UNIVERSITY OF TARTU
Institute of Computer Science, #UniTartuCS
Cybersecurity Curriculum

Edgar Miadzieles

Digital Twin and Blockchain-Driven Firmware Updates for the Internet of Vehicles

Master's Thesis (24 ECTS)

Supervisor: Mubashar Iqbal, PhD

Tartu 2024

Digital Twin and Blockchain-Driven Firmware Updates for the Internet of Vehicles

Abstract:

Blockchain has gained significant attention as a technology to provide decentralized solutions in various fields. It provides assurances of integrity, authentication, immutability, and transparency through a decentralized framework, instilling trust and security. The recent adoption of Digital Twins (DT) has enabled the simulation and testing in a virtual environment by mirroring the physical entities, their environment, and processes. This thesis aims to review the Over-the-Air (OTA) firmware update process in the context of Intelligent Transportation System (ITS) and, more specifically, the Internet of Vehicles (IoV). Current OTA updates depend on client-server architecture, whereas IoV and ITS benefit from a decentralized solution to remove a single point of failure in terms of various attacks and network congestion issues. We are using blockchain and DT technologies to meet the criteria of ITS and IoV of OTA firmware updates for vehicles in an IoV environment. This thesis presents a systematic literature review of existing OTA firmware update literature that uses blockchain and DT technologies. Furthermore, a solution is proposed for the OTA firmware updates, realized using Ethereum and Microsoft Azure DTs to satisfy the requirements of secure firmware updates for vehicles in an IoV environment. The proposed solution smart contract is evaluated based on gas consumption and unit tests. The proposed solution is developed as a console application and evaluated based on design criteria derived from a systematic literature review and contextual analysis of the IoV and ITS.

Keywords: blockchain, digital twins, firmware update, internet of vehicles, intelligent transportation system

CERCS: P170 Computer science, numerical analysis, systems, control

Digitaalse Kaksiku ja Plokiahelal Põhineva Püsivara Värskendus Sõidukite Internetis

Lühikokkuvõte:

Plokiahel on pälvinud märkimisväärset tähelepanu kui tehnoloogia, mis pakub detsentraliseeritud lahendusi erinevates valdkondades. Plokiahela raamistik pakub autentimist, terviklikkust ja muutumatust tagades usaldust ja turvalisust. Digitaalsete kaksikute hiljutine kasutuselevõtt on võimaldanud simuleerida ja testida virtuaalses keskkonnas, peegeldades füüsilisi üksusi, nende keskkonda ja protsesse. Selle lõputöö eesmärk on uurida püsivara uuendamise protsessi kasutades kaablita sidevahendeid intelligentse transpordisüsteemi ja täpsemalt sõidukite interneti kontekstis. Praegused kaablita sidevahenditega tarkvara värskendused sõltuvad klient-serveri mudelist, samas kui sõidukite internet ja intelligentne transpordisüsteem saavad kasu detsentraliseeritud lahendusest, mis eemaldab nõrgima lüli tõrke erinevate rünnakute ning võrgu ummistuse mured. Kasutame plokiahela ja digitaalse kaksiku tehnoloogiaid, et täita sõidukite interneti ja intelligentse transpordisüsteemi kriteeriume, võimaldamaks kaablita sidevahenditega püsivara värskendamise protsessi sõidukite interneti keskkonnas olevate sõidukite jaoks. See lõputöö esitab süstemaatilise ülevaate olemasolevast kaablita sidevahenditega püsivara värskenduste kirjandusest, mis kasutab plokiahela ja digitaalse kaksiku tehnoloogiat. Lisaks pakutakse välja lahendus kaablita sidevahendiga püsivara värskenduste jaoks, mis on realiseeritud Ethereumi ja Microsoft Azure Digital Twins technoloogia abil, et rahuldada sõidukite interneti keskkonnas olevate sõidukite turvalise püsivara värskenduste nõudeid. Pakutava lahenduse nutilepingut hinnatakse gaasitarbimise ja ühikutestide põhjal. Kavandatav lahendus töötatakse välja konsoolirakendustena mida hinnatakse kirjanduse süstemaatilise ülevaate ning sõidukite interneti ja intelligentse transpordisüsteemi kontekstuaalsest analüüsist tuletatud disainikriteeriumide alusel.

Võtmesõnad: plokiahel, digitaalne kaksik, püsivara värskendus, sõidukite internet, intelligentne transpordisüsteem

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Acknowledgements

This work is part of the **Cyber-security Excellence Hub in Estonia and South Moravia** (CHESS: https://chess-eu.cs.ut.ee) project funded by the European Union under Grant Agreement No. 101087529. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Supervisor
I would like to express my sincere gratitude to my supervisor, Dr. Mubashar Iqbal, who provided timely assistance, learning materials, and well-structured recommendations throughout the writing of this thesis.

ChatGPT
During the process of writing this thesis, the AI chatbot tool ChatGPT[1] version 3.5 and 4 was used. ChatGPT is a natural language processing chatbot that allows having human-like conversations using a text prompt. The tool was used as a knowledge-broadening and brainstorming tool to learn about subjects related to the thesis. Additionally, ChatGPT was used to find synonyms of different words and as an assistant to find different structures of sentences to broaden the vocabulary for expressing ideas.

Grammarly
Grammarly[2] is a writing assistance tool that uses artificial intelligence and natural language processing to check and improve the quality of written text. Grammarly was used to find issues in grammar, spelling, and punctuation in the near final draft of this thesis.

[1] https://chat.openai.com/
[2] https://www.grammarly.com/

Contents

1 Introduction

Internet of Things (IoT) has a multidisciplinary vision that has seen application in several domains [27]. Although the definition varies on multiple factors, it roughly refers to a network of interconnected devices ("things"), often with sensor capabilities, that autonomously communicate data with other things over a network [16][27][38][10]. These devices can range from everyday objects like smart thermostats and wearable fitness trackers to industrial machines and vehicles. This interconnectedness of devices facilitates the automation of processes, enhances efficiency, and enables the development of innovative applications across various industries. The proliferation of IoT has the potential to enhance the capabilities of Intelligent Transport System (ITS) [58][10]. For instance, IoT in ITS enables the integration of various sensors, actuators, and other smart devices within transportation infrastructure and vehicles [58][10]. This integration allows for real-time monitoring, data collection, and analysis to improve the efficiency, safety, and sustainability of transportation systems [58]. IoT in ITS facilitates functionalities such as traffic management, vehicle-to-vehicle communication, remote diagnostics, and monitoring, ultimately leading to enhanced mobility experiences, optimized transportation networks and a reduction in traffic accidents[8][44][41][10].

In recent times, automotive design has undergone a notable transformation, with traditional mechanical functions in various vehicle components being substituted by electronic counterparts [34]. As a result, modern vehicles now feature enhanced electronic functionalities that offer improved performance and new capabilities. Due to the increasing complexity of onboard electronic units, there has been a shift towards a software-defined approach in vehicle design [18]. This shift has introduced the capability for Over-The-Air (OTA) firmware updates. As a result, vehicle components can now receive updates remotely, enhancing functionality and safety.

With these advancements, the need for robust vehicle connectivity has become more pronounced. Today's vehicles are equipped with the necessary technology to connect and communicate over networks, marking a key development in the Internet of Vehicles (IoV). IoV, as a special case of IoT, stands to enhance the safety and efficiency of transportation [12]. In addition to vehicular connectivity, IoV aims to integrate vehicles intelligence to provide transportation services on a larger scale [12]. The IoV facilitates not only the OTA updates but also enables real-time data exchange between vehicles and infrastructure, further enhancing vehicle intelligence and operational efficiency within ITS. This integration of connectivity and IoV capabilities is necessary for supporting the complex software ecosystems of modern vehicles.

This thesis investigates the process of OTA firmware updates utilizing blockchain technology and Digital Twin (DT) to provide a secure firmware update process. Blockchain technology, Ethereum, is well-suited for the IoV environment as it enhances system availability and ensures verifiable transactions and data exchanges. Moreover, Ethereum provides observability and facilitation of the firmware update process via smart contracts.

To increase observability and maintenance possibilities we use the DT technology. DT is a new and emerging technology enabling the virtualization of physical entities, their relationships with the environment, and processes in the system. This thesis explores the integration of the aforementioned technologies to improve the security, availability, and monitoring of the OTA firmware update process for modern vehicles in an IoV environment. Furthermore, we propose an OTA update process incorporating Ethereum and DT technology and provide an evaluation of the components and process.

1.1 Problem Statement

The implementation of OTA firmware updates within the IoV presents various security challenges considering attacks like rogue updates, replay attacks, Denial-of-Service (DoS), eavesdropping attacks, impersonation attacks, and modification attacks [50]. For instance, traditional OTA update mechanisms often depend on a client-server model, which inherently includes a single point of failure for DoS, where server outages could render the firmware update feature inoperative [4]. Moreover, a large number of vehicles accessing the server simultaneously for updates can lead to network congestion. On the other side, vehicles do not always have connectivity in order to perform the update [13]. The availability of the firmware update can be vital to improve the overall safety of the vehicle. Vehicular OTA firmware updates must ensure availability and resistance to DoS attacks. It is important that the update process upholds integrity, authenticity, and authorization to mitigate impersonation-, modification-, eavesdropping- and rogue attacks. Each update must maintain integrity to ensure that no unauthorized modifications compromise the software. Additionally, the authenticity of each firmware update must be verified to confirm that it is genuine and comes from a trusted source. Proper authorization mechanisms must be in place to ensure that only authorized entities can initiate and apply updates. Finally, the firmware update process mitigates rollback and replay attacks by providing mechanisms to check the package order validity. By following the aforementioned details, the primary focus of this work is to investigate the secure management of firmware updates across an extensive network of connected vehicles (e.g., IoV in our case). It is not practical to examine the firmware update across every potential configuration of a physical vehicle's systems. This increases the likelihood of software malfunctions. Therefore, it is crucial to implement robust oversight, monitoring, and simulation of the updates to ensure security and proper functioning.

The envisioned solution involves leveraging DT and blockchain technology to address these challenges. DT can provide a dynamic, virtual model of each vehicle, enabling simulation of the update process. Vehicle DT, configured according to the physical vehicle, enables monitoring and oversight of the update process. Concurrently, blockchain technology offers a decentralized and immutable ledger, ideal for securely recording and facilitating firmware updates, ensuring integrity and availability in a verifiable manner.

1.2 Research Questions

The primary research question of this thesis is as follows. *How to build a digital twin and blockchain-enabled firmware update system for the Internet of Vehicles?* To address the primary research question, the following questions have been outlined:

RQ$_1$ *What are the latest advancements contributing to the current state-of-the-art in firmware updates within IoT systems?*

RQ$_2$ *How can OTA firmware updates be adjusted to meet the unique connectivity and security challenges present in the IoV ecosystem?*

RQ$_3$ *How to implement blockchain and DT-based OTA firmware update?*

RQ$_4$ *What are the performance, robustness, and security outcomes of implementing a blockchain and DT-based OTA firmware update system?*

Table 1. Design-science guidelines and their descriptions from [23].

Guideline	Description
DG.1 Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
DG.2 Problem relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
DG.3 Design evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
DG.4 Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
DG.5 Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
DG.6 Design as a search process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
DG.7 Communication of research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

1.3 Research Method

This thesis uses the methodology of design-science research. The design-science method is rooted in engineering and is fundamentally a problem-solving process [23]. Design science aims to understand the underlying problem and the solution by building and application of an artifact [23]. The guidelines in Table 1 specify the requirements of design-science research that are used in this thesis. The artifact **DG.1** in this thesis is the proposed solution in Section 5. This thesis finds the **DG.2** by reviewing the context of IoV and ITS in Section 3. The proposed solution is evaluated (**DG.3**) based on selected criteria in Section 6 and further discussion in Section 7. The research contribution (**DG.4**) and research rigor (**DG.5**) is based on providing the foundation in Section 2, Section 4 and Section 3 to produce a proposed solution and its evaluation in Section 5 and Section 6. The design guidelines **DG.6** and **DG.7** are achieved by finding the requirements for the proposed solution as well as the descriptive diagrams and high-level architectural overviews and descriptions of the proposed solutions and technologies used.

