Towards a formal verification of secure vehicle software updates

Computers & Security 161 (2026) 104751

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full Length Article

Towards a formal verification of secure vehicle software updates

Martin Slind Hagen [a], Emil Lundqvist [a], Alex Phu [a], Yenan Wang [a], Kim Strandberg [a,b,*], Elad Michael Schiller [a]

[a] Computer Science and Engineering, Chalmers University of Technology, Gothenburg, 41296, Sweden
[b] Department of Research and Development, Volvo Car Corporation, Gothenburg, 40531, Sweden

ARTICLE INFO

Keywords:
Provable security
Vehicular systems
Secure software updates

ABSTRACT

With the rise of software-defined vehicles (SDVs), where software governs most vehicle functions alongside enhanced connectivity, the need for secure software updates has become increasingly critical. Software vulnerabilities can severely impact safety, the economy, and society. In response to this challenge, Strandberg et al. [escar Europe, 2021] introduced the Unified Software Update Framework (UniSUF), designed to provide a secure update framework that integrates seamlessly with existing vehicular infrastructures. Although UniSUF has previously been evaluated regarding cybersecurity, these assessments have not employed formal verification methods. To bridge this gap, we perform a formal security analysis of UniSUF. We model UniSUF's architecture and assumptions to reflect real-world automotive systems and develop a ProVerif-based framework that formally verifies UniSUF's compliance with essential security requirements — confidentiality, integrity, authenticity, freshness, order, and liveness —demonstrating their satisfiability through symbolic execution. Our results demonstrate that UniSUF adheres to the specified security guarantees, ensuring the correctness and reliability of its security framework.

1. Introduction

Connected cars are quickly becoming the norm, with 96 % of manufactured cars in 2030 expected to have connectivity features (Council, 2020). These connected cars feature a multitude of electronic control units (ECUs), with more than 100 ECUs per vehicle (Narisawa et al., 2022). Maintaining and regularly updating these ECUs is critical to prevent security vulnerabilities. The massive scale of the automotive industry, with over 70 million cars sold worldwide annually (Scotiabank, 2024), makes it a prime target for malicious actors. If an attacker compromises the software update process, the result can be malware installation, sensitive data leakage, and even vehicle hijacking, leading to devastating financial, social, and potentially fatal consequences.

Despite the importance of secure updates, ensuring the confidentiality, integrity, and correct execution of the software update process for connected vehicles remains a highly challenging task. In particular, attackers could exploit weaknesses in the update process to install malicious software, eavesdrop on sensitive information, or revert vehicle software to an obsolete version containing vulnerabilities. Moreover, the sheer number of vehicles and ECUs in each car presents a scalability challenge, complicating the implementation of robust security measures across the entire fleet.

1.1. Existing solutions and their shortcomings

Frameworks have been proposed to address these challenges, including the Unified Software Update Framework (UniSUF) (Strandberg et al., 2021a). UniSUF proposes a reference architecture intended to serve as input to standards for secure software updates. UniSUF's specifications were driven by the following development goals (Strandberg et al., 2023).

Confidentiality: To hinder eavesdropping.

G1: Ensure software confidentiality during the software update process.
G2: UniSUF session keys can only be viewed in decrypted format by authorized software components.

Integrity and Authenticity: To hinder spoofing and tampering.

G3: The software is authentic against a certificate and remains unchanged during the update process.
G4: Only authentic resources are processed.

Freshness: To hinder replay attacks.

G5: An adversary should be unable to revert a vehicle's software to a previously installed version.
G6: The system creates unique software distribution files per software update. Each such file can only be used for a designated vehicle.

* Corresponding author.
E-mail addresses: slindm@student.chalmers.se (M.S. Hagen), emilundq@student.chalmers.se (E. Lundqvist), alexp@student.chalmers.se (A. Phu), yenan@chalmers.se (Y. Wang), kim.strandberg@volvocars.com (K. Strandberg), elad.schiller@chalmers.se (E.M. Schiller).

https://doi.org/10.1016/j.cose.2025.104751
Received 21 October 2024; Received in revised form 30 September 2025; Accepted 3 November 2025
Available online 11 November 2025

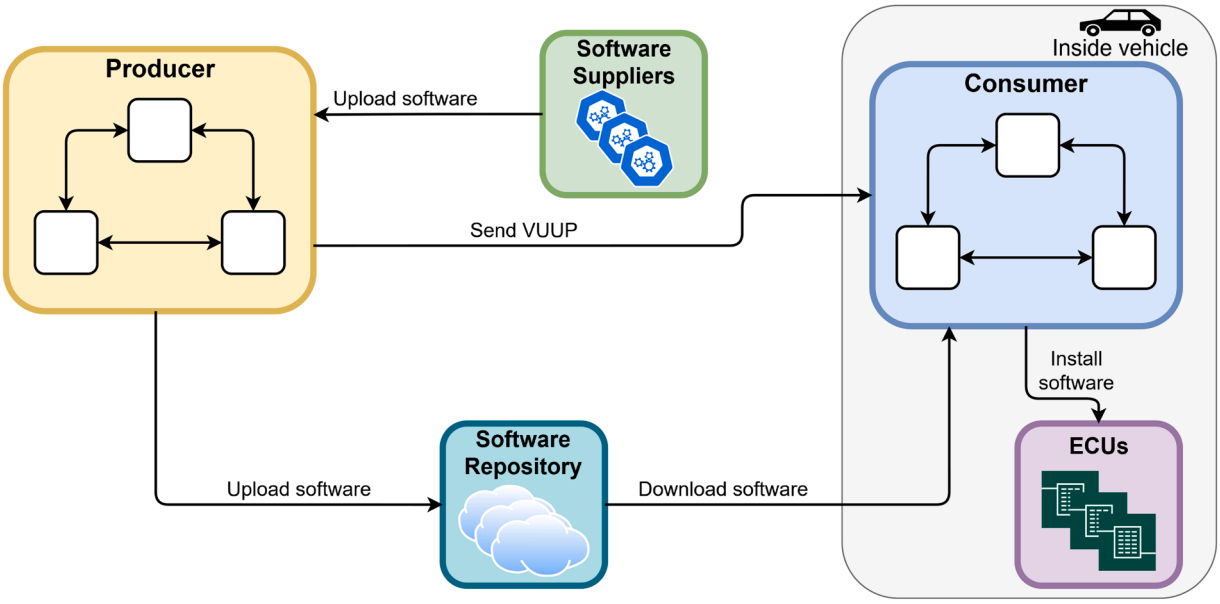

Fig. 1. High-level overview of the UniSUF architecture showing the main entities and update flow. Section 5 details the architecture components, see Figs. 15–31. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Order: To hinder vulnerabilities that take advantage of running the process in an unintended order.

G7: The software update process should follow the correct order. **Liveness:** Hinder DoS attacks.

G8: The software update process should eventually terminate, regardless of success or error.

Fig. 1 provides a high-level overview of the UniSUF architecture, illustrating the main entities involved in the update process: the Producer, the Consumer, Software Suppliers, the Software Repository, and Electronic Control Units (ECUs). Software update packages, referred to as VUUPs, are generated by the Producer based on the latest software versions provided by the Software Suppliers, which have also passed internal testing. The Producer delivers the VUUPs to the Consumer and uploads the software to the Software Repository. Upon receiving a VUUP, the Consumer processes it and installs the updates on the respective ECUs according to the instructions specified in the VUUP. Section 4 presents detailed architectures and protocol steps.

UniSUF's prior evaluations focus primarily on practical deployments and lack formal verification of its security guarantees. This leaves the potential for subtle vulnerabilities that attackers could exploit, notably by exposing cryptographic keys or violating the sequence of operations during the update process.

Several scientific challenges are associated with the formal verification of systems like UniSUF. These include **[1: Confidentiality]** ensuring that the software update process maintains the confidentiality of secret information throughout execution; **[2: Integrity and Authenticity]** verifying that no unauthorized modifications occur during the update process; and **[3: Order and Liveness]** guaranteeing that the update process follows the correct sequence of actions and terminates appropriately. These challenges are compounded by the complexity of modeling such systems in formal verification tools such as ProVerif (Blanchet, 2016; Blanchet et al., 2018), which requires a precise representation of the security assumptions and the adversary model.

The challenges associated with the formal security of UniSUF raise the following research questions.

RQ1: UniSUF has certain secrets, essential for its operation, for example, cryptographic keys and disseminated software. The question is whether UniSUF's operation might expose any of its secrets.

RQ2: How can we guarantee that the software that UniSUF disseminates is obtained from an authentic source and not manipulated?

RQ3: How can we guarantee that in UniSUF, it is impossible to perform a software update with obsolete software versions?

RQ4: Appropriate software updates require a specified order of actions. How can we guarantee that UniSUF's update operation proceeds orderly?

RQ5: How do we know that the software update process always ends?

1.2. Our contribution

We address the above challenges and research questions by conducting a formal security analysis using symbolic execution in ProVerif. Our key contributions to advancing the state of the art are as follows:

- We model UniSUF's architecture (Section 4) and assumptions (Section 3) to reflect real-world automotive systems. This model is represented in ProVerif (Section 5) to show the formal satisfiability of essential security properties, including confidentiality, integrity, authenticity, freshness, order, and liveness.
- We introduce novel techniques (Section 6) to simulate UniSUF in a symbolic execution environment within ProVerif, allowing us to formally analyze critical aspects, such as software authenticity and the correct sequence of operations.
- Through formal proofs and experimental results, we ensure that UniSUF's update process terminates and follows the correct procedural order.

The primary outcomes of our results are as follows.

- A rigorous formulation of UniSUF's security requirements, emphasizing confidentiality, integrity, authenticity, freshness, order, and liveness in the software update process.
- An open-source ProVerif-based framework to formally verify UniSUF's compliance with these security guarantees.
- Through ProVerif's symbolic execution environment, we demonstrate that UniSUF can satisfy the proposed security requirements under realistic adversary models while considering a system architecture that represents real-world deployment.

Our results show that UniSUF's architecture effectively prevents attacks, such as secret exposure and replay attacks while ensuring that software updates proceed in the correct sequence and terminate as expected. This demonstrates that UniSUF's architecture, assumptions, and security requirements can be formally satisfied, providing strong assurances for its security in real-world deployments (Section 7).

An important clarification concerns the notion of computational cost and scalability. The effort reported in this paper concerns the formal specification and verification of requirements, rather than the execution of update protocols in deployed vehicles. Consequently, any computational overhead arises exclusively at *design time*, when symbolic verification tools such as ProVerif analyze the model and proof obligations. These costs depend on model size and tool runtime, but are incurred only once during the verification process. They do not translate into runtime overhead for the vehicle platform or its backend infrastructure. Moreover, symbolic verification techniques, including ProVerif, are conceptually scalable in the sense that they reason over unbounded sessions and adversaries. Our approach, therefore, provides strong security assurances without introducing performance or scalability penalties in production systems.

To ensure the reproducibility of our results and to encourage further development, prospective implementation details to the source code can be found in our complementary technical report (Hagen et al., 2025).

2. Related work

Analyzing the security of a complex system requires consideration of potential threats. Strandberg et al. (2021b) proposes a security and resilience framework, i.e., Resilient Shield, utilizing a security enhancement methodology (Strandberg et al., 2018) along with mitigation mechanisms from Rosenstatter et al. (2020) in light of an analysis of attacks targeting vehicles. Based on the identified attacks, the authors establish security goals and specify the directives required to achieve them. Strandberg et al. (2023) details specifications for meeting security goals with an emphasis on vehicle software updates. The authors provide a detailed threat analysis of UniSUF for different threat scenarios, aligned with security goals (Strandberg et al., 2021b), and detailed requirements for the UniSUF architecture (Strandberg et al., 2021a).

2.1. Formal verification tools

There are formal verification tools that can either assist or automate proofs. For instance, we can express systems and their properties as logical formulas and use theorem provers such as Coq (Chlipala, 2013) and Isabelle (Wenzel et al., 2008) to either interactively or automatically prove the specified properties. However, formulating the entire implementation of complex systems is tedious and error-prone. Instead, it can be more intuitive to verify a system based on a formal model using model checkers such as SPIN (Holzmann, 1997), UPPAAL (Behrmann et al., 2006), and TLA+ (Lamport, 2002). Model checkers take a formula and a model and then verify whether the formula holds within the model. The appropriate model checker can simplify the process of deriving a system model. For example, UPPAAL is a model-checking environment that provides both a graphical interface and a modeling language to model real-time systems (Behrmann et al., 2006), which makes it well suited for time-critical systems.

For our purposes, we also need to model the adversary, to be explicitly defined in general model checkers. In contrast, cryptographic protocol verifiers, such as ProVerif (Blanchet et al., 2018), CryptoVerif (Blanchet, 2007), and Tamarin Prover (Meier et al., 2013), are designed to focus on security and implicitly incorporate the adversary model. For instance, these cryptographic protocol verifiers verify their models under the assumption of the Dolev-Yao adversary model (Dolev and Yao, 1983). Therefore, cryptographic protocol verifiers are more appropriate for security properties as they eliminate the need to model an adversary.

Wang (2024) provides protocols to attest that the manufacturer has approved vehicle hardware. These protocols enable the replacement of old components with new ones and the attestation of all vehicle components during start-up. Wang (2024) used formal methods to prove the correctness of his protocols. He defined the system model, assumptions, and requirements and used ProVerif (Blanchet et al., 2018) to formally verify that his protocols fulfill the specified requirements given the system model and assumptions. The work by Wang has been an inspiration for our own research in this area.

Basin et al. (2018) used Tamarin Prover to find weaknesses in the Authentication and Key Agreement protocol used by 5G. Tamarin Prover has also been used to analyze WiFi Protected Access 2 (Cremers et al., 2020) and Transport Layer Security 1.3 (TLS 1.3) (Cremers et al., 2017). In Bhargavan et al. (2022), an analysis of TLS 1.3 was performed with ProVerif (Blanchet et al., 2018), where they also looked at a privacy extension for TLS 1.3 called Encrypted Client Hello.

2.2. Formal verification of automotive and IoT software update protocols

Several formal methods have been applied to analyze the security of software update mechanisms in automotive and IoT systems. A significant body of work has focused on *Uptane*, which is a widely adopted framework for secure automotive software updates, considered by many in the industry as a de facto standard. For example, Kirk et al. (2023) develop a Communicating Sequential Processes (CSP) model of the Uptane protocol together with an attacker inspired by the Dolev and Yao (1983), and use the Failures–Divergences Refinement (FDR4) checker to derive exhaustive traces representing potential security violations. These symbolic traces are then translated into executable test cases and run against a reference Uptane implementation on a hardware testbed. Their analysis validated Uptane's defenses against the modeled threats, but also demonstrated that certain attacks, such as freeze or spoof, become feasible when specific defenses (e.g., expiration checks) are omitted in practice. Kirk et al. illustrate how CSP-based refinement and test generation can reveal both specification-level exposures and implementation-dependent weaknesses.

In a complementary study, Lorch et al. (2024) present a comprehensive automated verification of Uptane by combining the Kind 2 infinite-state model checker with the Tamarin cryptographic protocol verifier in an eager combination. Unlike prior Uptane analyses that faced state-space explosion or termination issues, their workflow achieves termination in most instances while covering fine-grained message structures. This analysis rediscovered all five previously known vulnerabilities and identified six new ones, all of which were acknowledged by the Uptane standards body. Their work illustrates how combining model checking with cryptographic reasoning can provide both scalability and coverage under diverse adversary capabilities, including multiple key-compromise scenarios.

Another formal analysis of Uptane is provided by Boureanu (2023), who used the Tamarin prover to model version 2.0 of the protocol. Building on the threat model in the Uptane 2.0 standard, this work introduced a hierarchical set of attacker tiers, including compromise of primary or secondary ECUs and of individual repositories. The authors validated a set of security and privacy requirements that go beyond those specified in the standard, including properties such as agreement and temporal correctness. They identified several flaws that were responsibly disclosed to the Uptane Alliance. While the verification required substantial computational resources, the study demonstrated that symbolic analysis of Uptane 2.0 at scale is feasible and can directly inform improvements to the evolving standard.

Earlier, Mahmood et al. (2020) introduced the first model-based security testing approach for automotive over-the-air (OTA) updates, targeting the Uptane reference implementation. Their framework combined attack trees for threat modeling with an automated tool that

generates and executes corresponding test cases. In a proof-of-concept demonstration, they showed a simulated attack compromising Uptane repositories with malicious firmware. While this approach provides systematic coverage of known threats, its reliance on attack trees limits its ability to capture zero-day or composite attacks, motivating later exhaustive analyses based on formal models.

Not all formal verification studies target Uptane; some address other software update protocols or different aspects of the OTA software update process. Ponsard and Darquennes (2021), for instance, apply the Tamarin prover to an IoT-oriented firmware update scheme called *UpKit* (Langiu et al., 2019). Their work focuses on proving selected properties, most notably firmware integrity and the freshness of update requests, under a Dolev-Yao adversary. By modeling the cryptographic operations and message flows in Tamarin's rewrite-rule framework, they verify that UpKit meets its targeted requirements, though the analysis does not cover the full range of software update properties.

Other researchers have looked at implementation-level assurance. Mukherjee et al. (2021) propose an Uptane-based OTA update solution deployed entirely inside a Trusted Execution Environment (TEE) on commercial off-the-shelf embedded hardware (ARM TrustZone). Using SAW, the Software Analysis Workbench, they verify code-level security properties of the Uptane client within OP-TEE and demonstrate the approach on a Raspberry Pi 3B. Their threat model assumes that the normal-world OS and network may be compromised, while the TEE and server remain trusted. This work highlights how hardware-enforced isolation can be combined with formal software analysis to strengthen OTA software update implementations.

An earlier foundational effort by Pedroza et al. (2011) introduced a holistic formal methodology for secure OTA updates in vehicles. Their approach extended the AVATAR SysML-based framework to incorporate both security and safety requirements, and integrated formal verification into a model-based development flow. Specifically, AVATAR models of OTA update protocols were translated to ProVerif for symbolic analysis under a Dolev-Yao adversary, while safety aspects were verified using UPPAAL. This allowed properties such as secure authentication, confidentiality, and data integrity of updates to be formally checked already at the design stage. The emphasis was on combining early requirements engineering with formal security and safety verification, thereby demonstrating as early as 2011 the feasibility of security-by-design for vehicular update systems.

Our work on UniSUF diverges from these studies in both goal and approach. Rather than assessing an existing protocol for hidden flaws, we focus on *formally specifying a new multi-ECU update framework and verifying that it satisfies a comprehensive set of security requirements by design*. We encode UniSUF's update procedures in ProVerif and prove, under a standard Dolev–Yao adversary model, that critical properties such as authenticity of the update source, integrity of firmware payloads, confidentiality of sensitive data, and freshness of update commands hold. Ordering is ensured through replay-protection and monotonicity checks, while liveness is argued at the design level rather than mechanically proved. Our verification approach is requirement-driven: each high-level requirement is formalized as a ProVerif query, and the framework is decomposed into interrelated sub-protocols to mirror the modular structure of the UniSUF framework. This decomposition facilitates scalability and clarity. In positioning, UniSUF extends the requirements-oriented perspective already seen in early frameworks such as AVATAR, while differing from vulnerability-focused analyses of Uptane and its implementations (e.g., attack-tree testing, CSP/FDR model-based testing, or hybrid model checking with cryptographic reasoning). Taken together with Boureanu's symbolic proof of Uptane 2.0 and Mahmood's attack-tree-based framework, these efforts trace the evolution of formal verification for software update protocols, from semi-formal test generation, through symbolic and hybrid analyses, to requirements-driven verification by design for new multi-ECU frameworks.

3. Preliminaries

We provide our definitions, assumptions, and requirements.

3.1. System settings

The system consists of computing entities that interact through communication channels. We assume the system is synchronous and that all entities can access universal time. Every entity has a state, including its variables and all messages in its incoming communication channels. The entities update their states by taking atomic steps. Each step performs an internal computation that takes one time unit. These steps can also receive or send messages. An unbounded sequence of atomic steps, X, denotes an execution. For a given entity E, an execution of E is a subsequence of X from which all steps not taken by E are omitted. Each of our studied problems is solved by a distributed algorithm. The system entities collectively execute an algorithm by individually running a sequence of tasks. Each problem is divided into the sub-problems we analyse in Section 5. The last task in each sequence is the *halt* task.

3.2. Threat model

Based on the Dolev-Yao model (Dolev and Yao, 1983), the adversary has complete control over the communication between entities. In addition, message interception, injection, and modification are also possible. The adversary may also delay the delivery of the message by a bounded time η; therefore, communication channels are assumed to be reliable but without guarantees of FIFO ordering. This is derived from Wang (2024).

3.3. Cryptographic primitives, notations, and assumptions

We assume access to the standard cryptographic primitives in Table 1. We emphasize the requirement for an authenticated symmetric encryption scheme, such as AES-GCM, which ensures that the encrypted data remain confidential and are authenticated to verify the sender (McGrew and Igoe, 2015, Sec. 1). This is necessary because some secrets must be authenticated, such as inputs to the Trusted Execution Environment (Strandberg et al., 2023, Tab. 1). To simplify our model, we define a certificate as valid if it is signed by the root certificate. For simplicity, we omit the explicit notation of these signatures in our cryptographic descriptions, assuming that all valid certificates are implicitly signed. UniSUF operates under the assumption of a trusted root certificate (Strandberg et al., 2021a, 2023). Consequently, we assume that this root certificate is securely pre-installed in the vehicle.

UniSUF assumes secure and reliable communication between entities. This can be achieved, for example, by using SSH (Lonvick and Ylonen, 2006) or mutual TLS (Rescorla, 2018; Campbell et al., 2020). Therefore, adversaries cannot eavesdrop, tamper, or replay messages, with the exception of one link (see Section 5.3.5) for which UniSUF uses reliable non-FIFO communication without security guarantees. UniSUF uses cryptographic materials (see Section 4.1), such as symmetric keys and cryptographic signatures.

3.4. Update rounds

Where applicable, cryptographic materials are assigned to individual vehicle identification numbers (VIN), v_{id}, and must be distributed within a specified deadline, t_e (expiration time) (Strandberg et al., 2023; Strandberg, 2024). The mapping is secured by appending the v_{id} and t_e to the cryptographic material and signing the resulting data. We use the pair (v_{id}, t_e) to refer to the software *update rounds* in UniSUF. We choose the term rounds, rather than sessions, to avoid confusion with the term sessions used, e.g., for SSH (Gasser et al., 2014, Sec. 2). When no v_{id} can be specified, we omit the VIN from our update round notation and use only t_e.

Table 1
Notations used in the communication schemes. Inspired by Wang (2024, Table 1).

Notation	Description
Obj_{Key}	Symmetric key of type Obj.
E_{Cert}	Entity E's certificate is an asymmetric key pair. The key pair's public key is signed by the root certificate and only E knows the private key. The private key is omitted when E_{Cert} is included in a data structure or message.
$E_{PrivateKey}$	The private key belonging to E_{Cert}.
$E_{PublicKey}$	The public key belonging to E_{Cert}.
$AsymEnc(Message, E_{PublicKey})$	Asymmetrically encrypts the given $Message$ with $E_{PublicKey}$, thereby creating $CipherText$.
$AsymDec(CipherText, E_{PrivateKey})$	Asymmetrically decrypts the given $CipherText$ with the private key $E_{PrivateKey}$, thereby recovering $Message$.
$SymEnc(Message, Obj_{Key})$	Symmetrically encrypts the given $Message$ with the symmetric key Key, thereby creating $CipherText$.
$SymDec(CipherText, Obj_{Key})$	Symmetrically decrypts the given $CipherText$ with the symmetric key Obj_{Key}, thereby recovering $Message$.
$AuthSymEnc(Message, Obj_{Key})$	Encrypts the $Message$, similar to $SymEnc(Message, Obj_{Key})$, but will also include an authentication tag that hinders $Message$ from being changed, and validates if the Obj_{Key} is used to encrypt $Message$.
$AuthSymDec(CipherText, Obj_{Key})$	Decrypts the $CipherText$, similar to $SymDec(CipherText, Obj_{Key})$, but any change to the encrypted message or any message encrypted by $Obj'_{Key} \neq Obj_{Key}$ will be detected.
$Hash(Message)$	Creates a hash H of $Message$, such that $Message$ cannot be retrieved from H.
$Sign(H, E_{PrivateKey})$	Creates a signature S of the hash H by encrypting H with $E_{PrivateKey}$, such that decrypting S with $E_{PublicKey}$ returns H, i.e., $AsymDec(S, E_{PublicKey}) = H$.
$[Message]_E$	Represents data that is signed by $E_{PrivateKey}$. It is shorthand for $Message \mid\mid Sign(Hash(Message), E_{PrivateKey})$.
$Create_{Obj}(args)$	A function that creates an object of type Obj. Optional arguments $args$ can also be included. The creation details may vary with different values of Obj and $args$.
$Request(Obj)$	Creates a flag for requesting an item or functionality of type Obj. Such flags are used in messages to model the various requests sent between entities in UniSUF (see Section 5).
$Success(Obj)$	Creates a flag stating that item of type Obj was successfully initiated. Such flags are used in messages to model success statuses sent between entities in UniSUF (see Section 5).
$i_1 \mid\mid \ldots \mid\mid i_n$	Represents concatenation of multiple items, more specifically from i_1 to i_n. When the three dots operator (…) notation is used at the end of the concatenation, e.g. $i \mid\mid \ldots$, it indicates that additional but unspecified items are being concatenated after i. Note that an item can be any data.
$(i_1, \ldots, i_i, \ldots) = I$	Items i_1 and up to i_i are extracted from the set of items I. The optional dots at the end signify that more items left in I are ignored.

For a given problem and its algorithm, an execution of an algorithm is denoted as an update round execution. We assume that message transmissions include an update round identifier. Therefore, entities can learn about new update rounds and associate each execution of their task sequences with an update round. We denote this execution as an entity's execution of an update round, which is a subsequence of the update round execution. We also assume that all entities have a persistent log of all received messages. Each message is identified by the tuple (r, d), where r is the update round identifier and d is the cryptographic material in the message. All entities drop any message that is already in the persistent log.

3.5. Problem definition

We present our requirements for UniSUF, using the goals derived in Section 1.1.

System-level Requirements 3.5.1 to 3.5.6 specify UniSUF at the system level. The System-level Requirements 3.5.1 to 3.5.3 and 3.5.5 depend on requirements specified for each UniSUF sub-problem. These requirements are presented in Section 5. Namely, one derives the specifications of UniSUF sub-problems by specifying the sub-problems' *set of secrets* (S), the *set of cryptographic materials* (D), the *set of procedures* ($\{\ell_1, \ell_2, \ldots\}$) that handles cryptographic materials, and the ordering constraints on the procedure invocations, which we call the *handling partial order* ($\mathcal{P}(\ell)$).

