e.g., over-the-air, with a workshop computer, at factory production or using a diagnostic update tool. We propose UniSUF, a Unified Software Update Framework for Vehicles, well aligned with automotive industry stakeholders. All data needed for a complete software update is securely encapsulated into one single file. This vehicle unique file can be processed in multitudes of update scenarios and executed without any external connectivity since all data is inherently secured. To the best of our knowledge, this comprehensive, versatile and unified approach cannot be found in previous research and is a contribution to an essential requirement within the industry for handling the increasing complexity related to vehicle software updates.

Keywords: Vehicle software update framework · Vehicle security.

1 Introduction

A vehicle can contain more than 150 ECUs (Electronic Control Units) and over 100M lines of code. The complexity of software within the automotive domain is increasing and with it the risk for vulnerabilities. To address this, there are ongoing activities for vehicle software updates, such as ISO/CD 24089 [1] and UN Regulation No. 156 regarding vehicle software update requirements [2]. The latter states, among other, requirements for vehicle manufactures to have a secure software update process. While standards and regulations typically focus on high-level requirements, technical design and implementation requirements are left up to the automotive organizations. There is a risk that if the software

update process is vulnerable, it can be exploited by attackers who could potentially introduce malicious code at some stage into the software update process that finally reaches in-vehicle systems causing life-threatening hazards such as manipulated brakes, steering, or engine control.

A secure software update framework that can support numerous different update scenarios, such as over-the-air, in workshops, and in factories, with or without Internet access, is required for automotive organizations in order to apply software updates to address vulnerabilities in a timely and regular manner. Our approach is to provide a cost-effective, open architecture, with increased security through isolation and separation of duties that is comprehensive to support numerous use cases. Thus, we propose UniSUF, a versatile and unified approach for secure vehicle software updates. By using multiple signing and encryption keys, all data needed for a complete software update is securely encapsulated into one single file, the *Vehicle Unique Update Package (VUUP)*. This vehicle unique file can be processed by a vehicle ECU, using a workshop computer, at factory production or with a diagnostic update tool, hence considerably simplifying software management processes. At the receiving vehicle side, this file is decapsulated and validated layer by layer, where cryptographic material and sensitive operations are isolated within a trusted execution environment to ensure both the integrity and the confidentiality of the data. The main contributions of this paper are:

- We have analyzed and reviewed several software update use cases in the automotive industry and as a result, defined a number of constraints and conditions for a unified and versatile approach.
- Considering these constraints and conditions, we suggest an approach for vehicle software updates, well aligned with automotive industry stakeholders. In-depth details give a comprehensive overview for a possible secure implementation covering the whole software chain from producer to receiver.
- We have reviewed the suggested approach with automotive software update architects to ensure that the proposed approach can be practically deployed and efficiently adopted for vehicle software updates.

2 Problem Statement

Considering the different existing use cases for vehicle software updates, such as over-the-air, using a workshop computer, at factory production, or with a diagnostic update tool, each use case typically has its own approach which causes complexity. Moreover, new use cases for software updates need to be considered with future demands to support 3rd party component updates ([3], [4]). Therefore, to simplify, reduce costs, allow flexibility, and to make the update process manageable, all while considering security aspects, we propose a unified and versatile approach to handle all the use cases.

After reviewing the above-mentioned use cases, the following constraints and conditions are defined for a unified software update framework:

- Support for online updates (software update files and/or cryptographic credentials/operations require online access).

- Support for offline updates (software update files and cryptographic credentials/operations are accessible offline).
- Should not rely on additional input for cryptographic keys or installation instructions, e.g., from a diagnostic update tool (i.e., all data needed for a complete software update is securely encapsulated into one single file and no additional input is required).
- No dependency on the data distribution model (i.e., software update files can be provided through different means and it does not matter how they are distributed to the vehicle).
- No dependency on software update storage location (i.e., software update files should be independently protected regardless of where they are stored).
- Flexible and modular to support 3rd party component updates.

We have taken these constraints and conditions into consideration when designing a software update framework to allow for a unified and versatile approach to support different use cases. Our proposed software update framework is described in the next section.

3 UniSUF: A Unified Software Update Framework

In this section, we present the Unified Software Update Framework (UniSUF). First, an overview of the involved entities in the framework is presented, followed by a brief explanation on how to secure data distribution and data execution, and finally, the procedure for preparing software update files is given.

3.1 Entities

There are three main entities involved in the software update process: the producer, the consumer, and the repository. The producer is responsible for producing the software. The consumer is responsible for the download and installation process of the software, and the repository is a storage point for software preferably located in various cloud sources, enabling both proximity and redundancy for data in relation to the vehicle.

An overview of the data distribution in the backend handled by the ***Producer Agent (PA***) is shown in Figure 1. The main entities it contains are:

- ***Producer Security Agent (PSA)*** facilitates functionalities for secure key generation using ***Secure Key Generator (SKG)***, secure storage for cryptographic material using ***Cryptographic Material Storage (CMS)*** and signing of data using ***Producer Signing Service (PSS)***.
- ***Version Control Manager (VCM)*** has control over available software versions w.r.t. current vehicle status to create both download instructions using ***Producer Download Agent (PDA)*** and installation instructions using the ***Producer Installation Agent (PIA)***.