1.4 Contributions

Considering the proliferation of the IoV paradigm, the infrastructure introduces many security and network challenges. This thesis aims to provide and analyze a firmware update framework for IoV using blockchain in combination with DT technology. Four main contributions are made. The contributions are as follows:

- $\mathbf{RQ_1}$ aims to highlight the current cases and solutions for implementing a blockchain-enabled firmware update model and a DT-enabled firmware update model in an IoT system in Section 4.
- In order to answer $\mathbf{RQ_2}$ we review the IoV and ITS paradigms and find the criteria for vehicular OTA firmware updates in the context of IoV. Additionally, a summary of the proposed OTA firmware update system is given in Section 3.
- Based on the findings of current implementations of blockchain and DT-enabled OTA firmware updates and the OTA firmware update criteria in the context of IoV, the proposed model is implemented as a proof of concept in order to answer $\mathbf{RQ_3}$ in Section 5.
- In order to answer $\mathbf{RQ_4}$, we evaluate the proposed solution based on the OTA firmware update requirements in an IoV environment. Secondly, we evaluate the functionality of the created smart contract using unit tests and static analysis, and provide a gas cost evaluation based on different cases. Finally, two scenarios are presented where the proposed solution OTA update process is showcased.

1.5 Thesis Structure

The thesis is structured as follows. Background in Section 2 describes the main technologies and concepts used in this thesis. It explains the concept of blockchain and its types and discusses the definition of DT. Section 3 introduces the concept of IoV and ITS. An overview is given for the components involved in OTA firmware updates for vehicles. The systematic literature review is performed in Section 4 reviewing DTs and blockchain in the context of OTA firmware updates. In Section 5 the solution for OTA firmware update is proposed using blockchain and DT. The solution is evaluated in Section 6 in terms of security, functionality, and performance in terms of gas consumption. Discussion in Section 7 lays out the benefits and issues with the proposed solution. Finally, Section 8 concludes the thesis with a summary of the thesis.

2 Background

This section describes the technologies used in the proposed OTA firmware update solution in Section 5 and contributes to design-science guidelines **DG.4** and **DG.7** from Table 1. We use blockchain and InterPlanetary File System (IPFS) to enable a decentralized OTA firmware update process. First, we provide an overview of blockchain in Section 2.1. Additionally, this section categorizes and describes various types of blockchains, with a detailed examination of Ethereum, Hyperledger Fabric (HLF), and IOTA. We have examined the chosen blockchain platform and the rationale behind its selection for integration into our project in Section 2.1.5. We discuss IPFS in Section 2.2. To enable oversight and monitoring of the OTA firmware update process we are using DT technology, which is discussed in Section 2.3. We provide a definition and describe architectures used for a DT. Finally, an overview is given in Section 2.4 to highlight the importance of chosen technologies.

2.1 Blockchain

Blockchain is one of the main technologies we use in this thesis for the proposed solution of OTA firmware updates in the IoV system. A blockchain is composed of blocks, with a block consisting of multiple transactions [37]. New blocks can be appended to the previous blocks with a hash value of the previous block. A nonce, which is a random number for verifying the hash, and a timestamp [37]. The block is added to the chain if the nodes on the network agree on the validity of the block, which is achieved using a consensus mechanism. During this process, a blockchain is formed with an immutable state with every participant of the network reaching the current state of the blockchain.

A consensus mechanism is a protocol for the majority of the network participants to agree on the state of a blockchain. Bitcoin uses Proof of Work (PoW) as its consensus

mechanism, where participants of the network called miners try to solve a complex cryptographic puzzle requiring significant computational effort [32]. Proof of Authority (PoA) is another consensus mechanism that relies on chosen confirmed validator nodes explicitly allowed to create new blocks to validate the transactions or new blocks added to the chain [35]. Proof of Stake (PoS) is a consensus algorithm that selects validators to create a new block based on the amount of currency they hold and are willing to stake or lock up as collateral [32].

A smart contract is code that is deployed on the blockchain on a specific address. Because the smart contract is deployed on the blockchain, it inherits properties of the blockchain like immutability, transparency, and decentralization, and once a smart contract has been executed, it cannot be reverted. The network's consensus mechanism allows for the collective verification of each contract's execution. Smart contracts are used for agreements among parties without the need for a central authority and can be used for voting systems, identity verification, and financial services, among others.

2.1.1 Types of Blockchains

Blockchain can be categorized into three types: public or permissionless, private or permissioned, and hybrid [39]. These blockchain types share similar properties of working on the Peer-to-Peer (P2P) network, are decentralized, distributed, use consensus, and have the ability to initiate, receive, and verify transactions [39]. Below we discuss the types of blockchains:

- **Public or permissionless blockchains** are open blockchain networks where any party can participate and have access to the network. Public blockchains offer high decentralization and distribution, no centralized regulations, and full transparency of the system and they are pseudonymous by nature. The nodes within the network rely on a distributed consensus mechanism, determined by the network, to validate transactions. The two primary types of consensus mechanisms are PoW and PoS. Some of the common public blockchains are Ethereum, Bitcoin, and Cardano among others.

- **Private or permissioned blockchains** are restricted and work on closed systems where the participants of the system need explicit permission to join the network. The transactions within the network are not accessible to other parties except the nodes participating in the network. Private blockchains are considered to have high transactional throughput and low decentralization. Private blockchain consensus models prioritize speed, control, and permissioned access. For example, HLF uses pluggable consensus mechanisms that are best suited for the network and application considering that privacy, confidentiality, and trust are important between the parties.

- **Hybrid blockchains** also known as consortium blockchains, combine public and private blockchains. This combination enables higher customization and provides the possibility of giving different levels of privacy and decentralization capabilities to different participating parties. These types of blockchains may offer higher transactional throughput while offering security and scalability.

2.1.2 Ethereum

Ethereum is a public blockchain platform that extends beyond simple cryptocurrency transactions by facilitating the execution of smart contracts and the development of decentralized applications. Instead of a distributed ledger, Ethereum is a distributed ledger that is described as a distributed state machine. Ethereum native cryptocurrency Ether (ETH) serves as a method for compensating the participants for computational work or execution, validation, and broadcasting transactions to the network. The primary components of the Ethereum platform include the following:

- **Ethereum Virtual Machine (EVM)** is the runtime environment for smart contracts which allows execution in a decentralized and global manner and specifies the protocols for transitioning states between blocks.
- **Smart Contracts** are snippets of code that are deployed on the blockchain on an address that is executable when the preconditions set by the contract are met. Deploying smart contracts and executing its functionality that introduces state change requires gas.
- **Gas** is used as a measurement of computation required to execute a transaction and acts as a fee. This mechanism is required to ensure that Ethereum is not vulnerable to spam.
- **Ether (ETH)** is the native currency of Ethereum blockchain.
- **Consensus Mechanism** of Ethereum blockchain is currently PoS.
- **Accounts**: Ethereum has two types of accounts: An externally owned account and a contract account. Externally-owned account has a public and private key and can initiate transactions. Contract accounts are controlled by the smart contract code, don't have private keys, and act only when executing contract code.
- **Transactions** are cryptographically signed actions initiated by externally owned accounts that change the state of the network. Creating transactions on the network requires fees. There are three types of transactions:
  - **Regular** transactions are transactions from one account to another.

  - **Contract deployment transactions** for deploying contracts on the blockchain.
  - **Contract execution transactions** for interacting with smart contract functionality.

- **Nodes** run Ethereum client software that participates in the Ethereum network where a node consists of the execution client, which executes the transactions in EVM and stores the latest state of the network, and the consensus client which employs the PoS consensus mechanism to facilitate network consensus based on data verified by the execution client. Ethereum nodes are categorized into two types:
  - **Full nodes** that perform a block validation from a specific starting block or from the genesis block.
  - **Light nodes** that only download block headers and rely on a full node for requesting the block information.

2.1.3 Hyperledger Fabric

HLF is an implementation of a distributed ledger platform for running smart contracts, called Chaincode, that is run by peers. HLF was established as a project of the Linux Foundation in early 2016 [7]. HLF has two kinds of peers: validating peers, who are responsible for transaction validation and ledger maintenance, and non-validating peers, which function as a proxy to connect transaction-issuing clients to validating peers [7]. HLF is a permissioned blockchain with a pluggable consensus and is meant for enterprise use mostly where privacy and confidentiality between parties are important.

Since HLF is a permissioned, private blockchain, it needs a mechanism to grant and validate access to identities within the network that sign new transactions. This is handled by the Membership Service Provider which also enrolls new members to the network. HLF stores its data on the ledger, which consists of a world state and the blockchain where the world state holds the current state of the ledger values and the blockchain, which serves as a transaction log. The ledger may be further scoped per channel, where the channel is a mechanism to isolate or privatize the ledger and transactions to only a specific set of participants.

2.1.4 IOTA

IOTA is a recent distributed ledger technology that focuses on zero cost, and fast transfers and is aimed for the Internet of Things [45]. Unlike traditional blockchains, which are composed of sequential blocks, IOTA stores the data in a directed acyclic graph (DAG) called Tangle, where each new transaction references two previous transactions where immediate consensus is not necessary [45]. The transaction-adding node checks for a

double spend or inconsistency of chosen references and does a light PoW before adding the new transaction. If a transaction receives additional references or approvals it is considered to have a higher level of confidence [45].

IOTA previously relied on the consensus mechanism of PoA which is achieved by the coordinator node. The consensus of IOTA demanded that a confirmed transaction is referenced directly or indirectly by the coordinator which sends signed blocks called milestones [45] [25]. This has been considered a fault in IOTA in terms of decentralization and scalability and also opens a possibility for a single point of attack [45]. On September 12, 2023, IOTA replaced the coordinator with a decentralized validator committee in order to avoid an unusable network when the coordinator is down [24]. The validator committee is an intermediary step, before IOTA 2.0, where the PoA mechanism is finally replaced by an on-tangle permissionless proof of stake voting mechanism [24]. The smart contracts in IOTA require gas to operate.

2.1.5 Blockchain Platform Choice

In the proposed solution we choose the Ethereum platform over other blockchain technologies. Ethereum was chosen mainly because of the maturity and decentralization aspects of the platform. As one of the earliest and most successful blockchain platforms, Ethereum benefits from significant network effects with wide adoption and continuous development. Moreover, Ethereum benefits from extensive tooling options for development and testing. Ethereum operates on a permissionless, decentralized network, providing robust security features and is less susceptible to a single point of failure. Furthermore, Ethereum was the most used blockchain platform for firmware update solutions in an IoT environment as was found in the SLR in Section 4.5.1.

2.2 IPFS

InterPlanetary File System (IPFS) is an open-source peer-to-peer distributed file system [5] where data is shared among multiple nodes. IPFS replaces location-based addressing, like URL, with content-based addressing and content can be accessed using a content identifier (CID). One of the advantages of IPFS is serving on-demand content. If many users access the same file, it can be distributed from all nodes that have it, not from a single server. IPFS uses Kademlia, a type of Distributed Hash Table (DHT) optimized for use in decentralized peer-to-peer networks. It's a system that helps locate the peers on the IPFS network that have the specific data being looked for. This setup allows for a highly effective and self-managing network that remains stable even as nodes frequently join or leave. IPFS builds a Merkle-directed acyclic graph, creating cryptographically hashed links between objects [5]. This provides tamper resistance and verifiable data fetching. Once content has been added to the network it cannot be deleted. The only deletion possible is local.

2.3 Digital Twin

Digital twin is a relatively new technology that has gained momentum in recent years. The concept has evolved during the years changing the definition based on its application. Generally, DT is a virtual replica of the physical asset or process. DT follows the replicated entity through its lifetime by sharing near real-time data. DTs are used for simulation, monitoring, and predictive analysis of the physical asset without direct involvement with the physical counterpart.

