applied sciences

MDPI

Article

MT-SOTA: A Merkle-Tree-Based Approach for Secure Software Updates over the Air in Automotive Systems

Abir Bazzi [1,*], Adnan Shaout [1] and Di Ma [2]

[1] Department of Electrical and Computer Engineering, Dearborn, MI 48128, USA; shaout@umich.edu
[2] Department of Computer and Information Science, Dearborn, MI 48128, USA; dmadma@umich.edu
* Correspondence: aybazzi@umich.edu

Featured Application: Automotive Industry.

Abstract: The automotive industry has seen a dynamic transformation from traditional hardware-defined to software-defined architecture enabling higher levels of autonomy and connectivity, better safety and security features, as well as new in-vehicle experiences and richer functions through software and ongoing updates of both functional and safety-critical features. Service-oriented architecture plays a pivotal role in realizing software-defined vehicles and fostering new business models for OEMs. Such architecture evolution demands new development paradigms to address the increasing complexity of software. This is crucial to guarantee seamless software development, integration, and deployment—all the way from cloud or backend repositories to the vehicle. Additionally, it calls for enhanced collaboration between car manufacturers and suppliers. Simultaneously, it introduces challenges associated with the necessity for ongoing updates and support ensuring vehicles remain safe and up to date. Current approaches to software updates have primarily been implemented for traditional vehicle architectures, which mostly comprise specialized electronic control units (ECUs) designed for specific functions. These ECUs are programmed with a single comprehensive executable that is then flashed onto the ECU all at once. Different approaches should be considered for new software-based vehicle architectures and specifically for ECUs with multiple independent software packages. These packages should be updated independently and selectively for each ECU. Thus, we propose a new scheme for software updates based on a Merkle tree approach to cope with the complexity of the new software architecture while addressing safety and security requirements of real-time and resource-constrained embedded systems in the vehicle. The Merkle-tree-based software updates over the air (MT-SOTA) proposal enables secure updates for individual software clusters. These clusters are developed and integrated by diverse entities with varying release timelines. Our study demonstrates that the MT-SOTA scheme can enhance the speed of software update execution without significantly increasing the process overhead. Additionally, it offers necessary defense against potential cyberthreats. The results of the performed technical analysis and experiments of the MT-SOTA implementation are presented in this paper.

Keywords: cyber physical systems; digital signatures; distributed software development; Merkle tree; safety-critical systems; software over the air updates; software-defined vehicle

Citation: Bazzi, A.; Shaout, A.; Ma, D. MT-SOTA: A Merkle-Tree-Based Approach for Secure Software Updates over the Air in Automotive Systems. *Appl. Sci.* **2023**, *13*, 9397. https://doi.org/10.3390/app13169397

Academic Editors: Mashael Maashi and Majed Aborokbah

Received: 21 July 2023
Revised: 13 August 2023
Accepted: 16 August 2023
Published: 18 August 2023

1. Introduction

Software updates over the air (SOTA) have gained high interest in the automotive industry. An electronic control unit (ECU) can only run a new version of received software after empirically verifying that it has successfully received the entire correct image file from the repository. Software updates need to be applied across a distributed system of automotive devices, which can be designed and serviced by different suppliers. The original equipment manufacturer (OEM) must shift from the traditional vehicle architectures

toward the more flexible and scalable approach required for the next generation architecture. Software-defined vehicles are the next evolution of the automotive industry [1]. New automotive trends such as e-mobility, automated driving, and connectivity services are made possible mainly by software. Main features and operations will be enabled and managed by software. Software updates not only allow for fixes of software defects or bugs but also enable management of new and paid features provided as a service throughout the entire life cycle. The change in business model is made possible by the separation of hardware and software. Significant variation is expected across the vehicle's various software. ECU software is becoming more similar to mobile software when different packages are provided separately and need to be integrated together in the vehicle. Virtualization, cloud-based approaches, AUTomotive Open System ARchitecture (AUTOSAR), and adaptive and flexible classic multi-layer software architecture [2,3] are examples which split today's binary image into several software binary images which can be independently developed, integrated, tested, released, and programmed on the target ECU (Figure 1).

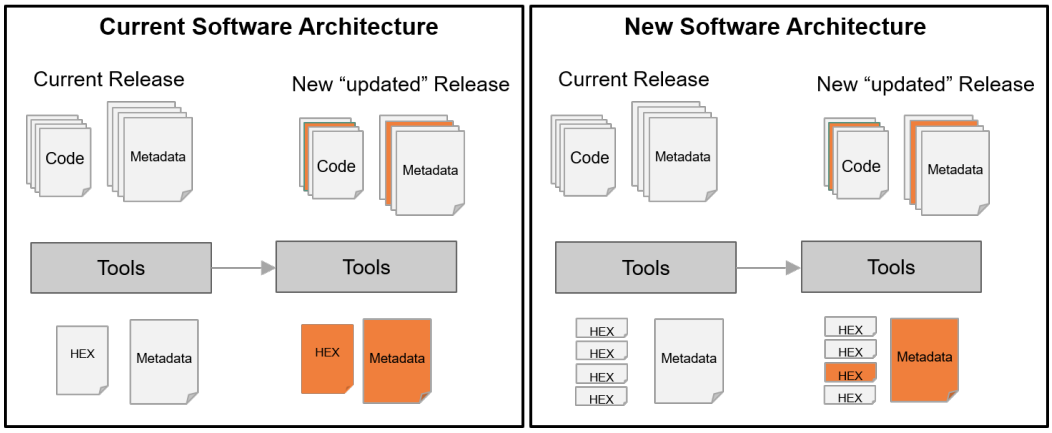

Figure 1. ECU binary images are divided into multiple software clusters, each of which can be developed, tested, released, and programmed onto the target ECU independently.

Digital signatures [4] have been proven to be the most fundamental and useful strategies for securing software updates in the automotive industry. Existing solutions [5] range from full over-the-air update (OTA) frameworks to approaches based on online/offline, symmetric/asymmetric algorithms, full and partial verification, hardware and software implementation, and many other approaches. These solutions have mainly been applied to classic ECUs with single image application software (one image binary per ECU). With newer ECU architectures and frameworks, OEMs and suppliers face the challenge to enable on demand a flexible deployment of portable, consistent and interoperable software (multiple of software clusters) in the ECU throughout the entire lifecycle of the vehicle. Software clusters and data are selectively updated to add or remove software functionality, enable paid services, implement a security fix, or improve performance. OEMs have also to support both legacy ECUs (e.g., classic AUTOSAR architecture) in addition to the high-performance computing ECUs (e.g., adaptive AUTOSAR architecture). Among the existing approaches, we did not find any related work to verify at once the authenticity, integrity, compatibility, and harmonization of a single binary image with the remaining software images using the single authentication signature appended to the software code image. Different approaches should be considered for ECUs with multiple independent binary executions. Thus, our Merkle-tree-based software updates over the air concept (MT-SOTA) is proposed to complement existing approaches to address new software architectures in the automotive industry. The main contributions of this work can be summarized as the following:

- An overview of the process for software updates over the air and the current solutions employed in the automotive industry.

- An overview of the new vehicle architectures and the challenges in dynamically updating selective software clusters.
- The design of a novel hash-based scheme that improves on the existing literature, efficiently stores integrity information for ECU software images, and addresses the various architectures of the software and vehicle markets. Our approach allows us to maintain one key security pair for signing and the same signature size to obtain the target security level. We define our scheme to accommodate interactions and compatibility between distinct software images, whether logically or physically interconnected within the vehicle.
- The development of the suggested approach prototype and demonstration that our strategy fulfills the specified requirements.

The remainder of this paper is organized as follows. Section 2 provides a summary of related work. Section 3 defines our system and the adversary model. Section 4 describes our scheme, including Merkle tree details and defining the interactions between entities. In Section 5, we analyze the security of our approach. Section 6 presents our performance evaluation and results. Section 7 discusses different methods to optimize the MT-SOTA approach to better address different use-cases. Finally, we conclude the paper in the last section.

2. Related Work

Consumer electronics, commercial aviation, and medical devices are adjacent industries that have many hardware devices containing loadable firmware components and have the task to track and verify the installation of the software updates across their fleets. In the NHTSA report [6], the authors present a literature review of the state-of-the-art of software updates in the industries related to the automotive industry and conclude that there are mainly two common existing defense mechanisms: trusted content distribution networks and digitally signed software updates.

The Internet of Things (IoT) devices have similar resource constraints (e.g., energy, computation, and storage capacity) to the automotive ECUs [7,8]. The Trusted Computing Group (TCG) has released a reference report [9] describing how secure software and firmware update for embedded systems can be performed using trusted computing technologies. The Internet Engineering Task Force (IETF) Software Updates for Internet of Things (SUIT) working group is actively working on specifying a software update architecture for IoT devices [10].

Numerous mechanisms have been considered, standardized, and adopted by the automotive industry for software updates over the air in the vehicle. Uptane [11] and other proprietary frameworks (e.g., [12]) have been introduced to manage the software update process. In addition, different security mechanisms have been used to secure the software updates by means of hashing mechanisms [13,14] as well as symmetric and asymmetric cryptographic mechanisms [15,16]. SOTA approaches have been extended by using compression [17] and delta flashing [18] to reduce the network utilization and bandwidth used for software updates. Blockchain-based approaches [19,20] have also been considered for performing software updates for vehicles to guarantee the authenticity and integrity of new updates. The implementation of blockchain for automotive software updates is promising for the new electrical/electronic (E/E) architecture; however, it requires considerable efforts for OEMs as well as high resource consumption at the ECU level. We categorize these SOTA solutions in the automotive industry based on the security mechanisms used by the researchers and summarize them in Table 1. In a typical new-generation vehicle architecture, the topology of an ECU has several software entities (known as software clusters). Software updates of such clusters need to be applied across a distributed system of automotive ECUs that are designed and serviced by different suppliers. Hence, OEMs have to shift from the traditional software update approaches toward a more flexible and scalable approach required for such software-defined architecture.

Table 1. Common software OTA update approaches used in automotive.

Approach	Description	Pros	Cons
Message-digest/ Hash-based [13,14]	Software package is hashed, and result is compared with a trusted known-good value. Different types of hash chain have been proposed to increase security.	Simple to implement. One-way only. No security keys used. Can be software or hardware implemented. Memory efficient (typically 256 bits).	Inefficient when there are many collisions. Known-good value must be read/write protected.
Symmetric-based [15]	Software package is encrypted with a common key and then decrypted at the receiver side with the same key. Another approach is to generate a message authentication code (MAC) using the common key and verify the same MAC code at reception.	Fast cryptography. Easily implemented (AES accelerators are widely available in microcontrollers.) Small key lengths. Example of symmetric keys used: AES (128/192/256 bits), CHACHA20 (128+ bits). Modes of sharing: Diffie-Hellman and physically.	Key security (key must be read/write protected). Managing keys is a challenge. Complexity consistent regardless of number of users or frequency of use.
Asymmetric-based [16]	Software package is encrypted with a private key and then decrypted at the receiver side with the corresponding public key. Another approach is to generate a code signature by hashing the software package and then encrypt the hash value using a private key (e.g., RSA 2048, Ed25519).	Key flexibility. Relatively long key lengths. Example of asymmetric keys used: RSA-2048 (256 bytes), RSA-7096 (512 bytes), ECDSA (secp521r1:1042-bit for public key, 132 bytes for signature), EdDSA (32/57 bytes for keys, 64/114 bytes for signature). Modes of sharing: PKI.	Key security (key must be read/write protected). Slower cryptography than symmetric schemes. Complex to implement for some algorithms (e.g., ECDSA and EdDSA). Complexity grows with number of users and frequency of use.
Blockchain-based [19,20]	Distributed peer-to-peer database. Data are saved on each node, transactions are saved into blocks. Cryptography is based on hashing and digital signatures (using asymmetric cryptography e.g., ECDSA.)	Immutable (permanent and tamper-proof). Decentralized control. Redundant decentralized copy on every node of the network. Consensus-based, creates trust and integrity in an untrusted environment	New to Automotive. High memory consumption in the ECU.
Secure Software Repository Framework [11,12]	Framework introduced to allow repositories to build different security models that provide varying degrees of security and usability (e.g., Uptane framework).	Separation of trust. Explicit and implicit revocation of keys. Flexible implementation (full or partial verification, asymmetric or symmetric keys, encrypted or unencrypted update image, online or offline keys.	OEM has to setup and maintain the repositories.

3. System Model

A typical implementation of a software update over-the-air architecture in a vehicle is shown in Figure 2. The OTA master residing in the ECU, usually the Telematics or Gateway modules within today's vehicles, is capable of connecting wirelessly to the repositories. The OTA master receives the software image from the repository and distributes it to the target ECU in the vehicle. A simplified view is shown in Figure 3 for the SOTA process carried out in several successive steps. There are four main actors in the system:

- OEM Repository "OemR": This repository is responsible for signing software images. Given the role of OemR in common SOTA processes, OemR contains all of the information about images to be installed on ECUs, generally using an OEM inventory database containing information on vehicles, ECUs, software images, etc. Each soft-

ware image must have a unique identifier linked to an ECU identifier that specifies the ECU installing the image. Our scheme extends the OemR with the capability of creating and maintaining the Merkle trees. In our study, we focus on a scenario in which software clusters reside in the same ECU. Thus, the OemR has to create a Merkle tree for each ECU in the vehicle. OemR has to store the hash value of each software cluster in addition to existing image metadata such as image name, version, size, hashing function used, and any additional proprietary OEM information, such as the download URL for the image file located in the image repository. OemR is trusted and responsible for protecting the harmonization among software clusters as well as the integrity of each software image to prevent any tampering.

- Image repository "ImgR": This repository contains the software binary images to install as well as the signatures (authentication tags) for these images. These signatures are generated by OemR.
- OTA Master "OtaM": This entity is responsible for receiving software images in the vehicle and distributing them to other ECUs in the vehicle. OtaM is capable of connecting to backend repositories through wireless channels, such as Wi-Fi or cellular communications.
- End-Target ECU "EcuX": This is the ECU in the vehicle where software image is installed or updated. Upon reception of new image software, EcuX validates the integrity and authentication of the image as well as proves complete possession and compatibility of the software image in question. EcuX can be directly connected to OtaM through in-vehicle communication channels, e.g., CAN or Ethernet, or can be indirectly connected through a gateway or domain controller. In case the software image is intended for the OTA Master itself, OtaM will play the role of EcuX.