On the receiving side, Figure 4 shows the ***Consumer Agent (CA***) handling data distribution to the vehicle. The main entities it contains are:

- ***Consumer Download Agent (CDA)*** downloads required data, e.g., instructions and software files, verifies the authenticity of the data and initiates installation using the ***Consumer Installation Agent (CIA)***.
- ***CDA*** and ***CIA*** uses the ***Consumer Security Agent (CSA)*** which requires a Trusted Execution Environment (TEE) in order to support secure operations and store cryptographic keys securely.

By using isolation mechanisms and implementing each entity as a module according to the principles of *least privilege* and *separation of duties* a potentially compromised entity cause the least possible harm to the complete system. These modules can be secured either locally or in the cloud.

3.2 Securing Data Distribution and Data Execution

To be able to secure the *data distribution* and *data execution*, we propose using signed asymmetric and symmetric keys in conjunction with key wrapping mechanisms. Symmetric session keys are used to encrypt sensitive cryptographic material needed for the update processes, such as keys for unlocking ECUs and keys for decryption of software. The symmetric session keys are encrypted with a public vehicle unique asymmetric key, ensuring the secure storage and transfer of key material. Using an asymmetric key for key wrapping ensures that only the vehicle with the corresponding private key can decrypt the encrypted session keys.

Policies dictate rules for each individual encrypted session key, where policies and keys in conjunction are signed i.e., giving rise to a *Key Manifest (KM). KMs* are securely processed at the receiving side, where session keys are appointed to certain trusted applications according to the stated policies. The functionality of trusted applications can be decryption of software files, unlock ECUs for software updates, and signing of installation reports and logs.

3.3 Preparation of Software Update Files

The individual files that contain the actual software need to be secured, ensuring both confidentiality and authenticity. Considering the entities in the framework the procedure to secure software files is as follows.
1. The producer of software signs software files with a supplier specific signing certificate to provide authenticity. If supported, this signature is later validated by end receivers (e.g., ECUs) before installation. Software files are uploaded to the *Producer Local Secure Storage*, shown in Figure 1.
2. *VCM* receives the software files and validates the software supplier's signature.
3. *VCM* requests a symmetric encryption key (hereafter called sw_key) from *PSA* and encrypts the software file with this key to provide confidentiality.
4. *VCM* requests a signature of the hash of the encrypted software file from *PSS*. The signature is added to the encrypted file metadata to provide authenticity.
5. *VCM* performs mutual authentication towards the cloud software repository and uploads the signed encrypted file to the cloud and stores the url to this file

in a database.
6. *VCM* securely stores the symmetric encryption key (i.e., the corresponding sw_key) in *CMS* to be retrieved later, and encrypted and included into a *Secure Key Array (SKA)* for a future software update (cf. Step 6. in Section 4.1).

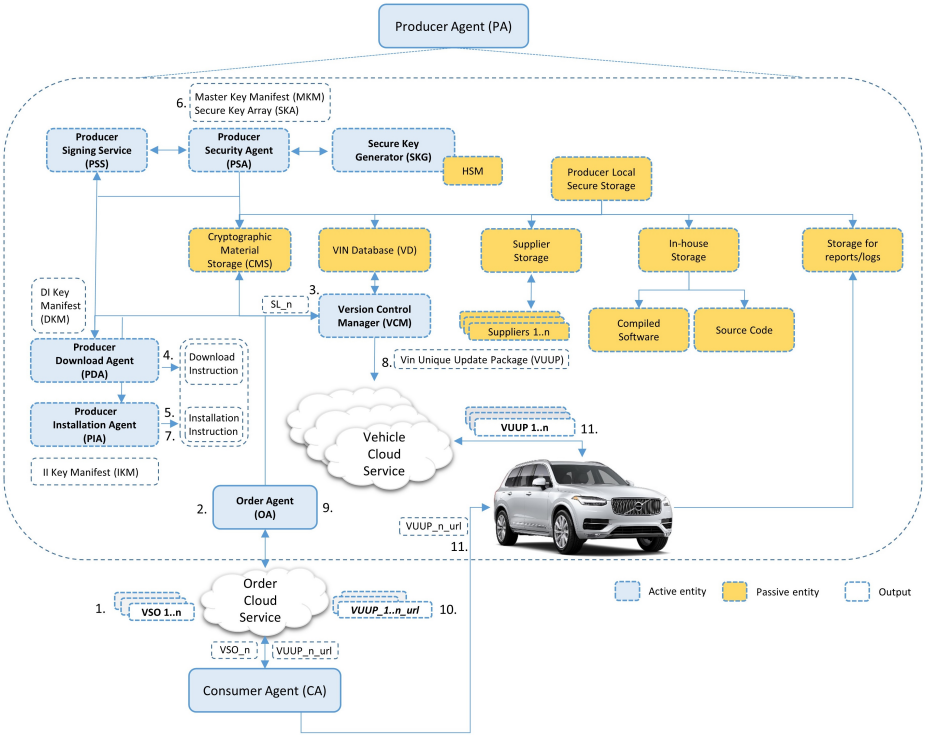

Fig. 1. Data distribution in the backend

4 The Software Update Process

In this section, we dive into the details of the complete software update process in UniSUF. Explanations of the abbreviations used can be found in Table 1.