System-level Requirement 3.5.1 (Confidential Secrets). *Let X be an update round execution and S be the* set of secrets *in X, which we specify per sub-problem (see Section 5). There is no $s_i \in S$ such that an adversary A can obtain s_i during X.*

System-level Requirements 3.5.2 to 3.5.4 consider a set of cryptographic materials D, which we specify for each sub-problem in Section 5. Each element in D is a pair; the first element is the cryptographic material itself, and the second element is the material's designated origin entity. As certificates are pre-existing cryptographic materials, we state that the origin entity of each certificate is the root CA.

System-level Requirement 3.5.2 requires that the adversary must not manipulate the cryptographic materials in D.

System-level Requirement 3.5.2 (Integrity of Cryptographic Materials). *UniSUF only uses cryptographic materials created by their designated origin entity, which we specify in Section 5, and is not modified by any other entity.*

System-level Requirement 3.5.3 considers procedures that handle cryptographic materials, such as material production, sending, receiving, validation, and software installation. To safeguard the vehicle during the most critical part of the update process, the vehicle enters offline mode (Strandberg et al., 2021a, 2023; Strandberg, 2024). Additionally, the ECUs are normally locked for security reasons, but must be unlocked to install new software. Therefore, we also classify ECU unlocking and vehicle offline mode activation as handling events.

Let X be an execution of update round r, d (documents) a subset of cryptographic materials, and ℓ a name of a procedure that handles d in event $e(r, d, \ell) \in X$. In Section 5, we list these procedures per task. We state that $e(r, d, \ell)$ is a handling event in X. Note that $d \subseteq D$.

System-level Requirement 3.5.3 specifies that cryptographic materials coupled with their handling procedures and associated with a specific update round are processed only during that update round; i.e., replays of round unique cryptographic materials between update rounds are not allowed.

System-level Requirement 3.5.3 (Inter-Round Uniqueness). *Let $e(r, d, \ell)$ and $e(r', d', \ell')$ be handling events during update round executions X and X', respectively. Suppose $(d, \ell) = (d', \ell')$. It holds that $r = r'$.*

System-level Requirement 3.5.4 specifies that cryptographic materials coupled to a specific update round are only processed once per procedure during that update round. In other words, replays of materials in an update round are not allowed.

System-level Requirement 3.5.4 (Intra-Round Uniqueness). *Let X be an update round execution and $e(r, d, \ell) \in X$ be a handling event. No event $e'(r, d, \ell)$ exists in X.*

System-level Requirement 3.5.5 specifies that procedures are executed in an order that follows UniSUF's specification.

System-level Requirement 3.5.5 (Integrity of Handling Events). *Let X be an update round execution. The occurrences of handling events, $e(r, d, \ell) \in X$, must follow a handling partial order $\mathcal{P}(\ell)$, which depends only on ℓ, where $\mathcal{P}(\ell)$ is specified per task (see Section 5).*

System-level Requirement 3.5.6 prevents non-termination of update rounds, given our system assumptions. The termination is considered timely if all UniSUF entities take the halt task before the update round expires; if not, it is considered late.

System-level Requirement 3.5.6 (Termination). *All executions of update rounds must terminate.*

Using the goals specified in Section 1.1, we discuss how the requirements satisfy the goals. Firstly, **G1** and **G2** are covered by **Confidential Secrets**, as we ensure that both the software and the cryptography keys are part of the set of secrets S. Note that **G2** is only partially covered because ProVerif (Blanchet et al., 2018) can only prove that the adversary is unable to learn the secrets and not that each secret is only available to a certain set of entities. Secondly, **G3** and **G4** are fulfilled by **Integrity of Cryptographic Materials** because the software and other materials produced by the origin entities are never modified throughout the update process. Thirdly, **Inter-Round Uniqueness** and **Intra-Round Uniqueness** together satisfy **G5** because any update round that processes the installation of software cannot be replayed to execute previous versions. **G6** is fulfilled because any VUUP can only exist in a single update round, and such an update round is directly coupled to a vehicle via the VIN (see Section 3.1). Fourthly, **G7** is fulfilled by **Integrity of Handling Events**, and lastly, **G8** is achieved by **Termination** since this requires that all executions terminate, legitimate or illegitimately.

4. UniSUF architecture

In this section, we detail the UniSUF architecture and its functionalities.

4.1. Cryptographic materials

As detailed in Table 2 and previously mentioned in Section 3.3, UniSUF uses different cryptographic materials. The signing process is shown in Fig. 2.

UniSUF uses Vehicle Unique Update Packages (VUUP) to install vehicle updates. A VUUP is an update package produced by UniSUF for a specific vehicle. It contains all the necessary cryptographic keys, certificates, and instructions for the software update. The internal structure of a VUUP file is shown in Fig. 3. The VUUP does not contain the actual software files; instead, Download Instructions are included to specify where software files can be downloaded. The specific implementation of Download Instructions is unspecified but can be seen as URLs to the software update files. Note that the Download Instructions is encrypted by a unique session key coupled to its update round. This key is retrieved from the Download Instruction Key Manifest (DKM).

As shown in Fig. 4, a key manifest consists of a symmetric session key that has been asymmetrically encrypted, accompanied by a policy defining the key's usage (see DKM, IKM, and MKM). Note that the UniSUF term session is equivalent to update rounds (see Section 3.4). In Table 2, none of the key manifests are signed, to remain consistent with the notation used by Strandberg et al. (2021a). However, when the key manifests are transmitted during tasks (see Section 5), all key manifests are signed with the certificates according to Fig. 2.

Additionally, a VUUP includes Installation Instructions encrypted by the session key from the Installation Instruction Key Manifest (IKM). The Installation Instructions contains the diagnostic instructions for installing software, the Master Key Manifest (MKM), and the Secure Key Array (SKA), as shown in Table 2. To guide task-specific requirements of Confidential Secrets (see System-level Requirement 3.5.1), we define the notations MKM_{Key} and SKA_{Key} to refer to all keys in MKM and SKA respectively.

Thus, the DKM and IKM session keys are packaged into key manifests (i.e., a master key plus a policy). In contrast, the MKM can contain multiple master keys, whereas the DKM and IKM each contain only one.

Strandberg (2024) defines encased software as software that undergoes a multi-layered protection process. Initially, the software is signed by the software supplier to ensure its integrity and authenticity. Next, as part of UniSUF, it is encrypted to ensure confidentiality. Finally, the encrypted software is secured with an additional signature, ensuring that the encrypted package can be validated to avoid initiating the decryption process in case of validation failure. The word encased should not be mixed up with the term encapsulated used by Strandberg et al. (2021a) to represent materials contained in the VUUP file.

UniSUF uses the term category to classify master keys based on purpose, such as decrypting software files, or unlocking ECUs to enable update capabilities.

The Vehicle Signed Order (VSO) is a readout per vehicle that contains detailed information about the vehicle, such as the onboard software versions. UniSUF uses this information and the latest available software versions to construct a software list containing the software files and configurations for a specific and vehicle unique software update. Download Instructions and Installation Instructions are then created based on the software list.

The software itself is located in external sources and not present in the actual VUUP file. The software files are signed by the software suppliers and associated with version numbers to prevent installations of older software versions (Strandberg, 2024). UniSUF validates the supplier signature, further encrypts the software, and appends another signature. Finally, the signed encrypted software is uploaded to the software repository.

4.2. System entities

As shown in Fig. 5, UniSUF consists of three entities: Producer, Consumer, and the Software Repository (Strandberg et al., 2023, Sec. 4). Additionally, UniSUF interacts with external entities, such as Software Suppliers and ECUs (Strandberg, 2024). UniSUF uses redundant entities and interacts with multiple Software Suppliers and vehicles with multiple ECUs (Strandberg et al., 2021a, 2023). However, for our proof, we consider a simplified system in which each vehicle has exactly one Consumer and one ECU. Additionally, all vehicles communicate with exactly one Producer and one Software Repository, and there exists only one Software Supplier.

4.2.1. Software repository

The UniSUF Software Repository is an entity that represents multiple distributed repositories (Strandberg et al., 2021a), mainly responsible for software storage, where each software file is associated with a specific download URL. However, in offline cases, the software can also be stored on Network-Attached Storage (NAS) or a USB stick (Strandberg, 2024).

4.2.2. Producer

Strandberg et al. (2021a) define the Producer as a collection of different sub-entities responsible for producing and securing software update packages. The Producer is also responsible for disseminating software

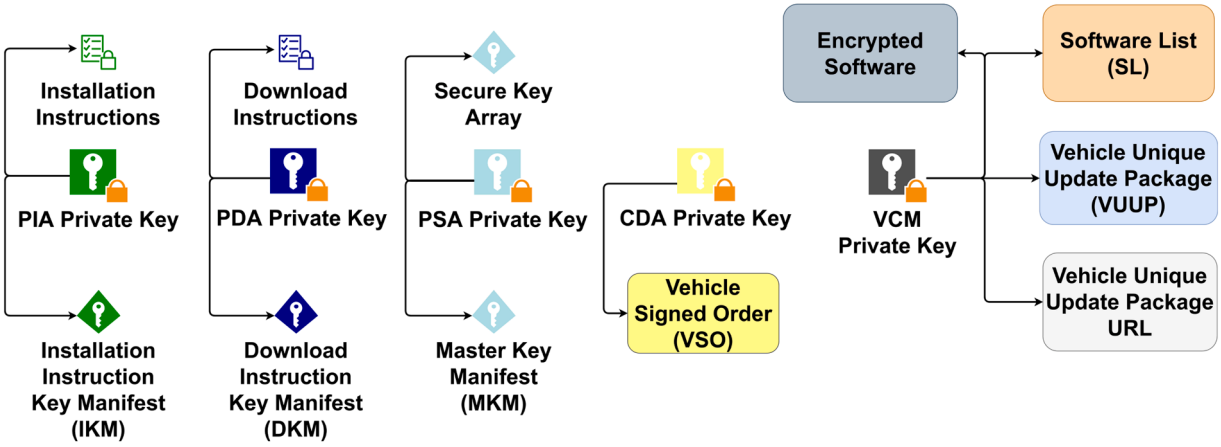

Fig. 2. Different cryptographic materials in UniSUF are shown, each with its respective key. Figure derived from Strandberg et al. (2021a, Fig. 3).

Table 2
UniSUF cryptographic materials, sorted in alphabetical order.

Cryptographic Material	Description
Certificate Package	$PDA_{Cert} \mid\mid PIA_{Cert}$
Download Instructions	$Create_{Download\,Instructions}([SoftwareList]_{VCM})$
Download Instruction Key Manifest (DKM)	$AsymEnc(DKM_{Key}, Vehicle_{PublicKey}) \mid\mid DKM_{Policy}$
Installation Instructions	$Create_{InstallationInstructions}([SoftwareList]_{VCM} \mid\mid [SKA]_{PSA} \mid\mid [MKM]_{PSA} \mid\mid\mid PSA_{Cert})$
Installation Instruction Key Manifest (IKM)	$AsymEnc(IKM_{Key}, Vehicle_{PublicKey}) \mid\mid IKM_{Policy}$
Master Key Manifest (MKM)	$MKM_{SecurityAccess} \mid\mid MKM_{Software} \mid\mid \ldots$
$MKM_{SecurityAccess}$	$AsymEnc(MKM_{SecurityAccess_{Key}}, Vehicle_{PublicKey}) \mid\mid MKM_{SecurityAccess_{Policy}}$
$MKM_{Software}$	$AsymEnc(MKM_{Software_{Key}}, Vehicle_{PublicKey}) \mid\mid MKM_{Software_{Policy}}$
Secure Key Array (SKA)	$SKA_{SecurityAccess} \mid\mid SKA_{Software} \mid\mid \ldots$
$SKA_{SecurityAccess}$	$AuthSymEnc(SecurityAccess_{Key_1}, MKM_{SecurityAccess_{Key}}) \mid\mid \ldots \mid\mid AuthSymEnc(SecurityAccess_{Key_n}, MKM_{SecurityAccess_{Key}})$
$SKA_{Software}$	$AuthSymEnc(Software_{Key_1}, MKM_{Software_{Key}}) \mid\mid \ldots \mid\mid AuthSymEnc(Software_{Key_n}, MKM_{Software_{Key}})$
Software	$Version \mid\mid Content$
$Software_{Encased}$	$[SymEnc([Software]_{Supplier}, Software_{Key})]_{VCM}$
Vehicle Unique Update Package (VUUP)	$VCM_{Cert} \mid\mid [VUUP_{Content}]_{VCM}$
$VUUP_{Content}$	$CertificatePackage \mid\mid [SymEnc(DownloadInstructions, DKM_{Key})]_{PDA} \mid\mid [DKM]_{PDA} \mid\mid [SymEnc(InstallationInstructions, IKM_{Key})]_{PIA} \mid\mid [IKM]_{PIA}$

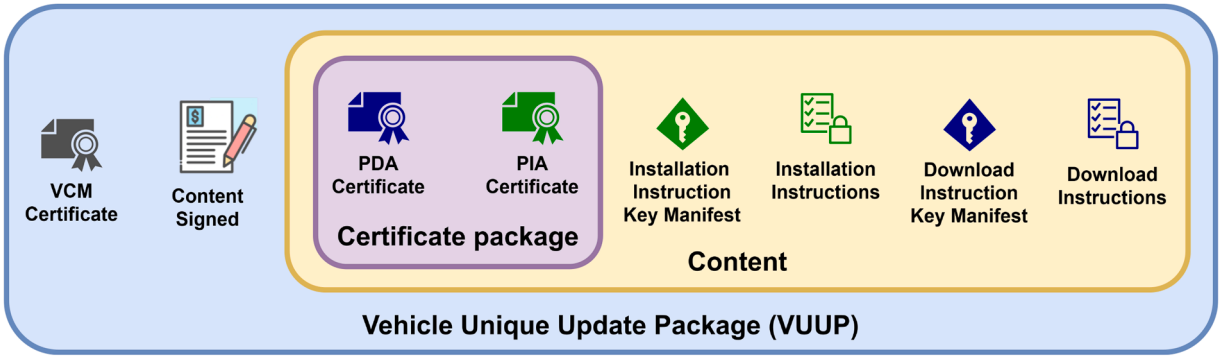

Fig. 3. Internal structure of a VUUP file. The blue items are used in the download process, while the green ones are used for the installation process. Note that the VUUP content has been signed by VCM_{Cert}. The figure is derived from Strandberg et al. (2021a, Fig. 3). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

to different storage repositories. As shown in Fig. 6, UniSUF has the following 12 producer entities (cf. Strandberg et al., 2021a, Table 1).

- **Producer Local Secure Storage** – Stores software files received from software suppliers.
- **Version Control Manager (VCM)** – Coordinates the producer entities and finalizes the creation of the VUUP file for a specific vehicle.
- **Producer Signing Service (PSS)** – Produces signatures on behalf of other entities.
- **Cryptographic Material Storage (CMS)** – Securely stores cryptographic materials, such as keys for decrypting software or unlocking ECUs, as well as certificates.
- **Producer Security Agent (PSA)** – Generates session keys and retrieves additional keys from the CMS, such as keys for unlocking ECUs, performing privileged diagnostic requests, and decrypting software. These additional keys are further encrypted with session keys, which are, in turn, encrypted using a vehicle-specific public certificate.

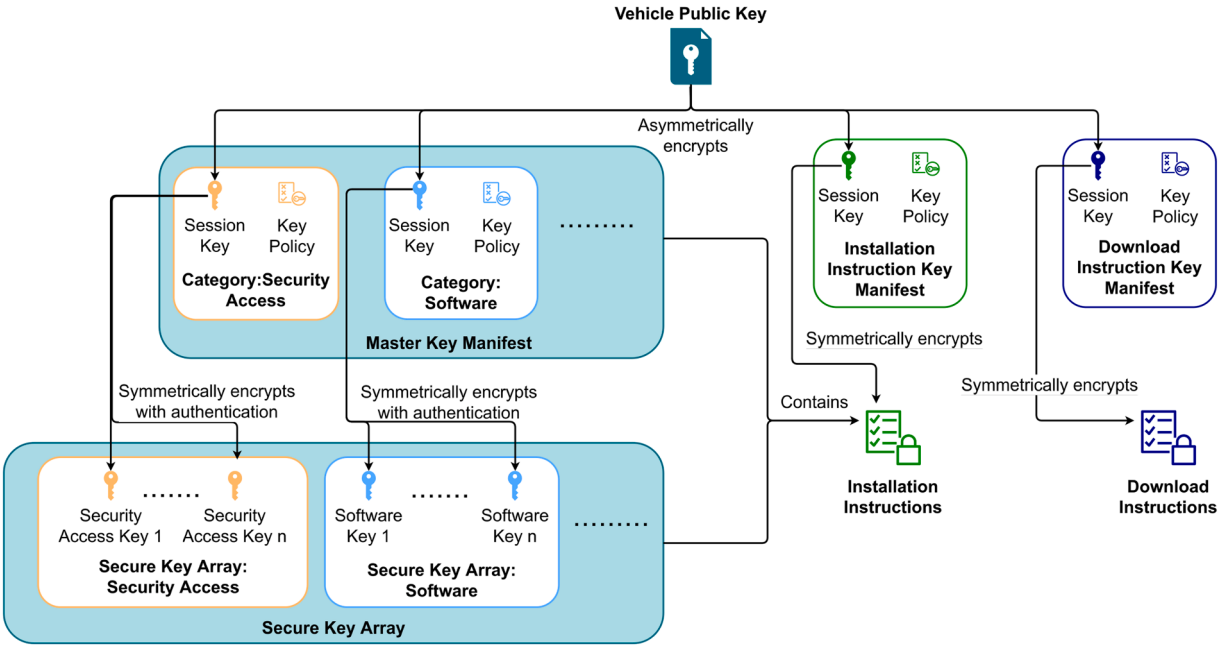

Fig. 4. Different cryptographic materials in UniSUF and how they are encrypted. Figure derived from Strandberg et al. (2021a, Fig. 2).

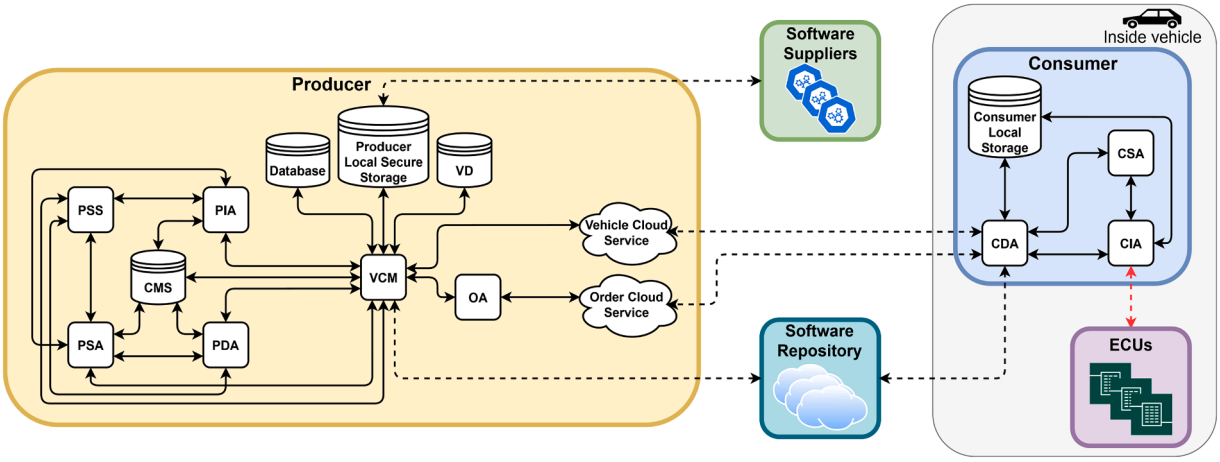

Fig. 5. Diagram of all the entities and their communication in UniSUF. Dotted arrows denote communication channels between entities from different modules. The black arrows denote secure communications, while the sole red arrow denotes an insecure communication link. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

- **Database** – Store URLs to software files located in software repositories.
- **Order Cloud Service** – Stores the Vehicle Signed Order (VSO) in a queue and URLs to VUUP files.
- **Order Agent (OA)** – Verifies the validity of incoming VSOs and starts the updating process by forwarding the request to the Version Control Manager (VCM).
- **Producer Download Agent (PDA)** – Creates download instructions from the software list received from VCM. Later in the process, the download instructions are encrypted, signed, and sent to VCM.
- **Producer Installation Agent (PIA)** – Creates installation instructions from the software list received from VCM. The installation instructions are bundled with cryptographic material, encrypted, signed, and sent to VCM.
- **VIN Database (VD)** – Stores data about unique vehicles and software versions.
- **Vehicle Cloud Service** – Stores the VUUP files and VCM certificates that can be downloaded by the Consumer via a VUUP URL.

4.2.3. *Consumer*

The Consumer is the Producer's counterpart and is responsible for decapsulating the VUUP and the processing of the download and installation instructions (Strandberg et al., 2021a, Sec. 4.2). UniSUF has the following four Consumer entities as shown in Fig. 7.

- **Consumer Local Storage** – Stores signed VUUP, and signed and encrypted software files.
- **Consumer Download Agent (CDA)** – Executes the download instructions and retrieves software from software repositories.
- **Consumer Security Agent (CSA)** – Provides a trusted execution environment where, e.g., decryption can occur in an isolated and secure space.

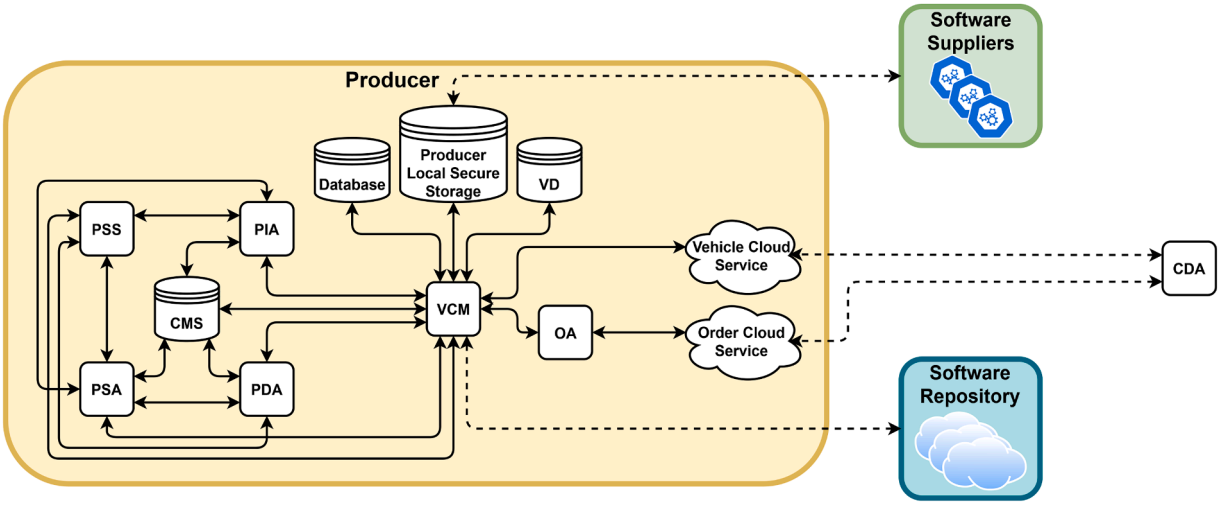

Fig. 6. The communication flow between the Producer entities. Dotted arrows denote communication channels between a Producer entity and a non-Producer entity. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

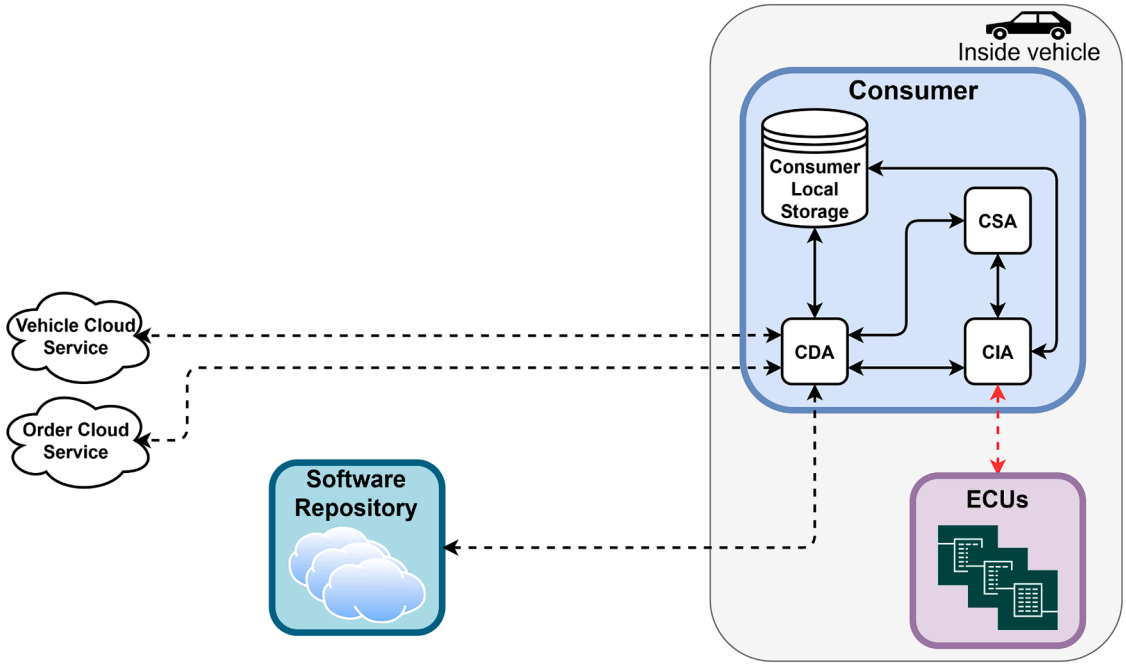

Fig. 7. The communication flow between the Consumer entities is denoted by the solid arrows. Dotted arrows denote communication between a Consumer and a non-Consumer entity. The red arrow denotes an insecure communication channel. All other channels are secure. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

- **Consumer Installation Agent (CIA)** – Executes the installation instructions and then streams the decrypted software to the unlocked ECUs with the help of CSA.

4.2.4. Software suppliers

Software suppliers create and deliver software to the Producer for the installation in different vehicles (Strandberg et al., 2021a, Sec. 3.3). The software suppliers also sign their software to provide authenticity. We assume a simplified model that considers a single software supplier supplying software to a single ECU (see Section 4.2.5).