2.3.1 Definition

DTs originated in the United States military aerospace industry and have now found usage in transportation, industrial production, education, and other industrial sectors [55]. The concept was first presented by M. Grieves in the context of Product Lifecycle Management (PLM) in 2002 and had the properties that are considered to define today's DT [15]. Similar concepts have been introduced before that mimic the information input from the physical world by D. Gelernter called "Mirror Worlds" and a system where a physical entity has a virtual counterpart by K. Främling at al [46]. Gathered from the works of [46] [55] the concept has further evolved during time as follows:

- The name "digital twin" was first used by NASA in 2012 for the simulation of a physical system that makes use of virtual models, real-time sensors, and historical data to mirror the physical twin.
- In 2013, DT technology was conceptualized as a concept that uses data-driven analytics and various physical models to accurately replicate the operational conditions of a specific entity.
- Definition of DT from a function perspective in order to simulate, predict, and control the feedback from the physical entities.
- Refinement of product description in DT where simulations should reflect the digital model properties and behavior more realistically.
- Additional update from an architectural perspective where the DT is not limited to entities but linked relationships.

The definitions of DT are various, and there is no consensus on what exactly a DT should be and which properties it should have. The rough definition of a DT is that it is a virtual model or simulation of a physical environment or object that represents the exact state given identical conditions that stores the historical data as well as exchanges on the real-time data where the states of the physical and virtual are the same. The lack of consensus on the definition means that DTs can currently have characteristics

based on the level of integration or completeness, however, they share these common characteristics as pointed out by [46]:

- **High-fidelity**: The DT is a near-identical copy of the physical entity or environment that mimics its appearance, state, and functionality. Depending on the DT completeness, the fidelity might be very accurate.
- **Dynamic**: The DT is not a static representation of a state but follows the changes of the physical counterpart.
- **Self-evolving**: Similarly to the previous definitions of "dynamic" criteria, the virtual entity changes according to the changes of the physical counterpart.
- **Identifiable**: Each physical entity with its relations has at least one unique virtual counterpart that is related to the physical entity.
- **Multiscale**: The DT copies the physical counterpart and its relations on multiple levels, ranging from geometry to physical phenomena related to the entity.
- **Multidisciplinary**: DT encompasses a wide range of fields like computer science, mechanics, electronics, etc.
- **Hierarchical**: DT consists of multiple components like physical devices and their environment with links to other devices and phenomena, which are all integrated within a virtual system.

2.3.2 Architecture

Due to the evolving definition of DTs, the integration and architecture can also consist of various levels and architectural models. Initially, the architecture was laid out as a 3D system [46] as depicted in Fig. 1. This system consists of three components: physical, virtual, and connection. Fig. 1 shows (a) the definition of a digital model, where the data is shared manually in both directions. Secondly, a digital shadow, as described by [46], is depicted as (b) in Fig. 1, where the automatic data is sent to the virtual model and manual data to the physical object. A DT definition where the data flow is automatic in both ways is shown by (c) in Fig. 1.

The 3-dimensional DT model was extended to a 5-dimensional model [49] consisting of physical, virtual, service, and data components and a connection between these components. Fig. 2 shows that the data is shared by the physical, virtual, and service components of the system using connections (d), (e), and (f). The component interactions are shown as (a), (b) with (c) also mapping the data between virtual and physical components in Fig. 2. The virtual component consists of sub-components in four dimensions, including the geometry model, physics model, behavior model, and rule

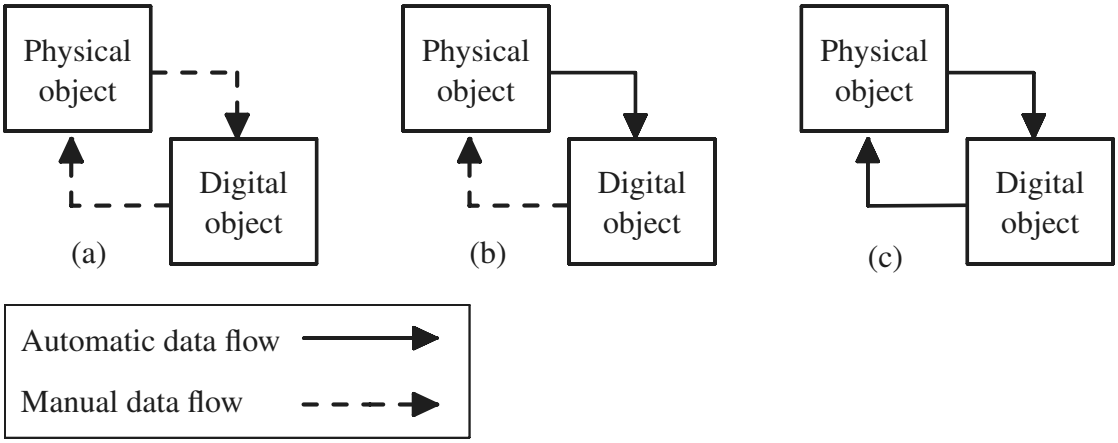

Figure 1. Three-dimensional DT architecture with communication flow types and directions, adapted from [46].

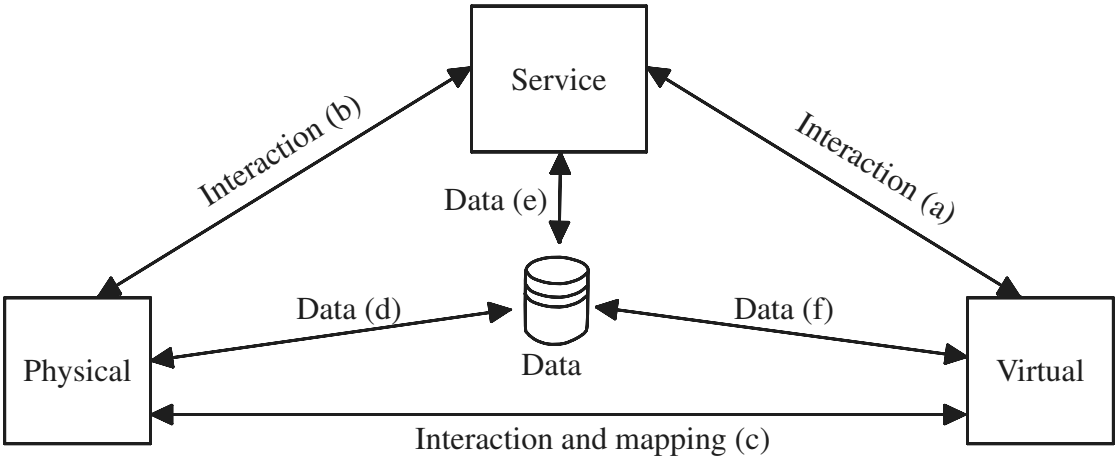

Figure 2. Five-dimensional DT showing communication directions between the components, adopted from [49].

model, and evolves with the physical part. The service layer can enhance the physical and virtual systems by providing optimization strategies and control for the system.

In terms of hierarchies, DTs can be classified into unit level, system level, and system of systems level based on their implementation and functionality [48]. These levels may be attributed to hierarchies of the system or entity in question. In the example of a production-like system, the system of system level might be attributed to the production floor and the machines, where the machine is the system and machine components are the unit level [49]. The same hierarchy might hold true where the specific equipment or

machine might be viewed as a system of systems with inner subsystems and the functional execution units like motors as a unit level [49]. The same reasoning applies to other fields and disciplines like mechanics, chemistry, electronics, etc. The actual implementation and tooling used to create a DT depends on the problems and requirements of the application or scenario.

2.4 Significance of Using Blockchain and DT in IoV

Implementing OTA firmware updates for software-defined components in vehicles is becoming increasingly common. The conventional client-server architecture depends on the server being secure, available, and capable of handling the volume of requests. This introduces issues of potential downtime, network congestion, and in many cases a single point of failure. Using a distributed technology for issuing new firmware updates can potentially solve these issues. The sharing and storage of firmware metadata can be facilitated by distributing the responsibility for its availability. Smart contracts can be designed to manage firmware update procedures. The blockchain network ensures that the smart contract functionality is executed correctly. The immutable and timestamped records on a blockchain offer a method of auditability and provide a simple process for tracing issues.

DT is used in many scenarios and applications to improve operational or product-related decision-making. The technology provides the possibility to optimize or design processes and products in a virtualized environment, potentially reducing cost and increasing the speed of development. DTs can be used to follow the product through its whole life cycle, providing improved oversight for maintenance and utility. With data-driven solutions, the issues of a real-life asset can be mitigated or predicted based on the virtualized product behavior. In general, DT serves as a solution for monitoring and analyzing the properties and behavior of an asset with unique conditions, eliminating the need for direct interaction with or reliance on the physical device itself.

2.5 Summary

In this section, we discussed the technologies we use for the proposed solution in Section 5. We defined different types of blockchains like public, private and hybrid blockchains. We covered the different aspects of Ethereum, IOTA, and HLF. The chosen platform for the proposed solution is Ethereum, mainly for its maturity, decentralization and security aspects. We covered IPFS, which is a decentralized peer-to-peer storage system and suitable for the proposed solution. DT technology and concept were covered in detail to give an overview of the definition and provide the different architectures of DTs. The use case of the technologies in the context of OTA firmware update in an IoV environment is further discussed Section 3 and in the conducted systematic literature review Section 4.

3 Use Case: IoV

In the background Section 2 we introduce the technologies taken under consideration as a part of the proposed firmware update process. This section aims to answer **RQ_2**: **How can OTA firmware updates be adjusted to meet the unique connectivity and security challenges present in the IoV ecosystem?**. We review the problem relevance according to **DG.2** from Table 1. We explore the concept of the IoV and its connection to ITS in Section 3.1. The benefits of relocating firmware update processes closer to the vehicles within this system are emphasized. Moreover, the IoV context is reviewed from the networking perspective. In Section 3.2 we give an overview of the current state of vehicle components and OTA firmware updates. Additionally, the requirements and criteria for the vehicle firmware updates are found. Finally, we introduce the proposed solution for OTA firmware updates in the IoV system as a summary in Section 3.3.

3.1 IoV Context

ITS refers to the application of advanced information and communication technologies to the field of transportation to improve transportation safety, efficiency, and sustainability. ITS is not limited to highway traffic, it provides services for air transport systems, water transport systems, rail systems, etc [40]. From the standpoint of road transport vehicles, ITS aims to enhance safety, traffic efficiency, and traveler information using traffic management systems, vehicle-to-everything communications, smart parking solutions, electronic toll collection systems, etc. As transportation management is becoming more data-driven, it requires vehicles to have a connection to their environment, other entities within the system, and the possibility to process and store data that exceeds the limitation of a single vehicle.

VANETs are a subset of mobile ad-hoc networks specifically designed for enabling connectivity using vehicles on-board-units (OBU) in the form of vehicle-to-vehicle (V2V), Vehicle-To-Infrastructure (V2I) through the Roadside Units (RSU) or Vehicle-to-Everything (V2X) [33]. V2X is a broader term that encompasses both V2V, V2I, vehicle-to-pedestrian, etc. VANETs are considered to be conditional networks because the vehicles as nodes are temporary and random and the performance is affected by vehicular density and distributions [12]. Another problem with VANETs is the limited capacity for processing information collected by vehicles and other actors of the VANET network making them suitable for small-scale services such as collision prevention or other notification services [9]. To integrate vehicles into ITS and target the aforementioned limitations, the vehicles need to have connectivity and computational resources on a larger scope, marking the transition from VANETs to IoV.