4.1 Encapsulating Data into a VUUP file

Producer Agent (PA): data distribution in backend. Figures 1, 2 and 3 describe the process of creating a complete *VUUP* file. The 11 steps described below are indicated by numbers where relevant in the Figures 1, 2 and 3.
1. Order request. The *Consumer Agent (CA)* in the vehicle, local workshop, or any other consumer, places a signed order on behalf of a *Vehicle Identification*

Table 1. Abbreviations

Abbreviations	
Vehicle Identification Number (VIN)	The VIN number is a vehicle unique fingerprint, and is composed of 17 characters.
Producer Agent (PA)	Parent entity consisting of many children entities covering backend requirements.
Producer Security Agent (PSA)	Responsible for handling cryptographic material in the backend systems.
Producer Signing Service (PSS)	Executes signing requests i.e., returns signatures of hash values requested by authenticated entities.
Order Agent (OA)	Responsible for managing software requests from consumers.
Secure Key Generator (SKG)	PSA uses this module for the secure generation of key material.
Secure Key Array (SKA)	An array that PSA creates with cryptographic material related to a VIN unique software package.
Version Control Manager (VCM)	Responsible for management of software versions related to unique vehicles and for repackaging of data into the final VUUP file.
Producer Download Agent (PDA)	Creates the instructions for the download of software for a certain VIN.
Producer Installation Agent (PIA)	Creates the diagnostic instructions for installation of software for a certain VIN, including retrieving necessary cryptographic material.
VIN Database (VD)	Stores VIN unique data related to software.
Cryptographic Material Storage (CMS)	Secure storage of cryptographic material.
Download Key Manifest (DKM)	The manifest that contains the DKM session key with the policy for decryption of the download instruction.
Installation Instruction Key Manifest (IKM)	Contains the IKM session key with policy for decryption of the installation instruction.
Master Key Manifest (MKM)	Contains MKM session keys with policies for decryption of cryptographic data.
Vehicle Unique Update Package (VUUP)	The update package that includes information to perform a complete vehicle software update, e.g., software download instructions, installation instructions and cryptographic material.
Consumer Agent (CA)	The parent entity to the children entities covering vehicle requirements for the software installation process. The localization for children entities can be adapted to accommodate various use cases, e.g., OTA, workshop, and factory.
Consumer Download Agent (CDA)	Executes download instructions and retrieves required software files to local storage.
Consumer Installation Agent (CIA)	A diagnostic client responsible for the execution of installation instructions and requests to CSA for the execution of cryptographic material.
Consumer Security Agent (CSA)	A trusted execution environment (TEE), with pre-stored certificates between vehicle manufacture and CSA; which enables secure transfer and execution of cryptographic material from the backend to the vehicle.
Key Wrapping (KW)	The process of encrypting one key with the use of another symmetric or asymmetric key, to securely store or transmit it over an untrusted channel.
Key Manifest (KM)	Used to define policies and relations for certain keys. Keys are secured with KW, where encrypted keys and policies are signed, giving rise to a KM.

Number (VIN) (i.e., a *Vehicle Signed Order (VSO)*). A *VSO* should contain a complete vehicle readout and be signed by the entity which creates the order. *VSOs* are placed in the *Order Cloud Service* queue. *Output: VSO_n.signed;*

2. Initiate VCM with VSO file. The *Order Agent (OA)* pulls *VSOs* from the *Order Cloud Service* queue, verifies the *CA* signature of the *VSO*, and requests initiation by *VCM* with this *VSO*.

3. VCM creates an SL file with VIN unique software information. *VCM* receives a *VSO* from *OA* for a certain *VIN*. *VCM* validates the signature of the *VSO* and retrieves the latest available software versions and *VIN* vehicle data from the *VIN Database (VD)*. VIN data in *VD* is compared with actual vehicle software readout in the *VSO*. Software deviations are handled, and a signed *Software List (SL)* is created from information in the *VSO* and *VD* and is sent to *PDA*, *PIA*, and *PSA*. *Output: SL.signed;*

4. PDA creates download instructions. *PDA* verifies the *SL* and creates download instructions (list of software urls) for all ECUs based on the *SL*. *PDA*