4.2.5. The electronic control unit

An Electronic Control Unit (ECU) is a vehicle computer responsible for various tasks, from simple signal processing to more advanced functionality, for instance, an infotainment system running various applications. For our simplified model, we assume a system with only one ECU. The ECU is first unlocked and put in programming mode by using security access and a secret key (Strandberg et al., 2021a, Sec. 4.2), to allow the ECU to receive and install software with Unified Diagnostic Services (UDS) (Strandberg, 2024; ISO14229, 2020).

4.2.6. Adversary

As shown in Fig. 8, the adversary is based on the Dolev-Yao model, assuming an adversary with access to the communication channels.

The adversary is actively present on all communication links in the system. However, most of these links, except the one depicted in red, are secure and reliable communication channels; that is, the adversary cannot interfere, according to Dolev-Yao. The one exception is the link

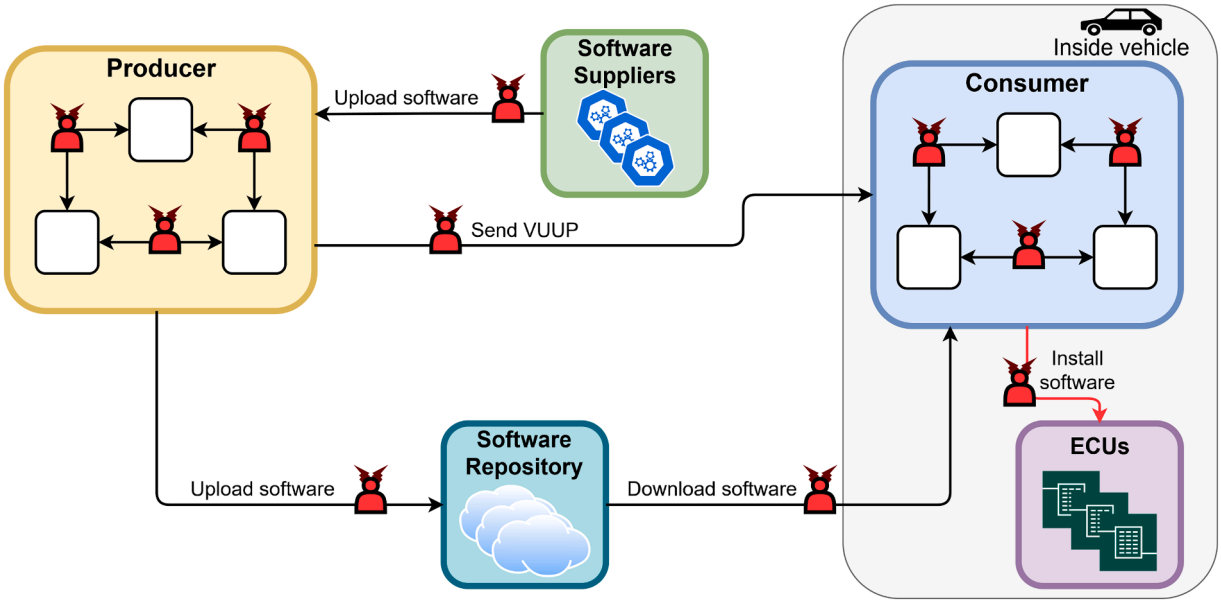

Fig. 8. The illustration depicts the presence of adversaries on the high-level UniSUF architecture shown in Fig. 1, where the adversaries are represented by the red devils. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

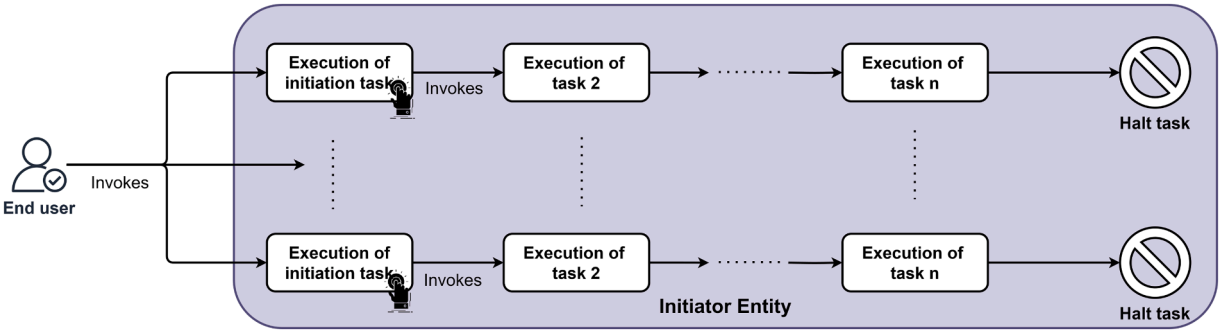

Fig. 9. The initial user-invoked task begins an execution of the initiator entity's tasks. The initiation tasks are marked with the symbol of a hand pressing a button.

between the Consumer and the ECUs, which is a reliable but not a secure communication channel, where the adversary can potentially read, modify, delete, or insert messages.

4.3. Modelling uniSUF

This section explains the modeling of entities and their respective executions in UniSUF.

4.3.1. Execution of system entities

Each system entity runs a sequence of tasks for a given problem. The entity running the initial user-invoked task is called the *initiator*; illustrated in Fig. 9. All other entities are listeners (see Fig. 10) since their first task is the listening task. This task listens for an initiation message specified for each listener. The listening task invokes the next task in sequence and a new listening task, enabling concurrent executions of the task sequence. We divide listeners into two categories: a passive listener that will wait for an initiation message and an active listener that will request an initiation message.

4.3.2. Lifecycle of update rounds

An update round execution begins when the round identifier is created during the initiation task. Throughout this round, listener entities start executing once their listening task receives a message containing the round identifier.

As shown in Figs. 11 and 12, an entity maintains a context for each update round. This context contains the update round identifier and the cryptographic materials (see Section 4.1) used throughout the execution of the round. The context is passed along the task sequence and can be updated by each task.

Each update round identifier encodes its expiration time and when such expiration occurs, all entities *halt* their local execution of the update round, by removing the context associated with the round and ignoring any further messages related to that round. The update round is terminated once all entities have halted their execution of the update round.

4.3.3. Passing contexts across segments of task sequences

We divide the main problems into sub-problems, where each sub-problem uses a subset of the system entities. In Fig. 13, *sub-problem 1* uses the *initiator* and *listener entity 1*. Fig. 13 shows that an entity's task sequence can be segmented so that each segment belongs to a single sub-problem. For example, *listener entity 1*'s *task 3* and *halt task* form a segment that belongs to *sub-problem 2*.

When an entity starts working on a sub-problem for an update round, it possibly already has an update round context. For instance, if the entity has previously participated in other sub-problems for the same update round, i.e., it has previously run either the initiation task or listening task for the same update round.

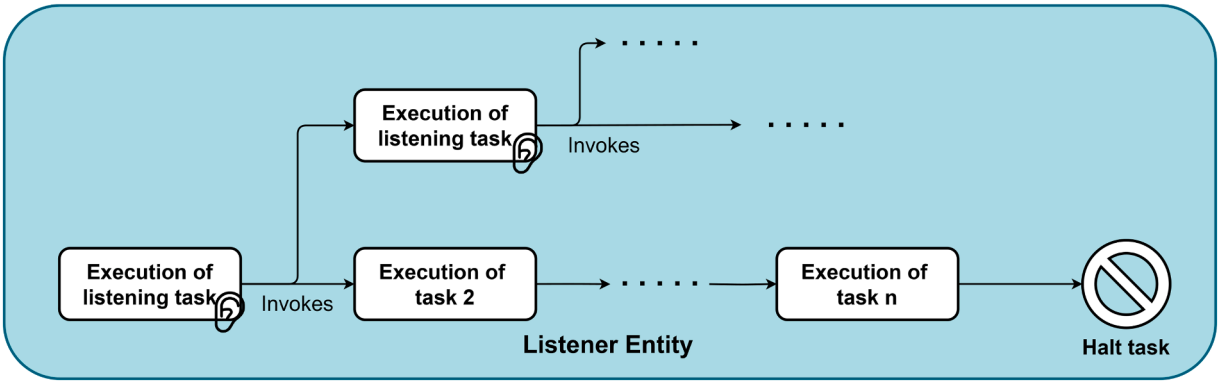

Fig. 10. The listening task invokes a copy of itself to continue listening on a new execution. It also begins an execution of the listening entity's tasks. The listening tasks are marked with a symbol of an ear.

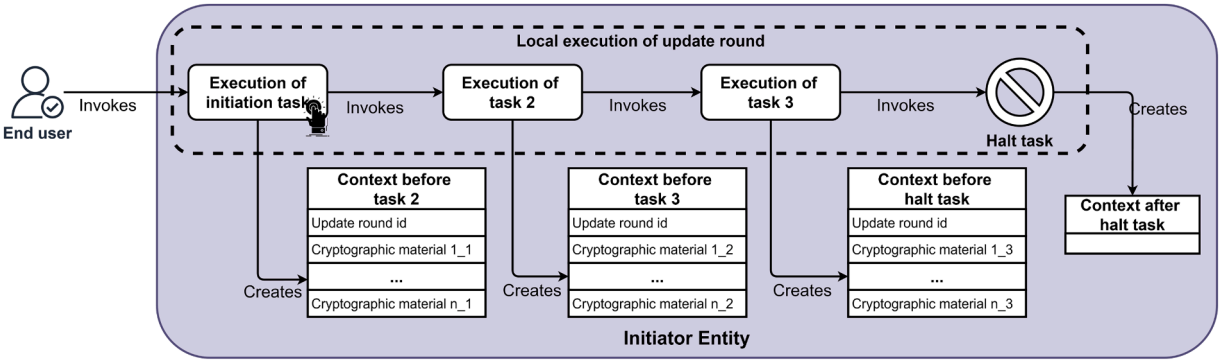

Fig. 11. The figure illustrates an example of executing an initiator entity's task sequence. For each task, we also show what the context can contain.

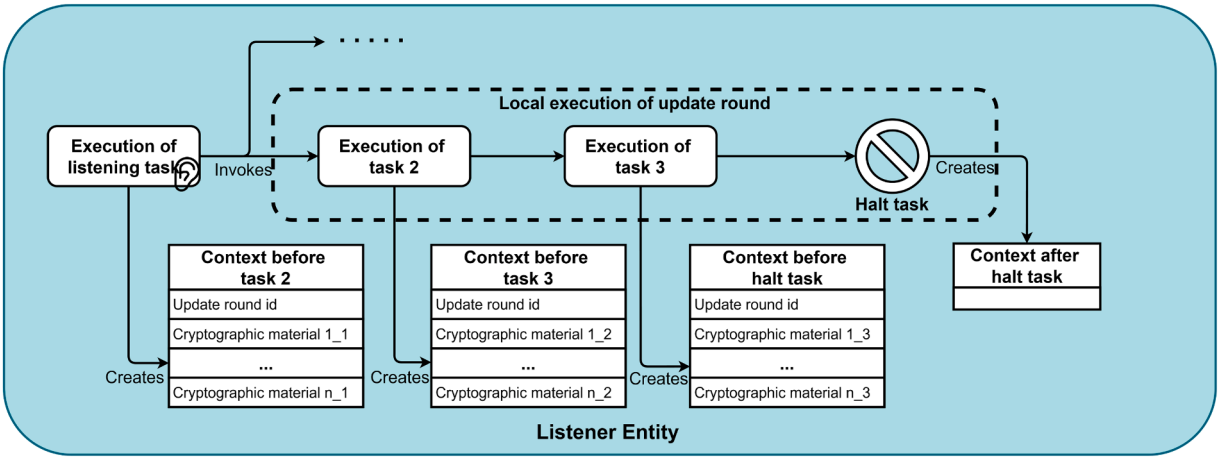

Fig. 12. The figure illustrates an example of an execution of a listening entity's task sequence. For each task, we also show what the context can contain.

Fig. 14 shows that *listening entity 2* has a context at the start of the sub-problem since it starts on *task 2*, i.e., it has previously run the *listening task*. The other listener, *listener entity 1*, does not start with any context, since it first needs to run its listening task at the start of the sub-problem.

Once a segment's execution finishes, the last context of this segment is passed to the next segment as the starting context. Therefore, we identify which entities have previously executed task segments in the same update round for a given sub-problem. For these entities, we specify the cryptographic materials present in their starting contexts (see Section 5).

4.3.4. Well-known addresses of UniSUF entities

We assume that all consumer entities, as defined in Section 4.2.3, are aware of the specific Vehicle Identification Number (VIN) of the vehicle they occupy. Each vehicle has a pre-stored public vehicle certificate containing metadata, including VIN-related information (Strandberg, 2024).

Additionally, we assume that all entities in UniSUF know each other's addresses, e.g., through existing protocols, such as DNSSEC, enabling entities to securely obtain IP addresses from domain names (Van Adrichem et al., 2014).

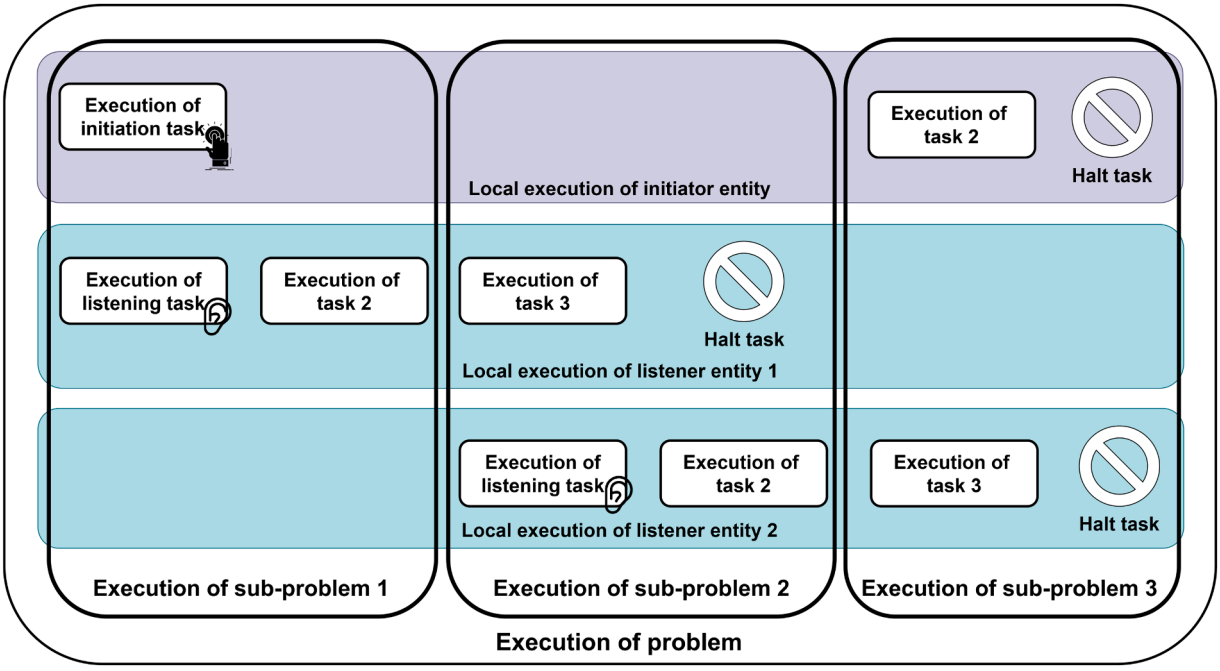

Fig. 13. The execution of an example problem. This problem considers three entities, whereas one is the initiator entity. The problem is divided into three sub-problems.

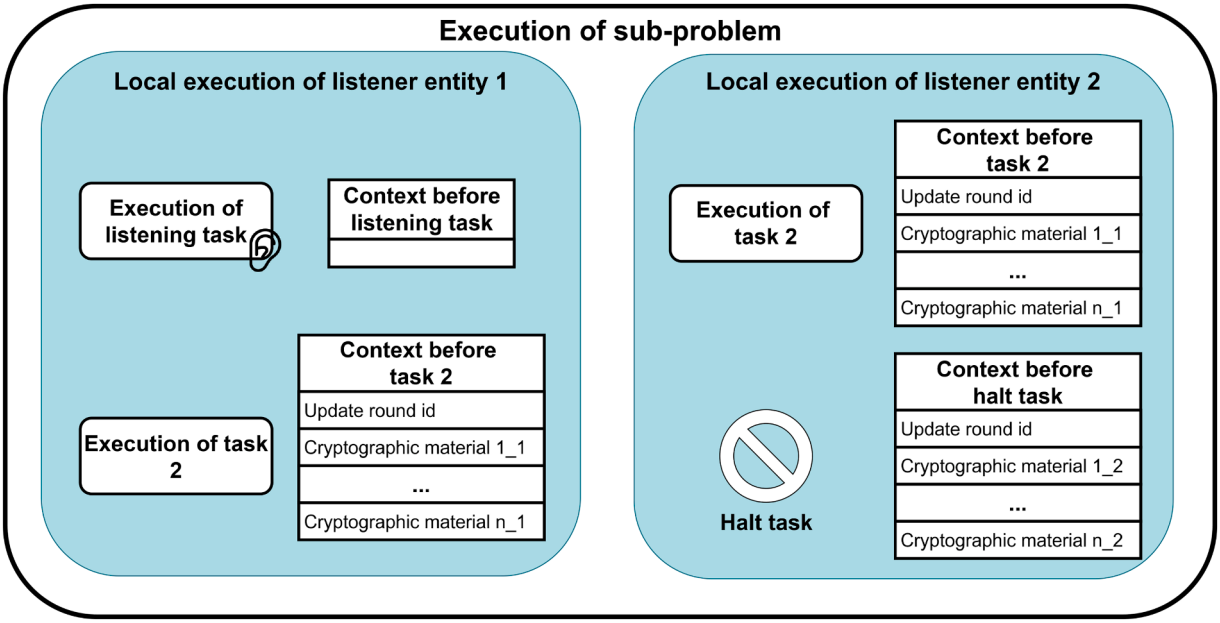

Fig. 14. Example of two entities solving a sub-problem together. The figure also shows how each task in the entities is connected to a context.

5. Sub-problems

From the UniSUF specifications (Strandberg et al., 2021a), we analyse two problems: the *software preparation* and the *software update*. The software preparation process (cf. Section 5.1) involves safeguarding software received from suppliers, ensuring the software's confidentiality and authenticity for it to be securely incorporated into future vehicle updates. Additionally, the software update is further divided into the *encapsulation* and the *decapsulation* stages (cf. Section 5.2 and 5.3) emphasizing securely updating vehicular software and configurations. Strandberg et al. (2021a) specifies an additional stage after decapsulation: the post-state. The post-state encompasses installation reports and logs, potentially affecting upcoming software updates. However, we do not consider the post-state in our simplified model.

Consequently, the main tasks in UniSUF are dissected into sub-problems, based on the steps provided by Strandberg et al. (2021a), where each sub-problem, has a description, a diagram depicting its algorithm, a communication scheme, and assumptions and requirements.

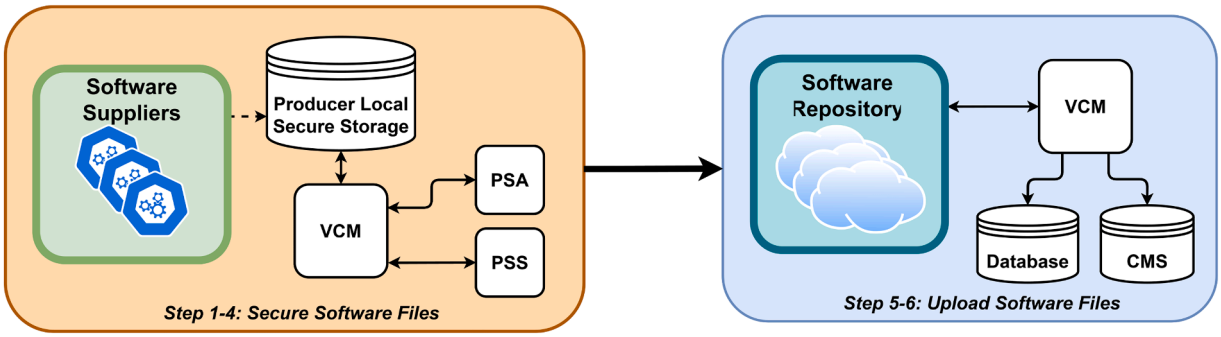

Fig. 15. An overview derived from Strandberg et al. (2021a, Sec. 3.3) for the communication between all entities involved in the preparation stage further divided into two sub-problems.

Table 3
Communication scheme for the sub-problem *Secure Software Files*. The dashed line represents parallel processes.

(1.2)	$Supplier \rightarrow ProducerLocalStorage$	$[Software]_{Supplier}$
(2.1)	$VCM \rightarrow ProducerLocalStorage$	$Request(Software)$
(2.2)	$ProducerLocalStorage \rightarrow VCM$	$[Software]_{Supplier}$
(3.1)	$VCM \rightarrow PSA$	$Request(Software_{Key})$
(3.2)	$PSA \rightarrow VCM$	$Software_{Key}$
(4.2)	$VCM \rightarrow PSS$	$SoftwareHash$ $SoftwareHash = Hash($ $SymEnc([Software]_{Supplier}, Software_{Key}))$
(4.4)	$PSS \rightarrow VCM$	$SignedSoftwareHash$ $SignedSoftwareHash = Sign(SoftwareHash, VCM_{PrivateKey})$

Table 4
Communication scheme for the sub-problem *Upload Software Files*.

(5.1)	$VCM \rightarrow SoftwareRepository$	$Software_{Encased}$
(5.2)	$SoftwareRepository \rightarrow VCM$	$Software_{URL}$
(5.3)	$VCM \rightarrow VINDatabase$	$Software_{URL}$
(5.4)	$VINDatabase \rightarrow VCM$	$Success(Software_{URL})$
(6.1)	$VCM \rightarrow CMS$	$Software_{Key}$
(6.2)	$CMS \rightarrow VCM$	$Success(Software_{Key})$

5.1. Preparation

An overview of the entities involved in the preparation stage and their communication links can be seen in Fig. 15.

In the preparation stage, software supplier files are processed before being used for software updates (Strandberg et al., 2021a). We have divided this stage into two sub-problems: the *Secure Software Files* and the *Upload Software Files*. The focus of the first sub-problem is the encrypting and signing of the software, whereas the second sub-problem handles the software upload and the creation of software URLs.

The preparation stage manages both software files applicable to multiple vehicles and unique files for specific vehicles (Strandberg, 2024). In our model, we assume the case when software is being prepared for multiple vehicles and we additionally assume that the update round is solely identified by the expiration time t_e (see Section 3.4).

5.1.1. Step 1–4: secure software files

The initial phase of the update process focuses on securing software files (see Fig. 16 and Table 3). The software supplier signs the software files before sending them to local storage on the producer side. VCM validates the signature and encrypts the software using a symmetric key obtained from PSA; this key is referred to as $Software_{Key}$. The encrypted software is then signed with the VCM certificate to finalize the $Software_{Encapsualted}$ assembly.

$$S = \{Software,\ Software_{Key},\ Supplier_{PrivateKey},\ VCM_{PrivateKey}\}$$
$$D = \{(Software,\ Supplier),\ (Software_{Key},\ PSA),\ (SoftwareHash,\ VCM),\ (SignedSoftwareHash,\ PSS)\}$$

To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define S, D and the partial order $\mathcal{P}(\ell) = \ell_i < \ell_{i+1}$ for $1 \leq i \leq 4$ with the following labels:

ℓ_1: The software suppliers upload the software to Producer Local Storage.
ℓ_2: PSA generates the software key.
ℓ_3: VCM generates a hash of the software.
ℓ_4: PSS generates a signature for the software.
ℓ_5: VCM assembles $Software_{Encased}$.

5.1.2. Step 5–6: upload software files

Once the software file has been encased, it is uploaded into a software repository. The repository then generates a URL to the uploaded encased software and returns this URL to VCM (see steps 5.1 and 5.2 in Fig. 17 and Table 4). Finally, the VCM stores the software URL in the VIN database and stores the $Software_{Key}$ in CMS; used for the encryption of software files.

Note that VCM has a starting context because its initiation task is in a previous sub-problem (see Section 5.1.1). VCM's starting context contains $Software_{Encased}$ and $Software_{Key}$.

$$S = \{Software,\ Software_{Key},\ Supplier_{PrivateKey},\ VCM_{PrivateKey}\}$$
$$D = \{(Software_{Encased},\ VCM),\ (Software_{Key},\ PSA),\ (Software_{URL},\ SoftwareRepository)\}$$

To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define S, D and the partial order $\mathcal{P}(\ell) = \ell_i < \ell_{i+1}$ for $1 \leq i \leq 3$ with the following labels:

ℓ_1: Software Repository receives the $Software_{Encased}$.
ℓ_2: VIN Database stores the $Software_{URL}$.
ℓ_3: CMS stores the $Software_{key}$.
ℓ_4: VCM receives status of $Software_{key}$ being stored in CMS.

5.2. Encapsulation

The encapsulation stage starts when the Order Agent has received an order request from CDA, whereafter producer entities work collaboratively to produce a VUUP (Strandberg et al., 2021a, Sec. 4.1). Fig. 18 shows a high-level flow diagram of the different sub-problems involved in the encapsulation stage. In steps 1–2, the encapsulation stage starts by producing a VSO, which is later processed to a Software List in step 3.

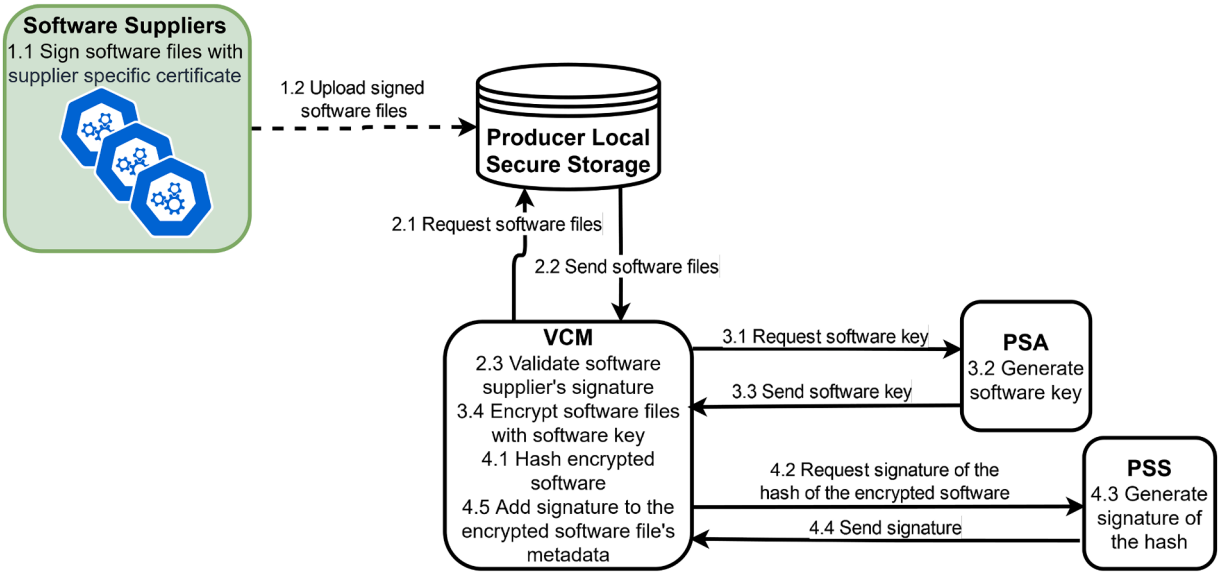

Fig. 16. Diagram of the sub-problem *Secure Software Files*.