IoV integrates VANETs as a part of its system and forms an integration and information exchange between humans, vehicles, and the environment. IoV integrates vehicle intelligence with networking and computing capabilities beyond VANETs [12]. IoV

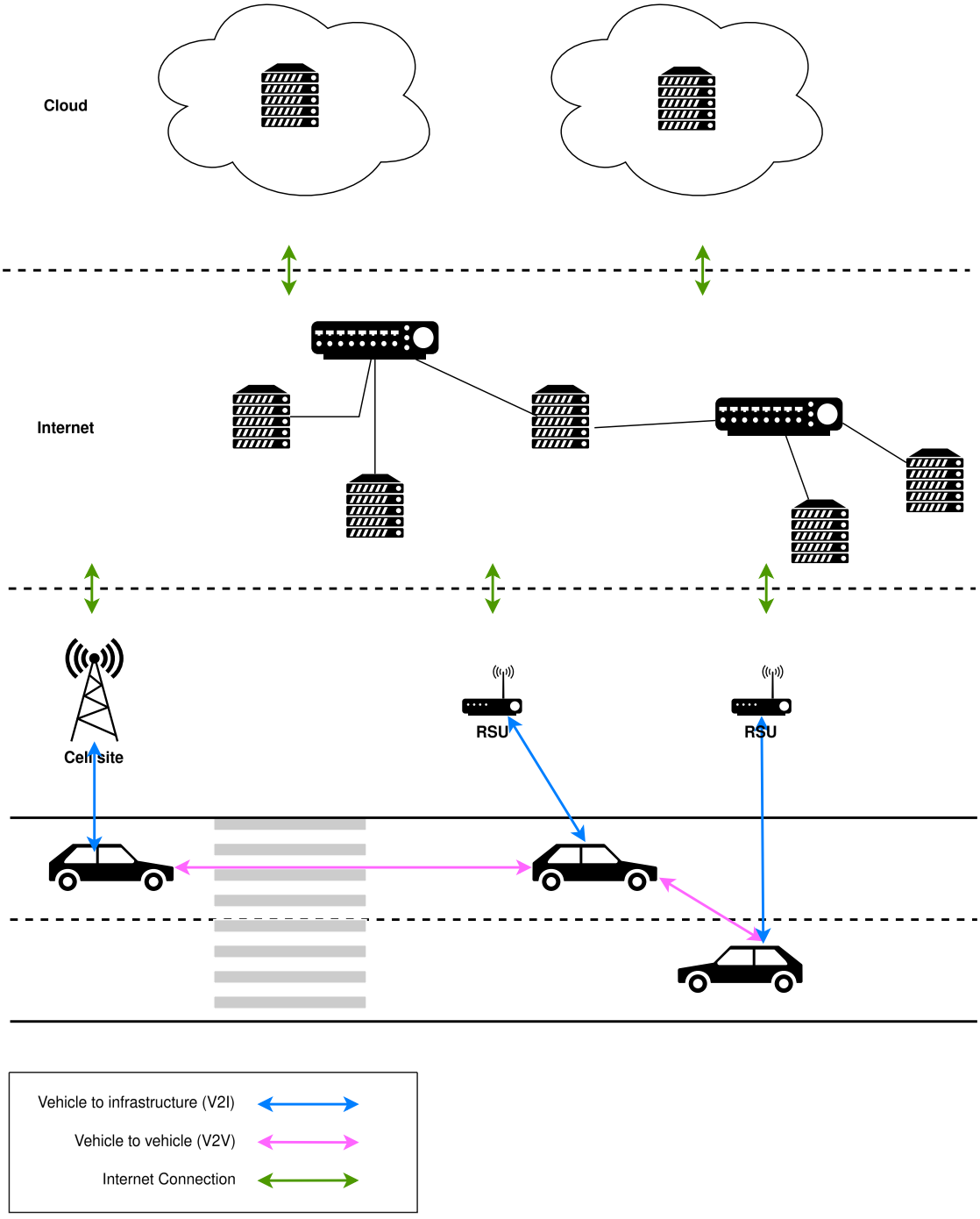

Figure 3. Abstract representation of IoV components, layers, and data communication flows between components and layers.

stands to serve as a technology to realize the demands of ITS by providing information for various ITS applications like road safety, management and control of traffic [12]. Fig. 3 presents a conceptual representation of the Internet of Vehicles (IoV), illustrating how vehicles establish vehicle-to-vehicle and vehicle-to-infrastructure (V2I) connections. These interconnected vehicles and infrastructure components collectively form the IoV, which is further integrated into the broader internet through cloud connectivity, showcasing a layered architecture encompassing vehicles, infrastructure, the internet, and the cloud. IoV allows to move the processing of big data generated by actors within the system from data processors like vehicles to technologies that provide scalable storage and more processing power through a communication layer with various networking technologies like Wifi, dedicated short-range communications (DSRC), 4G, 5G, etc.

Utilizing real-time traffic solutions through connected vehicles produces a large amount of spatio-temporal data as it depends on the location and time [12]. Paradigms like cloud computing, edge computing, or fog computing have been considered to offload the computation and data-intensive operations to decrease the load of centralized data centers forming additional layers between the vehicle and the server [12]. For IoV networks, it is advantageous to shift a portion of the computation and network traffic closer to the end-user, thereby meeting the demands of network services at the periphery, where direct interaction occurs. In terms of the use case of blockchain-enabled OTA firmware updates, the blockchain nodes can be positioned closer to the consumers given they have enough storage and networking capabilities to update the nodes to the latest states. Another advantage of blockchain in the context of IoV and ITS systems is its ability to support a decentralized and transparent communication framework. This enables a comprehensive oversight of data, while also ensuring that the blockchain ledger can be audited and reviewed for enhanced trust and integrity.

3.2 Current State of OTA Updates

Over the past decades, the software has grown exponentially alongside the Electronic Control Units (ECUs), which replace the mechanical functions of different units within the vehicle [34]. The ECUs are connected via Controller Area Network or Local Interconnect Network [13]. Together with the sensors, power distribution, and wiring, they form the electronic system (E/E) architecture [18]. Due to the rising complexity of onboard E/E architecture in a vehicle, model-based development has taken precedence to handle the growing requirements of vehicle system updates and development by Original Equipment Manufacturers (OEMs) increasing integration of OTA updates [18].

At present, OTA software updates are available through cellular networks [13] and require end-to-end connectivity in a client-server relationship to receive the update packages. The system for OTA updates has been researched in several papers with most of them targeting the issues of authentication, integrity, and confidentiality by proposing slight variations of the server-client model [13]. The general overview of the process is

depicted in the Fig. 4, where the vehicle, through a network connection enabled by a cell site, receives the update from the OEM server.

The current proposed development standard Adaptive Platform of the AUTomotive Open System ARchitecture (AUTOSAR) Adaptive Platform provides recommendations for service-oriented communication between the software components [1]. Firmware updates are one of the goals of AUTOSAR Adaptive Platform provided by the Update and Configuration Management (UCM) which handles the update requests and supports software packages from different suppliers. Retrieving updated statuses from the ECU is also a part of the specification. Firmware Over-The-Air (FOTA) is covered in AUTOSAR document [2] which specifies a FOTA target and UCM subordinate which receives the ECU software from the FOTA master ECU and a backend server that provides the firmware image. The FOTA master is also responsible for initial verification, authentication, integrity, and authenticity checks of the firmware.

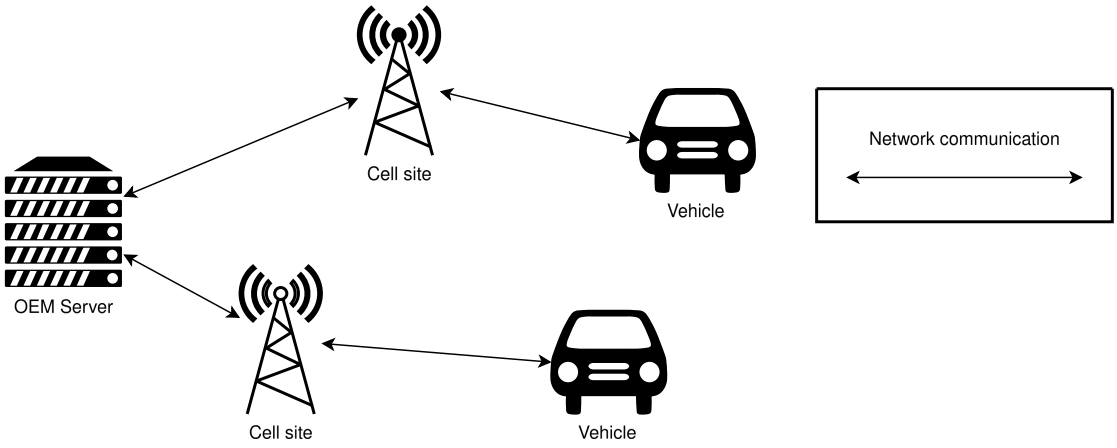

Figure 4. Generally used client-server OTA firmware update system with data communication between the vehicle (the client) and OEM server (the server) mediated via cell sites.

Another OTA update framework Uptane [28] is implemented by several automotive suppliers like Lear Corporation, OTAinfo, and Advanced Telematic Systems. Uptane provides customizability and security by dividing ECUs into primary ECUs with more computing power to verify the metadata and image and a resource-constrained secondary ECU that performs either full or partial verification. The image is also divided into directors for identifying the images necessary for updates and an image repository containing all versions of ECU software components. Another principle is securely signing the metadata of the update. However, researchers have pointed out, that Uptane is vulnerable to rollback attack [20].

The aforementioned techniques and frameworks are among many proposed [20] and

no single OTA update framework has been universally adopted. However, as described by [20], the installation procedure, generally, involves a Telematics Control Unit (TCU), also known as gateway ECU that downloads the update package from the server and distributes the package to the appropriate ECU after the package verification. Combined from [20], AUTOSAR, and Uptane a comprehensive overview of the essential criteria, challenges, and prerequisites for OTA updates for a vehicle can be compiled as such:

- **Reliability and validation**: ECUs in vehicles are interdependent, a failure in update may cause the vehicle to malfunction.
- **Authenticity**: Firmware is verified to be genuinely from the claimed source or originator and is not altered by unauthorized entities.
- **Integrity**: Firmware data, delivered via the OTA update process, is unaltered, and free from unauthorized modifications or corruptions. A verification mechanism should be in place to check the integrity of firmware.
- **Authentication**: The firmware update process hinges on the vehicle's ability to authenticate the update package.
- **Protection of the update transaction**: Traceability of update installations is critical, especially for safety updates and when addressing potential faults, necessitating verification of update authenticity by the OTA backend and target ECU.
- **Availability**: Safety-critical update packages need to be readily available to the target devices.
- **Update status from ECU**: The ECU must be able to respond with the update status if firmware was installed or failed.

3.3 Proposed Solution

Following the requirements stated in Section 3.2 we propose a system for OTA firmware updates for vehicles within the IoV system. To satisfy these requirements, we incorporate blockchain for integrity, authenticity, and authentication and DTs for validation and reliability. The firmware update status and verification is saved on the blockchain. The firmware updates for vehicles in an IoV system can take advantage of the decentralized IPFS and Ethereum networks increasing availability of the firmware update information for the vehicles.

The proposed solution, covered in Section 5, uses Ethereum smart contracts to store the firmware update metadata. Every vehicle is issued a smart contract. The firmware metadata is stored on the smart contract as a package. The packages have a reference to

the upcoming update forming a linked list. This is how the package ordering is achieved. The package consists of ECU firmware metadata compiled by the vendor of an ECU of the vehicle. This information is encrypted and shared with the manufacturer. The manufacturer compiles the updates into a package, encrypts the package, and adds it to the smart contract. The encrypted firmware is added to IPFS. The location of the firmware is embedded in the firmware metadata. Important smart contract functionality that changes the state of the smart contract is only allowed by the vehicle or the manufacturer. The firmware package structure provides the vehicle with the assurance of firmware authenticity, integrity, and authentication.

The vehicle polls these packages from the smart contract. The vehicle can verify the packages that have been signed by the manufacturer and decrypt the metadata. Each ECU metadata is read from the firmware metadata package and the vehicle can download the firmware from IPFS. Each ECU can, in turn, perform verification of the firmware, decrypt it, and install it. The installation fails or completes successfully. This information is added to the smart contract package information. Smart contract notifies the manufacturer via an event, that the vehicle has completed the firmware processing. This firmware update process can now be executed for the DT. DT uses the same smart contract to fetch the necessary data just like the vehicle. This provides the manufacturer oversight of the firmware updates and the ability to respond with the next steps for the vehicle if the vehicle firmware update fails. The verification of the smart contract by the vehicle and DT update process traceability provides oversight of the firmware update process.

3.4 Summary

This section aimed to answer the research question $\mathbf{RQ_2}$. We presented the unique conditions of the OTA firmware update process in an IoV environment. Vehicles within the IoV environment have intermittent connectivity to the RSUs and the internet featuring V2V, V2I, and V2X communication. The vehicle's internal components have become more software-defined making the OTA firmware update process more convenient for the vehicles. Taking advantage of the connected vehicles and software-defined components we can propose a firmware update process that is decentralized and secure. Decentralization allows us to omit the client-server architecture, avoid single point-of-failure issues, and move the update process closer to the vehicle. The requirements for the OTA firmware update process in an IoV context presented in this section is referenced to create a more decentralized solution in Section 5. The technologies used in the proposed solution are discussed in Section 2.