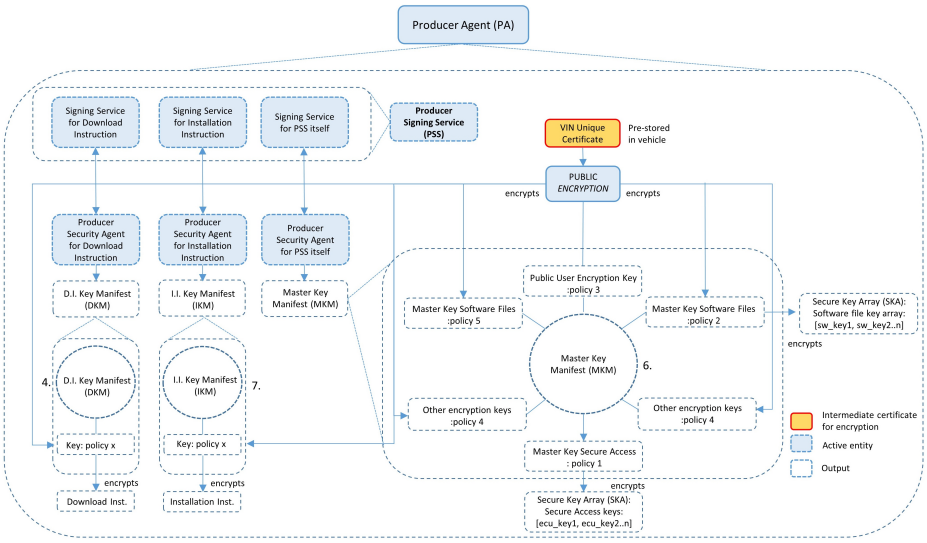

Fig. 2. Data distribution in the backend in relation to cryptographic material

requests a DKM (*Download Instruction Key Manifest*) session key from *PSA* and encrypts the download instructions with this key. Next, this key is encrypted with a vehicle unique public certificate retrieved from *CMS*, where the certificate is validated for authenticity towards the *Root CA* and *OCSP* (Online Certificate Status Protocol). The encrypted DKM session key and a policy that dictates the association to the download instructions give rise to the *DKM*. A hash is calculated of the encrypted download instructions and the *DKM* separately, and signature requests are sent to *PSS* on behalf of *PDA*, which replies with two separate signatures. *Output: download_instruction.signed; DKM.signed; PDA_cert;*

5. PIA creates installation instructions. *PIA* verifies the signature of the *SL* and creates installation instructions for all ECUs based on the *SL*.
Output: installation_instruction;

6. PSA requests cryptographic material. *PSA* verifies the *SL* and retrieves the required cryptographic material for software related to the received *SL* from *CMS*, such as keys for unlocking ECUs, privileged diagnostic requests, and software decryption keys. For each category of cryptographic material, *PSA* generates an MKM (*Master Key Manifest*) session key, where each key is associated with that specific category policy. MKM keys are in turn encrypted separately with a vehicle unique public certificate retrieved from *CMS* (same certificate as in Step 4), where the vehicle unique certificate from *CMS* is validated for authenticity towards *Root CA* and *OCSP*. The encrypted MKM keys with each respective category policy give rise to the *MKM*. A key array named *SKA* is created, which includes a sub-array for each category with separately encrypted key data, encrypted with the MKM key which belongs to that specific category. For example, *SKA* can include an array of encrypted symmetric keys used to

encrypt/decrypt the relevant software update files, so called sw_keys (cf. Section 3.3), and an array of encrypted security access keys used for unlocking relevant ECUs. A hash is calculated of the *SKA* and *MKM*, where after signature requests are sent to *PSS* which replies with two separate signatures. *Output: MKM.signed; SKA.signed;*

7. PIA retrieves the signed MKM and SKA from the PSA, and encrypts/signs the installation instruction. *PIA* request the signed *MKM* and the signed *SKA* from *PSA*. *MKM* and *SKA* signatures are validated where after *MKM* and *SKA* are included as part of the installation instructions. *PIA* requests an IKM (*Installation Instruction Key Manifest*) session key from *PSA* and encrypts the installation instructions with this key. The IKM session key is then encrypted with a vehicle unique public certificate retrieved from *CMS* (same certificate as in Step 4.), where the certificate is validated for authenticity towards the *Root CA* and *OCSP*. The encrypted IKM session key and a policy that dictates the association to the installation instructions give rise to the *IKM*. A hash is calculated of the encrypted installation instructions and the *IKM* separately, and signature requests are sent to *PSS* on behalf of *PIA*, which replies with two separate signatures.
Output: installation_instruction.signed; IKM.signed; PIA_cert;

8. VCM creates the VUUP file.
Input: download_instruction.signed; DKM.signed; installation_instruction.signed; IKM.signed; PDA_cert; PIA_cert;
VCM retrieves the generated data from *PDA* and *PIA*. Certificates are fetched from CMS and validated for authenticity towards the *Root CA* and *OCSP*, signatures are validated with the respective certificate and all the data is repackaged into *VUUP* content. A hash is calculated of the *VUUP* content, and a signature request is sent to *PSS* on behalf of *VCM*, which replies with a signature. The signed VUUP is uploaded to the *Vehicle Cloud Service* together with its VCM certificate. *Output: VUPP_n.signed; VCM_cert;*

9. VCM notifies OA. *VCM* notifies *OA*, that the order is ready and supplies a signed URL to the VUUP file. *Output: VUUP_1..n_url.signed;*

10. OA adds the url to the VIN unique VUUP in the Order Cloud Service. The *OA* validates the signature of url and thereafter adds the url in the *Order Cloud Service*.

11. CDA requests status. The *CDA* pulls status from the *Order Cloud Service* (via the *CA*) to indicate that updates are available for download via the signed VUUP_n_url. If no updates yet are available, the signed url will be empty.