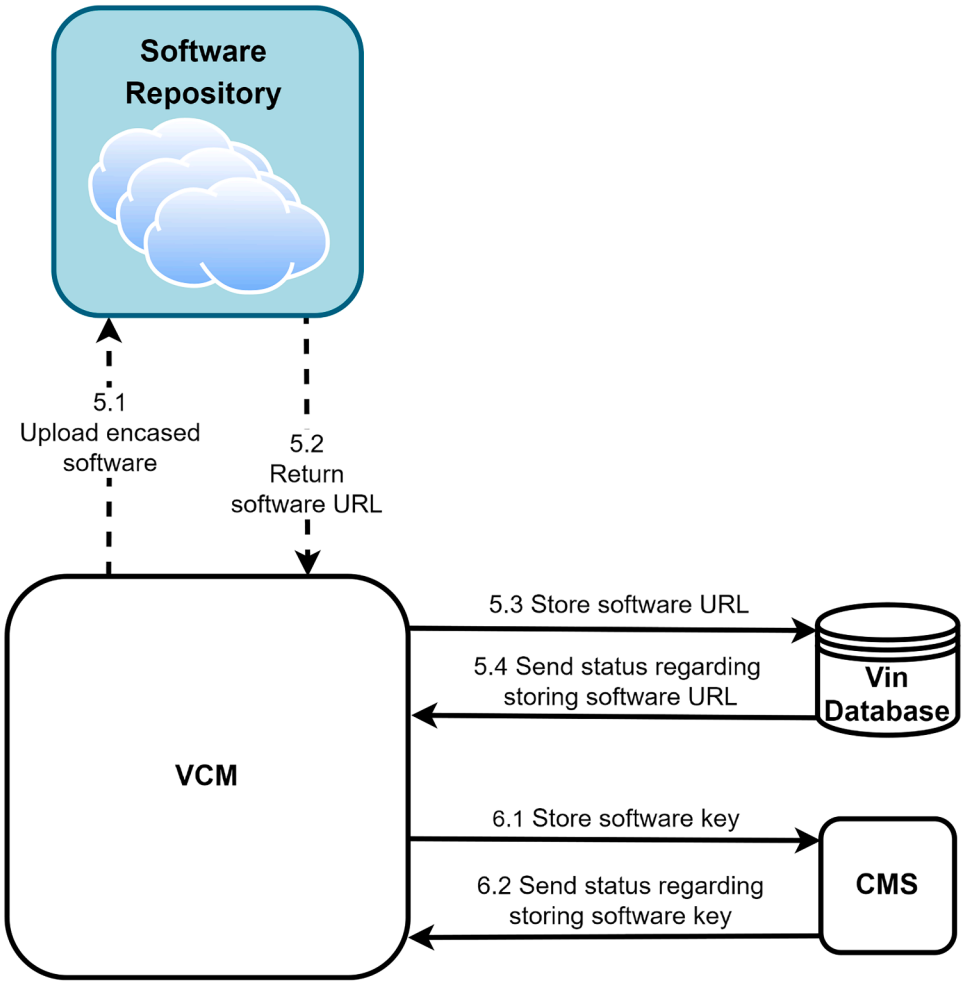

Fig. 17. Diagram of the sub-problem *Upload Software Files*.

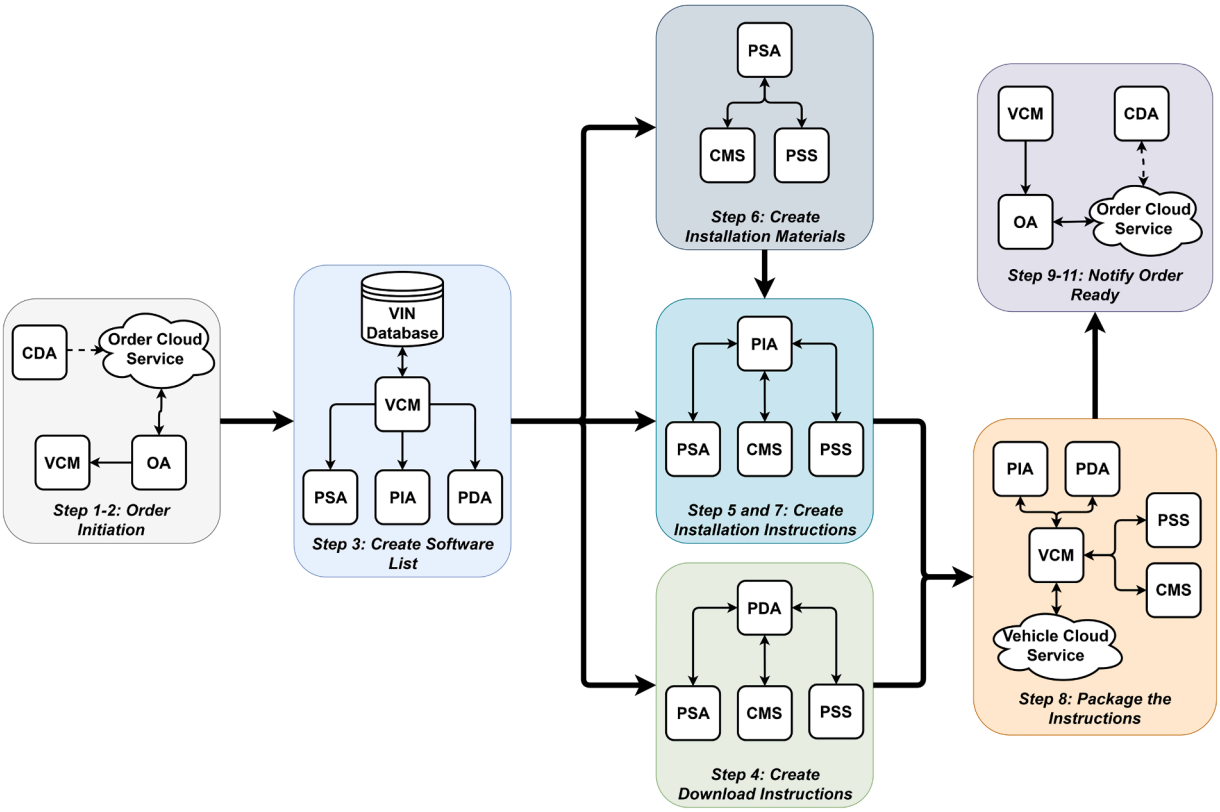

Fig. 18. Overview of communications between all entities that are involved in the encapsulation stage.

Table 5
Communication scheme for the sub-problem *Order Initiation*.

(1.3)	$CDA \rightarrow OrderCloudService$	$[VSO]_{Consumer}$
(2.1)	$OrderAgent \rightarrow OrderCloudService$	$Request(VSO)$
(2.2)	$OrderCloudService \rightarrow OrderAgent$	$[VSO]_{Consumer}$
(2.4)	$OrderAgent \rightarrow VCM$	$[VSO]_{Consumer}$

Table 6
Communication scheme for the sub-problem *Create Software List*.

(3.2)	$VCM \rightarrow VINDatabase$	VIN
		$(VIN, \ldots) = VSO$
(3.3)	$VINDatabase \rightarrow VCM$	$VIN_{Data} \parallel Software_{Versions}$
(3.6)	$VCM \rightarrow PDA$	$[SoftwareList]_{VCM}$
	$VCM \rightarrow PSA$	$[SoftwareList]_{VCM}$
	$VCM \rightarrow PIA$	$[SoftwareList]_{VCM}$
		$SoftwareList = Create_{SoftwareList}($ $VIN_{Data} \parallel Software_{Versions})$

Furthermore, in steps 4–7, the Software List is sent to PDA, PIA, and PSA to generate necessary materials, for instance, download and installation instructions. Instructions and other materials are included in a VUUP file (step 8), whereafter the encapsulation stage finalizes by notifying the consumer that updates are available (cf. steps 9–11).

5.2.1. Step 1–2: order initiation

The encapsulation process begins when the CDA requests an update by sending a signed order (denoted as VSO) to the Order Cloud Service (see steps 1.1–1.3 in Fig. 19 and Table 5). In step 1.4, Order Cloud Service stores the order in a queue, whereafter the Order Agent attempts to fetch an order from this queue. If an order is available, Order Cloud Service sends a VSO to Order Agent, which verifies the signature and initiates VCM using the VSO (steps 2.1–2.4).

To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define $S = \{CDA_{PrivateKey}\}$, $\mathcal{D} = \{(VSO,\ CDA)\}$, and the partial order $\mathcal{P}(\ell) = \ell_i < \ell_{i+1}$ for $1 \leq i \leq 4$ with the following labels:

ℓ_1: CDA generates a signed VSO.
ℓ_2: Order Cloud Service stores the signed VSO in its queue.
ℓ_3: Order Agent pulls signed VSO from Order Cloud Service.
ℓ_4: Order Agent initiates VCM with the signed VCM.
ℓ_5: VCM sends status on initialisation.

5.2.2. Step 3: create software list

Given a VSO, the VCM will produce a software list (see step 3.1–3.5 in Fig. 20 and Table 6). The software list contains the software to be installed in the vehicle. After creating the software list, it is transmitted to the PIA, PSA, and PDA for further processing (step 3.6). The software list is used to generate the download and installation instructions.

The entity VCM has a starting context because its listening task has been included in the previous sub-problem *Order Initiation* (see Section 5.2.1). The starting context of VCM contains the signed VSO.

$$S = \{CDA_{PrivateKey},\ VCM_{PrivateKey}\}$$
$$\mathcal{D} = \{(VSO,\ CDA),\ (SoftwareList,\ VCM),\ (VIN_{Data},\ VINDatabase),\ (Software_{Versions},\ VINDatabase)\}$$

To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define S and $\mathcal{D}$ as seen above, and the partial order $P(\ell) = \{\ell_1 < \ell_2,\ \ell_2 < \ell_3,\ \ell_2 < \ell_4,\ \ell_2 < \ell_5,\ \ell_3 < \ell_6,\ \ell_4 < \ell_6,\ \ell_5 < \ell_6\}$. In other words, ℓ_1 happens before ℓ_2 happens, which then precedes ℓ_3, ℓ_4, and ℓ_5 occurring in parallel, and finally ℓ_6 happens last.

ℓ_1: VCM obtains the most up-to-date software versions and vehicle data from VIN Database.
ℓ_2: VCM creates a signed Software List.

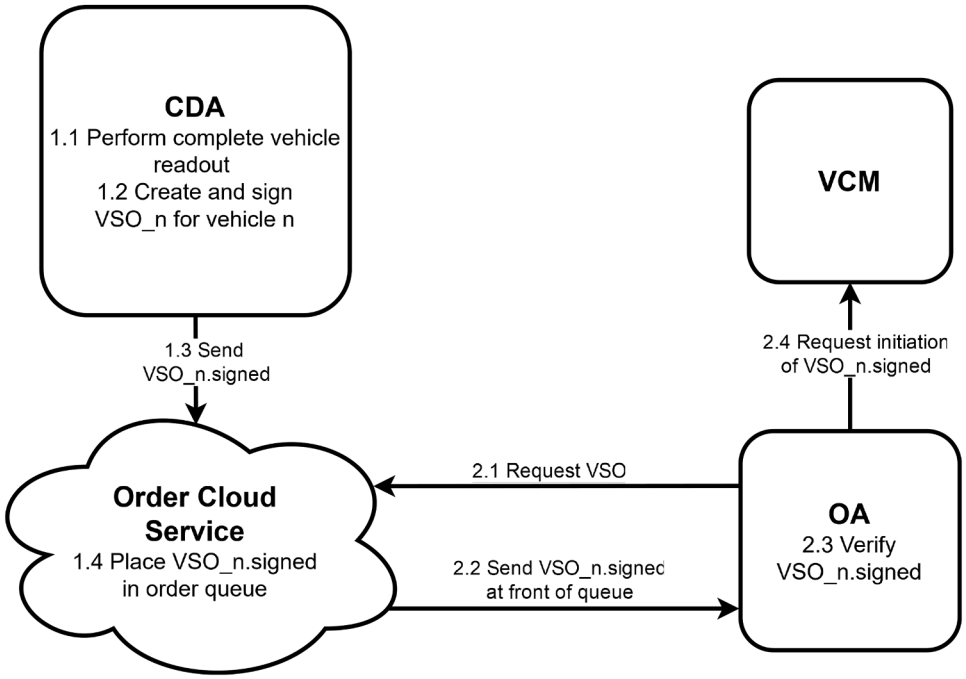

Fig. 19. Diagram of the sub-problem *Order Initiation*.

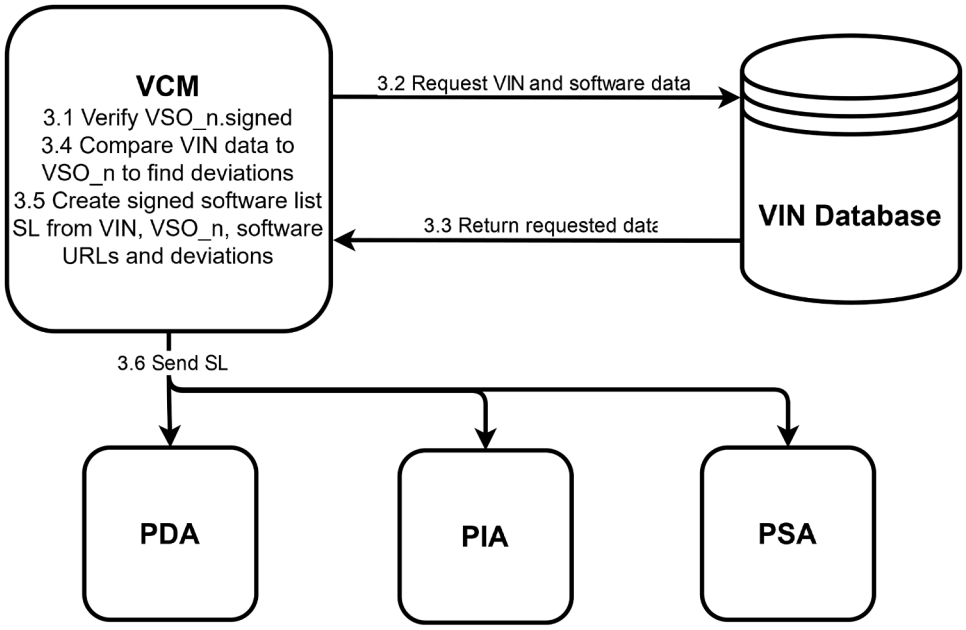

Fig. 20. Diagram of the sub-problem *Create Software List*.

ℓ_3: VCM sends the signed Software List to PDA.
ℓ_4: VCM sends the signed Software List to PIA.
ℓ_5: VCM sends the signed Software List to PSA.
ℓ_6: VCM receives status of Software List being sent to PDA, PIA and PSA.

5.2.3. Step 4: create download instructions

Based on the Software List received from the VCM in the previous sub-problem *Create Software List* (see Section 5.2.2), PDA creates the Download Instructions (see steps 4.1–4.2 in Fig. 21 and Table 7). The Download Instructions is encrypted using a generated session key. The session key is then used to produce the key manifest DKM (steps 4.3–4.10). This sub-problem finishes by signing the Download Instructions and DKM (steps 4.12–4.16).

The entities PDA and PSA have starting contexts because their listening tasks have been included in a previous sub-problem (see sub-problem *Create Software List*). The starting context of PDA contains the signed Software List and VCM_{Cert}.

$$S = \{SoftwareList,\ DownloadInstructions,\ DKM_{Key},\ Root_{PrivateKey},\ PDA_{PrivateKey}, VCM_{PrivateKey}\}$$

$$D = \{(SoftwareList,\ VCM),\ (DownloadInstructions,\ PDA),\ (DKM_{Key},\ PSA),\ (Vehicle_{Cert},\ Root),\ (PDA_{Cert},\ Root),\ (DKM,\ PDA)\}$$

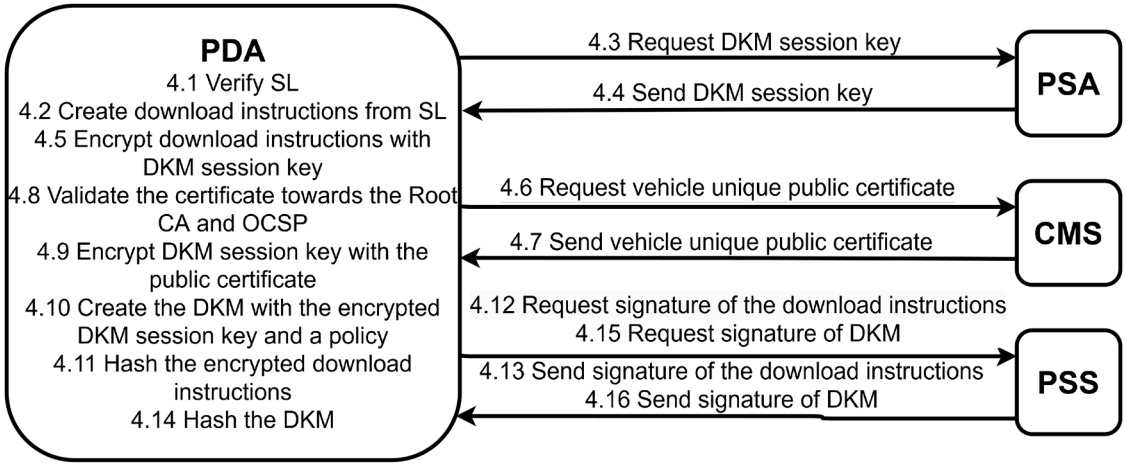

Fig. 21. Diagram of the sub-problem *Create Download Instructions*.

Table 7
Communication scheme for the sub-problem *Create Download Instructions*.

(4.3)	$PDA \rightarrow PSA$	$Request(DKM_{Key})$
(4.4)	$PSA \rightarrow PDA$	DKM_{Key}
(4.6)	$PDA \rightarrow CMS$	$Request(Vehicle_{Cert})$
(4.7)	$CMS \rightarrow PDA$	$Vehicle_{Cert}$
(4.12)	$PDA \rightarrow PSS$	$Hash($ $SymEnc($ $DownloadInstructions,$ $DKM_{Key}))$
(4.13)	$PSS \rightarrow PDA$	$Sign($ $Hash($ $SymEnc($ $DownloadInstructions,$ $DKM_{Key})),$ $PDA_{PrivateKey})$
(4.15)	$PDA \rightarrow PSS$	$Hash(DKM)$
(4.16)	$PSS \rightarrow PDA$	$Sign(Hash(DKM),\ PDA_{PrivateKey})$

To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define the sets S and D as seen above, and the partial order $\mathcal{P}(\ell) = \ell_i < \ell_{i+1}$ for $1 \leq i \leq 10$ with the following labels:

ℓ_1: PDA verifies the Software List.
ℓ_2: PDA creates the Download Instructions.
ℓ_3: PSA generates DKM_{Key}.
ℓ_4: PSA sends DKM_{Key} to PDA.
ℓ_5: CMS sends $Vehicle_{Cert}$ to PDA.
ℓ_6: PDA generates DKM.
ℓ_7: PSS generates a signature for the Download Instructions.
ℓ_8: PSS sends the signed encrypted Download Instructions to PDA.
ℓ_9: PSS generates signature for DKM.
ℓ_{10}: PSS sends the signed encrypted DKM to PDA.

5.2.4. Step 6: generate installation materials

In parallel to the creation of Download Instructions and Installation Instructions (see Section 5.2.3 and 5.2.5), the PSA generates and secures cryptographic materials necessary for the installation of software updates, e.g., SKA and MKM (see Fig. 22 and Table 8). For instance, ECU keys for the unlocking of ECUs, to gain extended privileges.

The entities PSA, PSS, and CMS have starting contexts because their listening tasks have been included in previous sub-problems (see sub-problem *Create Download Instructions*). PSA's starting context contains the signed Software List and VCM_{Cert} and CMS's starting context contains the $Vehicle_{Cert}$.

Table 8
Communication scheme for the sub-problem *Generate Installation Materials*.

(6.2)	$PSA \rightarrow CMS$	$SoftwareList$
(6.3)	$CMS \rightarrow PSA$	$Vehicle_{Cert} \parallel SKA_{Key}$
(6.10)	$PSA \rightarrow PSS$	$Hash(SKA)$
(6.11)	$PSS \rightarrow PSA$	$Sign(Hash(SKA), PSA_{PrivateKey})$
(6.13)	$PSA \rightarrow PSS$	$Hash(MKM)$
(6.14)	$PSS \rightarrow PSA$	$Sign(Hash(MKM), PSA_{PrivateKey})$

$S = \{SoftwareList,\ MKM_{Key},\ SKA_{Key},\ Root_{PrivateKey},\ PSA_{PrivateKey},\ VCM_{PrivateKey}\}$
$D = \{(SoftwareList,\ VCM),\ (Vehicle_{Cert},\ Root),\ (PSA_{Cert},\ Root),\ (MKM,\ PSA),\ (SKA,\ PSA)\}$

To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define the sets S and D as seen above, and the partial order $\mathcal{P}(\ell) = \ell_i < \ell_{i+1}$ for $1 \leq i \leq 9$ with the following labels:

ℓ_1: PSA verifies the Software List.
ℓ_2: CMS sends cryptographic material to PSA.
ℓ_3: PSA generates $MKM_{ECU_{Key}}$ and $MKM_{SW_{Key}}$.
ℓ_4: PSA generates MKM.
ℓ_5: PSA generates SKA.
ℓ_6: PSS generates a signature for the SKA.
ℓ_7: PSS sends the signed SKA to PSA.
ℓ_8: PSS generates signature for MKM.
ℓ_9: PSS sends the signed MKM to PSA.

5.2.5. Step 5 and 7: create installation instructions

The PIA receives Software List from VCM and creates Installation Instructions based on the Software List (see steps 5.1–5.2 in Fig. 23 and Table 9). Similarly to sub-problem *Create Download Installation*, the Installation Instructions are encrypted with a generated session key, which gives rise to the IKM (see steps 7.1–7.11). The Installation Instructions is also appended with the materials generated in the sub-problem *Generate Installation Materials* (steps 7.1–7.3).

The entities PIA, PSA, PSS, and CMS have starting contexts because their listening tasks are included in previous sub-problems (see sub-problems *Create Software List* and *Create Download Instructions*). PIA's starting context contains the signed Software List and VCM_{Cert}, PSA's starting context contains the signed SKA and signed MKM, and CMS's

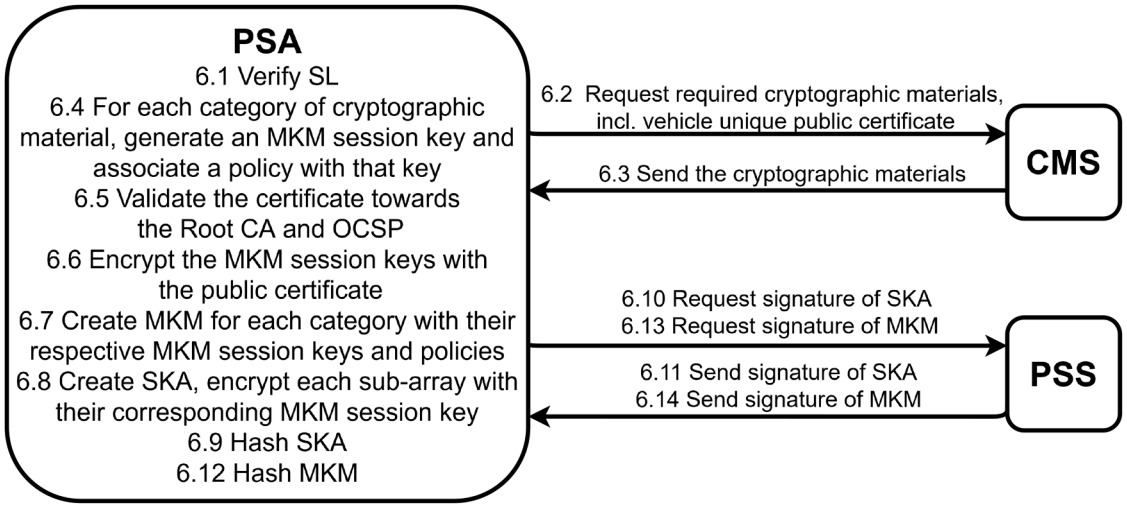

Fig. 22. Diagram of the sub-problem *Generate Installation Materials*.