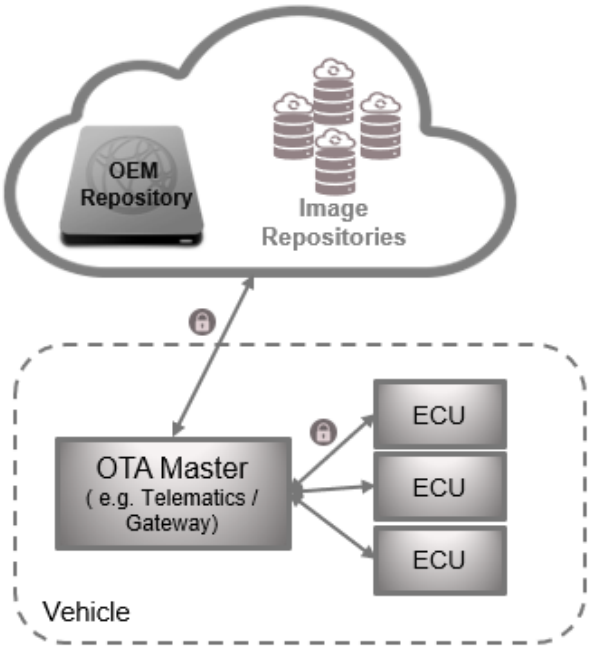

Figure 2. OTA System Architecture: The OEM and image repositories contain all the information about the software update process and the software images to be wirelessly transferred to the vehicle. The OTA Master manages the update process for all ECUs within the vehicle. It receives the images from the repositories and transfers the image to the target ECU.

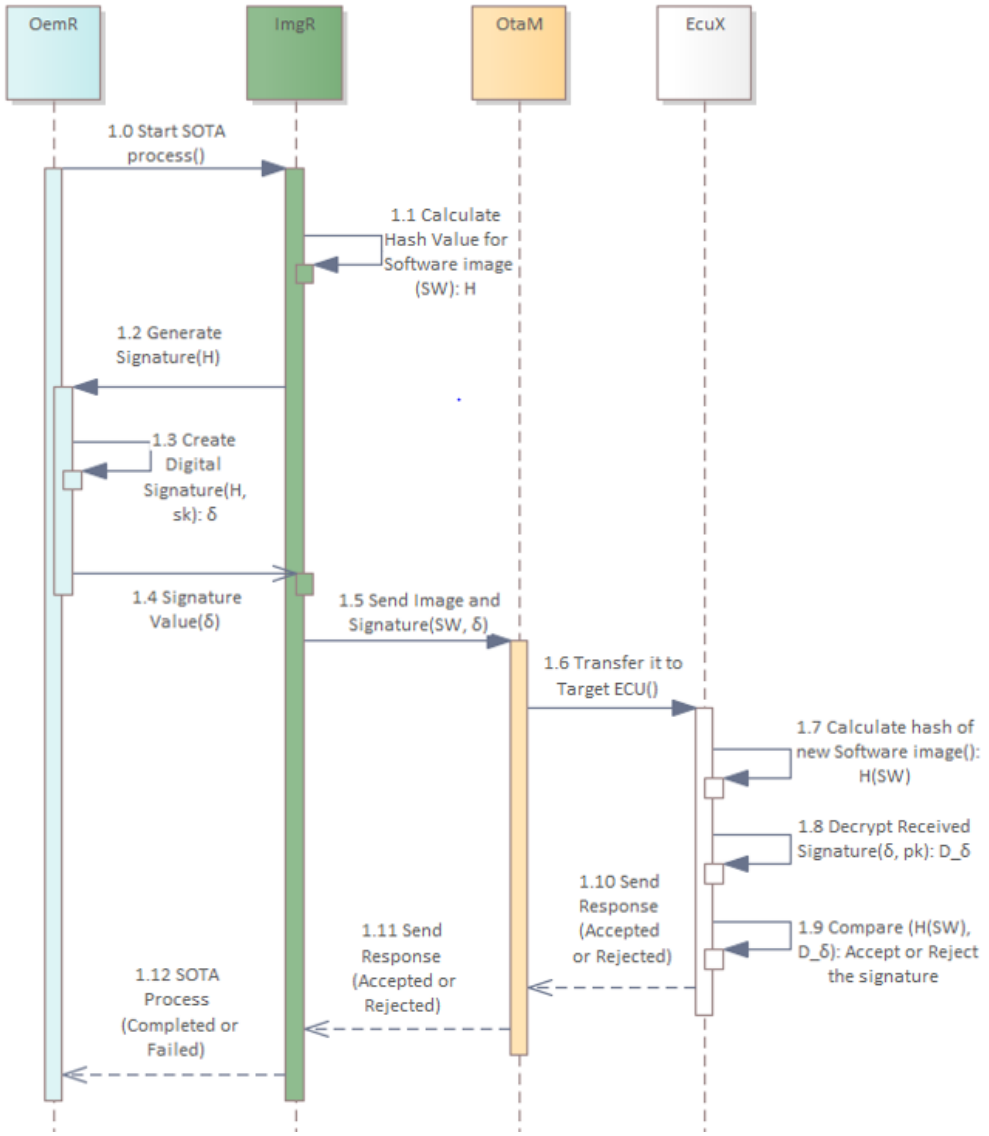

Figure 3. Software updates using a digital signature scheme: The signature generation process is used by the repositories to generate the signature on the software images, and the signature verification process is used by the ECUs in the vehicle to verify the received images.

3.1. Adversary Model

Before discussing the details of the system, we want to introduce the adversary model and attack strategies considered for our system. We assume a powerful adversary, ***A***, who has managed to find a way to get hold of ImgR, OtaM, or EcuX to modify the code of a software cluster image to control the ECU in the vehicle or even the vehicle itself, perform reverse engineering of the software image, or read the contents of the ECU software to discover confidential or intellectual property information. The adversary also knows the details of the software update algorithm and how the software is organized within an ECU, so it aims to modify one software entity or revert to an old software version. Due to the storage of the keys in secure protected memory, the attacker can only alter the software image itself or part of it but cannot modify any signature values generated by OemR. Our threat model encompasses attacks not only on data integrity and authentication but also on the compatibility of the complete software. Such attacks are mainly categorized into spoofing, splicing, and replay attacks. Spoofing attacks correspond to ***A*** being able to replace an existing software image with a malicious code. Splicing attacks or mix-and-match attacks correspond to ***A*** being able to swap software between different ECUs or

vehicles. Replay attacks correspond to $\boldsymbol{A}$ being able to revert a software image to an older image.

We assume that adversary $\boldsymbol{A}$ can obtain direct access to the ECU or the image repository and can repeat the process of modifying the image as many times as it wants until it can create an image that can evade the detection of software update verification. Any compromised applications can directly influence other applications in the ECU as well as other ECUs in the vehicle. Thus, the weakest image repository and ECU that can be attacked by $\boldsymbol{A}$ determines the security level of all applications running in the ECU and vehicle. Any of the actors having access to the image software can potentially be taken up by malicious actors in the supply chain and are prone to man-in-the middle attacks. On the one hand, adversary $\boldsymbol{A}$ can intercept and modify traffic between the vehicle and image software repositories or inside the vehicle between the ECUs or directly accessing and compromising an ECU. In contrast, $\boldsymbol{A}$ can compromise and control the image repository.

3.2. Assumptions

Our basic assumptions for the system are:

- A1: ECUs are able to perform cryptographic digital signature operations and key management throughout the entire lifecycle, including key establishment, storage, and usage. By using existing methods such as public key infrastructure (PKI) [21], OemR owns a private and public key pair for each EcuX. OemR uses the private key for signature generation and EcuX uses the public key for signature verification. We assume that the public keys are distributed and installed in EcuX before any software updates can be performed. The public keys shared between the OemR and EcuX are not guessable and cannot be accessed by the adversary.
- A2: The communication between OemR and ImgR is secure. We assume that they are able to connect and communicate whenever the ImgR has a new software image version to be deployed to the vehicle or whenever OemR finds out that the vehicle does not have the latest software images.
- A3: The hash function used is not breakable, meeting the requirements of preimage-resistant, second-preimage-resistant, and collision-resistant for a hash function [22]. In other words, for a hash value H(SW) calculated for a given image software SW, it is infeasible to find the software image SW (preimage-resistant feature). It is also infeasible to find another image SW′ such that Hash(SW) = Hash(SW′) and SW $\neq$ SW′ (second-preimage-resistant feature), as it is also infeasible to find any two images SW and SW′ such that Hash(SW) = Hash(SW′) and SW $\neq$ SW′.
- A4: The chosen signature scheme is strongly unforgeable. Strong unforgeability assures the adversary cannot generate a new signature for a previously signed software image. In other words, assume an adversary obtains the software image and signature pair (SW, δ); the signature is strongly unforgeable if the adversary cannot generate a new signature δ' for the same software image SW.
- A5: The OemR and EcuX share the same information about the Merkle tree (e.g., max number of software images, left-right bottom-up approach, static or dynamic). Otherwise, EcuX, as the verifier, cannot validate the correctness of the software image.

3.3. Requirements

The E/E architecture is going through fundamental changes. Driven by software-defined vehicle architecture, the vehicle is evolving into ECUs in which new functions and features are implemented and deployed primarily through software. The adaptation of the vehicle-centralized, zone-oriented E/E architecture takes place as an iterative process, introduced gradually in the vehicle. The E/E architecture transitions from being a decentralized system, connected by a central gateway in which functions are running on dedicated ECUs with software binding to hardware, towards more centralized systems with dedicated domain controllers handling a set of functions related to a specific area or domain before evolving into zonal controllers arranging functions according to their

positions in the vehicle [23]. Throughout this transition, significant variation is expected across the various vehicle software. As shown in Figure 4, the software evolution goes in parallel with hardware evolution. The majority of software in decentralized architecture is combined with a function-specific ECU in a single package and fully controlled by OEMs. A typical new-generation vehicle likely has an architecture composed of five or more domains. The functions are abstracted from discrete ECUs and centered in software in fewer hardware modules. Domain-specific functions are optimized for their application and distributed flexibly to any cross-domain vehicle controllers and no longer tied to a particular ECU. High performance computing ECUs and service-based approaches (e.g., service-oriented communication) are essential for the zonal architecture. Software services can be subscribed across application and ECU boundaries. Signal-based real-time-controlled software continues to be used for sensors and control actuators. Due to the disruption in the E/E topology, OEMs cannot change the complete vehicle setup quickly and the transition will take place at different rates depending on OEM strategies. Therefore, legacy classical ECUs will remain part of the vehicle, and a mixed architecture will be implemented in tomorrow's vehicles. An application software entity can be placed either in a central computing controller, a zonal controller, a domain controller, or a persistent specialized ECU (e.g., actuator, sensor).

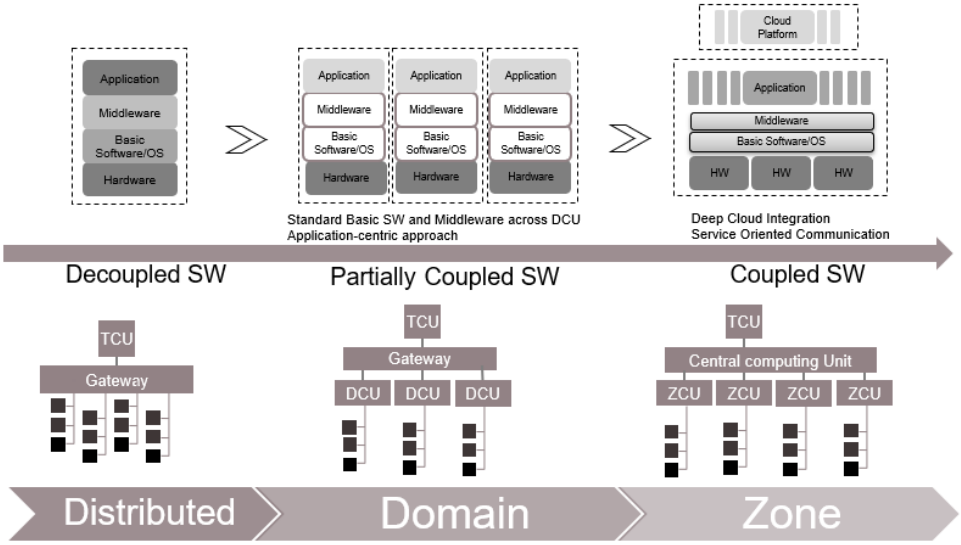

Figure 4. Evolution of the E/E and software architecture from a distributed approach to a hierarchical, software-centric architecture with centralized computers and domain or zone controllers.

The contents of software within an ECU can differ hugely due to many factors, such as segment (e.g., ADAS, Infotainment, Chassis), powertrain type, level of driving automation (SAE AV level), etc. The software functionalities range from real-time safety systems to interactive apps. The vehicles are expanded with cloud and backend solutions as well as services, enabling the vehicle to access and participate in mobility ecosystems to guarantee high availability and fast delivery of the updates. OEMs and suppliers have an opportunity to grow their business by adding on the fly certain software capabilities, e.g., functional safety, artificial intelligence, security capabilities. Optimization can be implemented globally across all vehicle platforms, as well as locally, limited to a particular vehicle model or market. Small, large, low-end, high-end vehicles are usually equipped with ECUs with different sets of software. Such flexibility and scalability are mainly achieved through software modularity and over-the-air update capabilities. Software modules are deployed based on the vehicle models or markets as well as relevant configurations. While variant handling and configuration options are traditionally performed at the time of vehicle purchasing, they will increase and be activated at a later point in time depending on available features

and paid services. The most promising feature of the future architecture is the ability to add new functions to the vehicle by simply adding a software driver, very similar to the case of adding a new application to a mobile phone. Software is transforming capabilities in the vehicle but also creating development challenges for OEMs. OEMs have to manage the software complexity introduced by new and advanced functionalities; distinguish which software is running into an ECU regardless of the vehicle models, markets, and activation states; as well as provide dynamic software updates at any time.

To address such automotive demands, we aim to design a software update scheme with the following requirements:

- **Requirement 1**: The scheme shall allow detection of any tampering with the software image.
- **Requirement 2**: The scheme shall allow detection of any incompatibilities between the logically or physically dependent software images in the vehicle.
- **Requirement 3**: The scheme shall work with any frameworks or platforms supporting multiple binaries programming using digital signatures. The scheme shall support different types of hash algorithms as well as different signature mechanisms.
- **Requirement 4**: The scheme shall allow efficient addition or removal of software clusters. The vehicle should be able to receive requests to add a new software cluster or remove an existing one. Thus, the scheme shall enable efficient generation and secure storage of the signature.
- **Requirement 5**: As ECUs might be heterogeneous in terms of hardware, computation capabilities, and memory storage, the scheme shall make it possible to induce insignificant overheads in the lightweight ECUs with limited memory and computation resources on the installation of additional images.
- **Requirement 6**: The scheme may be used for intra-ECU and inter-ECU distribution of software clusters. Dependant software clusters can be located in the same physical ECU. However, it is possible they are physically running in different ECUs. In this case, it should be possible for the ECU to validate the signature even it doesn't have access to complete software clusters. This enables flexibility to move a software cluster from one ECU to another ECU while keeping the dependency among the software clusters.
- **Requirement 7**: The scheme shall use one key pair per ECU to generate and verify the signature regardless of how many software clusters are used per ECU.