4.2 Decapsulating the VUUP file

Consumer Agent (CA): data distribution to vehicle. For the *CA*, the process can be considered as the *PA* process reversed. The 17 steps described below are indicated by numbers where relevant in the Figures 4, 5 and 6.

1. The CDA requests software updates.
Input: VUUP_n_url.signed; *VCM_cert;*
If there are updates available, the CDA receives a signed VUUP_n_url and

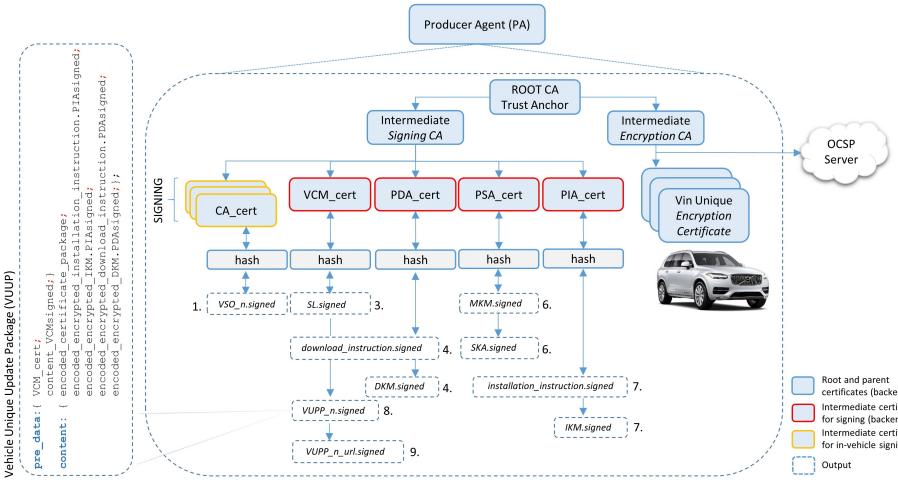

Fig. 3. Data distribution in the backend in relation to signing

VCM_cert, where the certificate is validated for authenticity towards the *Root CA* and *OCSP*, where after the signature of VUUP_n_url is validated using the received VCM_cert.

2. Download of VUUP. Mutual authentication is performed towards the *Vehicle Cloud Service* and the signed VUUP_n is downloaded to *Consumer Local Storage. Output: VUPP_n.signed;*

3. Validate VUUP. The signature of VUUP_n is validated with VCM_cert, and VUUP_n is decapsulated to produce the signed contents of download instructions, DKM, installation instructions, IKM as well as the included PDA_cert and PIA_cert. *Output:*
download_instruction.signed; DKM.signed; installation_instruction.signed; IKM.signed; PDA_cert; PIA_cert;

4. Validate data within VUPP. Certificates are fetched for online cases or retrieved from the VUUP file for offline cases. The signatures for the download instructions and the *DKM* are validated with the PDA_cert, and the signatures for the installation instructions and the *IKM* are validated with the PIA_cert.

5. DKM Key manifest initiation. The *CDA* requests the *CSA* to initialize the *DKM*, by providing the *DKM.signed* and PDA_cert.
Output: DKM.signed; PDA_cert;

6. CSA associates DKM with TEE application. The signature of *DKM* is validated with PDA_cert. The *DKM* session key is decrypted with the prestored vehicle unique private certificate and associated with the TEE application according to the policy in the *DKM* manifest, i.e., for decrypting download instructions.

7. Request decryption of download instruction. The *CDA* provides the signed download instructions to the *CSA* and requests decryption.
Output: download_instruction.signed;

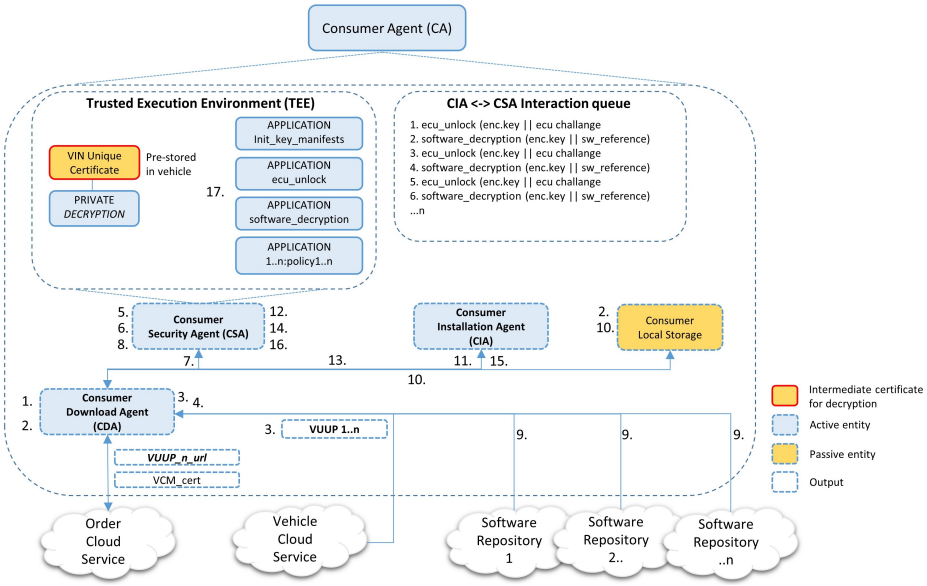

Fig. 4. Data distribution to the vehicle

8. Perform decryption of download instruction. The *CSA* validates signature of the download instruction with PDA_cert, decrypts with the DKM session key from the *DKM* in accordance with policy (i.e., decrypting download instructions) and returns the decrypted download instructions to the *CDA*.
Output: download_instruction;
9. Download of software files. The *CDA* performs mutual authentication towards various software repository sources and downloads encrypted signed software files to *Consumer Local Storage* using the download instructions. The *CDA* validates signatures of all encrypted software files with the VCM_cert.