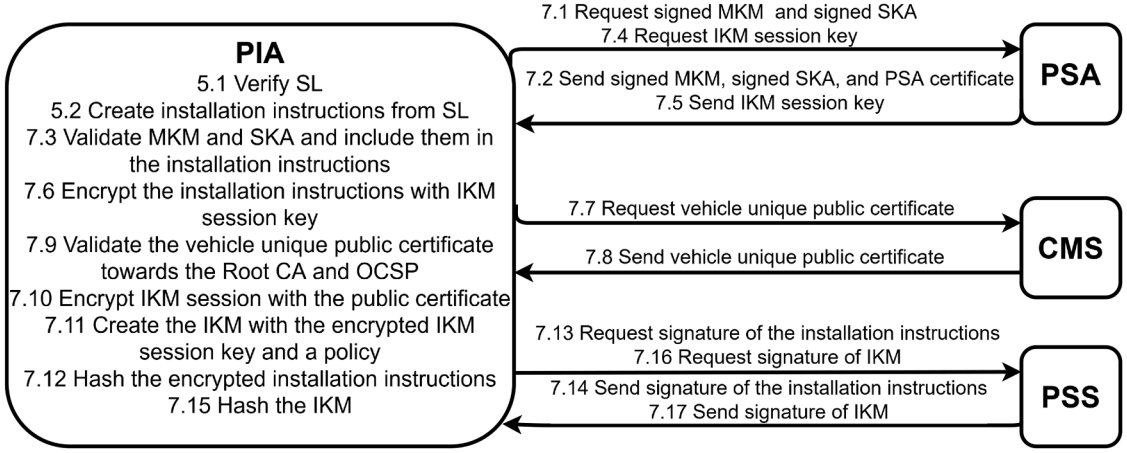

Fig. 23. Diagram of the sub-problem *Create Installation Instructions*.

starting context contains the $Vehicle_{Cert}$.

$$S = \{SoftwareList,\ InstallationInstructions,\ IKM_{Key},\ Root_{PrivateKey},\ PIA_{PrivateKey},\ VCM_{PrivateKey}\}$$

$$\mathcal{D} = \{(SoftwareList,\ VCM),\ (InstallationInstructions,\ PIA),\ (IKM_{Key},\ PSA),\ (Vehicle_{Cert},\ Root),\ (PIA_{Cert},\ Root),\ (MKM,\ PSA),\ (SKA,\ PSA)\}$$

To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define the sets S and $\mathcal{D}$ as seen above, and the partial order $\mathcal{P}(\ell) = \ell_i < \ell_{i+1}$ for $1 \le i \le 11$ with the following labels:

- ℓ_1: PIA verifies the Software List.
- ℓ_2: PIA creates the Installation Instructions.
- ℓ_3: PSA sends $[MKM]_{PSA}$, and $[SKA]_{PSA}$, and PSA_{Cert}.
- ℓ_4: PSA generates IKM_{Key}.
- ℓ_5: PSA sends IKM_{Key} to PIA.
- ℓ_6: CMS sends $Vehicle_{Cert}$ to PIA.
- ℓ_7: PIA generates IKM.
- ℓ_8: PSS generates signature for the Installation Instructions.
- ℓ_9: PSS sends the signed encrypted Installation Instructions to PIA.
- ℓ_{10}: PSS generates signature for IKM.
- ℓ_{11}: PSS sends signed encrypted IKM to PIA.

Table 9
Communication scheme for the sub-problem *Create Installation Instructions*.

(7.1)	$PIA \to PSA$	$Request(MKM) \parallel Request(SKA)$
(7.2)	$PSA \to PIA$	$[MKM]_{PSA} \parallel [SKA]_{PSA} \parallel PSA_{Cert}$
(7.4)	$PIA \to PSA$	$Request(IKM_{Key})$
(7.5)	$PSA \to PIA$	IKM_{Key}
(7.7)	$PIA \to CMS$	$Request(Vehicle_{Cert})$
(7.8)	$CMS \to PIA$	$Vehicle_{Cert}$
(7.13)	$PIA \to PSS$	$Hash($ $SymEnc($ $InstallationInstructions,$ $IKM_{Key}))$
(7.14)	$PSS \to PIA$	$Sign($ $Hash($ $SymEnc($ $InstallationInstructions,$ $IKM_{Key})),$ $PIA_{PrivateKey})$
(7.16)	$PIA \to PSS$	$Hash(IKM)$
(7.17)	$PSS \to PIA$	$Sign(Hash(IKM), PIA_{PrivateKey})$

5.2.6. Step 8: package the instructions

Once the required materials for software update have been created, the materials need to be inserted into the VUUP file. The procedure starts

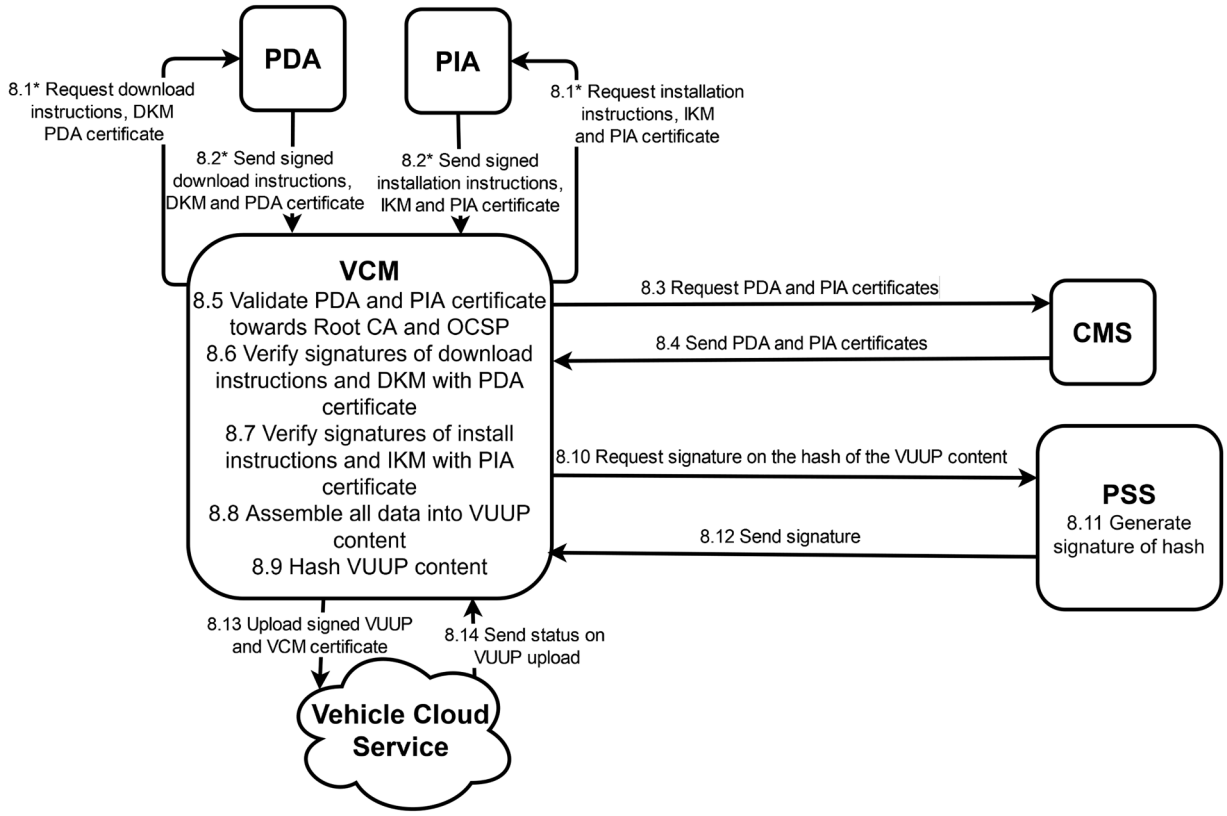

Fig. 24. Diagram of the sub-problem *Package the Instructions*. Steps marked with * can occur in parallel.

with PDA and PIA sending data to VCM (see steps 8.1–8.2 in Fig. 24 and Table 10). Before data is packaged into VUUP content, signatures are validated to ensure authenticity (steps 8.3–8.8). The VUUP file is signed by the PSS and later uploaded to the cloud (steps 8.9–8.14).

The entities PDA, PIA, VCM, CMS, and PSS have a starting context because their listening tasks are included in previous sub-problems *Order Initiation* (see Sections 5.2.2 and 5.2.5). PDA's starting context contains the encrypted and signed Download Instructions and the signed DKM, and PIA's starting context contains the encrypted and signed Installation Instructions and the signed IKM.

$$S = \{Vehicle_{PrivateKey},\ Root_{PrivateKey},\ PDA_{PrivateKey},\ PIA_{PrivateKey},\ PSA_{PrivateKey},\ VCM_{PrivateKey},\ DKM_{Key},\ IKM_{Key}\}$$

$$\mathcal{D} = \{(VUUP_{URL},\ VehicleCloudService),\ (VUUP,\ VCM),\ (DownloadInstructions,\ PDA),\ (InstallationInstructions,\ PIA),\ (PDA_{Cert},\ Root),\ (PIA_{Cert},\ Root),\ (VCM_{Cert},\ Root)\}$$

For any execution of this sub-problem, the entities PDA, PIA, VCM, CMS, and PSS are aware of which update round it belongs to since these entities have been in the same update round in sub-problems *Order Initiation* and *Create Download Instructions*. To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define S and $\mathcal{D}$ as seen above, and the partial order $\mathcal{P}(\ell) = \{\ell_1 < \ell_3,\ \ell_2 < \ell_3,\ \ell_3 < \ell_4,\ \ell_4 < \ell_5,\ \ell_5 < \ell_6,\ \ell_6 < \ell_7\}$. In other words, ℓ_1 and ℓ_2 happen parallel initially, followed sequentially by the remaining handling events, with ℓ_7 happening last.

ℓ_1: VCM receives Download Instructions and DKM from PDA.
ℓ_2: VCM receives Installation Instructions and IKM from PIA.
ℓ_3: VCM retrieves PDA and PIA certificates from CMS.
ℓ_4: VCM assembles the VUUP.
ℓ_5: PSS signs the VUUP.

Table 10
Communication scheme for the sub-problem *Package the Instructions*, where VCS is the Vehicle Cloud Service.

(8.1)	$VCM \rightarrow PDA$	$Request(DownloadInstructions) \parallel Request(DKM) \parallel Request(PDA_{Cert})$
	$VCM \rightarrow PIA$	$Request(InstallationInstructions) \parallel Request(IKM) \parallel Request(PIA_{Cert})$
(8.2)	$PDA \rightarrow VCM$	$[SymEnc(DownloadInstructions, DKM_{Key})]_{PDA} \parallel [DKM]_{PDA} \parallel PDA_{Cert}$
	$PIA \rightarrow VCM$	$[SymEnc(InstallationInstructions, IKM_{Key})]_{PIA} \parallel [IKM]_{PIA} \parallel PIA_{Cert}$
(8.3)	$VCM \rightarrow CMS$	$Request(PDA_{Cert}) \parallel Request(PIA_{Cert})$
(8.4)	$CMS \rightarrow VCM$	$PDA_{Cert} \parallel PIA_{Cert}$
(8.10)	$VCM \rightarrow PSS$	$Hash(VUUP_{Content})$
(8.12)	$PSS \rightarrow VCM$	$Sign(Hash(VUUP_{Content}), VCM_{PrivateKey})$
(8.13)	$VCM \rightarrow VCS$	$VUUP$
(8.14)	$VCS \rightarrow VCM$	$Success(VUUP)$

ℓ_6: VCM uploads the signed VUUP along with the VCM certificate to Vehicle Cloud Service.
ℓ_7: Vehicle Cloud Service receives a status of the signed VUUP and VCM certificate being uploaded to Vehicle Cloud Service.

5.2.7. Step 9–11: notify order ready

When the VUUP has been created, the Order Agent is notified that the order is ready for download (see step 9 in Fig. 25 and Table 11). The notification consists of a signed URL to the VUUP file. The signature of this URL is validated by the Order Agent before it is uploaded to the Order Cloud Service (steps 10.1 and 10.2).

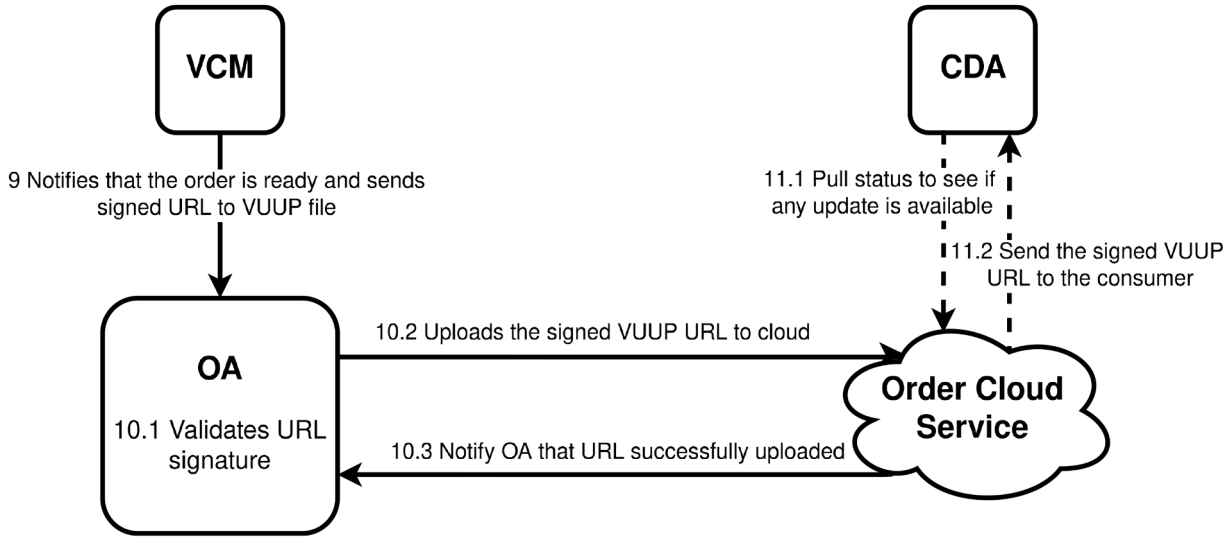

Fig. 25. Diagram of the sub-problem *Notify Order Ready*.

Table 11
Communication scheme for the sub-problem *Notify Order Ready*, where OCS is the Order Cloud Service.

(9)	$VCM \rightarrow OrderAgent$	$[VUUP_{URL}]_{VCM}$
(10.2)	$OrderAgent \rightarrow OCS$	$[VUUP_{URL}]_{VCM}$
(10.3)	$OCS \rightarrow OrderAgent$	$Success(VUUP_{URL})$
(11.1)	$CDA \rightarrow OCS$	$Request(VUUP_{URL})$
(11.2)	$OCS \rightarrow CDA$	$[VUUP_{URL}]_{VCM} \parallel VCM_{Cert}$

Table 12
Communication scheme for the sub-problem *Download Software Files*.

(1.1)	$CDA \rightarrow OrderCloudService$	$Request(VUUP_{URL})$
(1.2)	$OrderCloudService \rightarrow CDA$	$[VUUP_{URL}]_{VCM} \parallel VCM_{Cert}$
(2.1)	$CDA \rightarrow VehicleCloudService$	$VUUP_{URL}$
(2.2)	$VehicleCloudService \rightarrow CDA$	$VUUP$
(2.3)	$CDA \rightarrow ConsumerLocalStorage$	$VUUP$
(2.4)	$ConsumerLocalStorage \rightarrow CDA$	$Success(VUUP)$

In step 11 of this sub-problem, CDA will pull the status from Order Cloud Service to see if any update is available. We assume updates are always available since if no updates are available, there will be an illegitimate termination of the update process. The entities CDA, Order Agent, Order Cloud Service, and VCM have starting contexts because their listening tasks are included in a previous sub-problem (see sub-problem *Order Initiation*). VCM's starting context contains the $VUPP_{URL}$.

To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define $S = \{VCM_{PrivateKey}\}$, $\mathcal{D} = \{(VUUP_{URL}, VehicleCloudService), (VCM_{Cert}, Root)\}$, and the partial order $\mathcal{P}(\ell) = \ell_i < \ell_{i+1}$ for $1 \leq i \leq 4$ with the following labels:

ℓ_1: VCM sends $[VUUP_{URL}]_{VCM}$ to Order Agent.
ℓ_2: Order Agent sends $[VUUP_{URL}]_{VCM}$ to the Order Cloud Service.
ℓ_3: CDA pulls status from the Order Cloud Service.
ℓ_4: The Order Cloud Service sends $[VUUP_{URL}]_{VCM}$ to CDA.

5.3. Decapsulation

We summarise the decapsulation stage defined by Strandberg et al. (2021a, Sec. 4.2) as follows: CDA retrieves a $VUUP_{URL}$ from Order Cloud Service, further used to retrieve the VUUP from Vehicle Cloud Service. Once CDA has validated the VUUP, CDA uses CSA for the decryption and plain text retrieval of the Download Instructions. The Download Instructions is used to download software from the Software Repository. After that, CIA will use CSA to decrypt and for the plain text retrieval of the Installation Instructions, and with the help of CSA, install software to the ECUs.

We have divided the decapsulation stage into five sub-problems. A summary of the division can be seen in Fig. 26. In Strandberg et al. (2021a, Fig. 4), a more detailed version of the entire decapsulation process is presented.

5.3.1. Step 1–4: download VUUP

Once the encapsulation is done (see Section 5.2), CDA retrieves the $VUUP_{URL}$, and uses the URL to obtain a valid VUUP (see step 2.2 in Fig. 27 and Table 12). CDA also guarantees that the content of the VUUP is valid. For this sub-problem, CDA has a starting context because its initiation task is in a previous sub-problem (see Section 5.2.1). Order Cloud Service and the Vehicle Cloud Service also have starting contexts, which respectively contain the $VUUP_{URL}$ and VUUP for the update round (see Section 5.2.6 and 5.2.7). Note that the first steps of this sub-problem, steps 1.1 and 1.2 in Fig. 27 and Table 12, are the same as the last steps for the previous sub-problem (see Section 5.2.7).

$$
\begin{aligned}
S = \{&VUUP_{URL},\ DownloadInstructions,\ InstallationInstructions,\\
&DKM_{Key}, IKM_{Key},\ MKM_{Key},\ SKA_{Key},\ VCM_{PrivateKey},\\
&Vehicle_{PrivateKey},\ PDA_{PrivateKey}, PIA_{PrivateKey},\ PSA_{PrivateKey},\\
&Root_{PrivateKey}\}\\
\mathcal{D} = \{&(VUUP_{URL},\ VehicleCloudService), (VUUP,\ VCM),\\
&(DownloadInstructions,\ PDA), (InstallationInstructions,\ PIA),\\
&(DKM,\ PDA),\ (IKM,\ PIA),\ (VCM_{Cert},\ Root),\ (PDA_{Cert},\ Root),\\
&(PIA_{Cert},\ Root),\ (Root_{Cert},\ Root)\}
\end{aligned}
$$

To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define S, $\mathcal{D}$ and the partial order $\mathcal{P}(\ell) = \ell_i < \ell_{i+1}$ for $1 \leq i \leq 4$ with the following labels:

ℓ_1: Order Cloud Service has sent the signed $VUUP_{URL}$.
ℓ_2: CDA has validated the $VUUP_{URL}$.
ℓ_3: Vehicle Cloud Service has sent the signed VUUP.
ℓ_4: Consumer Local Storage has stored the signed VUUP.
ℓ_5: CDA has validated Download Instructions, DKM, Installation Instructions, and IKM.

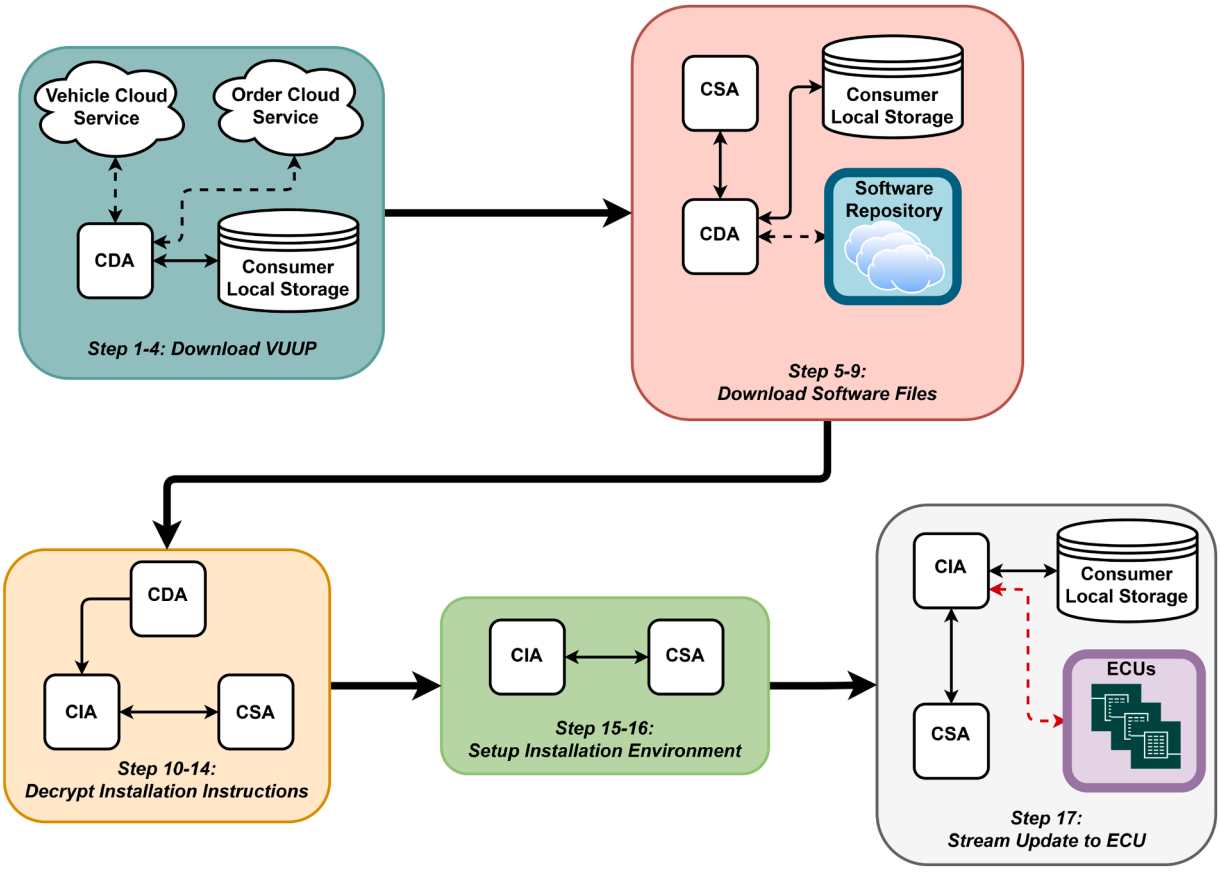

Fig. 26. Using the specifications of the decapsulation stage provided by Strandberg et al. (2021a, Sec. 4.2), we present the following division of sub-problems.

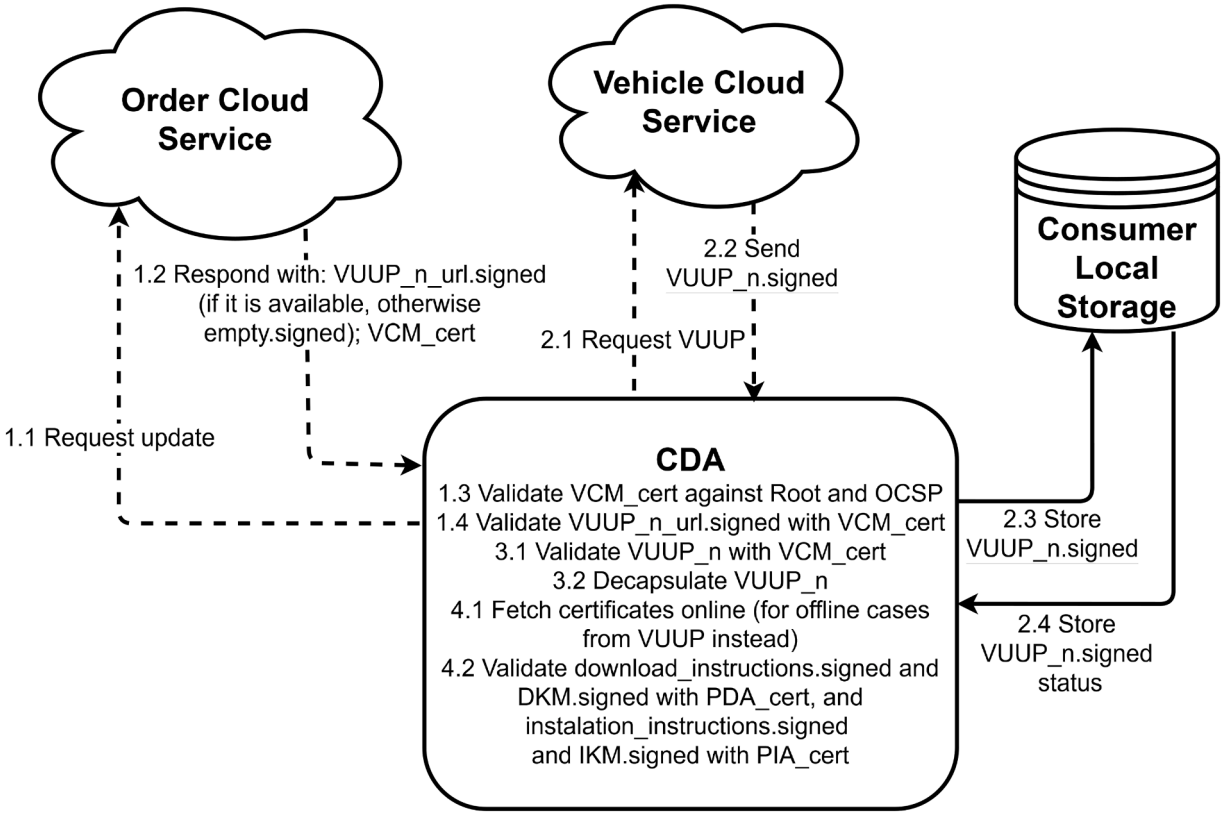

Fig. 27. Diagram of the sub-problem *Download VUUP*.