4 Systematic Literature Review

In this section, a Systematic Literature Review (SLR) is conducted to identify the latest advancements in the context of firmware update processes in IoT systems that use blockchain and DT technologies as a part of the process. This section is part of the design-science guidelines **DG.4**, **DG.5**, **DG.6** from Table 1. From the perspective of using these technologies for firmware updates, the main contributions of the aforementioned technologies are reviewed and summarized in order to provide an overview of the firmware update framework design, design goals, and use cases. Furthermore, blockchain attributes and constructs are extracted to outline the specific implementation details of the technology. The results from the SLR are used to define the design choices of the proposed solution in Section 5. The SLR methods and structure were guided by the Kitchenham review guidelines [26].

The focus of the review is to answer the research question **RQ_1: What are the latest advancements contributing to the current state-of-the-art in firmware updates within IoT systems?** The research question was divided into two sub-questions to gain an understanding of the relevancy of incorporating DT and blockchain technologies into the firmware update process in IoT systems.

$RQ_{1.1}$ *How does blockchain contribute to the firmware updates in the IoT systems?*

$RQ_{1.2}$ *How does DT contribute to the firmware updates in the IoT systems?*

In Section 4.1 we provide the search queries used to find the related research. Selection criteria used to filter the relevant papers is given in Section 4.2. Literature selection in Section 4.3 describes the selection process of the papers. Relevant data extraction strategy is described in Section 4.4. The results of the SLR are given in Section 4.5 based on data extraction strategy.

4.1 Search Queries

The primary data sources for the literature review are IEEE Xplore, ACM, ScienceDirect, and Springer databases. We also use literature search engines, e.g., Scopus and IEEE Xplore, to search the relevant studies from the aforesaid databases. The reason for choosing these databases is the availability of up-to-date technology and engineering-focused research, including topics like blockchain, IoT, and DT and their combined literature. The targeted search queries were difficult to compile, since terms like "firmware", might also be used as "software". Updates or modifications of the software might also be considered as device management. Considering these constraints, the search resulted in a more relaxed query searching from the title, keywords, and abstract. For literature review question **$RQ_{1.1}$** the search terms are the following: (iot OR "internet of things" OR iov OR "internet of vehicles") AND blockchain AND ((software or firmware) AND

update) targeting literature that broadly captures blockchain technologies used in IoT with the focus on firmware updates. For question $\mathbf{RQ_{1.2}}$ the query used was ("digital twin" AND ("software update" OR "firmware update" OR "device management")) in regards to the search for literature containing technologies and solutions that can enable firmware updates in a DT framework.

4.2 Selection Criteria

The characteristic of a relaxed query is that it yields a broad range of results. The focus of the selected Exclusion Criteria (EC) is to limit the search to literature that is recent, select only literature in English, exclude grey literature, and consider papers that are not limited in detail. Inclusion Criteria (IC) are compiled to find literature focused on blockchain-enabled firmware updates and DTs as a part of the firmware distribution system. Furthermore, the most benefit is gained if the literature explains how the use of the aforementioned technologies is relevant and beneficial to the firmware update process. The inclusion and exclusion criteria are presented below:

Inclusion criteria

- IC_1: Literature related to blockchain-enabled firmware updates.
- IC_2: Literature using DTs as a part of firmware distribution.
- IC_3: Literature discusses the benefits of blockchain- and DT-enabled firmware updates.

Exclusion criteria

- EC_1: Exclude literature not in the English language.
- EC_2: Exclude literature published before 2017.
- EC_3: Exclude literature shorter than five pages.
- EC_4: Exclude grey literature.

The IC was chosen to find literature that explicitly discusses blockchain- and DT-enabled firmware updates in order to find papers that answer questions $\mathbf{RQ_{1.1}}$ and $\mathbf{RQ_{1.2}}$. Moreover, this SLR is interested in the benefits that these technologies provide for the use case of firmware updates. Chosen selection criteria are the basis of the data extraction in Section 4.4. The EC focuses on recent literature in English language. Additionally, the chosen EC focuses on papers with detailed descriptions of the implementations of aforementioned technologies and, therefore, excludes literature that is shorter than

five pages. The combination of chosen inclusion and exclusion criteria should provide literature that covers blockchain- and DT-enabled firmware updates in an IoT environment in detail.

4.3 Literature Selection

The literature selection process is presented in Table 2. The initial literature was searched using the aforementioned search keywords and queries, which resulted in 301 papers in total. This is the sum of papers from both previously stated databases with included duplicates. After applying the exclusion criteria $EC_{1\text{-}4}$, 246 papers remained, counting the duplicates. The 246 remaining papers were checked for duplicate papers, where duplicate papers were filtered out, leaving 188 papers.

Inclusion criteria IC_1, IC_2 and IC_3 were applied for the unique papers based on abstract where, during the process, according to EC_3 and EC_4, literature shorter than 5 pages as well as grey literature were excluded leaving 35 papers. Snowballing and reverse snowballing methods were used to find additional related papers, resulting in 45 papers. After full-text reading 21 final papers remained.

Table 2. Literature selection process and count of papers after each step.

Process	**Count**
Search results based on initial queries	301
Apply EC_1, EC_2, EC_3 and EC_4	246
Remove duplicates	188
Filter by IC_1, IC_2 and IC_3 based on abstract. Apply EC_3 and EC_4	35
Snowballing and reverse snowballing	45
Filter by IC_1, IC_2 and IC_3 based on full-text reading	21

4.4 Data Extraction Strategy

Data extraction items for review sub-question $\mathbf{RQ_{1.1}}$ were chosen for two categories. Firstly, to give an overview of the design goals, benefits, and features provided by the architecture that integrates blockchain. The items are listed and a short description is given in Table 3. Furthermore, a short summary of the implementation is compiled during data extraction to explain the different approaches taken that enable the features extracted. Secondly, the use case and specific blockchain-related information was chosen, as outlined in Table 4. This was done to offer a detailed perspective on blockchain technology implementation as a part of the process of updating firmware. The data extraction items for sub-question $\mathbf{RQ_{1.2}}$ are compiled to find which DT technologies

are used, what role they play in firmware update life-cycle, and which industry they are applied in. The items are listed in Table 5 with their corresponding descriptions.

Table 3. Design goals of blockchain-enabled firmware update data extraction items targeting question $\mathbf{RQ_{1.1}}$.

Item	Description
Reference	Research paper reference
Firmware integrity	Integrity of delivered firmware update data
High availability	Receiving the firmware update data does not depend on a single source
Firmware authenticity	Firmware data can be authenticated by the receiving end device
Efficient key generation	Primarily important for resource-constrained devices for encryption and decryption functionalities
Financial incentive	System specifies blockchain-based financial incentives upon proof of delivery
Firmware update records	The status and history of updates are stored on the blockchain
Privacy	Privacy of end nodes is considered as part of the system

Table 4. Blockchain information and use case data extraction items for $\mathbf{RQ_{1.1}}$.

Item	Description
Use case	The context to which the system was intended for
Blockchain	Blockchain technology
Blockchain type	Permissioned, public, private, etc
Consensus	Consensus algorithm
Smart Contract	Whether the system uses smart contracts
Cryptography	Cryptography methods used in the framework to achieve firmware authenticity
Reference	Research paper reference

4.5 SLR Results

The literature review for $\mathbf{RQ_{1.1}}$ shows that a considerable amount of literature was dedicated to solving the IoT end device firmware updates via a blockchain-enabled system. Namely, blockchain-enabled firmware update-related papers were 15 out of 21 total papers. The results for data extractions are given in Table 6 for design goals and

in Table 7 for the blockchain attributes. Literature in the context of firmware updates using DT as a part of the system was found in 6 papers. The results of data extraction for question $\mathbf{RQ_{1.2}}$ are presented in Table 8. The next subsections contain the results for both literature review sub-questions as well as individual literature summaries.

Table 5. DT as part of firmware update data extraction items targeting question $\mathbf{RQ_{1.2}}$.

Item	**Description**
Reference	Research paper reference
Use case	The industry where the DT technology was researched or applied.
Life-cycle stage	which stage of the firmware update was the DT used in
Purpose	Core role or feature DT is implemented within the system
DT technology	Specific DT technology mentioned in the literature

4.5.1 $\mathbf{RQ_{1.1}}$: How Does Blockchain Contribute to the Firmware Updates in the IoT Systems?

The results of the data extraction for design goals and features presented in the Table 6 show, that the main benefits provided by the blockchain-enabled firmware updates were firmware integrity and firmware authenticity of which only 3 papers focused on efficient key-generation targeting resource-constrained devices. Although blockchain can provide auditability through storing the firmware version statuses, as well as firmware update process information, as an immutable record, only 9 out of 15 reviewed papers implemented the feature and stored records of firmware statuses on the blockchain. One of the justifications for using blockchain as a part of the firmware update process was to eliminate the issues of the currently used client-server update system, removing the problem of a single point of failure. Nevertheless, high availability was found to be the priority of 8 reviewed papers. High availability was achieved by using Swarm [51], IPFS [47] [36] [43], BitTorrent [29], unspecified decentralized storage network [31] with a suggestion to use IPFS or BitTorrent. Smart Contracts are a crucial element for establishing financial incentives. This integration is featured in five of the articles examined. Only 3 papers focused on privacy of the end-node or IoT device.

The majority chose an established blockchain with a smart contract feature like Ethereum as a part of the system as seen in Table 7. Works using Ethereum did not explicitly state the type of the blockchain. So the known "public" type of Ethereum was used in this SLR. The type of the blockchain was specified other than "public" for Ethereum if the work specifically stated it. Some of the papers do not explicitly specify the consensus mechanism used when the blockchain was used as Ethereum. This is due to the fact that little focus was placed on the consensus mechanism in the papers in general. The main focus was placed on the architecture of the firmware update

process using Ethereum. The consensus mechanism of Ethereum was updated from proof-of-work to proof-of-stake (PoS) in September 2022. In these cases, the consensus mechanism was deducted from the date of the paper. Some of the papers did not specify the consensus mechanism of the network in the proposed architecture. In these cases, the Ethereum PoW or PoA based on the date was used. Furthermore, systems based on a consortium blockchain like [4] did not mention any consensus mechanism; a PoA-like consensus is achieved in which only selected nodes can verify the blocks similarly to [43]. In addition to PoA, [29] uses a PoW consensus mechanism implemented on a custom blockchain. A custom blockchain implementation in [21] had the consensus mechanisms PoW, PoS, or PBFT given as a choice without a concrete implementation. A Kafka CFT-based consensus mechanism is launched in [22] as a transaction ordering service and Raft is used in [42] with both papers using a Hyperledger Fabric blockchain. The choice of cryptographic methods was mainly different for papers which considered resource-constrained devices as a part of their system using Ciphertext-Policy Attribute-Based Encryption (CP-ABE) in [4] [47] and Double Authentication Prevention Signature (DAPS) and Outsourced Attribute-Based Signature (OABS) in [57]. The use cases ranged from IoT, Wireless Sensor Networks (WSN), and LoRaWAN to a smart city in Ethereum blockchain implementations. Consortium and custom blockchains were used for autonomous vehicles, IoT, IIoT, and embedded devices networks. Hyperledger Fabric was the choice of blockchain for IoT and smart city systems.

Table 6. System design goals data extraction results for **RQ$_{1.1}$** based on Table 3.

Reference	Firmware integrity	High availability	Firmware authenti-city	Efficient key-generation	Financial incentive	Firmware update records	Privacy
[51]	✓	✓	✓				
[47]	✓	✓	✓	✓	✓	✓	✓
[53]	✓	✓	✓				
[57]	✓		✓	✓	✓	✓	✓
[14]	✓		✓			✓	✓
[29]	✓	✓	✓				
[56]	✓		✓				
[31]	✓	✓	✓		✓	✓	
[30]	✓		✓		✓	✓	
[36]	✓	✓	✓			✓	
[42]	✓		✓			✓	
[4]	✓	✓	✓	✓	✓	✓	
[43]	✓	✓	✓				
[21]	✓		✓				
[22]	✓					✓	

Witano et al. [53] developed a blockchain-based firmware update protocol compliant with the Open Connectivity Foundation specifications to bolster patch integrity and security. The protocol engages three entities: the Manufacturer (a full node) creating smart contracts with update metadata, the IoT Gateway (a lightweight node) retrieving and downloading the update, and the IoT device which, upon a client's update request,

Table 7. Blockchain and use case data extraction results for $\mathbf{RQ_{1.1}}$ based on Table 4.