Consumer Agent (CA): data execution in vehicle. After data distribution to the vehicle has been completed, the following steps describe the installation of the software update through data execution in the vehicle. These steps can be performed completely offline.
10. Initiation of pre-state phase. The *CDA* requests to start installation of software by sending the signed installation instructions, signed IKM, and the PIA_cert to the *CIA*.
Output: installation_instruction.signed; IKM.signed; PIA_cert;
11. Reboot to secure state. The *CIA* validates the PIA_cert for authenticity towards *Root CA* and *OCSP*. The signatures of the installation instructions and *IKM* are then validated with the PIA_cert. *CIA* then reboots to an offline secure state; ready for pre-state installation processes. PIA_cert is validated again after reboot, against *Root CA* and an offline CRL list; and the signature of IKM is validated with PIA_cert, where after the *CIA* requests *IKM* initialization by sending

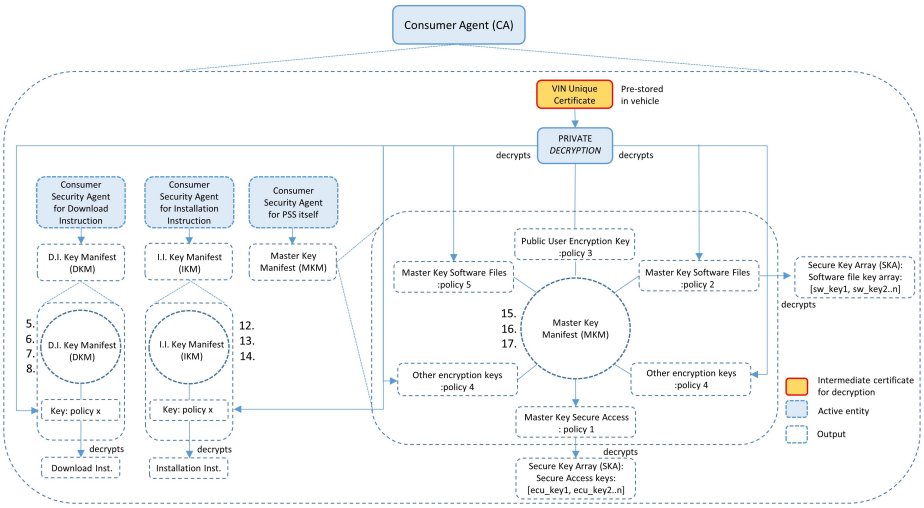

Fig. 5. Data distribution to the vehicle in relation to cryptographic material

the signed IKM and PIA_cert to the *CSA*. *Output: IKM.signed; PIA_cert;*

12. IKM key manifest initiation. The *CSA* validates the PIA_cert for authenticity towards *Root CA* and an offline *CRL*. The *CSA* then validates the *IKM* signature with PIA_cert, where after the IKM session key within *IKM* is decrypted with the pre-stored private asymmetric unique key and associated according to policy in the *IKM*, i.e., to be used for decrypting installation instructions.

13. Request of decryption of installation instruction. The *CIA* provides the signed installation instructions to the *CSA* and requests decryption. *Output: installation_instruction.signed;*

14. Decryption of installation instruction. The *CSA* validates the signature of the installation instructions with PIA_cert, decrypts with the IKM session key from the *IKM* in accordance with policy (i.e., decrypting installation instructions) and returns the decrypted installation instructions to the *CIA*. *Output: installation_instruction;*

15. Request MKM key manifest initiation. The *CIA* retrieves the encapsulated signed *MKM* and *SKA*, and PSA_cert from the decrypted installation instructions. The *CIA* validates the PSA_cert for authenticity towards *Root CA* and an offline *CRL*, and verifies signatures of the *MKM* and the *SKA* with the PSA_cert. *CIA* then requests *MKM* initialization by sending the signed *MKM* and PSA_cert to the *CSA*. *Output: MKM.signed; PSA_cert;*

16. MKM key manifest initiation. The *CSA* validates the *MKM* signature with the PSA_cert. MKM session keys within the *MKM* are decrypted with the pre-stored private asymmetric unique key and are associated with applications according to policy in the *MKM*.