4. Scheme Design

As explained in the previous section, functions are hard-coded and tied to the hardware for specialized ECUs and domain controllers, while they are implemented as software in domain-agnostic hardware for zonal controllers and central computers. Our approach aims to cover software over the air updates for any embedded systems in today's, tomorrow's, and future vehicles. We can categorize the software architectures into four main categories:

- Case 1: A single software package in an ECU (e.g., traditional classic AUTOSAR platform, software is decomposed into software blocks).
- Case 2: Multiple software packages in an ECU (e.g., adaptive AUTOSAR, flexible classic AUTOSAR platform).
- Case 3: Multiple software packages in different ECUs (e.g., zone controllers and connected ECUs).
- Case 4: Multiple software packages on-board and off-board (e.g., AI-based cloud applications, backend user applications).

Our approach is built on hash-based signatures [24] due to their advantages, specifically considered to be quantum-resistant and future proof. They are reasonably fast and result in a small signature size. Its security assumption is minimal as it relies only on the criteria of a hash function and unforgeability of a digital signature as stated in Section 5. It is parameterized to fit the need for specific applications and different use-cases. We choose hash tree versus hash table to cover use-cases where software is distributed or used

differently across vehicles. Most importantly, ECUs can verify complete software integrity and compatibility without requiring the entire software to be present at the ECU level. Rather, only the tree root and essential branch stubs for absent nodes are necessary.

Our SOTA scheme is based on a Merkle tree approach [25,26]. We present a theoretical explanation and analysis for our approach, irrespective of the specific software use-cases employed. Figure 5 outlines the workflow at the backend repositories:

1. The image software resides in the image repository (ImgR).
2. The ImgR provides the correspondent hash value of the software image with additional manifest data to the OEM repository (OemR) to generate the signature.
3. The OemR has all information about the vehicle, ECUs, and software clusters. The OemR checks which ECU the software cluster belongs to and uses the corresponding Merkle tree to calculate the root.
4. Using the private key assigned to EcuX, the Merkle tree root is signed to create the signature of the software image.
5. The OemR provides the generated signature to ImgR.
6. The OemR constructs the vehicle package containing the manifest data for the software images to be installed, and then sends it to the OtaM in the vehicle.
7. The ImgR sends the software image appended with the signature.
8. Once EcuX receives the software image, it updates the Merkle tree with the hash of the received software image, and then it determines whether the root tree is the same as the one received from the ImgR (after decrypting the received value with the public key of EcuX). Once verified, EcuX stores the value of updated node and root node in the Merkle tree in a secure location.

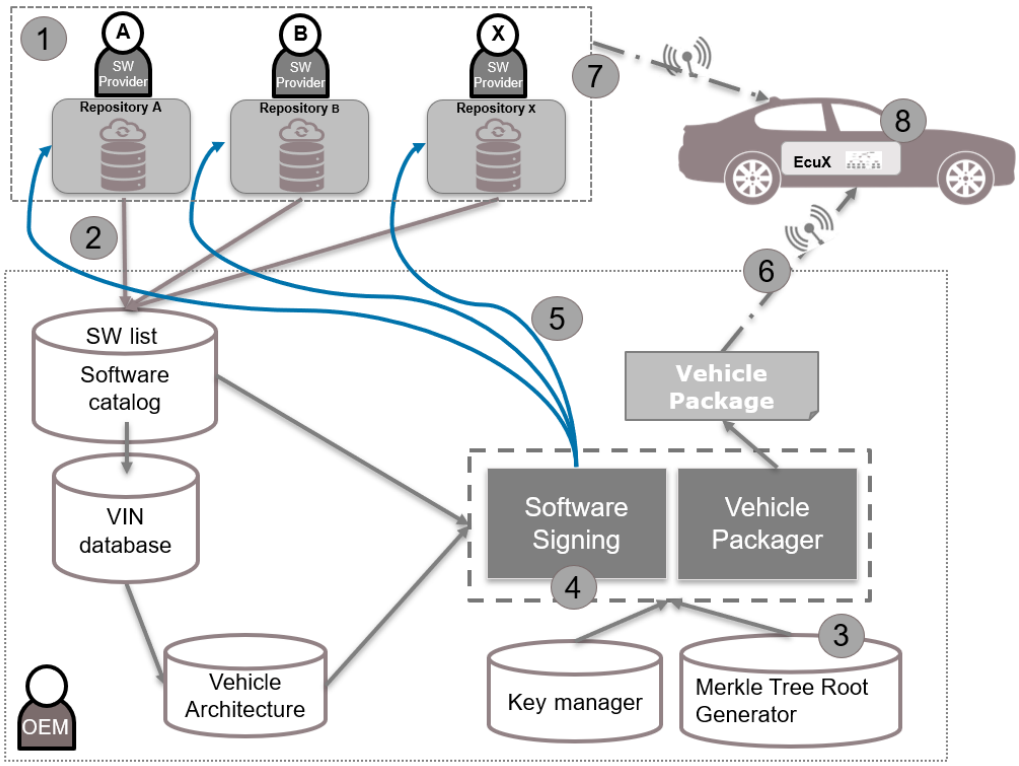

Figure 5. Software update over the air workflow, extended with Merkle tree scheme to safely and securely update software in the vehicle.

The Merkle-tree-based scheme is designed to work properly regardless of the quantity of verification or number of software components involved. MT-SOTA does not specify full implementation details on purpose. It describes the main necessary entities and methodology for the concept and leaves it up to OEMs to make their own technological

choices, for example, which hash algorithm to use or which public key cryptographic algorithm is used by the keys for signing the Merkle tree root (such as RSA or ECDSA [4]). When the ECU has access to existing software images installed on it, there is no need to send anything beyond the signature. In this case, the authentication path, the hash values of the neighbors of the nodes on the path from the leaf to the root, is also not transmitted. Thus, the communication cost is negligible. We use the following notations in our MT-SOTA scheme:

- SW = $SW_0, \ldots, SW_{N-1}$: SW_i is one of the software cluster images to be updated in an ECU.
- N is the number of software clusters in an ECU.
- N_{max} is the max number of software clusters that can reside in an ECU throughout the lifetime of the ECU.
- $S_i = \|SW_i\|$ is the size of the software cluster SW_i in bytes.
- H_i is the hash value of the software cluster SW_i image. $\|Hi\|$ is the size of H_i in bytes.
- HT: is the Merkle tree height. The height at the root node is equal to $\log_2(N)$.
- SW_M is the modified SW cluster, $SW_M \in SW$.
- MT: is the Merkle tree structure, MT[i][j] is the node at height i and index j.

The MT-SOTA scheme consists of two phases. At the OemR side, there are the setup and generation phases. At the EcuX side, there are the setup and verification phases. The setup phase of the Merkle tree is similar between OemR and EcuX. The setup phase consists of creating a full binary Merkle tree using the hashes of all installed software clusters. The tree needs to accommodate for N_{max} nodes that might be deployed throughout the ECU lifecycle. While a Merkle tree can be essentially infinite, we can use only a fixed number of levels due to embedded systems constraints. Our Merkle tree construction is a bottom-up approach, with height or level equal to HT, and can have up to 2^{HT} leaf nodes. Figure 6 shows the binary Merkle tree where the hashes of the software images are used for the leaves of the tree. The interior nodes, called non-leaf nodes, are constructed by hashing the child nodes. In addition to the tree information, there is additional management information that needs to be saved. We summarize the setup phase of the Merkle tree in Algorithm 1 as follows:

1. The first set of hash results of the software images SW_0 to SW_{N-1}, which are the H_0 to H_{N-1}, represents leaf nodes in the hash tree at height 0.
2. These nodes are combined in pairs to provide the first combined hash nodes. The value of the parent node is the hash of the concatenation of two child node hashes (left $\|$ right), where concatenation is represented by $\|$.
3. The hash results are further combined in pairs to an intermediate level hash results. These intermediate level hash results are repeatedly paired and combined until only a single combined hash result remains. This is the Merkle tree root.

During runtime, ImgR has a new software image update. Prior to downloading this software update to the vehicle, ImgR has to request OemR to generate the signature in the generation phase. Each image software is assigned a unique identifier which is used to locate the leaf nodes in the Merkle tree corresponding to EcuX. As shown in Figure 7, OemR inserts the new hash value in the leaf node, calculates the new hash of the parent hash for this updated leaf node and its sibling, and then the parent of the new hash value and its sibling has to be hashed. This will be repeated until we get to the tree root. This root is signed by the private key of EcuX to generate the signature. At the vehicle side, once EcuX receives a new software cluster update, it has to verify the signature to validate the received image. Depending on what the EcuX should perform, the corresponding verification should be performed. There are three operations that can be performed:

1. Update an existing software cluster;
2. Add a new software cluster;
3. Remove an existing software cluster.

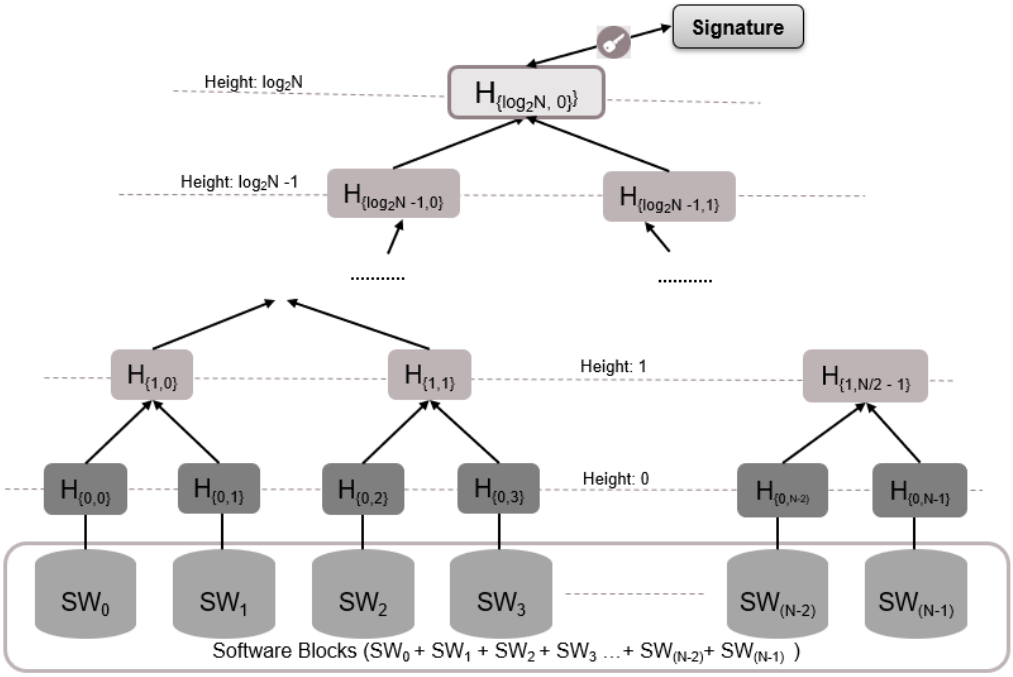

Figure 6. Merkle Tree of ECU composed of N software entities.

Algorithm 1 Merkle Tree Creation

INPUT: List of software ID(SW_i), Address (Memory location), and Size(S_i) of each software cluster
OUTPUT: Merkle Tree Root
FUNCTION: **BuildMerkleTree**

```
for i = 1 to N_max
    if (SW_i exist)
    Calculate H_i and save as a leaf node in tree
end
Calculate Merkle tree contents (MT[i][j]):
 (j is index of node in i^th level of the Merkle tree.
 i goes from 1 to HT, and j goes from 0 to (2^(HT−i) − 1) which is
 the number of nodes at the i^th level)
for i = 1 to Height of the tree (HT)
   for j = 0 to number of nodes at the current level
      Calculate Hash of the child nodes (e.g., MT[i][j] = hash(MT[i−1][2j], MT[i−1][2j+1]))
   end
end
```

Figure 8 summarizes the steps performed in the ECU to verify the signature. Verification is a bottom-up approach where it starts from the leaf node of the received software cluster (SW_M) and goes through the proof path to obtain the top root of the tree. Updating the Merkle tree follows the same method at OemR and EcuX once the leaf nodes are updated with the new hash values. The main difference is that OemR obtains the hash value of the software image from ImgR as OemR does not have access to the software image itself, while EcuX calculates the hash value of the software images if the software cluster belongs to it. Otherwise, it retrieves it from OemR.