Blockchain	Blockchain type	Consensus	Smart Contract	Cryptography	Use case	Ref.
Ethereum	Public	PoW	✓	PKI	IoT	[51], [53], [56]
					WSN / IoT	[30]
				DAPS / OABS / PKI	IoT	[57]
				PKI / Symmetric	LoRaWAN	[36]
		PoS	✓	CP-ABE / PKI	IoT	[47]
	Private	PoW	✓	PKI / Symmetric	Smart City	[14]
		PoA	✓	PKI	IoT	[31]
Custom blockchain	Permissioned	Not covered	✓	CP-ABE / PKI	Autonomous Vehicles	[4]
		PoA	✓	PKI	IoT	[43]
		PoW, PoS, or PBFT		PKI	IIoT	[21]
	Public	PoW and PoA		PKI	Embedded Devices	[29]
Hyperledger Fabric	Private	Raft		PKI	IoT	[42]
	Permissioned	Kafka CFT-based	✓	Not covered	Smart City	[22]

checks for and receives the patch via the IoT Gateway. Notably, the IoT device's firmware state is not stored on the blockchain.

This work [51] proposes a firmware update protocol that aims to provide firmware integrity, malicious code resistance, DDoS resistance and mitigate bandwidth issues. The system architecture consists of a genesis node, which creates the smart contract determining the firmware update upload initiator. The manufacturer node uploads the file to the distributed server with the help of other nodes. The nodes perform an out-of-chain antivirus and return the result to the smart contract. Smart contract assigns a node to upload the scanned file to the distributed server and save the address to the smart contract. IoT devices can request available updates and file addresses from the nodes for download. The IoT device firmware state is not stored on the blockchain.

Another work of [47] proposes a framework designed to enhance the security and efficiency of firmware updates, offering features like confidentiality, integrity, authenticated updates, streamlined authentication, and robust availability. The manufacturer announces updates on the blockchain and uploads them to a decentralized cloud storage system. Updates are delivered to IoT devices through a smart contract, which also ensures secure payment upon successful delivery. IoT devices decrypt updates using encryption based on device attributes and signal successful installation to the IoT owner, who then confirms it on the blockchain. This system provides a secure, transparent, and efficient method for managing firmware updates and transactions.

Zhao et al. work [57] introduces a privacy-focused model for software updates in IoT devices using smart contracts, ensuring unforgeable proof-of-delivery, fairness, authentication, and integrity. The IoT device vendor manages a list of devices, assigning a unique secret key to each. Updates are distributed through a smart contract, with a transmission node responsible for downloading and encrypting the update before sending it to gateways hosting the IoT devices. A double authentication preventing signature and attribute-based signature are employed for secure verification and decryption between the transmission node and the IoT device. Upon successful delivery, the transmission node is rewarded through the smart contract, reinforcing the system's security and efficiency.

Gong et al. [14] presented a system that aims to provide confidentiality, integrity, availability, auditability, and authentication capabilities. It includes a vendor node, blockchain nodes, a management node, and two types of end nodes, one of which is resource-constrained. End nodes periodically transmit encrypted messages detailing their firmware status and update requests to the management node. Firmware is obtained directly from the vendor, and the system records firmware statuses on the blockchain for improved security and tracking.

A firmware update scheme by paper [29] outlined a custom blockchain network that includes various nodes: a standard node (an embedded device), a blockchain node, a vendor-operated verification node, and a vendor node. In this system, the standard node generates an authenticated firmware update request using a pre-shared public key. This request is sent across the network, leading to the receipt of a metadata file containing details for firmware downloading. The standard node then proceeds to download the firmware from network participants using a peer-to-peer mechanism, streamlining the update process.

The paper's [56] objective is to address and mitigate various security threats, including firmware modification attacks, impersonation attacks, man-in-the-middle attacks, and replay attacks. The framework it proposes involves five key participants: the vendor, a broker, blockchain nodes, an IoT gateway, and the IoT device itself. Updates are initiated in two ways: either through a push request from the vendor via a smart contract, which blockchain nodes verify, or a pull request directly from the IoT device through the IoT gateway. The firmware is then sourced by the IoT gateway from either the vendor or a third-party repository, as directed by the vendor's smart contract, and subsequently transmitted to the IoT device.

To address the issue of file availability in peer-to-peer sharing, especially for less popular files, paper [31] introduces a protocol that offers a solution by incentivizing independent peers. This incentive is provided through cryptocurrency payments, facilitated by smart contracts initiated by the vendor, in exchange for a proof-of-distribution submitted by the distributor. The protocol employs a modified version of the Zero-Knowledge Contingent Payments (ZKCP) protocol, which resolves the fair exchange problem. Within this system, the IoT device holds only a portion of the blockchain data and relies on information from a trusted full node that participates in the blockchain network. This approach ensures both the efficiency of the network and the integrity of the transactions.

Paper [30] proposes an incentivized system for software update delivery and target device discovery to overcome the performance issues of OTA delivery of software updates and lossy channels of wireless communication. The software update patch is encrypted by the vendor and decryptable by the recipient using a preshared key. Double encryption is used between the transporter and receiver. The receiver can decrypt the patch by initializing a smart contract receipt for payment. The transporter can then reveal the key in order to receive the payment.

The security framework in paper [36] is designed to guarantee integrity, confidentiality, availability, and authentication, with a specific emphasis on low-powered devices that have limited resources in a Long-Range Wide Area Network (LoRaWAN) architecture. A firmware update service (FUS) and manufacturer create smart contracts with firmware update metadata. The firmware is deployed by the manufacturer to the IPFS network. FUS can then send the encrypted firmware to the LoRaWAN network, which in turn sends it to the gateway of end devices. The device decrypts, verifies, and installs the new firmware. The update status is stored in the blockchain smart contract.

Seo et al. [42] propose a blockchain-based software update framework where IoT devices without an internet connection can be automatically updated using UAV verifying each other using the public key and bloom filter. Participants vendor, gateway, IoT device, and UAV register in the private blockchain network, verified by the membership service provider, share and record the firmware statuses.

A consortium blockchain consisting of manufacturer nodes, who have the write permissions on the blockchain and are responsible for update initiation and AVs participating in the blockchain to receive updates as specified by smart contracts is proposed in paper [4]. The system takes advantage of the CP-ABE encryption in order to decrypt the firmware update messages and zk-SNARK for proof of delivery with an incentive. AVs can also share the updates with other AVs within the system and gain a reputation.

Another permissioned consortium blockchain network is proposed in [43] with a key generation center that manages the public keys of participants and revokes accesses of malicious participants. The author node uploads the firmware update manifest file to the blockchain using a verification node and the firmware data to IPFS. The IoT device periodically checks for updates from a verification node authenticated by its public key.

He et al. work [21] proposes a firmware update architecture in an IIoT environment with a monitoring module that receives and compares trusted snapshots that are generated by software developers and stored on the blockchain. The IIoT devices receive their updates via gateways to reduce the computational overhead of a blockchain and report the firmware update results to an administrator which communicates with the monitoring module to detect system anomalies.

A firmware update system of [22] uses HLF. Using smart contracts, the vendor pushes a transaction with the firmware hash to the blockchain which is later transferred to the IoT device. The IoT device can then verify the firmware file using the smart contracts verify operation and send the firmware update status to the blockchain. The status can be later queried from the blockchain by the vendor.

4.5.2 $RQ_{1.2}$: How Does DT Contribute to the Firmware Updates in the IoT Systems?

DT literature mainly discussing the firmware update process was limited and a total of 6 related papers were found. DTs are incorporated into the life-cycle of firmware updates

mainly for two reasons: pre-deployment testing, post-deployment for monitoring, or both. In the pre-deployment stage, the DT was mainly used by supplying pre-existing data from the previous system inputs and outputs or testing by simulation. Pre-deployment was used in all reviewed literature. The post-deployment stage was mainly used for monitoring current system statuses and resolving system failures or rollbacks. Testing the new firmware was the purpose of all reviewed papers. DTs as cost-reduction mechanisms, were discussed in [3] and [6]. Monitoring was considered as functionality in literature where DT was used in post-deployment scenario [6] [17] [19] [52]. DTs were used for dependency information specifically in [19] where a knowledge graph was created to ease monitoring. Although testing the behavior of the system before and after deployment was the main focus of all considered papers, security was specifically mentioned in 4 of the reviewed documents [3] [17] [19] and [11].

Table 8. DT data extraction results for **$RQ_{1.2}$** based on Table 5.

Reference	**Use case**	**Life-cycle stage**	**Purpose**	**DT technology**
[3]	IoT	pre-deployment	cost-reduction, security, testing	not discussed
[6]	IIoT	pre- and post-deployment	cost-reduction, testing, monitoring	Azure
[17]	IoV	pre- and post-deployment	testing, security, monitoring	CARLA
[19]	Smart Home	pre- and post-deployment	security, testing, monitoring, dependency management	Orange Thing'in
[11]	IIoT	pre-deployment	security, testing	not discussed
[52]	IoV	pre- and post-deployment	testing, monitoring, root cause analysis	SimulIDE

An overview of the OTA software updates in the IIoT was analyzed in [3] to show which parts are most updated after device deployment, provides a step-by-step approach for software updates in the IoT system and quantifies the energy cost of each step. The software update process is divided into two phases: Software module management and secure software rollout. DT was used for pre-deployment behavioral verification as a part of the first phase to see if the network behaves as expected in a simulation. DT can be used to identify bugs, version incompatibilities, and performance issues before new software rollout.

The challenge of monitoring the setups of different customer environments initiated

authors of [6] to propose a method for keeping the DT and virtual representations of the environments up to date by the example of relays. The firmware update of a relay device notifies the update status on every step to the DT keeping it updated. The DT can be used to simulate the real device for later product testing.

Paper [17] presents a continuous, contract-based strategy for designing, validating, and deploying modular updates in automotive systems with diverse variations. This approach encompasses both a structured process and a comprehensive methodology tailored to enhance the adaptability and efficiency of updates in complex automotive environments. They propose a UPDateable Automotive Test dEmonstratoR (UPDATER), which serves as a prototype for developing, deploying, and monitoring automotive OTA updates. DT is implemented to test the system during the verification phase to see if the system, after a firmware change, behaves according to the predefined contract.

Firmware updates as a part of the framework, [19] are using DTs to build a decision-support framework for inferring a topology of threatening dependencies in IoT systems in the context of device management. A knowledge graph of dependencies is exposed as a DT feature and represents current devices and dependencies, allowing for monitoring and system failure resolution.

DT application for mitigating potential attacks and malicious firmware updates is discussed in [11]. Before the firmware update takes place, the new firmware is first emulated on a virtual Active Neutral Point Clamped (ANPC) inverter with signals from the Digital Signal Processor (DSP). If the new firmware is malicious, it is rejected or applied if not malicious.

In the context of software-defined systems within the AV industry, a firmware update concept based on DTs is included in the architecture in [52]. Data analysis is divided into two parts, direct comparison of pre- and post-update data and using simulations that imitate the intended behaviour and actual outcome comparison. This enables the testing of the new firmware before production and conduction of root cause analysis.

4.6 Summary

Integrating blockchain into the process of firmware updates in an IoT system mainly provides assurance of firmware integrity, firmware authenticity, high availability, and the ability to store immutable update records for later verification or monitoring. The technology of smart contracts enables a built-in mechanism for updating metadata transactions and the network participants to gain financial incentives from active participation in the firmware update process. Permissioned, private, and public networks are suitable with different consensus mechanisms depending on the framework setup. Although, in many cases, the consensus mechanism was not directly specified. The use case differs based on the framework where an IoT network with restrained resource devices can take advantage of encryption mechanisms that are computation-efficient or delegate the computation-intensive tasks to their respective trusted gateways.