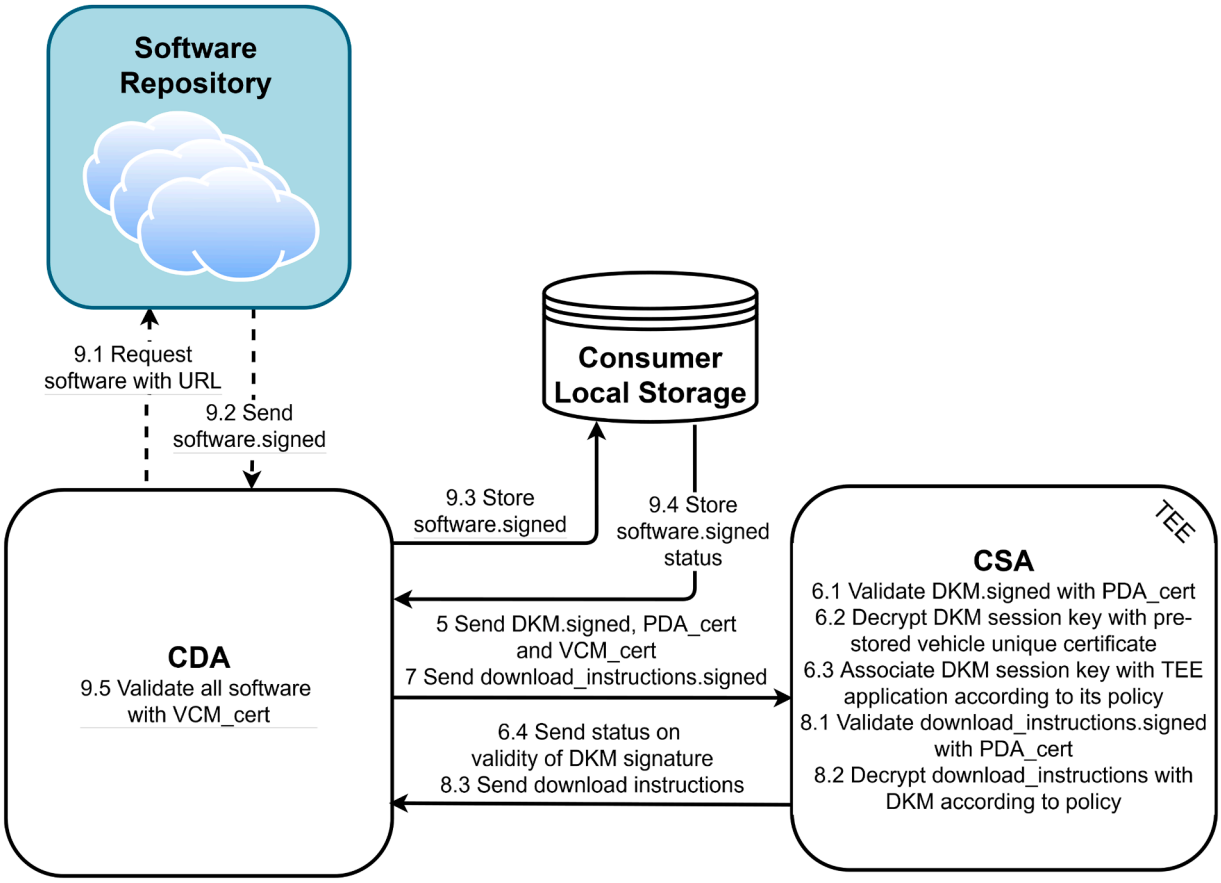

Fig. 28. Diagram of the sub-problem *Download Software Files*.

5.3.2. Step 5–9: download software files

CDA requests CSA to associate master keys in DKM to specific trusted applications within the Trusted Execution Environment (TEE). This is followed by the decryption of the Download Instructions on behalf of CDA. Then CDA uses the download instructions to retrieve the software from Software Repository (see step 9.1 in Fig. 28 and Table 13). For this sub-problem, CDA and Consumer Local Storage have starting contexts because their respective initiation and listening tasks are included in previous sub-problems (see Sections 5.2.1 and 5.3.1). CDA's starting context contains the signed DKM, PDA_{Cert}, VCM_{Cert}, and the signed and encrypted Download Instructions (see Section 5.3.1).

$$S = \{Software,\ DownloadInstructions,\ DKM_{Key},\ SKA_{Software_{Key}},\ VCM_{PrivateKey},\ Vehicle_{PrivateKey},\ PDA_{PrivateKey},\ Supplier_{PrivateKey},\ Root_{PrivateKey}\}$$

$$\mathcal{D} = \{(Software,\ Supplier),\ (DownloadInstructions,\ PDA),\ (DKM,\ PDA),\ (VCM_{Cert},\ Root),\ (Vehicle_{Cert},\ Root),\ (PDA_{Cert},\ Root),\ (Root_{Cert},\ Root)\}$$

To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define S, $\mathcal{D}$, and the partial order $\mathcal{P}(\ell) = \ell_i < \ell_{i+1}$ for $1 \le i \le 5$ with the following labels:

- ℓ_1: CSA has associated the DKM.
- ℓ_2: CSA decrypts the Download Instructions.
- ℓ_3: CDA has received the decrypted Download Instructions.
- ℓ_4: Software Repository has sent $Software_{Encased}$.
- ℓ_5: Consumer Local Storage has stored $Software_{Encased}$.
- ℓ_6: CDA has validated $Software_{Encased}$.

Table 13
Communication scheme for the sub-problem *Download Software Files*. Note that the VCM_{Cert} is sent to CSA in step 5, so that CSA has access to the certificate in step 17 (see Section 5.3.5) when it needs to validate software signatures (Strandberg, 2024).

(5)	$CDA \rightarrow CSA$	$[DKM]_{PDA} \parallel PDA_{Cert} \parallel VCM_{Cert}$
(6.4)	$CSA \rightarrow CDA$	$Success(DKM)$
(7)	$CDA \rightarrow CSA$	$[SymEnc(DownloadInstructions, DKM_{Key})]_{PDA}$
(8.3)	$CSA \rightarrow CDA$	$DownloadInstructions$
(9.1)	$CDA \rightarrow SoftwareRepository$	$Software_{URL}$
(9.2)	$SoftwareRepository \rightarrow CDA$	$Software_{Encased}$
(9.3)	$CDA \rightarrow ConsumerLocalStorage$	$Software_{Encased}$
(9.4)	$ConsumerLocalStorage \rightarrow CDA$	$Success(Software)$

5.3.3. Step 10–14: decrypt installation instructions

On behalf of CIA, CSA initiates the IKM, followed by the decryption of Installation Instructions (see steps 12.5 and 14.3 in Fig. 29 and Table 14). For this sub-problem, CDA and CSA have starting contexts because their respective initiation and listening tasks are run in previous sub-problems (see Section 5.3.1 and 5.3.2). CDA's starting context contains PIA_{Cert}, the signed IKM, and the signed and encrypted Installation Instructions (see Section 5.3.1).

$$S = \{InstallationInstructions,\ IKM_{Key},\ MKM_{Key},\ SKA_{Key},\ PIA_{PrivateKey},\ PSA_{PrivateKey},\ Root_{PrivateKey}\}$$

$$\mathcal{D} = \{(InstallationInstructions,\ PIA),\ (IKM,\ PIA),\ (PIA_{Cert},\ Root),\ (Root_{Cert},\ Root)\}$$

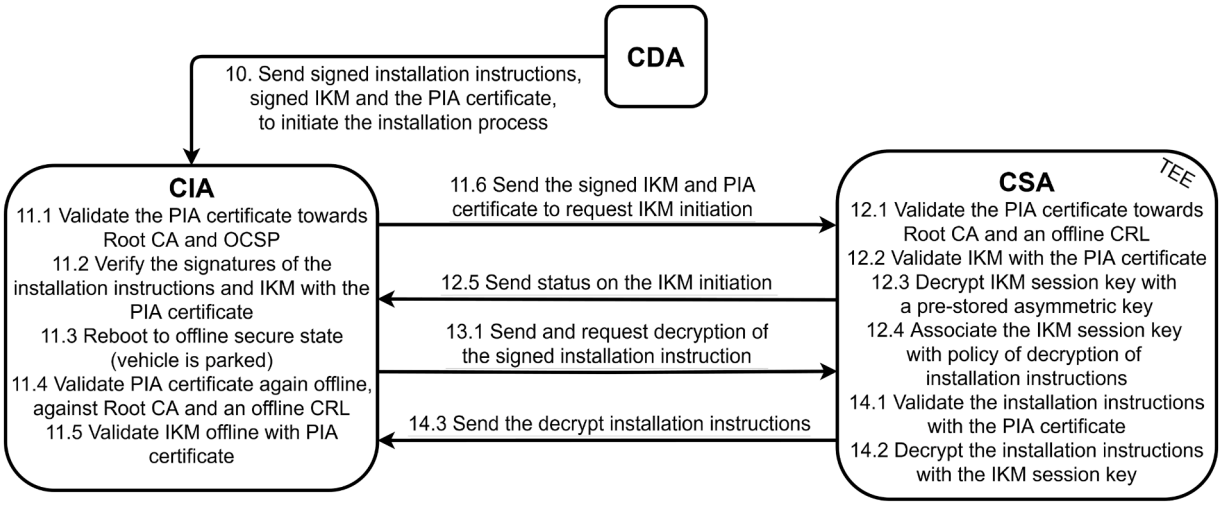

Fig. 29. Diagram of the sub-problem *Decrypt Installation Instructions*.

Table 14
Communication scheme for the sub-problem *Decrypt Installation Instructions*.

(10)	$CDA \rightarrow CIA$	$[SymEnc(InstallationInstructions, IKM_{Key})]_{PIA} \parallel [IKM]_{PIA} \parallel PIA_{Cert}$
(11.6)	$CIA \rightarrow CSA$	$[IKM]_{PIA} \parallel PIA_{Cert}$
(12.5)	$CSA \rightarrow CIA$	$Success(IKM)$
(13.1)	$CIA \rightarrow CSA$	$[SymEnc(InstallationInstructions, IKM_{Key})]_{PIA}$
(14.3)	$CSA \rightarrow CIA$	$InstallationInstructions$

Table 15
Communication scheme for the sub-problem *Setup Installation Environment*.

(15.4)	$CIA \rightarrow CSA$	$[MKM]_{PSA} \parallel PSA_{Cert}$
(16.4)	$CSA \rightarrow CIA$	$Success(MKM)$

To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define S, D and the partial order $\mathcal{P}(\ell) = \ell_i < \ell_{i+1}$ for $1 \leq i \leq 3$ with the following labels:

ℓ_1: CIA initiates offline mode.
ℓ_2: CSA associates the IKM session key with its policy.
ℓ_3: CSA decrypts the Installation Instructions.
ℓ_4: CIA receives the decrypted Installation Instructions.

5.3.4. Step 15–16: setup installation environment

On behalf of CIA (see Step 15.4 in Fig. 30 and Table 15), CSA sets up an installation environment using the MKM. Afterwards, CSA is ready to decrypt software and unlock the ECUs (see Section 5.3.5). For this sub-problem, CSA and CIA have starting contexts because their listening tasks are run in previous sub-problems (see Section 5.3.2 and 5.3.3 respectively). CIA's starting context has access to a set of decrypted Installation Instructions (see Section 5.3.3).

$$S = \{InstallationInstructions,\ MKM_{Key},\ SKA_{Key},\ PSA_{PrivateKey},\ Root_{PrivateKey}\}$$
$$D = \{(InstallationInstructions,\ PIA),\ (SKA,\ PSA),\ (MKM, PSA),\ (PSA_{Cert}, Root),\ (Root_{Cert}, Root)\}$$

To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define S, D and the partial order $\mathcal{P}(\ell) = \ell_i < \ell_{i+1}$ for $1 \leq i \leq 2$ with the following labels:

ℓ_1: CIA validates MKM and SKA.
ℓ_2: CSA associated the MKM keys with their policy.
ℓ_3: CIA receives MKM status from CSA.

5.3.5. Step 17: stream update to ECU

The goal of this sub-problem is to stream software updates to ECUs. This can mainly be divided into two processes; first, an ECU needs to be unlocked via a challenge-response schema, i.e., security access. The CSA solves the unlocking on behalf of CIA (see step 17.1 – 17.9 in Fig. 31 and Table 16). Second, the software is transmitted to the ECU from CIA, after it has been decrypted by the CSA. The software signature is validated, and depending on whether the software was successfully installed or not, a status message is sent to CIA (see steps 17.12 – 17.21). These two steps are then repeated to install different software on different ECUs (Strandberg et al., 2021a, Fig. 4). However, as mentioned in Section 4.2.5 we only consider the update of one single ECU for one occasion.

For this sub-problem, Consumer Local Storage, CSA, and CIA have starting contexts because their respective initiation and listening tasks are run in previous sub-problems (see Section 5.3.1 to 5.3.3). Consumer Local Storage's starting context contains $Software_{Encased}$. CSA's starting context contains $MKM_{SecurityAccess_{Key}}$, $MKM_{Software_{Key}}$ and VCM_{Cert}. CIA's starting context contains $SKA_{SecurityAccess_{Key}}$, $SKA_{Software_{Key}}$, $MKM_{SecurityAccess_{Key}}$ and $MKM_{Software_{Key}}$.

$$S = \{SKA_{Software_{Key}},\ SKA_{SecurityAccess_{Key}},\ MKM_{Software_{Key}},\ MKM_{SecurityAccess_{Key}},\ Vehicle_{PrivateKey},\ Supplier_{PrivateKey},\ VCM_{PrivateKey},\ Root_{PrivateKey}\}$$
$$D = \{(Software,\ Supplier),\ (SKA_{Software_{Key}},\ PSA),\ (SKA_{SecurityAccess_{Key}},\ PSA),\ (MKM_{Software_{Key}},\ PSA),\ (MKM_{SecurityAccess_{Key}},\ PSA),\ (Supplier_{Cert},\ Root),\ (VCM_{Cert},\ Root),\ (Root_{Cert},\ Root)\}$$

To derive the specific sub-problem requirements from the system requirements (see Section 3.5), we define S, D and the partial order $\mathcal{P}(\ell) = \{\ell_1 < \ell_2, < \ell_3 < \ell_4 < \ell_9,\ \ell_5 < \ell_6, < \ell_7 < \ell_8 < \ell_9\}$. In other words, both sequences ℓ_{1-4} and ℓ_{5-8} happen before ℓ_9 but can be executed concurrently.

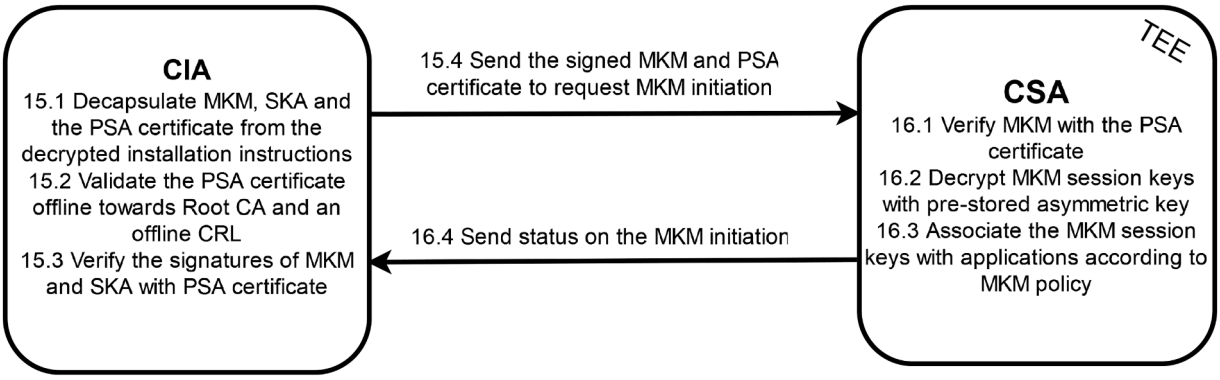

Fig. 30. Diagram of the sub-problem *Setup Installation Environment*.

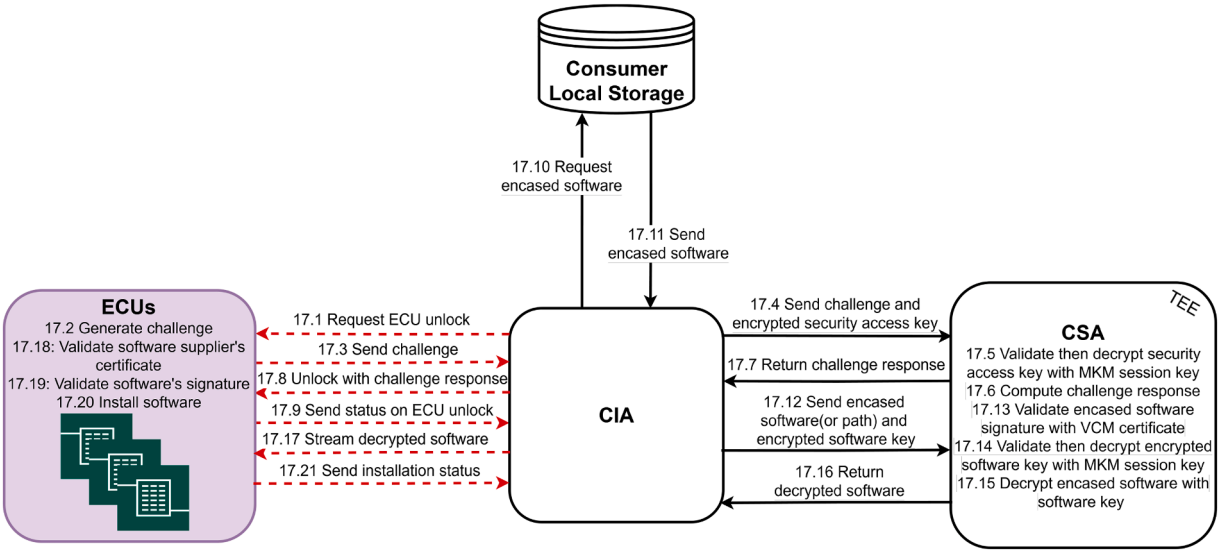

Fig. 31. Diagram of the sub-problem *Stream Update to ECU*.

ℓ_1: ECU generates challenge.
ℓ_2: CIA forwards the challenge to CSA.
ℓ_3: CSA responds to the challenge.
ℓ_4: ECU accepts the challenge.
ℓ_5: Consumer Local Storage sends $Software_{Encased}$ to CIA.
ℓ_6: CIA receives the $Software_{Encased}$ from Consumer Local Storage.
ℓ_7: CIA sends the $Software_{Encased}$ and $Software_{Key}$ to CSA.
ℓ_8: CSA decrypts software.
ℓ_9: ECU installs software.

6. Methods

We present the methods for simulating our assumptions and system settings (see Section 3) in ProVerif. Furthermore, we outline the methods used to model our requirements and provide a summary of the verification results. We also formally demonstrate our proofs for intra-round uniqueness and termination, i.e., System-level Requirements 3.5.4 and 3.5.6.

6.1. Simulation of cryptographic primitives in ProVerif

In this section, the simulation of cryptographic primitives is described.

6.1.1. Unauthenticated symmetric encryption

Blanchet et al. (2018) illustrates an example of unauthenticated symmetric encryption, as seen in Listing 6.1, by defining functions for symmetric encryption `senc` and symmetric decryption `sdec`. Both functions take arguments of a `bitstring` (a built-in type) and a `key`. Note that `key` is a user-defined type that represents symmetric keys. The functions `senc` and `sdec` output the ciphertext and plaintext, respectively. The equations on lines 4 and 5 describe the relationship between the functions `senc` and `sdec`, where `m` is the message and `k` is the key. These equations ensure that whenever the algorithm decrypts some data, `sdec` outputs the original plaintext if and only if the same key was used during encryption and the ciphertext has not been modified. Otherwise, it outputs some arbitrary data.

Listing 6.1: Unauthenticated symmetric encryption (Blanchet et al., 2018, Sec. 4.2.2).

```
1 type key.
2 fun senc(bitstring, key): bitstring.
3 fun sdec(bitstring, key): bitstring.
4 equation forall m: bitstring, k :key; sdec(senc(m,k), k) = m.
5 equation forall m: bitstring, k :key; senc(sdec(m,k), k) = m.
```

6.1.2. Authenticated symmetric encryption

The authenticated encryption in Listing 6.2 ensures that the same key is used for encryption and decryption and that modified ciphertexts

Table 16
Communication scheme for the sub-problem *Stream Update to ECU*. Note that this sub-problem can be repeated (Strandberg et al., 2021a, Fig.4), and therefore $SKA_{SecurityAccess_{Key}}$, $SKA_{Software_{Key}}$ and $Software$ might refer to different keys and software for different iterations.

(17.1)	$CIA \rightarrow ECU$	$Request(ECU)$
(17.3)	$ECU \rightarrow CIA$	$ECU_{Challenge}$
(17.4)	$CIA \rightarrow CSA$	$ECU_{Challenge} \parallel AuthSymEnc(SKA_{SecurityAccess_{Key}}, MKM_{SecurityAccess_{Key}})$
(17.7)	$CSA \rightarrow CIA$	$ECU_{Challenge-Response}$
(17.8)	$CIA \rightarrow ECU$	$ECU_{Challenge-Response}$
(17.9)	$ECU \rightarrow CIA$	$ECU_{Unlocked}$
(17.10)	$CIA \rightarrow ConsumerLocalStorage$	$Request(Software)$
(17.11)	$ConsumerLocalStorage \rightarrow CIA$	$Software_{Encased}$
(17.12)	$CIA \rightarrow CSA$	$Software_{Encased} \parallel AuthSymEnc(SKA_{Software_{Key}}, MKM_{Software_{Key}})$
(17.16)	$CSA \rightarrow CIA$	$[Software]_{Supplier}$
(17.17)	$CIA \rightarrow ECU$	$[Software]_{Supplier}$
(17.21)	$ECU \rightarrow CIA$	$ECU_{Installation-Status}$

are detected. In ProVerif, we model this by defining the decryption as a destructor through the reserved word `reduc` (Blanchet et al., 2018, Sec. 3.1). If the mentioned abnormalities are detected during the decryption, ProVerif blocks. Alternatively, the `authSdec`: `let m = authSdec(c) in P else Q` syntax can be used, such that process `P` is run if decryption succeeded and `Q` is run on failure. If `else Q` is omitted, then the process `P` terminates on failure.

To clarify, the behavior of the destructor is specified through reduction rules. These can be expressed with `reduc forall`, which introduces universally quantified rewrite rules. In particular, the rule in Listing 6.2 means: for all keys `k` and messages `m`, decrypting a ciphertext produced by `authSenc(m, k)` with the same key `k` yields the original message `m`. In other words, `reduc forall` tells ProVerif how to symbolically simplify terms when the left-hand side pattern matches. If no rule applies (e.g., due to a wrong key or a tampered ciphertext), the reduction does not occur, and the process either blocks or executes the else branch.

Listing 6.2: Authenticated symmetric encryption (Blanchet et al., 2018, Sec. 3.1.2).

```
fun authSenc(bitstring, key): bitstring.
reduc forall m: bitstring, k: key; authSdec(authSenc(m, k), k) = m.
```

6.1.3. Asymmetric encryption

Asymmetric encryption (see Listing 6.3) is similar to authenticated symmetric encryption. The main difference is in the consideration of key pairs. In ProVerif, the keypair is modeled such that the public key can be retrieved from the private key, but not the other way around (Blanchet et al., 2018). Also, a message decrypted with a private key must have been encrypted with the corresponding public key.

Listing 6.3: Asymmetric encryption (Blanchet et al., 2018, Sec. 3.1.2).

```
type skey.
type pkey.
fun pk(skey): pkey.
fun aenc(bitstring, pkey): bitstring.
reduc forall m: bitstring, k: skey; adec(aenc(m, pk(k)), k) = m.
```

6.1.4. Hash function

The hash function (see Listing 6.4) takes a `bitstring` as input and outputs an arbitrary `bitstring`, and has no associated destructors or equations (Blanchet et al., 2018, Sec. 4.2.5). By excluding destructors and equations, the hash function resembles a random oracle model, making it impossible to reverse the hash to obtain the original value.

Listing 6.4: Hash function (Blanchet et al., 2018, Sec. 4.2.5).

```
fun hash(bitstring): bitstring.
```

6.1.5. Digital signature

The modeling of digital signatures can be more complex in comparison to other cryptographic primitives. Blanchet et al. (2018) describe a method to simulate the signing function. It is modeled as a function that takes a message of type `bitstring` and a signing private key of type `sskey` as input and outputs the signed message as a `bitstring` (see Listing 6.5). A reducer is used to validate the authenticity of a signed message. The reducer is specified to pattern match on the correct private key, similar to how asymmetric encryption is simulated.

Listing 6.5: Digital signatures schema presented in the ProVerif manual (Blanchet et al., 2018, Sec. 3.1.2.).

```
type sskey.
type spkey.

fun spk(sskey): spkey.
fun sign(bitstring, sskey): bitstring.

reduc forall m: bitstring, k: sskey; getmess(sign(m, k)) = m.
reduc forall m: bitstring, k: sskey; checksign(sign(m, k), k) = m.
```

However, UniSUF requires a more complex signing schema, since data signing in UniSUF considers multiple steps and messages. Therefore, the previously stated method of signing function fails to capture all our use cases.

We specify a more complex signing schema (see Listing 6.6). This schema allows for building a signature step-by-step:

- The `sgnHash` is used to sign a hash.
- The `createSgn` function takes the signed hash of a message and appends it to the message.
- The `validateSgn` function checks that a properly signed message contains the message and the signed hash of the message.
- The `equation` allows the more traditional `sgn` signing function to be used interchangeably with `createSgn`.

In summary, `sgnHash` and `createSgn` are used to properly model the signing process in UniSUF, while `sgn` serves as a shorthand to create signatures.

Listing 6.6: Digital signatures (Blanchet et al., 2018, Sec. 3.1.2).

```
type sskey.
type spkey.
fun spk(sskey): spkey.

fun sgnHash(bitstring, sskey): bitstring.
fun sgn(bitstring, sskey): bitstring.
fun createSgn(bitstring, bitstring): bitstring.

equation forall m: bitstring, ssk: sskey;
    createSgn(m, sgnHash(hash(m), ssk)) = sgn(m, ssk).
reduc forall m: bitstring, k: sskey;
    validateSgn(createSgn(m, sgnHash(hash(m), k)), spk(k)) = m.
```

6.1.6. Certificates

For certificates, we use a simplified version of the implementation presented by Wang (2024, Appendix A.1). On line 2 (see Listing 6.7), the `createCert` function outputs a certificate from a signing public key and a signing private key. We define two destructors `validateCert` and `getCert` for the function `createCert`. The destructor `validateCert` validates whether the public key corresponds to the entity that issued the certificate, and outputs the holder's public key if and only if this

validation passes. The destructor `getCert` is similar to `validateCert`, except that `getCert` does not validate the certificate.

Listing 6.7: Certificates (Wang, 2024, Appendix A.1).

```
type cert.
fun createCert(spkey, sskey): cert.
reduc forall holderSpk: spkey, issuerSsk: sskey;
    validateCert(createCert(holderSpk, issuerSsk), spk(issuerSsk)) =
    ↪ (holderSpk, spk(issuerSsk)).
reduc forall holderSpk: spkey, issuerSsk: sskey;
    getCert(createCert(holderSpk, issuerSsk)) = (holderSpk,
    ↪ spk(issuerSsk)).
```

6.2. Representing requirements in ProVerif

ProVerif allows assertions of the system properties that the model must fulfill. All such assertions are declared using the `query` keyword (Blanchet et al., 2018). In this section, we explain our methods for specifying the security requirements of UniSUF in ProVerif.