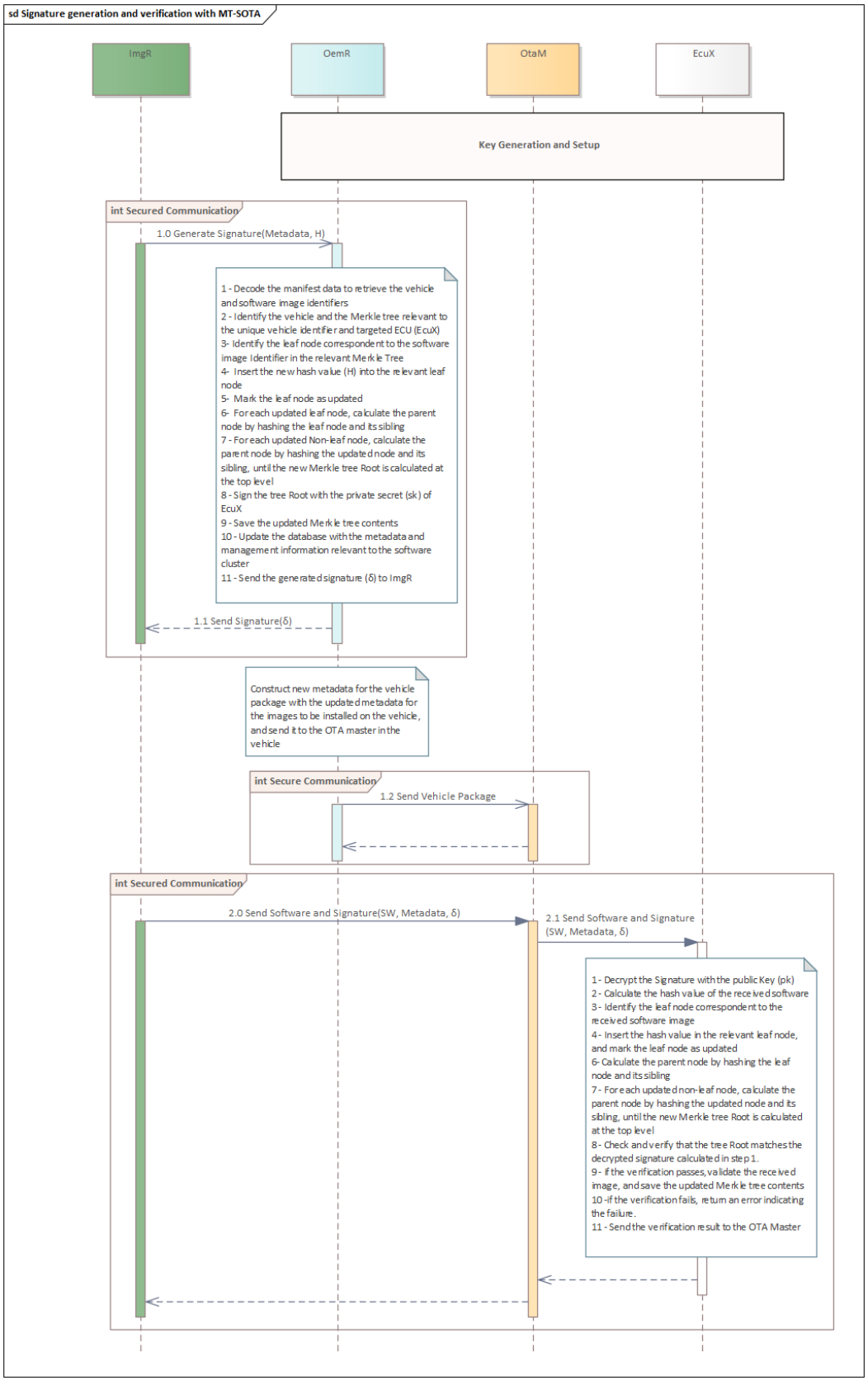

Figure 7. Signature Generation and Verification with MT-SOTA.

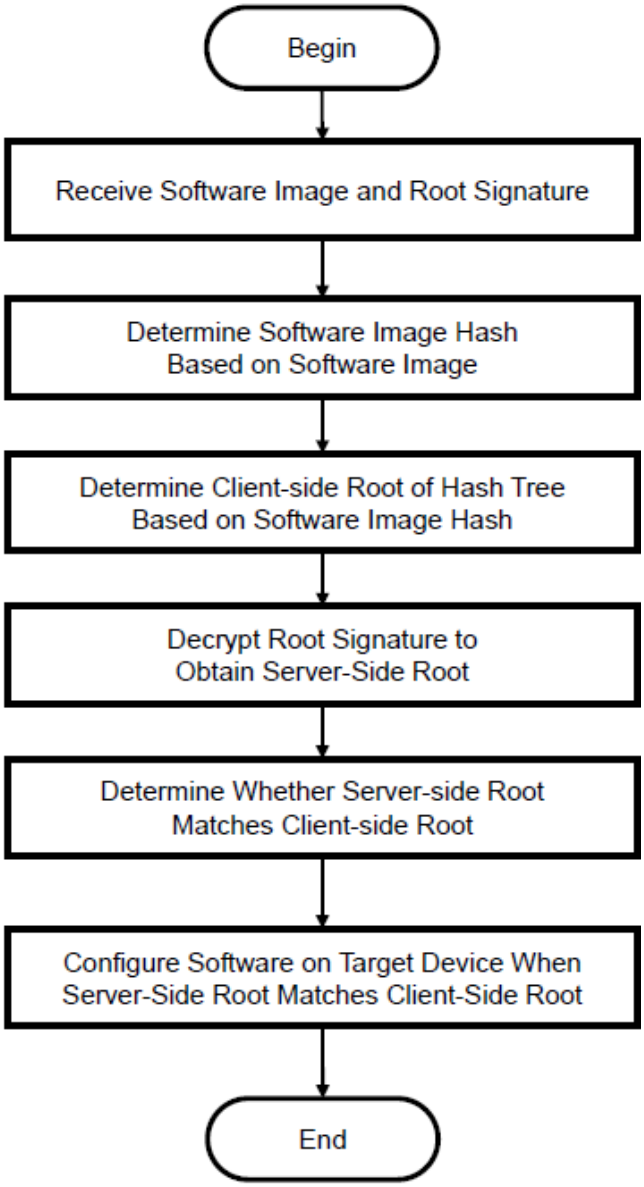

Figure 8. Verification process at the target ECU through use of the Merkle (hash) tree: a received root signature is decrypted to obtain a server-side root that is compared to a client-side root calculated at the target ECU based on an updated hash tree incorporating the hash of the received software.

Figure 9 shows how a Merkle tree is modified to add or update a software cluster. When EcuX receives a new software image SW_M, it first has to perform hashing operation on SW_M, then the hash value H_M is inserted into the corresponding leaf node in the tree, and the node is marked as updated. Next, the parent hash for this updated node and its sibling is calculated. Then, it needs to find the sibling node of its parent node and calculates their hash. This process is iterated until reaching the tree root. Mathematically, this can be represented as follows:

Let j be the position of the node to be verified; then,

if j is even

$$\mathrm{MT[i][j/2]} = \mathrm{Hash(MT[i-1][j], } MT\mathrm{[i-1][j+1])}$$

if j is odd

$$\mathrm{MT[i][(j-1)/2]} = \mathrm{Hash(MT[i-1][j-1], MT[i-1][j])}$$

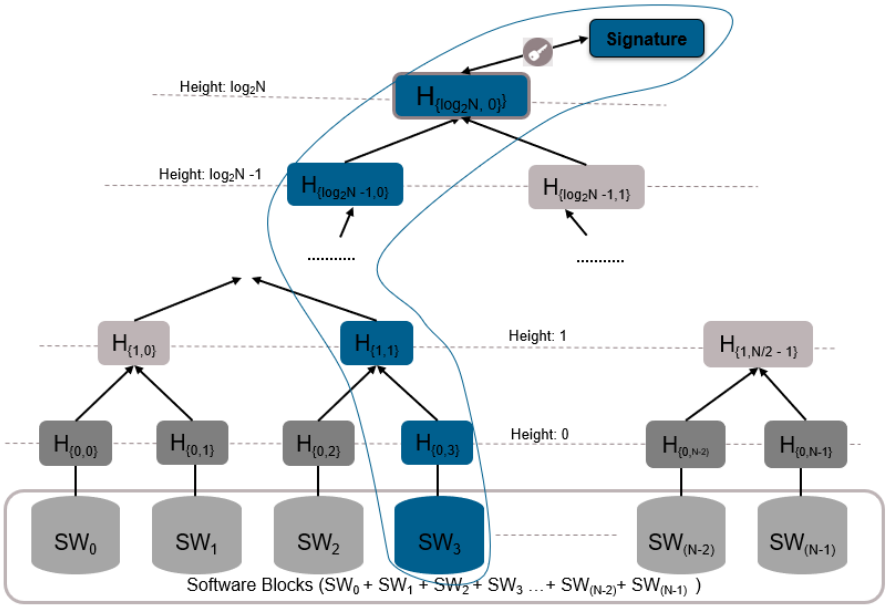

Figure 9. Visual view of Merkle tree modification: update of a software cluster.

When EcuX has to remove a software image, it first identifies the leaf node associated with the removed software image and designates it as invalid. The hash of its parent is then updated to match the value of the sibling node of the removed leaf node. Subsequently, it computes the hashes for all intermediary levels until reaching the tree root. Figure 10 shows how a Merkle tree is modified when a software cluster is removed.

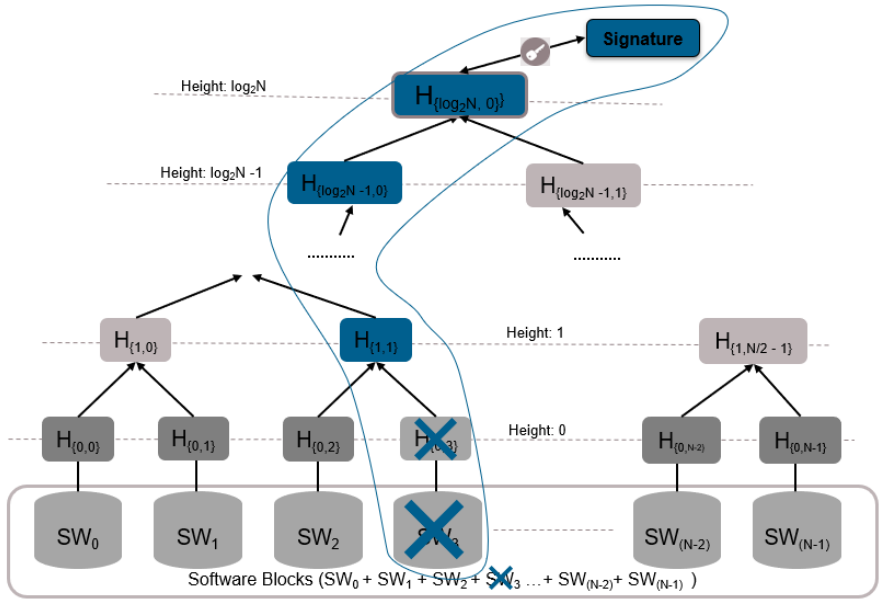

Figure 10. Visual view of Merkle tree modification: removal of software cluster.

Regardless of the verification types EcuX has to perform, the calculated Merkle tree root is compared with the decrypted received value. If they match, the integrity, authentication, and compatibility of the updated software cluster is verified. Otherwise, EcuX should indicate a failure error and report it to OtaM, and then to OemR. The path composed of all modified nodes is called the proof path of the software cluster node. For a Merkle tree of height HT, the proof path includes HT nodes to obtain the root. Thus, the higher the tree level is, the longer the proof path is.

The storage size of the Merkle tree depends on the hash output size. The Merkle tree height increases exponentially with the number of nodes. The higher the tree is, the more memory storage it requires. The MT-SOTA scheme has the capability to retain all nodes in the tree, encompassing both leaf and non-leaf nodes. This expedites the signature verification procedure by eliminating the necessity to recompute values for unaffected nodes within the tree. It is possible to store only the leaf nodes to save on memory storage at the cost of increased computation because the values of all non-leaf nodes have to be calculated. The Merkle tree needs to accommodate for $(2^{HT+1}-1)$ nodes. Thus, the storage of the tree (in bytes) is equal to $(2^{HT+1}-1) \times \|H_i\|$. Assuming that SHA-2-256 is used as the hash algorithm, each node requires 32 bytes for the hash value storage. Table 2 shows the number of bytes used to store the complete Merkle tree based on the number of software cluster nodes when using SHA-2-256.

Table 2. Merkle tree storage size based on number of used software clusters.

Height	Max Number of Node	Tree Storage in Bytes
1	2	96
2	4	224
3	8	480
4	16	992
5	32	2016
6	64	4064
7	128	8160

5. Security Analysis

A large amount of software and abstraction can make the vehicle susceptible to cyber-attacks. Thus, MT-SOTA security remains a significant requirement for software updates over the air. As explained previously, the fundamental security primitive of the MT-SOTA is based on the security of both the signature scheme and hash function. The digital signature scheme should be strongly unforgeable. The cryptographic hash functions should satisfy the three criteria of preimage resistance, second-preimage resistance, and collision resistance. The security advantage of our proposed Merkle-tree-based digital signature consists of the insertion of another layer of security using additional hash steps to generate the root.

Fact1: Security of Merkle-Tree-Based Signature

Given a randomly chosen software image, the probability that an adversary without knowledge of the entire tree (software image hashes) can forge a modified image with a valid signature is negligible if a cryptographic (specifically, one-wayness and collision-resistance) hash function is used to construct the tree.

Proof. Suppose that there is an adversary who knows some details about the Merkle tree (index of leaf node, number of leaf nodes, value of the root node). The first scenario would be that the attacker finds the same valid signature for a modified local software image. Finding such a signature would mean breaking the Merkle tree signature and obtaining the OEM private key. If the attacker is not able to accomplish this, it needs to find, for a given software image of a hash function, another software image with the same hash value. A hash function in which this is feasible is not second-preimage-resistant. Thus, if the hash function in use is second-preimage-resistant, the attacker will not be able to find the same signature. Thus, the Merkle-tree-based signature scheme is secure, as long as the signature is secure and the hash function used is second-preimage-resistant. □

Fact2: Authentication of the individual software image

Given a software image intended for a specific vehicle ECU, the adversary's capability to install the image on another ECU is negligible due to the unforgeability of the digital signature.

Proof. Suppose that there is an adversary who manages to alter the manifest information to change the destination ECU of the updated software image, for example from ECU-A to ECU-B. The prover has used the private key associated with the public key of ECU-A. If the attacker is able to change the update process and succeed into sending the image to ECU-B, the verifier decrypts it with the public key of ECU-B and thus the verification fails. Given that OemR has a unique private/public key pair assigned to each ECU and the keys are well protected and secured, the attacker will not succeed in installing unintended software on an ECU. □

Fact3: Resilience to the leakage of information

Given that the ECU has access to the other software images whose hashes constitute the leaf nodes of the tree, there is no need for the prover to send proof nodes (sibling path) as these can be calculated in the ECU when software clusters are local to the ECU. An adversary who does not know the details of the other software images cannot obtain any meaningful information through eavesdropping on the proposed MT-SOTA proposal. Proof 3: Since the size of the Merkle tree and the information about the software images are managed by the OEM repository and not by the image repository, there is no relationship between the signature size, structure, and size of the Merkle tree, making it impossible for the adversary to know about other software images and their hashes. The signature generated by our scheme includes the value of the root node in the tree encrypted with a private key. The adversary must uncover the work done by the OEM repository in order to compromise the software image signature.

6. Evaluation and Analysis

In this section, we evaluate the performance of our proposed scheme. First, we briefly discuss the performance overhead of the OemR and more thoroughly EcuX in terms of computation timing and storage memory. Our scheme does not add any additional overhead to ImagR because this entity is not engaged with the signature calculation beyond the calculation of the hash value based on the software image contents. Regarding OtaM, when EcuX is capable of computing and storing the Merkle tree hashes, there is no overhead applied to OtaM. Whenever a vehicle is requested to install an update, OtaM also receives a new piece of metadata from OemR with detailed information on the installed images bundled within a vehicle package. However, for lightweight ECUs, OtaM may need to manage the Merkle tree and provide the verification results to the ECU. In this case, the study we present for EcuX is applied to OtaM itself. It is worth noting that our scheme does not add much overhead to the network bandwidth because no additional traffic is needed as a single signature is still sent with the software image. Finally, we briefly discuss the optimization of the MT-SOTA scheme to address specific- use-cases.