17. Secure CIA - CSA interface established. Peri-state. The *CIA - CSA*

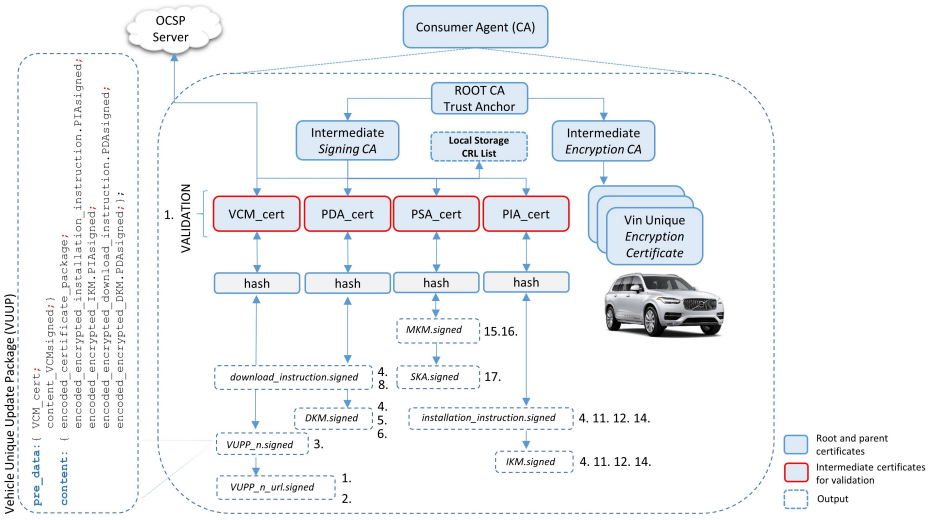

Fig. 6. Data distribution to the vehicle in relation to validation

communication interface is now initialized. The *CIA* can request decryption of software from the *CSA* by sending the encrypted file (or path/link) together with the corresponding encrypted sw_key retrieved from the *SKA*. Before decryption can start, the *CSA* validates the authenticity of software files with the VCM_cert and aborts the decryption request from the *CIA* if this fails. On the other hand, if it is successful, the *CSA* then decrypts the encrypted sw_key with the MKM session key for software files from the *MKM* and uses the sw_key to decrypt the software file. This interface is also used for unlocking ECUs using, e.g., security access to authorize the update. In this case, the challenge from the ECU is sent to the *CSA* together with the corresponding encrypted security access key from the array in *SKA*. *CSA* decrypts the encrypted security access key and processes the challenge from the ECU and can provide the results to the *CIA*. This approach allows the *CIA* to perform ECU unlock without exposing the security access key outside of *CSA*. The *CIA* is after this step ready to stream out software to the ECU.

4.3 Post-State Activities

CSA needs to have the possibility to sign post-state installation data, such as installation reports and logs potentially affecting upcoming software updates. A unique session signing key can be transported via *MKM* to *CSA* which can handle signing requests within a trusted application isolated within the *TEE*. The corresponding validation key can be stored in *CMS*. Part of post-state is to perform a complete vehicle software version request (readout). To provide authenticity, the readout can be signed by supported ECUs (e.g., if they contain pre-stored private keys) and validated by *CSA* with the help of the correspond-

ing validation keys attached to the *SKA*. These responses are then attached to the installation report. To provide confidentiality, *CSA* can encrypt installation reports and logs by using keys in *SKA*.

5 Implementation Considerations

The *CA* (all consumer entities) can as shown in Figure 4 be located in an ECU in the vehicle used for over-the-air updates or in a client workshop computer with a separated *CSA*. *CSA* in this case can be implemented in a hardware security device such as a Yubico key [5] or a smart card, with a pre-stored encryption/decryption certificate and a pre-stored *Root CA* acting as a trust anchor for validating certificates. Vehicle manufacturers can provide these hardware security devices to workshops, and also have full control to manage and revoke them. Depending on both security and performance requirements *CSA* can also be placed in a workshop HSM or even located in the cloud. Because of the proposed entity separation (implemented as modules), other approaches are also possible, such as integrating *CDA* and *CIA* in an update tool with *CSA* integrated in hardware or separated. It is also possible to use *CDA* separately (outside the vehicle) and securely push the update package to the in-vehicle *CIA* (e.g., via mutually authenticated communication). *CIA* then validates and executes the installation instructions and uses the ECU-internal *CSA* to perform secure transfer and execution of cryptographic operations. The different entities can be securely containerized out in the cloud or kept within vehicle manufacture premises. This solution fulfills the constraints and conditions stated in Section 2 and is therefore highly adaptable to accommodate various scenarios within the automotive industry.

6 Related Work

Samuel et al. suggest using a layered approach with the use of different roles and keys called *The Update Framework (TUF)* to ensure the integrity of the downloaded data, however, this approach does not consider the installation of these updates and is not adapted for vehicles [6]. In [7] T. Kuppusamy, propose an implementation and adaption of the TUF framework named *Uptane* for vehicle over-the-air updates, where the authors add more metadata to improve its resilience. Another approach proposed by Idress et al. [8] suggests deploying a new architecture where all in-vehicle ECUs use HSMs for over-the-air updates. In [9] Mahmud et al. propose an architecture that relies on sending multiple copies to secure the software update, an approach which we believe is not realistic due to infrastructure constraints. M. Steger et al. propose a framework named *SecUp* which uses handheld devices to wirelessly connect and update vehicles over an IEEE 802.11s mesh network for local environments (i.e. factory and workshop) [10]. In [11] Nilsson et al. present an approach for securing firmware updates over-the-air by combining encryption, hashing, and signing of firmware by chaining fragments. In [12] Nilsson et al. continue their work

on hash-chain verification and suggest an over-the-air update framework that validates firmware after it is flashed to the ECUs, however, this requires all in-vehicle ECUs to be adapted to this approach and that integrity verification of the download is solved by other means.