6.2.1. Modeling confidential secrets

One of the main features of ProVerif is the ability to verify the secrecy of the variables (Blanchet et al., 2018). For secrecy queries, such as `query attacker(new x).`, ProVerif will attempt to prove that there exists no reachable state in which the attacker can access the local variable `x`. This query is a shorthand for `not attacker(new x).`, which means the query will output true if there is no state where the attacker can obtain the secret. More complex queries can also be created. We use the signature function in Section 6.1.5 as an example: `query attacker(sgn(new x, new ssk))`. This checks that `x` signed with the private key `ssk` cannot be learned by the adversary.

6.2.2. Modeling integrity of handling events

ProVerif allows users to define events using the keyword `event` (Blanchet et al., 2018, Sec. 3.2.2). These user events represent our handling events, with the event's name serving as the label (ℓ). Also, data can be associated with user events, as shown in lines 1–3 in Listing 6.8, thereby representing the cryptographic materials (d) in the handling events. The update round identifier (r) is also passed in as data to the user events. By using the same r for all events, we ensure that the execution stays the same during the update round. However, d is specified for each handling event in the execution.

Listing 6.8: Modeling integrity of handling events in ProVerif. Line 6 asserts that event `C` is reachable. The nested correspondence assertion in line 7 specifies that if the event `C` has happened, then event `B` must have occurred at an earlier time, and event `A` must have happened before event `B`.

```
event A(bitstring, bitstring).
event B(bitstring, bitstring).
event C(bitstring, bitstring).

query r: bitstring, d1: bitstring; d2: bitstring, d3:bitstring;
    event(C(r, d1));
    event(C(r, d1)) ==> event((B(r, d2)) ==> event(A(r, d3))).
```

The integrity of handling events requires that handling events are executed according to a specified partial order. ProVerif correspondence assestions (Blanchet et al., 2018, Sec. 3.2.2) represents this requirement because they specify relationships between the events ensuring they occur in the desired order. For example, the assertion `event(e1) ==> event(e2)` states that whenever event `e1` has occurred, `e2` must have occurred previously.

Correspondence assertions can be extended to model a chain of events by using nested correspondence assertions (Blanchet et al., 2018, Sec. 4.3.1). The nested correspondence assertion in Listing 6.8 specifies that if event `C` has occurred, then event `B` must have occurred previously, and event `A` must have occurred before `B`. Note that line 6 is a reachability query that checks if the event `C` is reachable. If event `C` is never reached, then the entire nested correspondence assertion is vacuously true and, therefore, not meaningful. That is, any system will meet the requirement as long as it never executes `C`.

The partial order of handling events can be modeled by creating different queries for chains of correspondence assertions. Lines 1 and 2 in Listing 6.9 give an example of two independent sequences that can occur concurrently: `C` must have occurred before `B` and `E` must have occurred before `D`. Additionally, line 3 in Listing 6.9 illustrates a method for asserting that multiple events have occurred before a specific event (Blanchet et al., 2018, Sec. 4.3.1). By linking the events using conjunctions, the listed events must have happened before the target event but in no specific order. This enables modeling concurrent events in a sequence of events.

Listing 6.9: Code example of partially ordered events being modeled in ProVerif (Blanchet et al., 2018, Sec. 4.3.1). Note that declarations of events are excluded to avoid clutter.

```
query event(B) ==> event(C).
query event(D) ==> event(E).
query event(A) ==> (event(B) && event(D)).
```

6.2.3. Modeling integrity of cryptographic materials

The integrity of cryptographic materials requires that the materials being processed remain the same during a sub-problem (see Section 3.5). This can be modelled by extending the approach used for the integrity of handling events (see Section 6.2.2). By using correspondence assertions, we model the relationships between cryptographic materials that are consumed by different handling events. Because we look at relationships, we can verify that the materials remain unchanged even as they are involved in various cryptographic transformations.

The code snippet in Listing 6.10 demonstrates this approach. Line 3 specifies that the cryptographic materials `cm` have not been modified between the events. Additionally, by looking at the signing function `sgn` (see Section 6.1.5) and the private key `ssk`, we can see that the `cm` was signed by a specific private key. Namely, the key belonging to the public key we specify in `A` and `B`: `pk(ssk)`.

Listing 6.10: Modeling integrity of cryptographic materials. The assertion verifies that if event `C` occurs (`cm` is encrypted and then signed), then event `B` must have occurred earlier (`cm` was encrypted), and event `A` must have happened even earlier (the original unencrypted `cm` was available).

```
query cm: bitstring, ssk: sskey;
    event(C(sgn(aenc(cm, spk(ssk)), ssk)));
    event(C(sgn(aenc(cm, spk(ssk)), ssk))) ==> (event(B(aenc(cm,
    ↪ spk(ssk)))) ==> (event(A(cm, pk(ssk))))).
```

Because we divide UniSUF into sub-problems (see Section 5), there are assertions where we need cryptographic materials that are not used in the sub-problem. For example, some data could have been previously signed in another sub-problem. In that case, we need to create the signature with a private key only available during the system setup (see Section 6.3.1), but not for any entities in the sub-problem. We create a special `event started(r, d).` that is executed during the system setup and contains cryptographic materials (d) not used in the sub-problem. This event allows us to make assertions with these materials.

As explained in Section 6.2.2 and in this section, both integrity requirements can be specified using nested correspondence assertions. Therefore, we use a single nested correspondence assertion to specify the requirements for both handling events and the cryptographic materials.

Table 17
Summary of mapping from system-level requirements to ProVerif specifications and the corresponding verification results.

Requirement (Section 3.5)	**ProVerif formulation** (Section 6.2.1 to 6.2.4)	**Result**
Confidential Secrets (Req. 3.5.1)	Secrecy queries, e.g., `query attacker(new S)`, for all confidential materials `S`, such as cryptographic keys and software updates.	Verified
Integrity of Cryptographic Materials (Req. 3.5.2)	The mapping is similar to that of *Integrity of Handling Event*, but each event is extended with parameters specifying the expected cryptographic materials, e.g., `event(e_i(d_1,…,d_j))`, for all handling events and cryptographic materials `{d_1,…,d_j}` specified.	Verified
Inter-Round Uniqueness (Req. 3.5.3)	Correspondence assertions to check that no two distinct rounds produce the same data, e.g., `event(e_i(r_1, d)) && event(e_i(r_2, d)) ==> r_1 == r_2`, for any pair of rounds `r_1` and `r_2`, some event `e_i`, and all produced materials `d` in the current sub-problem.	Verified
Intra-Round Uniqueness (Req. 3.5.4)	Not verified using ProVerif. See proof in Section 6.5.	Proven
Integrity of Handling Event (Req. 3.5.5)	Reachability queries, e.g., `event(e_1)`, and correspondence assertions using event queries, e.g., `event(e_1) ==> … ==> event(e_j)`, for all handling events `{e_1,…,e_j}` specified.	Verified
Termination (Req. 3.5.6)	Not verified using ProVerif. See proof in Section 6.6.	Proven

6.2.4. Modeling inter-round uniqueness

Informally, inter-round uniqueness means that no two distinct update rounds produce the same data. We can verify this property in ProVerif using correspondence assertions, similar to the approach used by Wang (2024). Listing 6.11 demonstrates how the inter-round uniqueness property is expressed in ProVerif. In this example, line 2 specifies a precondition that checks whether both instances of the event `A` are reachable. Since both instances of `A` produce the same data, they must originate from the same update round. Consequently, the assertion on line 3 requires that if both events are reachable, they must have occurred within the same update round.

Listing 6.11: Code example of the inter-round uniqueness query for event `A`.

```
1 query round1: bitstring, round2: bitstring, data: bitstring;
2     event(A(round1, data)) && event(A(round2, data))
3     ==> round1 = round2.
```

6.2.5. Verification summary

Table 17 summarizes the mapping between each system-level requirement from Section 3.5 and the corresponding ProVerif artifacts used for verification, along with the outcomes obtained.

To assess the computational overhead introduced by our formal verification model, we measured the verification time for each sub-problem described in Section 5. The experiments were conducted on a Dell Latitude 5450 laptop using ProVerif 2.05. For each sub-problem, we recorded the average runtime of the ProVerif process over 50 repeated runs, discarding the maximum and minimum values. The resulting verification times are summarized in Table 18. Notably, verifying the entire set of sub-problems requires about two seconds. As discussed in Section 1.2, the formal verification overhead occurs exclusively at design time and is therefore incurred only once.

6.3. Simulation of system settings and assumptions

We describe the techniques for simulating the assumptions listed in Section 3.

Table 18
ProVerif verification time in milliseconds for each of the three main problems (e.g., preparation, encapsulation, and decapsulation) and their respective sub-problems, see Section 5. As mentioned in Section 1.2, the verification overhead occurs only at design time.

Task	Sub-problem	Time [ms]	% of Total
Preparation	Step 1–4	31 ± 1	1.47 %
Preparation	Step 5–6	21 ± 1	1.00 %
Preparation Total		52 ± 2	2.47 %
Encapsulation	Step 1–2	19 ± 1	0.90 %
Encapsulation	Step 3	58 ± 1	2.76 %
Encapsulation	Step 4	58 ± 1	2.76 %
Encapsulation	Step 5 and 7	605 ± 10	28.75 %
Encapsulation	Step 6	56 ± 2	2.66 %
Encapsulation	Step 8	497 ± 10	23.62 %
Encapsulation	Step 9–11	17 ± 1	0.81 %
Encapsulation Total		1311 ± 27	62.31 %
Decapsulation	Step 1–4	314 ± 6	14.92 %
Decapsulation	Step 5–9	129 ± 3	6.13 %
Decapsulation	Step 10–14	112 ± 2	5.32 %
Decapsulation	Step 15–16	29 ± 1	1.38 %
Decapsulation	Step 17	156 ± 3	7.41 %
Decapsulation Total		741 ± 14	35.22 %
Total		2104 ± 43	100 %

6.3.1. Setting up cryptographic materials and starting contexts

As mentioned in Section 4.2, we consider a system with a single producer responsible for producing updates for multiple vehicles. Additionally, vehicles occasionally need to update their software with the latest versions, i.e., each vehicle can be updated multiple times. While some of the cryptographic materials we identify in Section 4.1 are used for multiple updates, others are ephemeral, i.e., they can only be used during their designated update round. Next, we show how to simulate the relationships between update rounds and cryptographic materials.

We present the Mapping Tree in Fig. 32, which is a tree consisting of different processes. The root *process* invokes multiple vehicles in its child processes *setupVehicle*, and each vehicle has multiple update rounds (invoked in the child process *setupUpdateRound*). In each process, we create cryptographic materials. Because the materials for each sub-problem vary, we create a tree for each sub-problem. Additionally, the sub-problems related to software preparation (see Section 5.1) do not consider vehicles. Therefore, *setupVehicle* is not included for these sub-problems. Instead, *process* directly invokes *setupUpdateRound*. Moreover, for *Secure Software Files* (see Section 5.1.1) and *Order Initiation* (see Section 5.2.1), *setupUpdateRound* is omitted. This is because these sub-problems contain the initiation task, meaning the update round of the current execution has not been initiated (see Section 4.3).

In the root *process*, we create the software supplier certificate and all producer certificates because these are used in all update rounds. However, the vehicle certificate is only used in update rounds for its vehicle. Therefore, this certificate is created in *setupVehicle*. All other cryptographic materials are created in *setupUpdateRound* because they are only used in a single update round.

Each process passes down its cryptographic materials to its children processes. Therefore, *setupUpdateRound* can access cryptographic materials from *setupVehicle* and *process*, while *setupVehicle* can access materials from the root *process*.

6.3.2. Mapping starting contexts to cryptographic materials

As mentioned in Section 4.3, an entity participating in a sub-problem can have a starting context. This occurs when the entity has previously run either the initiation or listening task for another sub-problem in the same update round. We, therefore, extend our mapping tree from Section 6.3.1 to account for starting contexts (see Fig. 33).

All producer and software repository entities with no starting context are invoked by *process*. Therefore, they only have access to the

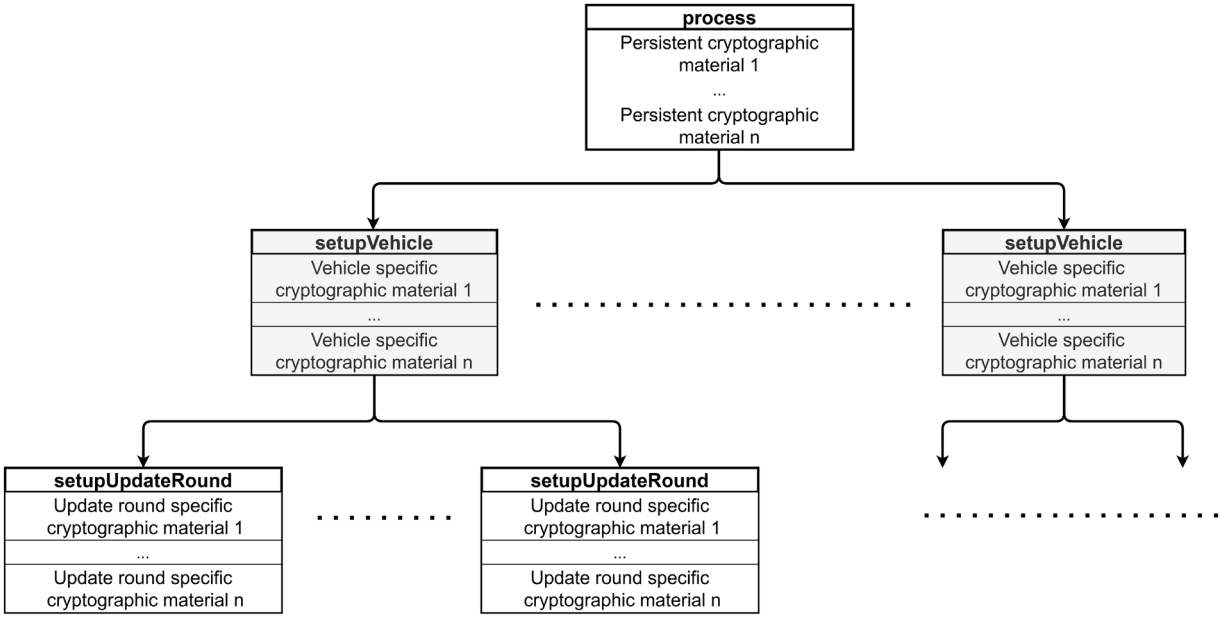

Fig. 32. Mapping Tree creates multiple update rounds for each vehicle. The tree also ensures that cryptographic materials are used in their assigned update round.

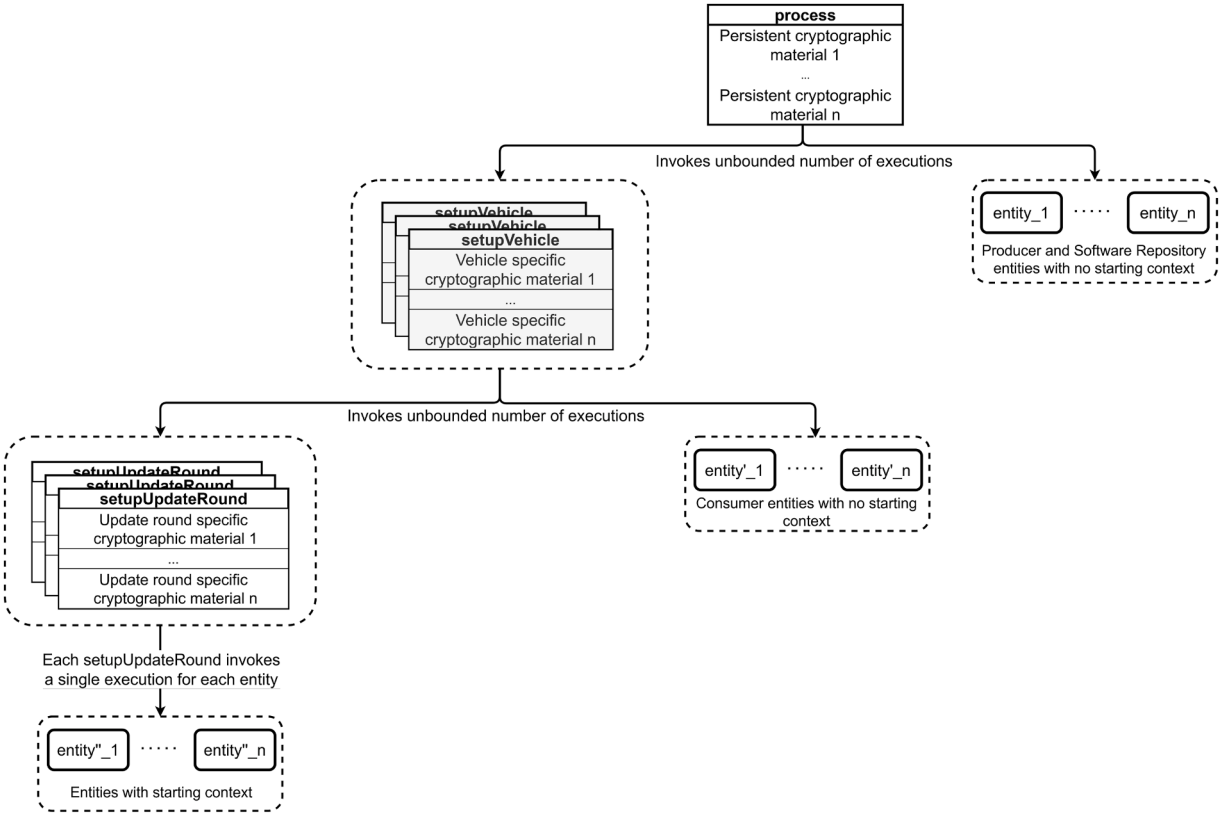

Fig. 33. Extension of the mapping tree in Fig. 32. This tree invokes the starting contexts for entities in the tree's sub-problem. The entities with starting contexts receive them when they are invoked by the tree contexts.

cryptographic materials used in all update rounds, i.e., they cannot access materials coupled to a vehicle or update round. Consumer entities with no starting context also have access to vehicle-specific materials and are therefore invoked by *setupVehicle*.

An entity with a starting context is invoked by *setupUpdateRound*. These entities can access the cryptographic materials created for the update round. We make sure to specify the materials in accordance with Section 5, such that only materials previously created in previous sub-problems are available.

Note that we create v_{id} and t_e (see Section 3.4) in *setupVehicle* and *setupUpdateRound* respectively. This allows us to separate the different update rounds from each other.

6.3.3. ProVerif simulation of the mapping tree with update rounds

ProVerif uses a single main process, defined with the reserved word `process`, and sub-processes can be defined as macros by using the reserved word `let` (Blanchet et al., 2018, Sec. 3.1). We use these two reserved words to implement our mapping tree (see Fig. 33), as seen in Listing 6.12. For easier comprehension, the names of the sub-processes follow the mapping tree's structure.

Listing 6.12: Code example of how entities are instantiated with their corresponding data, thereby simulating the update round mapping tree in Fig. 33.

```
let entity_1 ((*Parameters for entity_1*)) =
    (*Algorithm for entity_1....*).
(*
Define entity_n, entity'_1, entity'_n, entity''_1 and entity''_n as entity_1 was defined
but with different parameters and algorithms.
*)

let setupVehicle((*Parameters for the vehicle*)) =
    (*Initiate Vehicle specific material*)
    !setupUpdateRound((*Send material to each update round*)) |
    !entity'_1((*Send initial material to entity'_1*)) |
    (* .... | *)
    !entity'_n((*Send initial material to entity'_n*)).

let setupUpdateRound((*Parameters needed for the update round*)) =
    (*Initiate update round specific material*)
    (*Running all entities in an update round*)
    entity''_1((*Send initial material to entity''_1*)) |
    (* .... | *)
    entity''_n((*Send initial material to entity''_n*)).

process
    (*Initiate persistent cryptographic material*)
    !setupVehicle((*Send initial material to each vehicle*)) |
    !entity_1((*Send initial material to entity_1*)) |
    (* .... | *)
    !entity_n((*Send initial material to entity_n*))
```

Note the exclamation operator, `!`, in the listing, which in ProVerif invokes an unbounded number of replications of a process (Blanchet et al., 2018, Sec. 3.1.4). This operator corresponds to the arrows we mark with *Invokes unbounded amount of processes* in our tree. By using an unbounded amount of invocation per process, we ensure that each vehicle receives multiple updates.

To give an execution of an entity access to the cryptographic materials discussed in Section 6.3.2, we pass the materials as parameters when invoking the execution. In ProVerif, this is achieved by: `entity_1(cryptographicMaterial)` (Blanchet et al., 2018, Sec. 3.1).

Since all public cryptographic materials are available to all, the adversary is explicitly made aware of them. To simulate this, we send the public material out on a channel that the adversary can read on, as such (Blanchet et al., 2018, Sec. 3.1):

```
    out(publicChannel, (publicData1, publicData2, ..., publicDataN))
```

Note that this is done for all public cryptographic materials created in *process*, *setupVehicle* and *setupUpdateRound*.

6.3.4. Reliable communication

We simulate reliable communication in which the receiving-side messages are delivered according to the order in which the sender fetched them. The session identifier and message sequence number are included for all messages. The sequence number is incremented for each new message. This simulates a connection establishment and ensures the correct ordering of messages.

Listing 6.13: Simulation of reliable communication in ProVerif.

```
1  free alice_bob: channel.
2
3  table bobSessions(bitstring, bitstring).
4
5  let Alice() =
6      new sessionId: bitstring; (* initiate the session identifier *)
7      new data1: bitstring; (* data to be sent *)
8      out(alice_bob, (sessionId, data1, 1));
9      in(alice_bob, (=sessionId, data2: bitstring, =2)).
10
11 let Bob() =
12     new processId: bitstring; (* each instance of Bob has a unique process ID *)
13     (* receive the session ID and data from Alice *)
14     in(aliceBob, (sessionId: bitstring, data1: bitstring, =1));
15
16     insert bobSessions(sessionId, processId);
17     get bobSessions(=sessionId, processId': bitstring) suchthat processId <> processId' in
18     (* another execution of Bob is already in the session, so we abort *)
19         0
20     else (
21     (* no other execution of Bob is currently in the session, so we proceed *)
22         new data2: bitstring;
23         out(alice_bob, (sessionId, data2, 2))
24     ).
```

Our simulation is illustrated in Listing 6.13. Alice initiates the session by sending a message that includes a session identifier, data, and sequence number 1 (line 8). The session identifier tracks the session, while the hardcoded sequence number orders messages within it.

In line 14, Bob listens to the first message of any session by using the ProVerif pattern matching operator (=) (Blanchet et al., 2018, Sec. 3.1.2). When Bob receives such a message, we check if another execution of Bob is already in the session. This is because we use an unbounded number of processes (see Section 6.3.3), which means that multiple executions of Bob can join the session. To prevent another execution of the same process from joining the same session, we adapt Wang's solution (Wang, 2024, Sec. 9.3.2).

Specifically, in line 3, we set up a table to store the session and the process identifiers. Then, in lines 16–17, we add the session identifier and Bob's process identifier to the table. We then check if another instance of Bob is already in the session, i.e., if there are at least two different process identifiers for the given session. If so, we abort the session establishment process. Otherwise, Bob proceeds to join the session. Wang (2024, Sec. 9.3.2) states that this solution works because ProVerif schedules the processes such that, at most, one execution is allowed to proceed for a given session. Furthermore, ProVerif explores all possible schedules, including the schedules in which only a single execution is permitted to continue.

6.3.5. Secure and reliable communication

We enhance the reliable communication channel with security guarantees. In ProVerif, based on the Dolev-Yao threat model, the adversary has full control over the communication channels (Blanchet et al., 2018, Ch. 3). They can freely read, update, or insert channel messages. This means messages are vulnerable to eavesdropping and tampering by an attacker. To prevent the adversary from reading, updating, or inserting channel messages, we declare the channel as private (Blanchet et al., 2018, Sec. 6.7.4): `free c: channel[private].`

6.3.6. Update rounds

We describe how we simulate update rounds.

6.3.7. Replacing session identifiers with update round identifiers

We include the update round identifier in all messages, as mentioned in Section 3.4. The update round identifier provides context to the

cryptographic materials transmitted in our code and removes the need for the session identifiers discussed in Section 6.3.4. This is because an update round encapsulates multiple peer-to-peer sessions. This change is simple to implement. Instead of using session identifiers (`sessionId`), we use the update round identifier (see Section 3.4), i.e., VIN and the expiration time (`vin, expirationTime`), or just the expiration time (`expirationTime`).

6.3.8. Simulating the listening task in ProVerif

As mentioned in Section 4.3, the first task of all listeners is the listening task. We simulate this task so that listeners can discover new update rounds. As mentioned in Section 6.3.4, we adapt a solution by Wang (2024, Sec. 9.3.2), to hinder multiple executions from working on the same session. For update rounds, we do the same, but we use a table containing update round identifiers instead of session identifiers.

Listeners that are considered producer and software repository entities are made aware of the round's VIN and expiration time, as follows:

```
in(c, (vin: bitstring, expirationTime: bitstring, ..., =1);
```

In contrast, consumer listeners are already aware of the VIN of their vehicle (see Section 3.1). Therefore, consumer listeners only learn the expiration time:

```
in(c, (=vin, expirationTime: bitstring, ..., =1));
```

Moreover, the equals operator (=), which is used for pattern matching in ProVerif (Blanchet et al., 2018, Sec. 3.1.4), ensures that the consumer listeners only work on update rounds intended for their vehicle.

6.3.9. Unbounded number of processes in ProVerif and considerations

Using unbounded processes running concurrently in ProVerif (Blanchet et al., 2018) means that our update rounds can also occur concurrently in our model. However, Strandberg (2024) notes that this may not accurately represent reality, where update rounds occur sequentially for a given vehicle. The strict sequential update rounds could be achieved by allowing a vehicle to update only once. Although this would simplify the solution, it would also trivialize the problem, as the adversary would not be able to replay messages to the same vehicle since there would be no other update rounds to target. We believe that the use of unbounded concurrent processes is crucial to our proof. We argue that the set of all sequential schedules is a subset of the set of all concurrent schedules. Therefore, if our proof holds for all concurrent schedules, it will also hold for all sequential schedules.

6.3.10. Well-known addresses

We add a communication channel for each pair of entities communicating in a task. This allows us to separate the different peer-to-peer sessions inside an update round from each other. Without multiple channels, a message from Alice to Bob could end up at Charlie, that is, the channels simulate the well-known addresses discussed in Section 3.1. Specifically, channels alone simulate well-known addresses for the producer and the software repository. Meanwhile, for the consumer entities that belong to a vehicle, the VIN is also needed to simulate well-known addresses. The reason is that the VIN separates the multiple vehicles considered in our system setup (see Section 6.3.1).