6.1. OemR

The OEM repository is a key component in our scheme as it is responsible for generating the signature by computing the Merkle tree root and signing it for each software image update. OemR has full knowledge of the vehicle architecture and identification, ECUs, and software cluster distribution and activation among these ECUs; therefore, OemR knows how to create and maintain the Merkle trees used to generate the signature. OemR can extract information about used software images on a vehicle. Using this information, it can determine how to construct and update the Merkle tree with the new hash value of the software image. Thus, at the minimum, OemR has the storage overhead of the Merkle tree, which is highly dependent on the vehicle architecture. If Y is the number of ECUs in the vehicle, and Nmax is the total number of the software clusters per ECU and is a power of 2, then the Merkle tree height HT is set to $\log_2(N_{Max})$, and the storage overhead of the Merkle trees is given by:

$$Y \times (2^{HT+1} - 1) \times \|H_i\|$$

6.2. EcuX

As the process of updating software in an ECU is complex, and in the interest of measuring performance accurately at the ECU level, this study focuses on the MT-SOTA realization in the ECU to show the Merkle tree overheard on the end-target ECU, including resource demands, processor overheads, start-up time, and code sizes. It is possible that the initial Merkle tree contents are flashed at the end of line programming along the initial software to offload this task from EcuX. With our scheme, the key management is the same between the standard SOTA and the MT-SOTA schemes because this was a requirement for our design (Requirement 7 in Section 3.3) to maintain single-key usage for signatures regardless of the number and providers of the software clusters used.

6.3. Experimental Evaluation

In this section, we evaluate the performance of our concept through extensive experiments and verify its superiority in comparison to the standard traditional SOTA process. We built the test target ECU using an Infineon AURIX$^{(TM)}$ starter kit triboard [27] with a TC49x 32-bit microcontroller operating at a 400 MHz CPU clock and connected to a laptop acting as the repository through the UART/USB interface. We initially started our implementation with a TC39x 32-bit microcontroller [28]. However, given that the TC49x offers more support for different hash algorithms implemented in hardware, it is efficient to compare results with the same experimental conditions where hashes are calculated by hardware engines in the TC49x instead of being calculated by software for the cases where the TC39x does not support the required hash algorithms (e.g., the case of SHA3-512). The hardware security is typically a small cost adder (less than 10%) and nowadays it has become a necessity in ECUs. Thus, our experiments used security hardware engines to perform the hash calculations and signature verification operations. The full sequence of secure MT-SOTA implementation in the ECU is shown in Figure 11. To simplify the system, a new image is flashed to the triboard using programming tools. In addition, cryptographic key setup is out of the scope, and keys are loaded to the device using programming tools.

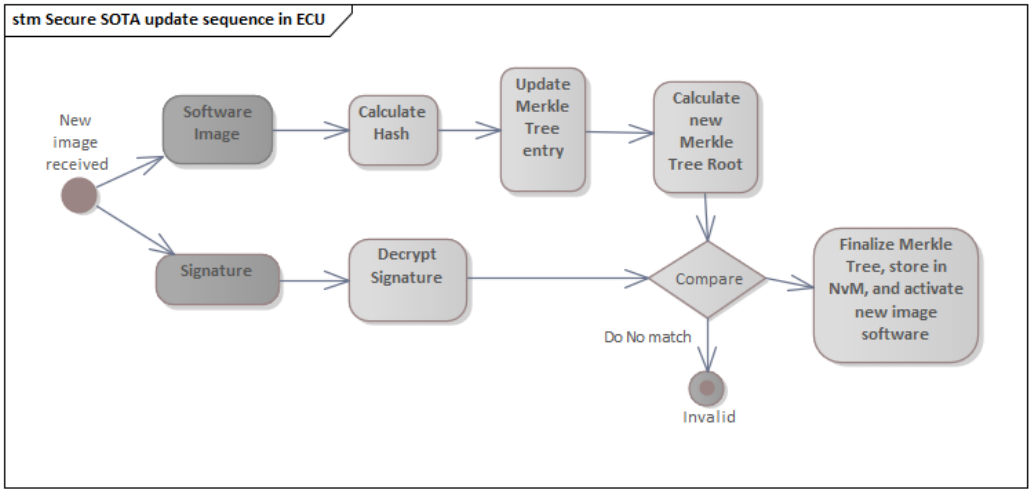

Figure 11. Secure MT-SOTA workflow sequence within the ECU. Once image is received, the signature verification is performed before installing the image.

The application software is stored in the physical non-volatile memory (NvM). Given the memory layout of the microcontroller we are using, we have six program flash (Pflash) banks used for application cores and one Pflash bank used for the security core. We designed and located the main scheduler and MT-SOTA scheme handler in Pflash0. The application software clusters are located in any of the Pflash1 through Pflash5 NvM banks. The security software, located in Pflashcs, is responsible for executing on-demand cryptographic functions (e.g., hash computations) initiated by the MT-SOTA scheme software

handler. As the smallest erasable Pflash size is 16 Kbyte, we assume that the smallest size of any software cluster or block is 16Kbyte and its size should always be a multiple of 16. We assume that we have N software images in addition to the security image and the main MT-SOTA handler image. Thus, we have N + 2 image binaries to flash on the device. We present the non-volatile memory map layout as shown in Figure 12.

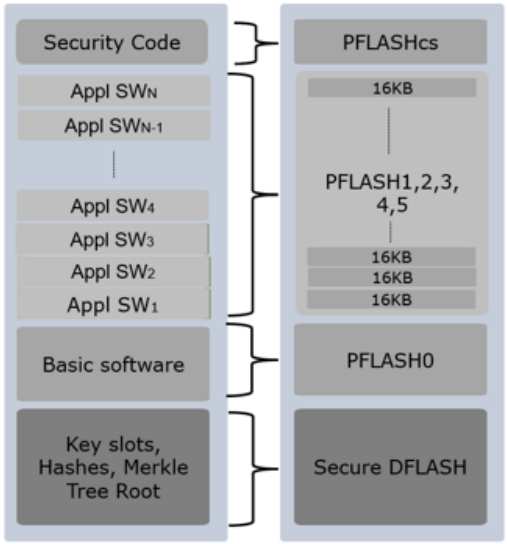

Figure 12. Software memory mapping for the ECU test case.

Software clusters (SWC) are usually separated based on various criteria such as safety level, functionality, virtual platforms, etc. We analyze different generic-use-cases to reflect different flash memory sizes, different numbers of software clusters, and different hash algorithms. We experimented with the performance of these use cases built with a combination of parameters, as shown in Table 3. Because the encryption and decryption of the Merkle tree root are the same for all use cases, we do not include it in the results to emphasize the performance results of the MT-SOTA scheme itself.

Table 3. Parameters used in MT-SOTA scheme study.

Parameter	Supported Cases
Hash algorithm (Hash output size in bytes)	SHA3-512 (64), SHA3-256 (32), SHA2-256 (32), SHAKE-256 (32), SHAKE-128 (16), SHA1-160 (20), MD5-128 (16), SM3 (256)
Number of SWCs	2 through 128
Non-volatile program flash sizes	1 MB through 20 MB
Size of software clusters	Multiple of 16 KB. Max size = Flash size/Number of software clusters

The computation of hashes is performed using cryptographic services (algorithms that map arbitrarily sized data to a fixed size output). These cryptographic routines can be implemented in software or hardware, and mixed setups can be supported. It should be possible for different applications to have different underlying cryptographic primitives or schemes. This flexibility is one of the advantages of this approach. For example, one software cluster might use SHA-2 as the hash algorithm, while another software cluster uses a SHA-3 hash algorithm. Excluding some exceptions, more hash output bits aim to achieve stronger security and higher collision resistance. As a general rule, 512-bit hash

functions are stronger than 256-bit hash functions, which are stronger than 128-bit hash functions. The main security profiles of computation of hashes [24]:

- SHA-2: Length: 224, 256, 384, 512
- SHA-3: Length: 224, 256, 384, 512
- BLAKE: Length: 224, 256, 384, 512
- BLAKE: Length: 224, 256, 384, 512
- RIPEMD-160: Length: 128, 160, 256, 320
- SM3: Length: 256

Since the hashing computation can be executed by hardware engines within the security module in AURIX$^{(TM)}$, our analysis shows that the usage of different hash algorithms does not have a major impact on execution timing. Thus, we decided to include the results of the work performed with the SHA2-256 hash algorithm in this paper.

The number of software clusters and the hash algorithm used have a direct impact on the data flash memory required to save the contents. The larger the size of the hash output is, the more memory it takes to construct and store the Merkle tree. The more software clusters there are, the more levels the Merkle tree has. In order to compare our MT-SOTA concept performance to existing concepts, we study the case when MT-SOTA is not used. Since our concept requires using a single security key pair for each ECU, we have the same requirement for a SOTA study without MT-SOTA. In this case, when any part of the software is modified, added, or removed, the complete software must be hashed, and then the hash value is signed (encrypted by OemR and decrypted by EcuX). We show the hashing time execution for different values of software cluster sizes and numbers in Figure 13. As the software cluster size increases, more time is required to perform hash operations. It is worth noting that there is an inverse correlation between the size and the number of software clusters as the complete software has to fit into the available Pflash memories. Thus, the maximum size of the software cluster becomes limited as the number of software clusters increases.

With our MT-SOTA implementation, we measured the time required to create and set up the Merkle tree data. Prior to this, hashing values of all software components had to be calculated and used for the leaf nodes of the tree, followed by creating different levels of hashes until reaching to the root, as explained earlier. The execution times are shown in Figure 14. As shown, the Merkle tree setup is independent of the software cluster size because only the hash values are used. The more nodes used, the more time is required for the tree setup.

Once the Merkle tree is built, we study the case of adding new software clusters or updating or removing them. Figure 15 shows the time required to add or update a software component. This includes hashing the added/updated software and updating the Merkle tree with a new hash value. Figure 16 shows the time required to remove a software component entry from the Merkle tree. Because removing a node does not require hashing for the software cluster, the time taken to remove a node is shorter than the time taken to add a node. It only requires updating the Merkle tree to retrieve the root.

Employing MT-SOTA for software cluster updates leads to a reduction in timing by as much as 80%, as illustrated in Figure 17. The measurement results in Figure 18, showcasing various Merkle tree sizes, further affirm the significant improvement provided by MT-SOTA. The time required for hashing smaller data blocks (software clusters) is significantly shorter than the time needed for hashing the entire software. As the number of nodes increases, the performance advantage of MT-SOTA over the traditional SOTA process becomes more noticeable. For example, when there are only eight nodes per ECU, MT-SOTA improves the process by 80% regardless of the size of the software clusters. Additionally, MT-SOTA demonstrates improved efficiency when removing a software cluster, as it eliminates the need for hash calculations on the removed software. Instead, the algorithm solely updates the Merkle tree contents. Consequently, the time depicted in Figure 13 becomes unnecessary, and MT-SOTA decreases the time interval from multiple milliseconds to microseconds.

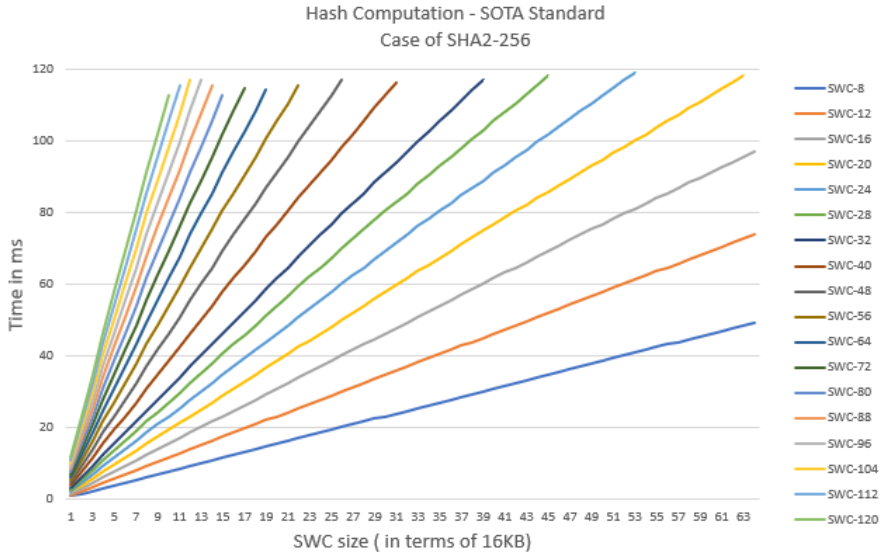

Figure 13. Processing time for generating hash of the entire software image. As the number of software clusters or the size of the software increases, the required time also increases.

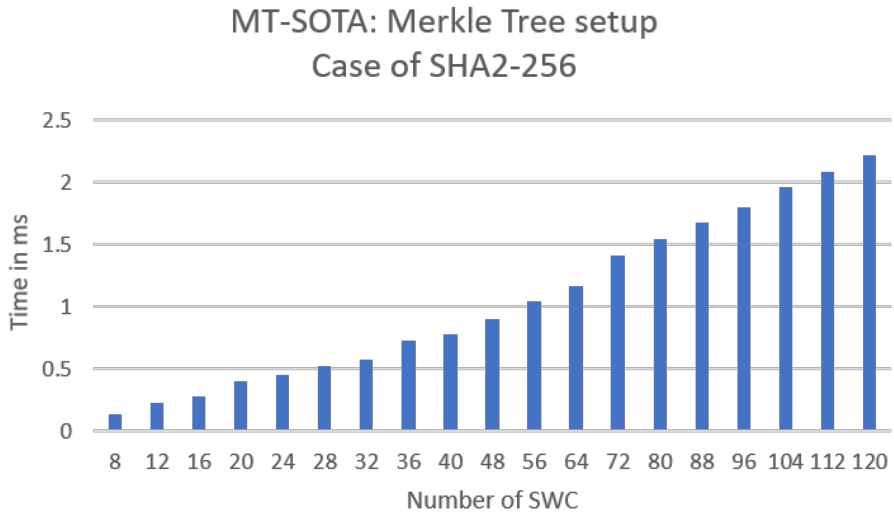

Figure 14. Execution time for setting up the Merkle tree.