However, the aforementioned solutions lack necessary details for a *unified* and *versatile* approach that supports updates over-the-air; from a workshop computer; at the factory production; use of diagnostic update tools; and third-party vehicle platform users e.g., using the vehicle as a base controlled by other autonomous systems. As a case in point, *Uber* is using the *Volvo Cars* platform in their fleet of cars [3]; a scenario which most likely will become more prevalent in the future due to increased sustainability requirements. Thus, solutions such as ride-sharing will probably be more common where collective fleets of cars require integrating 3rd party hardware and software which are dependent on a unified software update framework. UniSUF supports 3rd party components by appending its related data to the VUUP file i.e., adding additional instructions and at the same time keeping the VUUP file intact. Moreover, other details are missing in current solutions such as required installation instructions including handling of necessary pre-, peri- and post-state phases and secure transport and secure execution of ECU-specific cryptographic keys. UniSUF considers all these three states, and ensures a secure transport to a trusted execution environment, following a secure execution for all sensitive data. Many existing solutions also consider changes to all ECUs which usually is not possible; something which is not required by UniSUF. The mentioned versatility of UniSUF can keep required adaptions of the vehicle as well as the required cost to a minimum.

7 Future Work and Conclusion

We have contributed with a comprehensive and novel unified software update framework named UniSUF, well aligned with industry stakeholders. As part of future work, we have already begun defining an attacker model and started a security analysis of our proposed solution. We aim to perform a more detailed evaluation of UniSUF, including a discussion on the fulfilled requirements as well as a comparison to other approaches, in a future paper.

UniSUF is made to accommodate various scenarios for the automotive domain by encapsulating needed data into one single file, a Vehicle Unique Update Package (VUUP). This vehicle unique file can be processed within a vehicle ECU, using a workshop computer, at factory production, with a diagnostic update tool, or in other compositions. Moreover, the complete update process can be performed without any external communication dependencies, since all files are inherently secured. A continuous secure software update process is a prerequisite for facilitating vehicle resilience towards cyber attacks in a rapidly changing environment. We believe our contributions in this paper can facilitate further research in this area, towards securing the connected car.

Acknowledgment. This research was supported by the CyReV project (2019-03071) funded by VINNOVA, the Swedish Governmental Agency for Innovation Systems.

References

1. for Standardization, I.O.: Road vehicles — Software update engineering. https://www.iso.org/standard/77796.html (2021), accessed: 2021-06-02
2. for Europe (UNECE), U.N.E.C.: UN Regulation No. 156 - Software update and software update management system. https://unece.org/transport/documents/2021/03/standards/un-regulation-no-156-software-update-and-software-update (2021), accessed: 2021-06-02
3. Corp., V.C.: Volvo Cars and Uber present first autonomous drive-ready production car. https://group.volvocars.com/news/future-mobility/2019/volvo-and-uber-present-autonomous-drive-ready-xc90 (2019), accessed: 2021-06-02
4. Corp., V.C.: Volvo Cars teams up with world's leading mobility technology platform DiDi for self-driving test fleet. https://www.media.volvocars.com/global/en-gb/media/pressreleases/280668/volvo-cars-teams-up-with-worlds-leading-mobility-technology-platform-didi-for-self-driving-test-flee (2021), accessed: 2021-06-07
5. yubico: Protect your digital world with YubiKey. https://www.yubico.com/ (2021), accessed: 2021-06-02
6. Samuel, J., Mathewson, N., Cappos, J., Dingledine, R.: Survivable key compromise in software update systems. pp. 61–72 (12 2010)
7. Karthik, T., Brown, A., Awwad, S., McCoy, D., Bielawski, R., et al.: Uptane: Securing software updates for automobiles. 14th ESCAR Europe (2016)
8. Idrees, M.S., Schweppe, H., Roudier, Y., Wolf, M., Scheuermann, D., Henniger, O.: Secure automotive on-board protocols: A case of over-the-air firmware updates. In: Strang, T., Festag, A., Vinel, A., Mehmood, R., Rico Garcia, C., Röckl, M. (eds.) Communication Technologies for Vehicles. pp. 224–238. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
9. Mahmud, S., Shanker, S., Hossain, I.: Secure software upload in an intelligent vehicle via wireless communication links. In: IEEE Proceedings. Intelligent Vehicles Symposium, 2005. pp. 588–593 (2005)
10. Steger, M., Boano, C.A., Niedermayr, T., Karner, M., Hillebrand, J., Roemer, K., Rom, W.: An efficient and secure automotive wireless software update framework. IEEE Transactions on Industrial Informatics 14(5), 2181–2193 (2018)
11. Nilsson, D.K., Larson, U.E.: Secure firmware updates over the air in intelligent vehicles. In: ICC Workshops - 2008 IEEE International Conference on Communications Workshops. pp. 380–384 (2008)
12. 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)