Using DTs in the context of firmware updates is limited. DTs are used for testing and monitoring the firmware update during the whole deployment life-cycle. Pre-deployment phase offers a cost-effective way of testing the firmware in a virtual environment under different scenarios with either simulated or preexisting data. Continuous monitoring of the firmware changes can be done post-deployment phase to notice and solve any further issues. Testing in a virtual environment also offers a containerized mechanism to notice any security and other issues related to the firmware update. The proposed solution in Section 5 uses findings from this section as well as Section 3 to create an OTA firmware update process in the context of IoV. These findings include the requirements for a firmware update in the IoV context, proposed solutions from different papers, and the benefits that blockchain and DT provide.

5 Proposed Solution

In this section, we aim to answer the question $\mathbf{RQ_3}$: **How to implement blockchain and DT-based OTA firmware update?**. The proposed solution is the design-science artifact according to **DG.1** from Table 1. The design of the solution is communicated by providing high-level overviews via diagrams according to **DG.7** from Table 1 to provide a clear understanding of the solution. We are going to provide a detailed discussion of the proposed OTA firmware update solution for vehicle ECUs in an IoV system. This section is the precursor for the evaluation in Section 6, where the implementation components are evaluated. In Section 5.1, we describe the components and entities of the system and provide a description of their roles and functions. In Section 5.2, assumptions are given for the components and processes to scope the proposed solution. Additionally, the design goals for the proposed system are stated. Section 5.3 explains the high-level architecture as well as the smart contract and the firmware update process. Section 5.4 describes the implementation. The implementation defines how IPFS, blockchain network, and DT are set up. Lastly, the console applications are implemented as simulations of the system components like vehicle, manufacturer, and DT.

5.1 Component Descriptions

This section outlines the key components involved in the OTA firmware update process for vehicles in the proposed solution. It describes the roles and functionalities of each component. The overview includes a variety of elements, from the vehicle and its ECUs to supporting technologies like blockchain nodes and IPFS servers. These components are referenced directly and as a conceptual entity. The components of the system are listed below:

- **Vehicle**: Vehicle is an actor of the system with multiple ECUs that are subject to

the OTA firmware update. The vehicle in the proposed system is composed of the gateway ECU and secondary ECU.

- **Gateway ECU**: Otherwise known as Telematics Control Unit, is the ECU within the vehicle that provides connectivity services between the vehicle and external networks. In this case, the trusted blockchain nodes and IPFS server. The gateway ECU is responsible for the connectivity to IPFS and blockchain and firmware package verification and redistribution to other (secondary) ECUs. Gateway ECU is also responsible for verification of the installation status of the firmware package. The vehicle manufacturer controls the private and public key of the gateway ECU.
- **Secondary ECU**: This ECU is a module within the vehicle that is the end recipient of the firmware update.
- **Ethereum node**: Trusted Ethereum blockchain node that the vehicle queries firmware update information from and be used for updating the smart contract. The blockchain node holds the blockchain state and is able to use the Ethereum contract functionality.
- **Vehicle manufacturer**: The vehicle manufacturer is the manufacturer of the vehicle and issuer of blockchain contracts that specify the details for firmware updates. The manufacturer is also responsible for generating the firmware update packages for the vehicle and adding them to the smart contract.
- **Vendor**: Vendor of the vehicle ECU. The Vehicle can consist of ECUs from different vendors not limited to the vehicle manufacturer. The vendor provides the ECU firmware data to the vehicle manufacturer.
- **Digital Twin**: The virtual copy of the vehicle. DT is updated based on the blockchain state. DT instances is created for the vehicle and the secondary ECU.
- **IPFS node**: IPFS is used as the storage for firmware data and the trusted IPFS node is used for adding and retrieving the firmware data.
- **Smart contract**: Ethereum smart contract specifies the firmware update order and holds the firmware metadata. The smart contract and the blockchain also serve as a record-keeping mechanism for firmware updates.
- **Package**: Firmware update units is referred to as packages. The package specifies the metadata for the firmware update like encryption details and firmware data locations. The package is also referred to as the metadata stored on the smart contract.

5.2 Design Goals

As can be seen from Table 6 the most value to be gained from a blockchain-enabled firmware update is from gaining firmware integrity, high availability, firmware authenticity and retaining firmware update records. These values are also reflected at the end of Section 3.2. These are the areas the implementation is going to focus on. High availability and firmware update record features are achieved by using IPFS and blockchain. The implementation uses public-private key pairs for firmware signing and verification as well as encrypting symmetric keys to encrypt the firmware data ensuring firmware authenticity and integrity. The smart contract allows only authenticated updates to it. The state-changing functions of the smart contract is limited to the manufacturer and the vehicle. The vehicle can check if the correct package was received and receive update statuses from ECU. Based on the findings from Section 4.5.2, the most value gained from DTs is monitoring and testing. In order to improve the status monitoring of the firmware update, a DT of the vehicle is created in the Microsoft Azure environment. The manufacturer is able to update the DT based on the smart contract state. DT provides oversight of the update near real-time by listening to events of the smart contract on the blockchain. Additionally, the DT instance can be used to simulate the firmware update for testing in a virtual environment.

Each vehicle has a corresponding smart contract. The smart contract stores the firmware update metadata and the vehicle verification of the update in its state. A single firmware package may contain updates for multiple ECUs. This potentially means that the updates can be tested and deployed as a package to increase the reliability and functionality of the vehicle as a whole. The firmware metadata for each ECU can be created by the vendor of the ECU and packaged by the manufacturer. The firmware packages have links referencing the previous and next firmware packages to achieve ordering for the updates. These links can be changed by the manufacturer by adding new packages. If the vehicle encounters an error, it does not continue the update until a new package is added to resolve the issue. The implementation relies on the following assumptions and requirements:

- **Trusted blockchain node**: The vehicle does not have the full or partial blockchain state. It uses a trusted blockchain node in order to fetch the updates and update the smart contract. The gateway ECU has secure access to one or many trusted blockchain node(s). The communication between the gateway ECU and the blockchain should ideally employ a Transport Layer Security (TLS) connection to mitigate man-in-the-middle (MiTM) attacks, ideally, complemented by an authentication method to ensure authorized access. The list of addresses for trusted blockchain nodes may be added and removed as required; however, establishing a connection to a trusted node is desirable for guaranteeing the integrity of the data, ensuring it remains unaltered and reliable. The secure communication channel

is a desirable condition for production environments, but is not reflected in the evaluation simulations.

- **Encryption**: The ECUs in the vehicle have enough computational capacity to perform encryption, decryption, verification, and signing the firmware metadata and firmware data. The key generation process for the evaluation implementation is described in Section 5.4.3. Encryption methods for resource-constrained ECUs were not considered. This is a consideration due to the fact that in a real scenario, some ECUs might not employ encryption at all and rely on a separate ECU to handle the load of the computation. The on-board vehicle ECU variants can be multiple and have to be accounted for separately based on the case. These scenarios are not considered in the proposed solution and evaluation.

- **Preshared values**: The system relies on preshared keys for all ECUs in a vehicle that participate in the firmware update. Sensitive values are stored securely and inaccessible from any third party, for example in a hardware security module (HSM) in the production. The gateway ECU stores the manufacturer's public key, its own private key, and the address of the smart contract. Secondary ECUs store the vendor's public key and their own private key. The vendor has access to its own private key and the public keys of the ECUs they issue. The manufacturer stores its own private key, the public keys of vehicle gateway ECUs, and the smart contract addresses that correspond to its vehicles.

- **Firmware installation**: The firmware installation implementation in this thesis is simulated. If the firmware is verified, the integrity and authenticity are checked and the firmware can be decrypted by the ECU, the firmware installation is considered to be successful.

5.3 Architecture

This section describes the architecture of the OTA firmware update implementation. The high-level overview of the components and their relations can be seen on Fig. 5. The components are categorized into storage, virtual environment, physical environment, manufacturer, and vendor. Storage consists of two systems: IPFS and blockchain with each corresponding trusted node. The physical environment encapsulates the vehicle and the major components (ECU and gateway ECU) of the vehicle. The DTs belong to the virtual environment where each entity of the vehicle from the physical environment has a corresponding virtual copy.

The communication channels show the unordered data flow between the components of the system on Fig. 5. According to the DT architecture, the flow of the data should not only include storage and service, but bidirectional flow between the physical and virtual environment as seen on Fig. 2. This is left out of the current solution architecture because

it is not relevant to the firmware update flow. However, any other communication might still happen between the physical and virtual entities which is out of the scope of this work. Communication channel (a) on Fig. 5 is used by the vendor to share the metadata and encrypted data of the firmware update for ECUs with the manufacturer. Communication channel (b) marks the communication between the manufacturer and the firmware data storage system. Manufacturer uses IPFS to store the firmware data and smart contracts on the Ethereum blockchain to store the metadata of the firmware package. Communication channel (c) between the manufacturer and virtual environment is important to notify the DT of the status of the update. The DT itself cannot decrypt the verification data of the update status by the vehicle, it can only be decrypted by the manufacturer. If channel (c) produces a successful update message, the virtual environment can use channel (d) to update the firmware based on the storage state. Channel (e) is used by the physical vehicle to query firmware metadata from the smart contract and firmware data in IPFS.

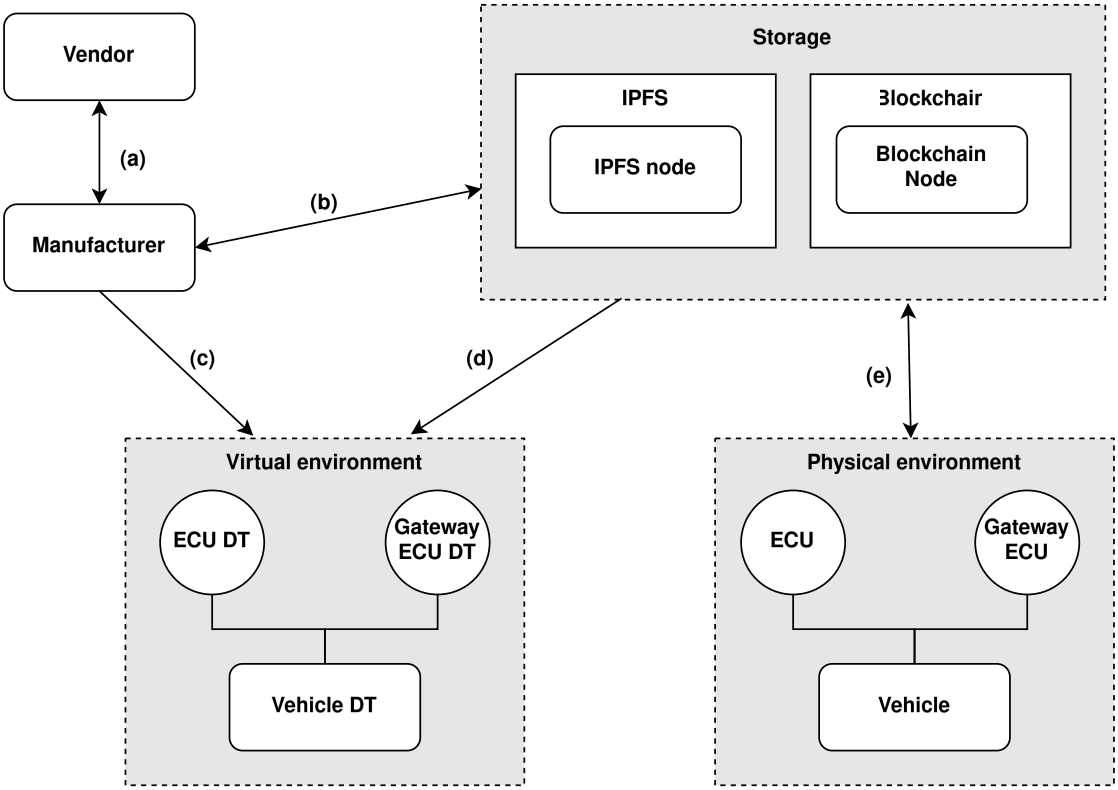

Figure 5. Proposed solution implementation architecture overview showing components and data flows between components.