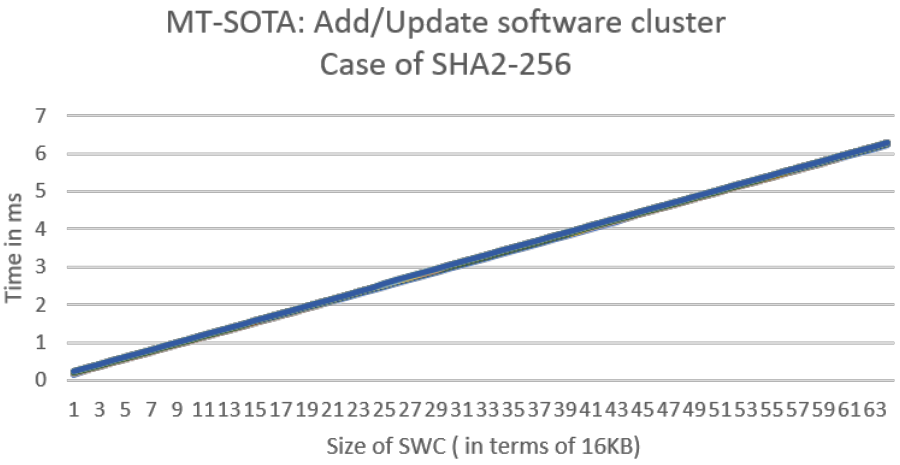

Figure 15. Execution time for adding/updating individual software clusters with MT-SOTA.

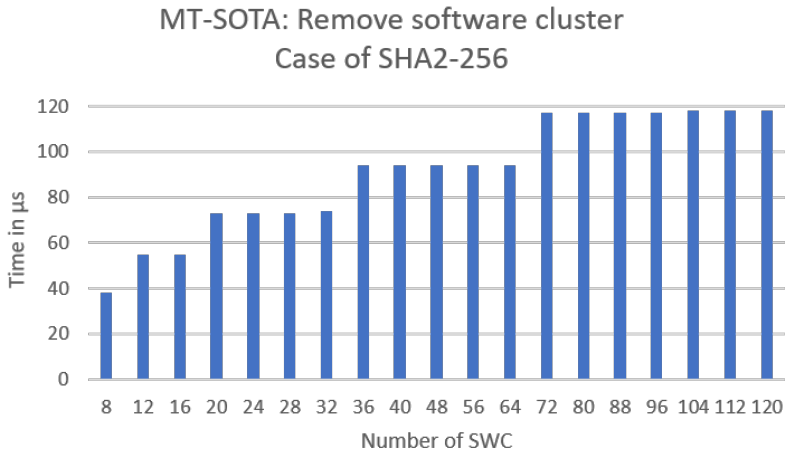

Figure 16. Execution time for removing individual software clusters with MT-SOTA.

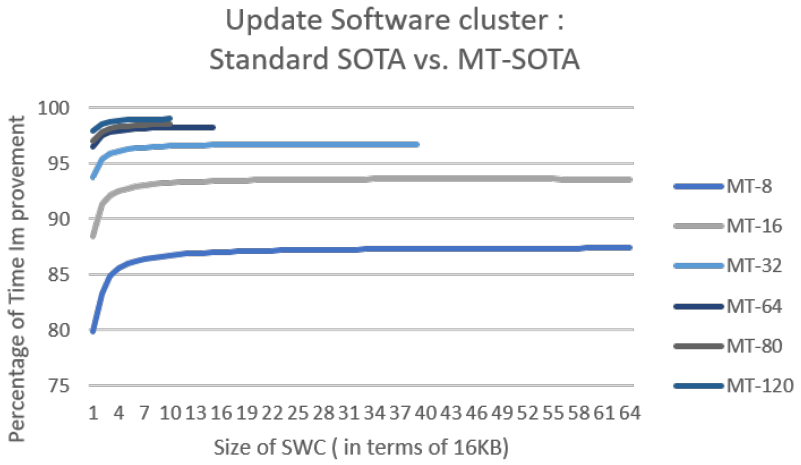

Figure 17. Time execution percentage improvement with MT-SOTA vs. standard SOTA.

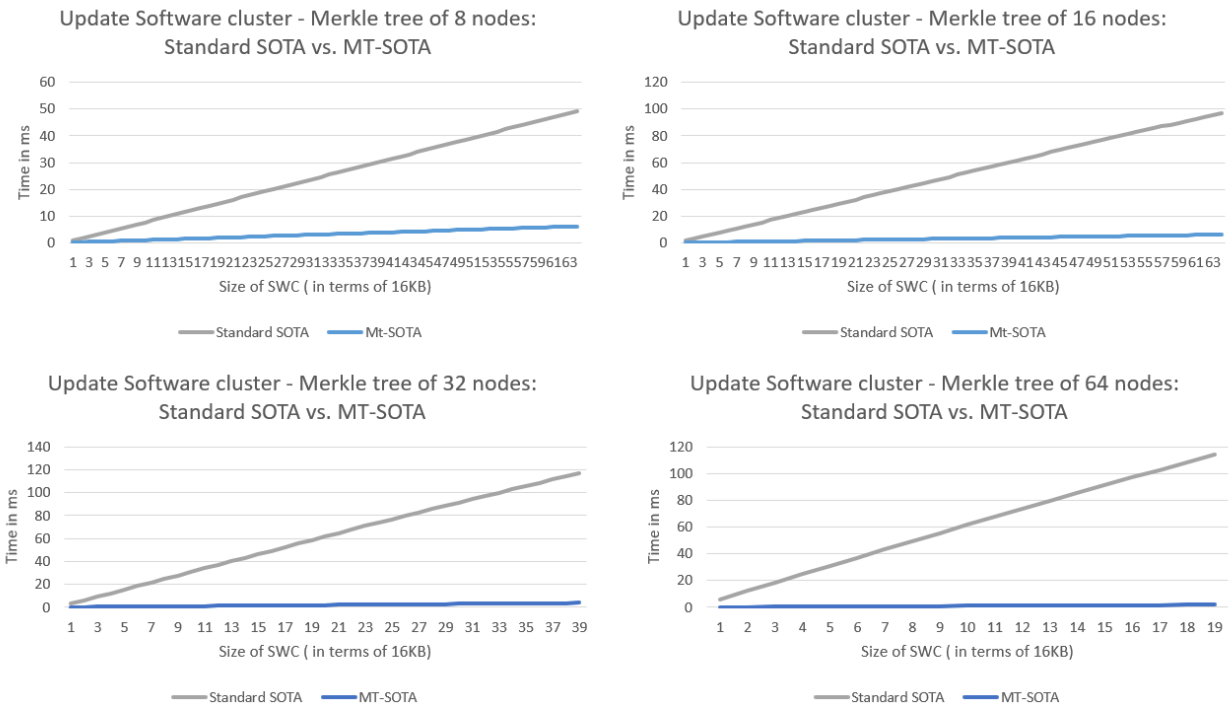

Figure 18. Comparing execution times: updating software clusters with MT-SOTA vs. standard SOTA.

Looking into memory needs, the overhead of the MT-SOTA scheme depends on the software clusters number as well as the hash algorithm used. In Table 4, we provide a summary of memory utilization for various hash output sizes and maximum node counts across our study cases.

Table 4. Memory utilization with MT-SOTA scheme.

Type of Memory	Amount of Memory Used	Description
Non-volatile program flash	7 KB	MT-SOTA handler code
Non-volatile data flash	Memory USED (Max Nodes/Hash Output Size): 16.5 KB (128/64) 8.5 KB (128/32) 5.5 KB (128/20) 4.5 KB (128/16) 6.3 KB (64/64) 4.2 KB (64/32) 3.5 KB (64/20) 3.2 KB (64/16)	Merkle tree contents (64 or 128 as max number of stored nodes). The larger the hash output size is, the more memory is needed.
RAM (data)	100 bytes + copy of Dflash Merkle tree contents	MT-SOTA handler-related variables. The Merkle tree contents have to be copied to RAM when a new software update has to be performed, and then the updated RAM copy will be copied to Dflash once the software update is validated and completed.

7. MT-SOTA Optimization Techniques

The Previous sections have demonstrated the performance of MT-SOTA in updating software clusters within the ECU. Nonetheless, specific challenges must be acknowledged based on the relevant use cases. In this section, we explore various approaches to address certain limitations associated with the utilization of MT-SOTA.

7.1. Dynamic Merkle Tree Algorithm

Up to this point, our emphasis has been on a static tree design, in which software nodes are sequentially assigned to the leaf nodes located at the bottom level. We showed that the smaller the tree height (maximal number of levels below the root) is, the better the software update performance is. As software nodes are anticipated to be dynamically added in the vehicle, the tree can be re-designed such that the addition and removal of nodes can be performed efficiently with the minimum tree height possible. The goal is to build a self-balancing Merkle tree that automatically keeps its height small for node insertions and deletions. Self-balancing trees have proven their efficiency for many binary search trees, such as red–black trees and AVL trees [29]. Given that nodes are added and removed independently, the algorithm should be meticulously designed to achieve optimal results, resulting in the minimal achievable tree height. Deviating from the bottom-up approach we have adopted in our study, we can follow a top–bottom approach to add the new leaf node. To explain our approach, we show in Figure 19 an example of a Merkle tree with eight nodes. To add node 6, it needs to move down node 5 to a lower level, add node 6 to the right side, and then calculate their parent node and the root node. When all nodes are added, it ends up with a balanced Merkle tree similar to the static tree we worked with previously.

When a node needs to be removed, we initiate the process from the node's location. In our pursuit to maintain a minimized tree height, we assess whether another individual leaf can be rotated to take the place of the removed node, facilitating the rebalancing of

the tree. When the tree has an even number of leaf nodes, the process of removing a node is straightforward. The designated node is removed, its adjacent node is elevated to its position, and the corresponding branch is adjusted to reconnect to the tree root. When the tree has an odd number of leaf nodes, removing a node mandates rebalancing the tree. A straightforward approach involves leaving the node in its position and replacing it with the other single leaf node. Subsequently, adjustments are made to the branch of the transitioning leaf node, followed by the recalculation of parent nodes for all nodes that have undergone modification. As shown in Figure 20, removing node 5 requires only the calculation of the root, while removal of node 3 requires transitioning node 6 to the node 3 position, resulting in reducing the tree height by 1.

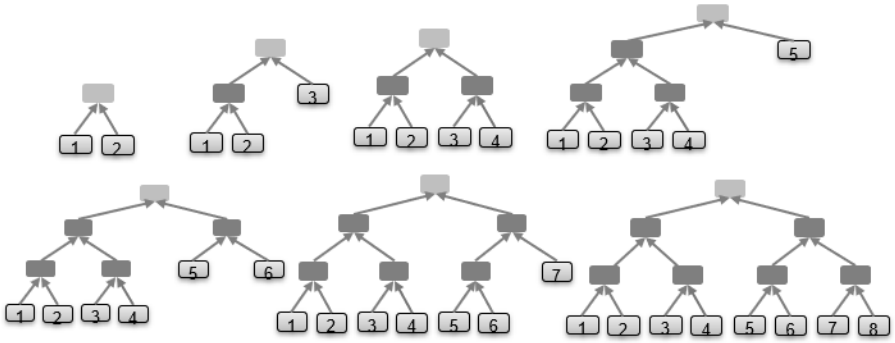

Figure 19. Nodes added dynamically to the tree.

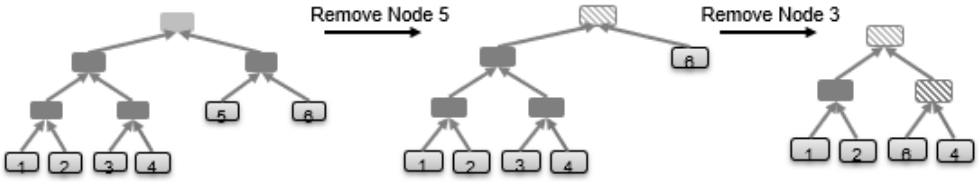

Figure 20. Node removal and re-balancing of Merkle tree.

To achieve a fast search when adding a node, each non-leaf node holds an additional bit to indicate if it has a full balanced structure. At a given time, the tree has N nodes with HT as height. If 2^{HT} is larger than the number of existing nodes, then the height of the tree remains the same when a new node is added. Otherwise, the height should be incremented, and in this case, as the tree is already full, a new level has to be added to accommodate the new node. The current root node becomes the left node, and the new added node is the right node, and their parent node is the new tree root. This is the case when node 5 is added in Figure 19. When the tree has space to add more nodes, we always start at the level below the root level and go down until we locate where the new node should be added. Algorithm A1 , shown in Appendix A, shows how to add a new software cluster (leaf node) to the tree, while Algorithm A2 shows how to delete a node from the tree. We have used array and index terms to make it simple to read and follow these algorithms. However, the implementation is using pointers to efficiently create and manipulate the dynamic tree contents. We summarize the process as follows:

If any of the nodes is a leaf node at the current level, then the new node will be added as the neighbor leaf node, and their parent node replaces the location where the leaf node was (e.g., node 4 or 6 added in Figure 19). If both nodes are non-leaf nodes, it first checks if the left node is complete (complete bit is set) or not. If the left node is set as complete, it checks if there is a leaf node in the right node. If the right node has only one leaf node, this leaf node is moved down to the left side at the lower level, and then the new node is added as a neighbor at the right side, and then their parent node replaces the location where the right node was (e.g., node 4 or 8 added in Figure 19). If the right node has two nodes (either

two leaf nodes or two non-leaf nodes), we need to move down the right node and make it the left node at the lower level, and then add the new node as a neighbor at the right side, and their parent node replaces the location where the right node was (e.g., node 7 added in Figure 19).

If the left node is not set as complete, then we start with the left side node and we repeat the same process. Once the new node is added, the correspondent branch from this node to the root is updated with new hashes.

7.2. Skewed (Unbalanced) Merkle-Tree-Based SOTA

The Merkle-tree-based approach provides a flexible balanced approach to build the tree of hashes. As described in this paper, the root is retrieved from the hash tree in an average time proportion to $\log_2(N)$, where N is the number of nodes (software clusters) in the tree. In theory, this could be optimal when all leaf nodes are utilized and updated at an equal rate. However, given that software modules within the vehicle are dynamically integrated and updated at varying rates, the objective is to reduce the path required for nodes that undergo more frequent updates.

To come up with an efficient approach to speed up the overall system performance, MT-SOTA is updated in a way to build an unbalanced tree to allow faster execution for more frequent software nodes. Let us first explain our approach with a simple use case with four nodes. On the left side of Figure 21, we have a balanced Merkle tree with four nodes and height of 2. Each updated leaf node requires changing two nodes (its parent node and the root node) and consequently two hash operations to retrieve the root. If we can move the leaf node, the mostly updated, to a higher level closer to root level, we will need to change only the root node in the tree whenever this leaf node is updated. Thus, by structuring the Merkle tree such that the path of the frequently accessed nodes is reduced, performance can be improved. Compared to the balanced Merkle tree, updating the leaf nodes which are on the shorter path to the root takes less time and has lower overhead in terms of memory access. In order to implement such an approach, each software node has to be given an attribute to indicate its priority level. On average, the unbalanced tree is two times deeper than the balanced one. The drawback for such approach is the height of the tree increases, resulting in increasing the path of some leaf nodes in the tree. The key difference between regular Merkle tree and an unbalanced or skewed Merkle tree is that the paths from the different leaf nodes to the root note have different lengths, depending on type of the skewed tree.