6.4. ProVerif libraries

ProVerif offers a method to organize frequently used functions and macros into a library file, allowing them to be imported into other files to minimize redundant code (Blanchet et al., 2018, Sec. 6.6). In addition, the libraries ensure that the data structures appearing in multiple proofs are modeled consistently. We have developed libraries for implementing cryptographic primitives, helper functions for setting up cryptographic materials, and common message types.

6.5. Correctness proof for intra-round uniqueness

Lemma 6.1. *Consider an entity E and a handling $e(r, d, \ell)$, E executes $e(r, d, \ell)$ at most once.*

Proof. Consider an entity, E, and a handling event, $e(r, d, \ell)$, which we denote e for brevity. To execute e, entity E must be in the update round r and have previously generated the cryptographic material d. Then, there can only be two cases: either the generation of d depends on some cryptographic materials sent to E in messages, m', or E generates d entirely from scratch. For the sake of simplicity, we assume that there is only one such message. Note that similar arguments are held when multiple messages are held.

In the first case, m' must contain (r, d') where all d' were used to generate d. To execute the event e more than once, E must have received multiple versions of message m' containing (r, d'). However, this is impossible because the entities never process duplicate messages, according to the assumption in Section 3.4, which specifies that entities omit any message already in their logs.

In the second case, d is generated from scratch. As we assume perfect cryptography and, hereby, randomness is based on a random oracle, two generated materials cannot be identical. Therefore, executing e with the same d multiple times is impossible.

Thus, executing e multiple times in both cases is impossible. In other words, E executes e at most once. □

From the Lemma 6.1, we derive that the *Intra-Round Uniqueness* requirement always holds.

Corollary 6.1.1. *The System-level Requirement 3.5.4 holds for all tasks.*

6.6. Correctness proof for termination

Lemma 6.2. *All entity executions terminate eventually. Each termination is either timely or late.*

Proof. Consider an update round; if an entity runs its *halt* task for this update round before the expiration time, it terminates timely by the definition of termination in Section 3.5. Otherwise, it fails to run the halt task before the expiration time, and by assumption (see Section 4.3.2), the entity halts and terminates late. Therefore, all entity executions always terminate and each termination is either timely or late. □

From the Lemma 6.2, we derive that the *Termination* requirement always holds because all update rounds must always terminate if all entities always terminate.

Corollary 6.2.1. *The System-level Requirement 3.5.6 holds for all tasks.*

7. Conclusions

Our work scrutinizes UniSUF's requirements and our research questions (see Section 1.1). To validate UniSUF's requirements and architecture, we developed a formal model using ProVerif to ensure that the ProVerif program satisfies the specified requirements and the technological assumptions that UniSUF relies on. Furthermore, we divided the UniSUF update process into smaller, more manageable sub-problems. We analyzed and created specific requirements for each sub-problem so that the requirements of the sub-problems together fulfill the System-level Requirements of UniSUF (see Section 3.5).

Our verification results show that our symbolic execution of UniSUF in ProVerif fulfils all requirements. The system-level requirements established for UniSUF collectively address the research questions posed in Section 1.1. The *Confidential Secrets* requirement (System-level Requirement 3.5.1) ensures that the system's secrets, such as cryptographic keys and disseminated software, are not exposed to the adversary, addressing **RQ1**. The *Integrity of Cryptographic Materials* requirement (System-level Requirement 3.5.2) guarantees that the software obtained and used by UniSUF originates from the right source and has not been modified by

any other entity, addressing **RQ2**. The *Inter-Round Uniqueness* and *Intra-Round Uniqueness* (System-level Requirement 3.5.3 and System-level Requirement 3.5.4) collaboratively prevent the use of obsolete software versions and the replay of cryptographic materials within and between update rounds. This guarantees that UniSUF always performs software updates with the correct up-to-date versions, thus addressing **RQ3**. The *Integrity of Handling Events* (System-level Requirement 3.5.5) ensures the operations in the software update process follow the order specified by UniSUF, thus addressing **RQ4**. The last requirement, *Termination* (System-level Requirement 3.5.6), ensures that the update process always terminates, addressing **RQ5**.

7.1. Discussion

Although our work proves the UniSUF model in ProVerif to be secure, it does not necessarily guarantee the security of a real-world implementation of UniSUF. There is an inherent discrepancy between the latter and our formal model. Namely, formal models are intended to be complete, e.g., nothing outside the model's specification can occur, such as compromised components. However, in real-world deployments, implementation errors and unexpected events, such as evolving attacker capabilities, can affect the system security.

In addition, like any formal analysis, our results depend on the correctness of the working assumptions and abstractions. Modeling errors or oversights could lead to missed vulnerabilities, and alternative attacker models (e.g., beyond the symbolic Dolev-Yao adversary considered here) may expose other risks that fall outside our current scope. Thus, important challenges remain in the area.

This does not imply that our formal verification has failed to establish meaningful security guarantees. On the contrary, our work has rigorously demonstrated the security properties of UniSUF in the formal model. Our work establishes the provability of UniSUF's security, which can be a starting point for real-world implementations of UniSUF.

However, the security assurances provided by our model do not automatically transfer to a real-world implementation. Verifying the correctness of an actual UniSUF implementation requires a substantially different and more comprehensive analysis that goes beyond the scope of our work. Thus, bridging the gap between the formal model and a real-world implementation remains an open challenge and requires further investigation.

7.2. Future work

Although UniSUF has been our case study, the methodology itself is general: by adapting the modeled roles, message formats, and requirements, a similar decomposition and verification approach can be applied to other software update frameworks in the automotive or IoT domains. As additional directions for future work, we propose to extend our approach to cover multi-ECU and large-scale vehicular networks. The modular decomposition of UniSUF into sub-problems (Section 5) and the requirement mapping in Table 17 naturally support compositional reasoning across ECUs, enabling scalability to larger deployments. Prior analyses of Uptane that incorporate secondary ECUs Kirk et al. (2023), Lorch et al. (2024), Boureanu (2023) highlight both the feasibility and the challenges of scaling symbolic verification, especially with respect to state explosion. We envision that assume-guarantee style reasoning and modular lemmas can address these challenges, while preserving strong guarantees at the system level.

Acknowledgments

This research was partly supported by the MAGIC project (2024-03687) funded by VINNOVA, the Swedish Governmental Agency for Innovation Systems.

CRediT authorship contribution statement

Martin Slind Hagen: Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis, Conceptualization; **Emil Lundqvist:** Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis, Conceptualization; **Alex Phu:** Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis, Conceptualization; **Yenan Wang:** Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis, Conceptualization; **Kim Strandberg:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Methodology, Conceptualization; **Elad Michael Schiller:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Methodology, Conceptualization.

Data availability

To ensure the reproducibility of our results and to encourage further development, prospective implementation details to the source code can be found in our complementary technical report (Hagen et al., 2025).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Implementations in ProVerif

A.1. Cryptographic primitives

Listing Appendix A.1: ProVerif library file `cryptography.pvl` that contains all our cryptographic primitives.

```
type key.

(* Helper to convert key to bitstring and back so that it can be
↪  encrypted*)
fun key2bits(key): bitstring [data, typeConverter].
fun bits2key(bitstring): key [data, typeConverter].

(* Authenticated symmetric encryption *)
fun authSenc(bitstring, key): bitstring.
reduc forall m: bitstring, k: key; authSdec(authSenc(m, k), k) = m.

(* Unathenticated symmetric encryption *)
fun senc(bitstring, key): bitstring.
fun sdec(bitstring, key): bitstring.
equation forall m: bitstring, k:key; sdec(senc(m,k), k) = m.
equation forall m: bitstring, k:key; senc(sdec(m,k), k) = m.

(* Asymmetric encryption *)
type skey.
type pkey.

fun pk(skey): pkey.
fun aenc(bitstring, pkey): bitstring.
reduc forall m: bitstring, k: skey; adec(aenc(m, pk(k)), k) = m.

fun hash(bitstring): bitstring.

type sskey.
type spkey.
fun spk(sskey): spkey.
fun sgnHash(bitstring, sskey): bitstring.
fun createSgn(bitstring, bitstring): bitstring.

reduc forall m: bitstring, k: sskey; getMess(createSgn(m,
↪  sgnHash(hash(m), k))) = m.
```

```
reduc forall m: bitstring, k: sskey; getHash(createSgn(m,
↪  sgnHash(hash(m), k)), spk(k)) = hash(m).
reduc forall m: bitstring, k: sskey; validateSgn(createSgn(m,
↪  sgnHash(hash(m), k)), spk(k)) = m.

fun sgn(bitstring, sskey): bitstring.
equation forall m: bitstring, ssk: sskey;
    createSgn(m, sgnHash(hash(m), ssk)) = sgn(m, ssk).

(* Certificate stuff *)
(*
    A ceritificate can be seen as an extension to a public key, that
    ↪  provides a
    signature of the public key signed by the issuer's secret key.
*)
type cert.

fun spkey2bits(spkey): bitstring [data, typeConverter].
fun bits2spkey(bitstring): spkey [data, typeConverter].

fun createCert(spkey, sskey): cert.
reduc forall holderSpk: spkey, issuerSsk: sskey;
    validateCert(createCert(holderSpk, issuerSsk), spk(issuerSsk)) =
    ↪  (holderSpk, spk(issuerSsk)).
reduc forall holderSpk: spkey, issuerSsk: sskey;
    getCert(createCert(holderSpk, issuerSsk)) = (holderSpk,
    ↪  spk(issuerSsk)).

(* Other type converters *)

(* In principle, these two type of secret keys should be interchangeable
↪  *)
fun skey2sskey(skey): sskey [data, typeConverter].
fun sskey2skey(sskey): skey [data, typeConverter].
(* similarily, here should also be interchangeable *)
fun pkey2spkey(pkey): spkey [data, typeConverter].
fun spkey2pkey(spkey): pkey [data, typeConverter].
```

A.2. Utilities

Listing Appendix A.2: ProVerif library file `uniSufHelpers.pvl` that contains all helper methods for setting up the cryptographic materials in Section 4.1.

```
(* -lib cryptography.pvl *)

fun setupVso(bitstring): bitstring[data].
reduc forall vin: bitstring; getVin(setupVso(vin)) = vin.

letfun setupVsoSgn(cdaSsk: sskey, vin: bitstring) =
    sgn(setupVso(vin), cdaSsk).

(* Authenticated encryption of key k, via authEncKey*)
letfun authSencKey (k: key, encK: key) =
    authSenc(key2bits(k), encK).

(*
    Generates a new signed key manifest, and returns the session key
    ↪ along with the signed manifest
*)
letfun genKeyManifestSgn(sessionK: key, vin: bitstring, expirationTime:
↪ bitstring, vehiclePk: pkey, sgnSsk: sskey) =
    let kEnc = aenc(key2bits(sessionK), vehiclePk) in
    sgn((kEnc, (vin, expirationTime)), sgnSsk).

(*
fun vuupSgn2vuupUrl(bitstring): bitstring.
reduc forall vuupSgn: bitstring;
↪ vuupUrl2vuupSgn(vuupSgn2vuupUrl(vuupSgn)) = vuupSgn.

letfun setupVuupUrlSgn(vuupSgn: bitstring, vcmSsk: sskey) =
    sgn(vuupSgn2vuupUrl(vuupSgn), vcmSsk).
*)
letfun setupVuupUrlSgn(vcmSsk: sskey) =
    new vuupUrl: bitstring;
    (sgn(vuupUrl, vcmSsk), vuupUrl).

letfun setupDkmSgn(vin: bitstring, expirationTime: bitstring, vehiclePk:
↪ pkey, pdaSsk: sskey) =
```

```
32      new dkmK: key;
33      let dkmSgn = genKeyManifestSgn(dkmK, vin, expirationTime, vehiclePk,
        ↪  pdaSsk) in
34      (dkmK, dkmSgn).
35
36  letfun setupIkmSgn(vin: bitstring, expirationTime: bitstring, vehiclePk:
    ↪  pkey, piaSsk: sskey) =
37      new ikmK: key;
38      let ikmSgn = genKeyManifestSgn(ikmK, vin, expirationTime, vehiclePk,
        ↪  piaSsk) in
39      (ikmK, ikmSgn).
40
41
42  fun createDownloadInstructions(bitstring): bitstring.
43  fun createInstallationInstructions(bitstring, bitstring, bitstring,
    ↪  cert): bitstring.
44  (*
45      Reducer does not include the signed software list as we do not want
        ↪  to be able to compute back to it
46      Same with no reducer for dowload instructions, as we do not wnat to
        ↪  be able to recompute the software list
47  *)
48  reduc forall softwareListSgn: bitstring, skaSgn: bitstring,
    ↪  mkmSgn:bitstring, psaCert: cert;
49      unpackInstallationInstructions(createInstallationInstructions(
        ↪  softwareListSgn, skaSgn, mkmSgn, psaCert)) = (skaSgn, mkmSgn,
        ↪  psaCert).
50
51
52  letfun setupSoftwareEncapsulated(supplierSsk: sskey, skaSoftwareK: key,
    ↪  vcmSsk: sskey) =
53      new software: bitstring;
54      sgn(senc(sgn(software, supplierSsk), skaSoftwareK), vcmSsk).
55
56  letfun setupDownloadInstructionsEncSgn(softwareListSgn: bitstring, dkmK:
    ↪  key, pdaSsk: sskey) =
57      let downloadInstructions =
        ↪  createDownloadInstructions(softwareListSgn) in
```

```
    sgn(senc(downloadInstructions, dkmK), pdaSsk).

letfun setupMkm(vin: bitstring, expirationTime: bitstring, vehiclePk:
↪  pkey) =
    (*Setting up MKM*)
    new mkmEcuK: key;
    new mkmSoftwareK: key;
    let mkmEcu = (aenc(key2bits(mkmEcuK), vehiclePk),
    (vin, expirationTime)) in
    let mkmSoftware = (aenc(key2bits(mkmSoftwareK), vehiclePk), (vin,
    ↪  expirationTime)) in
    let mkm = (mkmEcu, mkmSoftware) in
    (mkmEcuK, mkmSoftwareK, mkm).

letfun setupMkmSgn(vin: bitstring, expirationTime: bitstring, vehiclePk:
↪  pkey, psaSsk: sskey) =
    let (mkmEcuK: key, mkmSoftwareK: key, mkm:bitstring) = setupMkm(vin,
    ↪  expirationTime, vehiclePk) in
    (mkmEcuK, mkmSoftwareK, sgn(mkm, psaSsk)).

letfun setupSkaSoftwareK () =
    new skaSoftwareK: key;
    (skaSoftwareK).

letfun setupSka(mkmEcuK: key, mkmSoftwareK: key) =
    (* Setting up SKA, assuming only one ECU and only one sofware file
    ↪  therefore only
       one key per sub-array.

       In reality, e.g.: skaEcu = [authSenc(skaEcuK1, mkmEcuK), ...,
       ↪  authSenc(skaEcuKn, mkmEcuK)]
       However, here we have simplified it such that skaEcu =
       ↪  authSenc(skaEcuK, mkmEcuK)
    *)
    new skaEcuK: key;
    let skaEcu = authSencKey(skaEcuK, mkmEcuK) in
    let skaSoftwareK = setupSkaSoftwareK() in
```

```
88      (*new skaSoftwareK: key;*)
89
90      let skaSoftware = authSencKey(skaSoftwareK, mkmSoftwareK) in
91      (skaEcuK, skaSoftwareK, (skaEcu, skaSoftware)).
92
93  letfun setupSkaSgn (mkmEcuK: key, mkmSoftwareK: key, psaSsk: sskey) =
94      let (skaEcuK: key, skaSoftwareK: key, ska: bitstring) =
        ↪  setupSka(mkmEcuK, mkmSoftwareK) in
95      (skaEcuK, skaSoftwareK, sgn(ska, psaSsk)).
96
97  fun setupSoftwareList(bitstring, bitstring): bitstring.
98
99  letfun setupSoftwareListSgn(vcmSsk: sskey) =
100     (*Setting up softwareList*)
101     new vinData: bitstring;
102     new softwareVersions: bitstring;
103     sgn(setupSoftwareList(vinData, softwareVersions), vcmSsk).
104
105 letfun setupInstallationInstructionsEncSgn(softwareListSgn: bitstring,
    ↪  skaSgn: bitstring, mkmSgn: bitstring, psaCert: cert, ikmK: key,
    ↪  piaSsk: sskey) =
106     let installationInstructions: bitstring =
        ↪  createInstallationInstructions(softwareListSgn, skaSgn, mkmSgn,
        ↪  psaCert) in
107     sgn(senc(installationInstructions, ikmK), piaSsk).
108
109 letfun setupVuupContent (vin: bitstring, expirationTime: bitstring,
    ↪  pdaCert: cert, piaCert: cert, downloadInstructionsEncSgn: bitstring,
    ↪  dkmSgn: bitstring, installationInstructionsEncSgn: bitstring, ikmSgn:
    ↪  bitstring) =
110     (vin, expirationTime, (pdaCert, piaCert), downloadInstructionsEncSgn,
        ↪  dkmSgn, installationInstructionsEncSgn, ikmSgn).
111
112 letfun setupVuupSgn (vin: bitstring, expirationTime: bitstring,
113     pdaCert: cert, piaCert: cert, downloadInstructionsEncSgn: bitstring,
114     dkmSgn: bitstring, installationInstructionsEncSgn: bitstring,
115     ikmSgn: bitstring, vcmSsk: sskey, vcmCert: cert) =
116     (vcmCert, sgn(setupVuupContent(vin, expirationTime, pdaCert, piaCert,
        ↪  downloadInstructionsEncSgn, dkmSgn,
        ↪  installationInstructionsEncSgn, ikmSgn), vcmSsk)).
117
```

A.3. Message types

Listing Appendix A.3: ProVerif library file `uniSufMessageTypes.pvl` that contains all the message types used in our implementation.

```
type vcmInitSuccess.
type pdaSlSuccess.
type piaSlSuccess.
type psaSlSuccess.
type dkmSuccess.
type ikmSuccess.
type mkmSuccess.
type vuupUrlSuccess.

type vsoRequest.
type dkmKeyRequest.
type ikmKeyRequest.
type vehicleCertRequest.
type vuupUrlRequest.
type vuupRequest.
type vuupSuccess.
type softwareSuccess.
type softwareRequest.
type signatureRequest.
type skaSgnRequest.
type mkmSgnRequest.
type cmsCryptoRequest.
```

References

ISO14229, 2020. Road vehicles - Unified Diagnostic Services (UDS). ISO 14229.

Basin, D.A., Dreier, J., Hirschi, L., Radomirovic, S., Sasse, R., Stettler, V., 2018. A formal analysis of 5g authentication. In: SIGSAC Conference on Computer and Communications Security, CCS. ACM, pp. 1383–1396.

Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M., 2006. UPPAAL 4.0. In: Third International Conference on the Quantitative Evaluation of Systems QEST. IEEE Computer Society, pp. 125–126.

Bhargavan, K., Cheval, V., Wood, C., 2022. A symbolic analysis of privacy for TLS 1.3 with encrypted client hello. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, p. 365–379. https://doi.org/10.1145/3548606.3559360

Blanchet, B., 2007. Cryptoverif: computationally sound mechanized prover for cryptographic protocols. In: Dagstuhl Seminar "Formal Protocol Verification Applied. Vol. 117, p. 156.

Blanchet, B., 2016. Modeling and verifying security protocols with the applied pi calculus and proverif. Found. Trends® Priv. Secur. 1 (1–2), 1–135.

Blanchet, B., Smyth, B., Cheval, V., Sylvestre, M., 2018. ProVerif 2.00: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/manual.pdf.

Boureanu, I., 2023. Formally verifying the security and privacy of an adopted standard for software-update in cars: verifying uptane 2.0. In: IEEE International Conference on Systems, Man, and Cybernetics, SMC. IEEE, pp. 1301–1306.

Campbell, B., Bradley, J., Sakimura, N., Lodderstedt, T., 2020. OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens. RFC 8705. https://doi.org/10.17487/RFC8705

Chlipala, A., 2013. Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. The MIT Press.

Council, E. P. M.A., 2020. Share of new vehicles shipped worldwide with built-in connectivity in 2020 and 2030. https://www.statista.com/statistics/1276018/share-of-connected-cars-in-total-new-car-sales-worldwide/.

Cremers, C., Horvat, M., Hoyland, J., Scott, S., van der Merwe, T., 2017. A comprehensive symbolic analysis of TLS 1.3. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, p. 1773–1788. https://doi.org/10.1145/3133956.3134063

Cremers, C., Kiesl, B., Medinger, N., 2020. A formal analysis of IEEE 802.11's WPA2: countering the kracks caused by cracking the counters. In: 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, pp. 1–17. https://www.usenix.org/conference/usenixsecurity20/presentation/cremers.

Dolev, D., Yao, A., 1983. On the security of public key protocols. IEEE Trans. Inf. Theory 29 (2), 198–208. https://doi.org/10.1109/TIT.1983.1056650

Gasser, O., Holz, R., Carle, G., 2014. A deeper understanding of SSH: results from internet-wide scans. In: 2014 IEEE Network Operations and Management Symposium (NOMS), pp. 1–9. https://doi.org/10.1109/NOMS.2014.6838249

Hagen, M. S., Lundqvist, E., Phu, A., Wang, Y., Strandberg, K., Schiller, E. M., 2025. Towards a Formal Verification of Secure Vehicle Software Updates. arXiv, *arXiv*:2511.15479 [cs.CR].

Holzmann, G.J., 1997. The model checker SPIN. IEEE Trans. Softw. Eng. 23 (5), 279–295. https://doi.org/10.1109/32.588521

Kirk, R., Nguyen, H.N., Bryans, J.W., Shaikh, S.A., Wartnaby, C., 2023. A formal framework for security testing of automotive over-the-air update systems. J. Log. Algeb. Methods Program. 130, 100812.

Lamport, L., 2002. Specifying Systems: The TLA + Language and Tools for Hardware and Software Engineers. Addison-Wesley.

Langiu, A., Boano, C.A., Schuß, M., Römer, K., 2019. Upkit: an open-source, portable, and lightweight update framework for constrained iot devices. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 2101–2112. https://doi.org/10.1109/ICDCS.2019.00207

Lonvick, C.M., Ylonen, T., 2006. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253. https://doi.org/10.17487/RFC4253

Lorch, R., Larraz, D., Tinelli, C., Chowdhury, O., 2024. A comprehensive, automated security analysis of the uptane automotive over-the-air update framework. In: The 27th International Symposium on Research in Attacks, Intrusions and Defenses, RAID. ACM, pp. 594–612.

Mahmood, S., Fouillade, A., Nguyen, H.N., Shaikh, S.A., 2020. A model-based security testing approach for automotive over-the-air updates. In: 13th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW. IEEE, pp. 6–13.

McGrew, D., Igoe, K., 2015. AES-GCM Authenticated Encryption in the Secure Real-time Transport Protocol (SRTP). RFC 7714. https://doi.org/10.17487/RFC7714

Meier, S., Schmidt, B., Cremers, C., Basin, D., 2013. The TAMARIN prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (Eds.), Computer Aided Verification. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 696–701.

Mukherjee, A., Gerdes, R., Chantem, T., 2021. Trusted verification of over-the-air (OTA) secure software updates on COTS embedded systems. In: Workshop on Automotive and Autonomous Vehicle Security (AutoSec). Vol. 2021, p. 25.

Narisawa, F., Asada, Y., Sobue, T., Yano, M., Sakanoue, O., Maeda, K., Saito, M., 2022. Vehicle electronic control units for autonomous driving in safety and comfort. Hitachi Rev. 71 (1), 78–83.

Pedroza, G., Idrees, M.S., Apvrille, L., Roudier, Y., 2011. A formal methodology applied to secure over-the-air automotive applications. In: 2011 IEEE Vehicular Technology Conference (VTC Fall), pp. 1–5. https://doi.org/10.1109/VETECF.2011.6093061

Ponsard, C., Darquennes, D., 2021. Towards formal security verification of over-the-air update protocol: requirements, survey and upkit case study. In: Proceedings of the 7th International Conference on Information Systems Security and Privacy, ICISSP. SCITEPRESS, pp. 800–808.

Rescorla, E., 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://doi.org/10.17487/RFC8446

Rosenstatter, T., Strandberg, K., Jolak, R., Scandariato, R., Olovsson, T., 2020. Remind: a framework for the resilient design of automotive systems. In: 2020 IEEE Secure Development (SecDev), pp. 81–95. https://doi.org/10.1109/SecDev45635.2020.00028

Scotiabank, 2024. Number of cars sold worldwide from 2010 to 2023, with a 2024 forecast. https://www.statista.com/statistics/200002/international-car-sales-since-1990/.

Strandberg, K., 2024. Private communication.

Strandberg, K., Arnljung, U., Olovsson, T., Oka, D.K., 2023. Secure vehicle software updates: requirements for a reference architecture. In: 2023 IEEE 97th Vehicular Technology Conference (VTC2023-Spring), pp. 1–7. https://doi.org/10.1109/VTC2023-Spring57618.2023.10199410

Strandberg, K., Oka, D.K., Olovsson, T., 2021a. UniSUF: a unified software update framework for vehicles utilizing isolation techniques and trusted execution environments. In: 19th Escar Europe : The World's Leading Automotive Cyber Security Conference (KonferenzverÖffentlichung), pp. 86–100. https://doi.org/10.13154/294-8353

Strandberg, K., Olovsson, T., Jonsson, E., 2018. Securing the connected car: a security-enhancement methodology. IEEE Veh. Technol. Mag. 13 (1), 56–65. https://doi.org/10.1109/MVT.2017.2758179

Strandberg, K., Rosenstatter, T., Jolak, R., Nowdehi, N., Olovsson, T., 2021b. Resilient shield: reinforcing the resilience of vehicles against security threats. In: 2021 IEEE 93rd Vehicular Technology Conference (VTC2021-Spring), pp. 1–7. https://doi.org/10.1109/VTC2021-Spring51267.2021.9449029

Van Adrichem, N. L.M., Lua, A.R., Wang, X., Wasif, M., Fatturrahman, F., Kuipers, F.A., 2014. DNSSEC Misconfigurations: how incorrectly configured security leads to unreachability. In: 2014 IEEE Joint Intelligence and Security Informatics Conference, pp. 9–16. https://doi.org/10.1109/JISIC.2014.12

Wang, J., 2024. Remote Offline Attestation for Seldomly Connected Vehicular Systems. Dept. of CSE, Chalmers University of Technology, Sweden. Master's thesis.

Wenzel, M., Paulson, L.C., Nipkow, T., 2008. The isabelle framework. In: Mohamed, O.A., Muñoz, C., Tahar, S. (Eds.), Theorem Proving in Higher Order Logics. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 33–38.