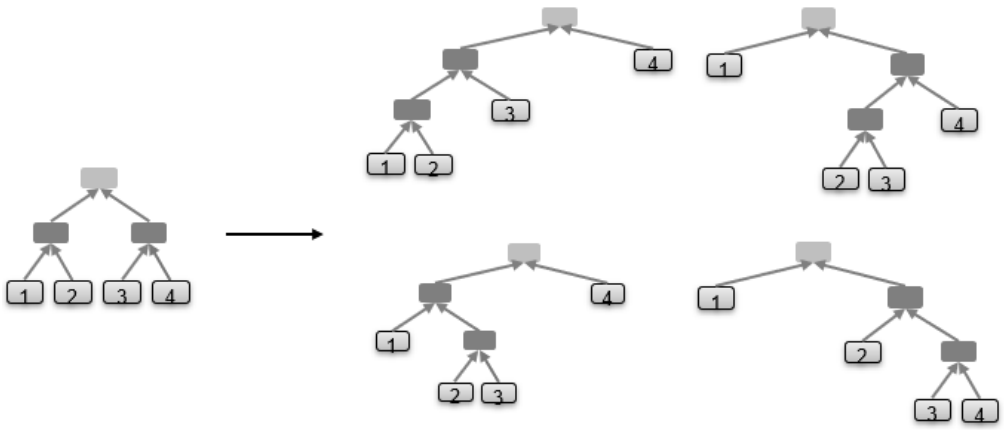

Figure 21. Transition from balanced to skewed (unbalanced) Merkle tree.

In order to deal with this trade-off, we can combine a characteristic of a B-tree [30] type with our Merkle tree. A leaf node is not anymore a single node hash but has a variable number of child nodes within some pre-defined range. The leaf node is a block of multiple individual nodes that share some common criteria. As shown in Figure 22, node 8 is the most frequently updated node and set at the top level below the root level as a single node, and then nodes 6 and 7 come at the next level and thus combine together to create a single

node with an array of two nodes. The nodes 1, 2, and 3 correspond to the lowest level with an array of three nodes. By structuring the tree in such block-based nodes, we can obtain the benefit of having shorter paths, without adversely affecting the paths of the other leaf nodes. In this paper, we focused our analysis on a balanced Merkle tree but we plan to incorporate unbalanced tree analysis in future work.

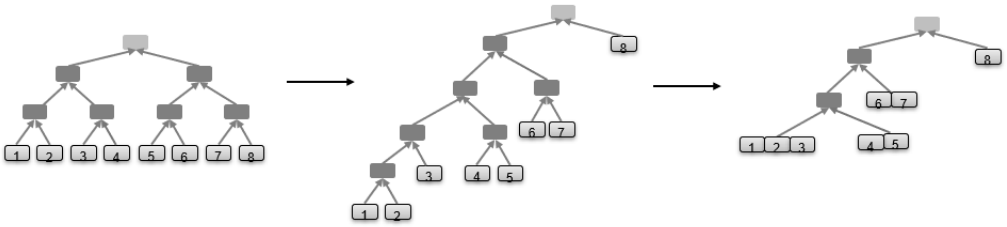

Figure 22. Unbalanced Merkle tree with arrayed leaf nodes.

7.3. Memory Reduction—Usage of Different Hash Algorithm for Non-Leaf Nodes

According to our study results, the larger the hash output size is, the more data flash and RAM memory are required. Some ECUs may have restrictions on the available memory in their devices. To reduce the amount of memory used for the Merkle tree without compromising the algorithm, it suffices to store the number of bytes correspondent with the sufficient security level in the leaf nodes and then modify the Merkle tree to use different hash algorithms with smaller output sizes in the non-leaf nodes versus the leaf nodes. This can save many bytes per non-leaf node because each node of the tree is a single hash value, thus reducing the overall memory used by the Merkle tree. This optimization requires only changes in OemR and EcuX, where the Merkle tree scheme is implemented, without any impact on ImgR. When using SHA3-512 or other hash algorithms with 64-byte hash values for hashing the software clusters, we can reduce the memory used by 1 Kbyte for 32 nodes, and by 2 Kbytes for 64 nodes and 4 Kbytes for 128 nodes if we use SHA3-256 or other hash algorithms with 32-byte hash values for non-leaf nodes. For the case of using SHA2-256 or other hash algorithms with 32-byte hash values for hashing the software clusters, we can reduce the memory by 0.5kbytes for 32 nodes, and by 1 Kbyte for 64 nodes and by 2 Kbytes for 128 nodes if we use SHA3-256 or other hash algorithms with 16-byte hash values for non-leaf nodes.

7.4. Time Execution Improvement—Multi Root Merkle Trees

The results of our study show that the execution time increases with more software clusters used. The time required to update a software cluster can be reduced if the number of nodes used is reduced in the Merkle tree. Thus, having multiple smaller Merkle trees instead of a single, big tree can achieve a faster execution time for updating a software cluster. Given that certain software clusters can be updated more often than the others, it is efficient to reduce the updates path for more frequent software clusters. Thus, we can divide the Merkle tree into coupled multi-root trees. This allows faster execution and enables the separation of nodes into different trees because not all software clusters are updated equally and frequently. In Figure 23, we show an example use case of a 16-node Merkle tree that can be divided into two 8-node Merkle trees or four 4-node Merkle trees. The roots of the multiple trees are used as the leaf nodes of an auxiliary Merkle tree whose root will be signed with the secret key to generate the signature. As an alternative, the roots of the multiple trees can be simply hashed together and the output hash value is signed by the secret key to generated the signature. Figure 24 shows the timing improvement achieved when we divide the single Merkle tree in the ECU into multiple trees with 8 or 16 nodes per tree. The more nodes exist, the better the timing improvement is. A Merkle tree of 8 nodes is 50% better when the number of nodes is more than 40, while a Merkle tree of 16 nodes is at least 45% better when the number of nodes is more than 72. An advantage of this optimization is its ability to add more nodes than anticipated for the ECU. Due to

the nature of traditional microcontrollers, we used a fixed number of levels for the Merkle tree based on the maximum number of software clusters or blocks anticipated for the ECU. With multi-root Merkle trees, a new Merkle tree can be dynamically added as long as the auxiliary Merkle tree can accommodate new leaf nodes or the roots are directly hashed.

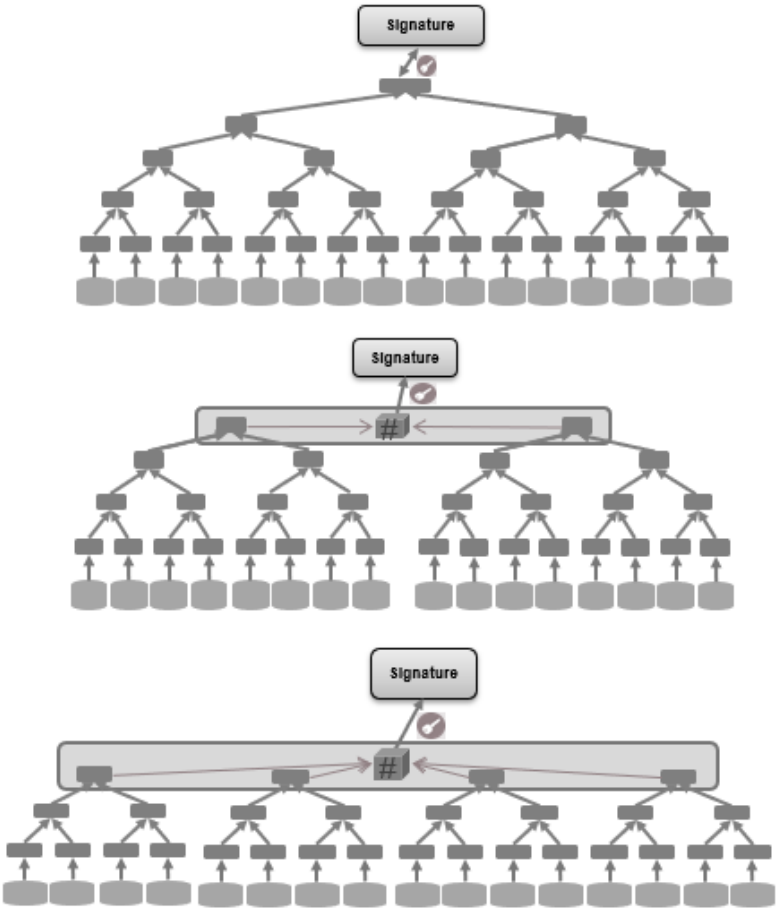

Figure 23. Decomposition of single Merkle tree into multi-root Merkle trees.

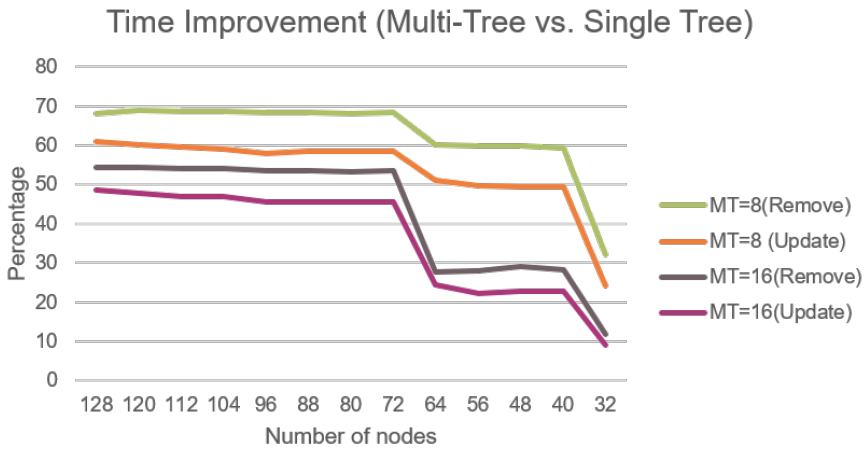

Figure 24. Time execution performance for Merkle tree setup with multi-root trees.

7.5. Pre-Flash of Merkle Tree Leaf Nodes

As demonstrated by this work, the MT-SOTA scheme accelerates the process of updating a software cluster. During the creation and configuration of the Merkle tree, each leaf

node involves the hashing of its corresponding software cluster or block. This procedure can be time-consuming during the initial startup, especially when the software's size is substantial. Nonetheless, this is not a significant concern, as it occurs only once during the ECU's initial power-on. To enhance setup efficiency, a solution could involve pre-flashing and pre-installing Merkle-tree leaf nodes into the ECU, along with the initial software (e.g., during end-of-line programming). This step would eliminate this task from the ECU and potentially expedite the setup time. In this case, there is no need to calculate the hash of each individual software cluster, and the Merkle tree can be setup in the ECU at a negligible cost as shown in Figure 14.

8. Conclusions and Future Research

In this paper, we presented MT-SOTA as our scheme for software over-the-air updates for software-based vehicle architectures. We furnished a theoretical explanation and analysis of this approach, considering various parameter values to encompass diverse use cases, ultimately representing various software architectures. The outcomes of our study illustrate the performance enhancement realized through MT-SOTA, coupled with a reasonable storage overhead. The MT-SOTA scheme is scalable, convenient, and flexible for integration into existing automotive software architecture platforms and new software-defined vehicle architectures. The execution time and memory requirements show that the Merkle-tree-based signature is suitable for deployment in backend repositories and most importantly in resource-constrained ECUs. The MT-SOTA is parameterized to fit the need of OEM applications and architecture-use cases. The intended performance and ECU resources, including factors such as the presence of hash engines in hardware, necessary calculation speed, and memory capacity, are carefully considered when determining the MT-SOTA parameters to achieve optimal outcomes. As shown through the measurement results and proposed optimization methods, MT-SOTA accelerates the time required to verify a software component update in the field. The removal of a software component exhibits enhanced efficiency with MT-SOTA, leading to reduced time due to the absence of hash calculations on the eliminated software component. While MT-SOTA offers several advantages for software over-the-air updates, it also comes with certain limitations. Implementing and managing Merkle trees can be complex. OEMs are required to establish the Merkle trees within their repositories and communicate the structure to the ECUs in the vehicle. Constructing the initial Merkle tree contents involves computing hash values for all software blocks, which can be time-consuming and resource intensive. The MT-SOTA's efficiency diminishes when the number of software clusters is restricted. Consequently, one potential proposal is to adopt a per-vehicle Merkle tree approach. Future work is to study the MT-SOTA for high Computing ECUs or zone controller ECUs with the AUTOSAR adaptive platform as well as the new AUTOSAR "Vehicle API" variant. Our forthcoming research endeavors encompass the exploration of the MT-SOTA methodology to tackle the complexities of variant management. This involves effectively managing diverse software configurations and adaptations within an ECU. The primary incentive is to establish a single signature, generated utilizing the Merkle tree scheme, for the ECU. This signature will subsequently undergo verification at the ECU level, accommodating distinct vehicle models, options, or features.

9. Patents

There is a pending patent resulting from the work reported in this manuscript.

Author Contributions: Writing—original draft, A.B.; Supervision, A.S. and D.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: We would like to thank the anonymous reviewers for their review of this paper.

Conflicts of Interest: The authors declare the following financial interest/personal relationships which may be considered as potential competing interests: Abir Bazzi is currently employed by Infineon Technologies. The evaluation platform used for the experiments is provided by Infineon. However, the author has taken the necessary precautions to generalize the results as much as possible.

Abbreviations

The following abbreviations are used in this manuscript:

AUTOSAR	AUTomotive Open System ARchitecture
ECU	Electronic Control Unit
ImgR	Image Repository
MT-SOTA	Merkle Tree-Based Software Updates over the Air
OEM	Original Equipment Manufacturer
OemR	OEM Repository
OtaM	OTA Master
SOTA	Software Updates over the Air

Appendix A

Algorithm A1 Dynamic Merkle Tree Node Insertion

INPUT: SW_M ID, H_M
OUTPUT: Merkle Tree Root, Position of SW_M
FUNCTION: **AddToMerkleTree**
if (2^{HT} = N) //Tree is full, Need to add another level
{
    MT[HT][1] = H_M;
    MT[HT+1][0] = Hash(MT[HT][0], MT[HT][1]);
    Pos(SW_M) = [Height = HT,Index = 1];
    HT++;
    N++;
    Return; //Exit
}
HT_{temp} = HT-1;//starting at the top level(below the Root)
i = 0;
NodeNotlocated = 1;
do //start of the (do- while)loop
 if(MT[HT_{temp}][i] AND MT[HT_{temp}][i+1] are leaf nodes)
 {
    MT[HT_{temp}- 1][2i] = MT[HT_{temp}][i];
    MT[HT_{temp} - 1][2i+1] = MT[HT_{temp}][i+1];
    MT[HT_{temp}][i] = MT[HT_{temp}+1][i/2];
    MT[HT_{temp}][i+1] = H_M;
    j = i;
    $HT^{'}$ = HT_{temp};
    while ($HT^{'}$ < HT)
    {
     MT[$HT^{'}$+1][j/2] =
     Hash(MT[$HT^{'}$][j], MT[$HT^{'}$][j+1]);
     j = j/2; if j = 1 then j = 0;
     $HT^{'}$++;
    }

Algorithm A1 *Cont.*

```
    Pos(SW_M) = [Height = HT_temp,Index = i+1];
    N++;
    NodeNotlocated = 0; //Exit while loop
  }
  else if ( MT[HT_temp][i] is leaf node)
  {
    MT[HT_temp- 1][i] = MT[HT_temp][i];
    MT[HT_temp - 1][i+1] = H_M;
    MT[HT_temp][i] =
    Hash(MT[HT_temp-1][i], MT[HT_temp-1][i+1]);
    j =i;
    HT' = HT_temp;
    while (HT' < HT)
    {
      MT[HT'+1][j/2] =
      Hash(MT[HT'][j], MT[HT'][j+1]);
      j = j/2; if j =1 then j = 0;
      HT'++;
    }
    Pos(SW_M) = [Height = HT_temp - 1,Index = i+1];
    N++;
    NodeNotlocated = 0; //Exit while loop
  }
  else if (MT[HT_temp][i+1] is leaf node)
  {
    MT[HT_temp-1][2i+2] = MT[HT_temp][i+1];
    MT[HT_temp-1][2i+3] = H_M;
    MT[HT_temp][i+1] =
    Hash(MT[HT_temp - 1][2i+2], MT[HT_temp -1][2i+3]);
    j =i;
    HT' = HT_temp;
    while (HT' < HT)
    {
      MT[HT'+1][j/2] =
      Hash(MT[HT'][j], MT[HT'][j+1]);
      j = j/2; if j =1 then j=0;
      HT'++;
    }
    Pos(SW_M) = [Height = HT_temp - 1,Index = 2i+3];
    N++;
    NodeNotlocated = 0; //Exit while loop
  }
  else
  {
    if (MT[HT_temp][i] is set as complete)
    {
      //add node on the right side at this level
      i = 2*(i+1);
      HT_temp = HT_temp-1;
    }
    else
     {
```

Algorithm A1 *Cont.*

```
      //add node on the left side at this level
      i = 2*i;
      HT_temp = HT_temp-1;
    }
  }
while (NodeNotlocated) //end of the (do- while)loop
```

Algorithm A2 Dynamic Merkle Tree-Node Deletion

```
INPUT: SW_M
OUTPUT: Merkle Tree Root
FUNCTION: DeleteFromMerkleTree
Pos_M = Position (SW_M);
if ( N % 2 == 0) // Node number is Even
{
    DeleteNode(Pos_M); //Just delete node, no re-balancing
}
else // Node number is Odd—Rebalancing might be needed
{
     HTt = HT; //starting at the top Root level
     i = 0;
     LineB :
    if ( MT[HTt][i] has 2 non-leaf nodes)
      if ( Pos_M at left side)
      i = 2i; else i = 2i + 1;
      HTt = HTt -1;
      goto LineB;
    else if ( MT[HTt][i] has 2 leaf nodes)
       [ SW_M2, Ht2, Index2] =
      CheckSingleleafNode(HTt, Pos_M);
      if exist
      if deleted node is the left one:
      MT[HTt-1][2i] = MT[HT2][Index2];
      MT[HTt][i] =
      Hash(MT[HTt-1][2i], MT[HTt-1][2i+1]);
      if deleted node is the right one:
      MT[HTt][2i+1] = MT[HT2][Index2]
      MT[HTt][i] =
      Hash(MT[HTt-1][2i], MT[HTt-1][2i+1]);
      j = i;if j =1 then j = 0;
      HT' = HTt;
      while (HT' < Ht2)
      {
        MT[HT'+1][j/2] =
        Hash(MT[HT'][j], MT[HT'][j+1]);
        j = j/2; if j =1 then j=0;
        HT'++;
      }
      DeleteNode(Pos_M2);
      Return;
```

Algorithm A2 *Cont.*

```
    if Do not exist
    if deleted node is the left one:
    MT[HTt][i] = MT[HTt - 1][2i+1]
    else if deleted node is the right one:
    MT[HTt][i] = MT[HTt - 1][2i]
    j = i;if j =1 then j = 0;
    HT' = HTt;
    while (HT' < HT)
    {
      MT[HT'+1][j/2] =
      Hash(MT[HT'][j], MT[HT'][j+1]);
      j = j/2; if j = 1 then j = 0;
      HT'++;
    }
  else if ( MT[HTt][i] has 1 leaf node)
    if it is a single node:
    {
      if deleted node is the leaf left node:
       MT[HTt+1][i/2] = MT[HTt][2i]
      else if deleted node is the leaf right node:
       MT[HTt+1][i/2] = MT[HTt][2i+1]
    }
    else
    {
      if deleted node is the leaf left node:
       MT[HTt][i] = MT[HTt - 1][2i+1]
      else if deleted node is the leaf right node:
       MT[HTt][i] = MT[HTt - 1][2i]
      else if deleted node is neither one:
       if ( Pos_M at left side)
       i = 2i; else i = 2i+1;
       HTt = HTt -1;
       goto LineB;
    }
    j = i;if j = 1 then j = 0;
    HT' = HTt;
    while (HT' < HT)
    {
      MT[HT'+1][j/2] =
      Hash(MT[HT'][j], MT[HT'][j+1]);
      j = j/2; if j =1 then j = 0;
      HT'++;
    }
  N = N- 1;
  Height-Adjustment();
}
```

INPUT: Pos_M //Position of SW_M
OUTPUT: Merkle Tree Root
FUNCTION: **DeleteNode**

Algorithm A2 *Cont.*

```
{
    HTt = HT; //starting at the top Root level
    i =0;
LineA :
   if ( MT[HTt][i] has 2 non-leaf nodes)
    if ( Pos_M at left side)
    i = 2i; else i = 2i+1;
    HTt = HTt -1;
    goto LineA;
   else if ( MT[HTt][i] has 2 leaf nodes)
    if deleted node is the left one:
    MT[HTt][i] = MT[HTt - 1][2i+1]
    if deleted node is the right one:
    MT[HTt][i] = MT[HTt - 1][2i]
    j = i; if j =1 then j = 0;
    HT' = HTt;
    while (HT' < HT)
    {
     MT[HT'+1][j/2] =
     Hash(MT[HT'][j], MT[HT'][j+1]);
     j = j/2; if j = 1 then j = 0;
     HT'++;
    }
   else if ( MT[HTt][i] has 1 leaf node)
    if deleted node is the leaf left node:
     MT[HTt][i] = MT[HTt - 1][2i+1]
    if deleted node is the leaf right node:
     MT[HTt][i] = MT[HTt - 1][2i]
     j = i;
     HT' = HTt;
     while (HT' < HT)
     {
      MT[HT'+1][j/2] =
      Hash(MT[HT'][j], MT[HT'][j+1]);
      j = j/2; if j = 1 then j = 0;
      HT'++;
     }
   N = N - 1;
   Height-Adjustment();
}
```

References

1. Dixon, R. Evolution of New EE Architecture. S&P Global. Available online: https://autotechinsight.ihsmarkit.com/shop/product/5003328/evolution-of-new-ee-architecture-october-2022 (accessed on 1 December 2022).
2. AUTOSAR, Adaptive Release R22-11, 2022. Available online: https://www.autosar.org/search?tx_solr%5Bfilter%5D%5B0%5D=category%3AR22-11&tx_solr%5Bfilter%5D%5B1%5D=platform%3AAP&tx_solr%5Bq%5D= (accessed on 1 January 2023).
3. Zeeb, A. AUTOSAR Classic Platform Flexibility Managing the complexity of distributed embedded software development: Invited Talk. In Proceedings of the IEEE 18th International Conference on Software Architecture Companion (ICSA-C), Stuttgart, Germany, 22–26 March 2021; p. 167.
4. NIST FIPS 186-5, Digital Signature Standard (DSS), 3 February 2023. Available online: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf (accessed on 1 April 2023).
5. Bazzi, A.; Shaout, A.; Ma, D. Secure Software Update in Automotive Modern Software Architecture. In Proceedings of the Women in Semiconductor Hardware (WISH) Conference, San Jose, CA, USA, 14–15 September 2022.
6. Bielawski, R.; Gaynier, R.; Ma, D.; Lauzon, S.; Weimerskirch, A. *Cybersecurity of Firmware Updates*; Technical Report DOT HS 812 807; National Highway Traffic Safety Administration: Washington, DC , USA, 2020.

7. Rehman, G.U.; Haq, M.I.U.; Zubair, M.M.; Mahmood, Z.; Singh, M.; Singh, D. Misbehavior of nodes in IoT based vehicular delay tolerant networks VDTNs. *Multimedia Tools Appl.* **2023**, *82*, 7841–7859. [CrossRef]
8. Rehman, G.U.; Zubair, M.; Qasim, I.; Badshah, A.; Mahmood, Z.; Aslam, M.; Jilani, S.F. EMS: Efficient Monitoring System to Detect Non-Cooperative Nodes in IoT-Based Vehicular Delay Tolerant Networks (VDTNs). *Sensors* **2023**, *23*, *99*.. [CrossRef] [PubMed]
9. *TCG Guidance for Secure Update of Software and Firmware on Embedded Systems*; Rep. Version 1, Revision 72; Trusted Computing Group: Beaverton, OR , USA, 10 February 2020.
10. *A Firmware Update Architecture for Internet of Things*; IETF RFC 9019; 2022. Available online: https://datatracker.ietf.org/doc/html/rfc9019 (accessed on 15 August 2023).
11. Kuppusamy, T.K.; DeLong, L.A.; Cappos, J. Uptane: Security and Customizability of Software Updates for Vehicles. *IEEE Veh. Technol. Mag.* **2018**, *13*, 66–73. [CrossRef]
12. Steger, M.; Boano, C.A.; Niedermayr, T.; Karner, M.; Hillebrand, J.; Roemer, K.; Rom, W. An Efficient and Secure Automotive Wireless Software Update Framework. *IEEE Trans. Ind. Inform.* **2018**, *14*, 2181–2193. [CrossRef]
13. Nilsson, D.K.; Sun, L.; Nakajima, T. A Framework for Self-Verification of Firmware Updates over the Air in Vehicle ECUs. In Proceedings of the IEEE Globecom Workshops, New Orleans, LA, USA, 30 November–4 December 2008; pp. 1–5.
14. Ghosal, A.; Halder, S.; Conti, M. STRIDE: Scalable and Secure Over-The-Air Software Update Scheme for Autonomous Vehicles. In Proceedings of the IEEE International Conference on Communications (ICC), Dublin, Ireland, 7–11 June 2020; pp. 1–6.
15. Mansour, K.; Farag, W.; ElHelw, M. AiroDiag: A sophisticated tool that diagnoses and updates vehicles software over air. In Proceedings of the IEEE International Electric Vehicle Conference, Greenville, SC, USA, 4–8 March 2012; pp. 1–7.
16. Mayilsamy, K.; Ramachandran, N.; Raj, V.S. An integrated approach for data security in vehicle diagnostics over internet protocol and software update over the air. *Sci. Direct-Comput. Electr. Eng.* **2018**, *7*, 578–593. [CrossRef]
17. Suzuki, N.; Hayashi, T.; Kiyohara, R. Data Compression for Software Updating of ECUs. In Proceedings of the IEEE 23rd International Symposium on Consumer Technologies, Ancona, Italy, 19–21 June 2019; pp. 304–307.
18. Bogdan, D.; Bogdan, R.; Popa, M. Delta flashing of an ECU in the automotive industry. In Proceedings of the IEEE 11th International Symposium on Applied Computational Intelligence and Informatics, Timisoara, Romania, 12–14 May 2016; pp. 503–508.
19. Baza, M.; Nabil, M.; Lasla, N.; Fidan, K.; Mahmoud, M.; Abdallah, M. Blockchain-based Firmware Update Scheme Tailored for Autonomous Vehicles. In Proceedings of the IEEE Wireless Communications and Networking Conference, Marrakesh, Morocco, 15–18 April 2019; pp. 1–7.
20. Steger, M.; Dorri, A.; Kanhere, S.S.; Römer, K.; Jurdak, R.; Karner, M. Secure Wireless Automotive Software Updates Using Blockchains: A Proof of Concept. In Proceedings of the Advanced Microsystems for Automotive Applications, Berlin, Germany, 11–12 September 2018; pp. 137–149.
21. Menezes, A.J.; van Oorschot, P.C.; Vanstone, S.A. *Handbook of Applied Cryptography*; CRC Press: Boca Raton, FL, USA, 2016.
22. Rogaway, P.; Shrimpton, T. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In *Fast Software Encryption*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 371–388.
23. Burkacky, O.; Deichmann, J.; Stein, J. Automotive Software and Electronics 2030. Availableonline:https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/the-case-for-an-end-to-end-automotive-software-platform (accessed on 22 March 2023).
24. *ISO/IEC 10118-3:2018*; IT Security Techniques—Hash-Functions —Part 3: Dedicated Hash-Functions. ISO: Geneva, Switzerland, 2018.
25. Merkle, R.C. A Certified Digital Signature. In *Advances in Cryptology—CRYPTO' 89 Proceedings*; Brassard, Gilles : New York, NY, USA, 1990; pp. 218–238.
26. Merkle, C. Method of Providing Digital Signatures. U.S. Patent US4309569A, 5 January 1982.
27. Infineon Technologies TC4xx Evaluation Board. Available online: https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc4x/ (accessed on 1 March 2023).
28. Infineon Technologies TC3xx Evaluation Board. Available online: https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/ (accessed on 1 March 2023).
29. Meguellati, F.M.; Zegour, D.E. A Survey on Balanced Binary Search Trees methods. In Proceedings of the International Conference on Information Systems and Advanced Technologies (ICISAT), Tebessa, Algeria, 27–28 December 2021; pp. 1–5.
30. Comer, D. Ubiquitous b-tree. *ACM Comput. Surv.* **1979**, *11*, 121–137. [CrossRef]