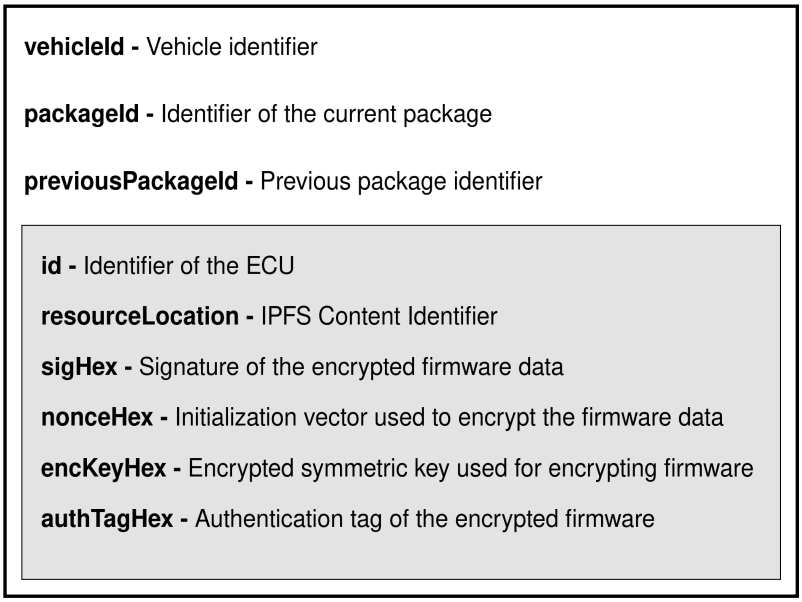

Figure 6. Proposed solution firmware package metadata structure.

5.3.1 Firmware Metadata

The firmware metadata is stored on the smart contract within a package. Each package and thus metadata entry in the smart contract corresponds to a single firmware update package for the vehicle. The metadata contains fields that are used to confirm that the firmware update is meant for the specified vehicle. Secondly, it provides a way to verify that the order of the package is correct. Other fields provide the location, decryption, and authentication of the data.

The schema of the metadata is presented in Fig. 6. At the top level, the metadata has information about the vehicle ID (*vehicleId*) assuring that the metadata corresponds with the vehicle. Additionally, it stores the current package ID (*packageId*) and the previous package ID (*previousPackageId*) to verify the package order to ensure that the package requested from the blockchain node corresponds to the package ID in metadata. Because the order is mainly defined in the smart contract state, there has to be a way to verify, that the requested package is actually received and that the package received is actually the next in order for the firmware updates. The vehicle checks if the previous package ID corresponds to the currently installed package on the vehicle. In case the response from the Ethereum node is incorrect, the vehicle does not proceed with the update.

Secondary ECU metadata is stored as an array of records where a single entry is shown with a grey background in Fig. 6. Each array entry corresponds to a separate ECU. Using the *id* field, the vehicle can check if the ECU is installed on the vehicle. The *resourceLocation* field holds the location of the firmware data. In our case, it is the

IPFS content identifier. The *sigHex* field stores the signature of the encrypted firmware, signed by the vendor. Field *nonceHex* is the initialization vector used in the encryption of the firmware data. Field *encKeyHex* is the symmetric encryption key, encrypted by the vendor using the ECU public key. The authentication tag is stored under *authTagHex*. All fields suffixed with "Hex" are base16 encoded. All the symmetric keys used are a random 32-byte value. The initialization vector is a random 12-byte value. The symmetric encryption is done using aes-256-gcm mode of operation. All metadata is encrypted by the manufacturer with a symmetric key. The details for decryption are stored on the firmware package in the smart contract. The format of the encryption details for the metadata in smart contract package is similar to the metadata encryption and is discussed in Section 5.3.2.

5.3.2 Solidity Smart Contract

The solidity smart contract is the main driver for the OTA firmware update metadata delivery and state. The contract is created by the vehicle manufacturer to set firmware update metadata for a specific vehicle and track the statuses of the update processing. During the creation of the contract, the blockchain address of the vehicle manufacturer and the blockchain address of the vehicle are stored on the contract. A firmware package can be added only by the vehicle manufacturer and is added to a linked list of packages within the contract where every package specifies the next package. These packages are available for the vehicle to query.

```
1 constructor(address vehicle_) {
2     owner = msg.sender; // Manufacturer
3     vehicle = vehicle_;
4     none = 0x00..;
5     initial = none;
6 }
```

Listing 1. Solidity smart contract constructor function.

The smart contract is developed as a hardhat project[3]. The full code of the smart contract can be found in the hardhat project via the link in Appenix I. The contract has 4 modifiers that allow only authorized usage of the contract. Modifier *onlyOwner* is applied to functions that are only allowed to be executed by the owner, the vehicle manufacturer. Similarly *onlyVehicle* allows function executions only by the vehicle. The third modifier *packageIdUnique* checks if the package ID to be added is unique in the contract. The modifier *onlyOwnerOrVehicle* allows both the manufacturer and vehicle to execute the functions. The contract constructor shown in Listing 1 on lines 1-6 is

[3] https://github.com/edgarmiadzieles/thesis_hardhat

initialized with a special *none* value set to a hexadecimal 32-byte 0 value. Ethereum solidity language has no null value, so the value represents an empty entry.

The firmware packages are stored in a mapping, where the key is the package ID, and the value is the firmware package, as seen on lines 21 and 32 in Listing 8. Combined, package and mapping consist of the following fields. The *firmwareMeta* field contains the firmware metadata shown in Fig. 6 and line 2 in Listing 8. The field is encrypted using symmetric key generated by the manufacturer. The *key* field, as shown in line 3 in Listing 8, is formed of initialization vector, authentication tag and encrypted symmetric key. The first 12 bytes are reserved for the nonce. The next 16 bytes are the authentication tag. The remaining 32 bytes are the encrypted symmetric key. The symmetric key is encrypted using the vehicle public key. Field *metadataSig* contains the bytes of a vehicle manufacturer signed sha256 hash of *firmwareMeta* field. The vehicle checks this information to verify that the package was indeed created by the manufacturer. The field *metadataSig* corresponds to line 4 in Listing 8. Field *packageId* is the unique identifier for the package is set on line 21 to the mapping on line 32 in Listing 8. The *status* (line 5 in Listing 8) can be PROCESSED or PENDING. The PENDING status packages are considered not yet installed. Packages with the status PROCESSED are processed by the vehicle. This means the vehicle has either installed the package successfully or processed the package resulting in an error. The status is accompanied by the *verification* field. When the vehicle has processed a package, it sets verification data, readable by the manufacturer to verify the state of PROCESSED packages. This field (line 6 on Listing 8) can indicate a successful install or error data related to the package processing. To increase the security, *verification* value can be encrypted by the vehicle using symmetric encryption and manufacturer public key. The decryption data can be part of the *verification* field value similar to the *key* value on line 3 on Listing 8. This won't be implemented in the evaluation, however, might be necessary for the production deployment. On line 7 in Listing 8, *nextId* specifies the next package to be processed, forming a linked list. This linked list formation is achieved by the function *addPackage* (shown on lines 11 - 30 in Listing 8). The function expects previous and next package IDs. The previous package ID is added as the *nextId* of the previous package and the next package ID points to the next package. This enables changing the order in case a package installation fails for the vehicle.

The functionality of the smart contract explained next can be found on the project[4]. Initially, the vehicle queries the *getInitial* function in order to get the initial update. If the update is the *none* value, the first package has not been added yet. If the initial package exists and has been processed, the function *getNextPackageId* can be queried with an argument of the last installed package ID. If any new packages have been added and linked, this function returns the next package ID. To get the package details *getPackage* function is used. The function returns the information necessary for the vehicle to

[4] `https://github.com/edgarmiadzieles/thesis_hardhat`

complete the update. Only the vehicle is able to execute the function *packageProcessed*, which marks the package with the requested ID into the PROCESSED status and adds the firmware update verification data. Using this function, the smart contract triggers an event notifying the manufacturer of the processed firmware update. The manufacturer can then use the *getPackageVerification* function to get the verification data of a specific package.

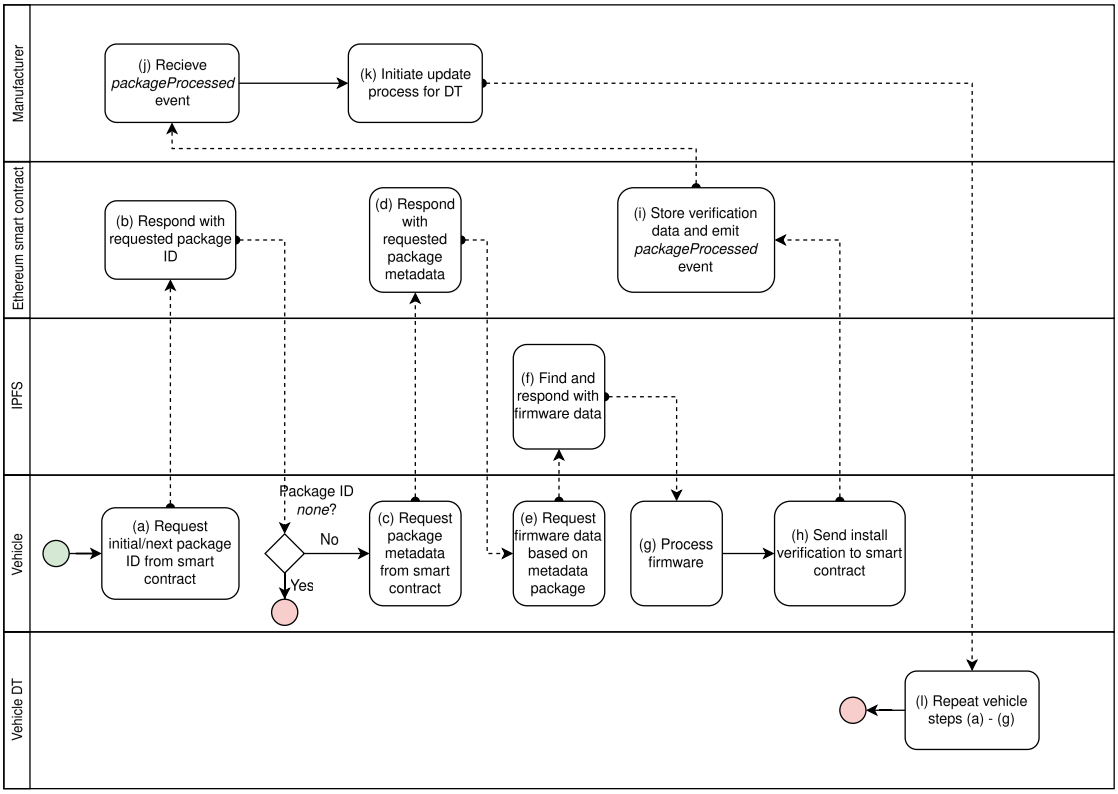

Figure 7. Proposed OTA firmware update solution sequence showing update process steps between major components of the system.

5.3.3 Update Process

This section describes the proposed process of OTA firmware update delivery for a physical vehicle and updating the DT based on verification data from the physical vehicle. The update process is shown in Fig. 7 as a sequence diagram. The update information is periodically requested by the vehicle for the initial or next package ID from the smart contract. This step is shown as step (a) in the diagram. The smart contract either responds with a "none" response or a package ID that is the next package to be processed, as shown

in step (b). If the package ID was a "none" response, the update sequence terminates and the process begins again after some timeout.

The package ID response is other than "none" when the vendor (which can also be the manufacturer) issues firmware for an ECU and shares the encrypted data and metadata with the manufacturer. The ECU metadata shared is described in Section 5.3.1 except the resource location (IPFS CID). The manufacturer receives the encrypted firmware and then uploads the firmware or multiple firmware data to IPFS. IPFS responds with a CID for each firmware data added to IPFS. The manufacturer creates the firmware package metadata described in Section 5.3.1. The package is created for the vehicle and added to the smart contract. The metadata package is compiled by adding the CID from IPFS by manufacturer. The manufacturer generates a new package ID which is added to the metadata along with a reference to the previous package ID and the vehicle ID. The metadata is encrypted using a symmetric key (line 3 on Listing 8). The authentication tag and nonce is stored within the *key* field on the package. The manufacturer adds the package using the smart contract of the vehicle with the key and signature of the encrypted firmware metadata under the same package ID that was added to the metadata. If the package is an intermediary package, then the next package ID can be added and linked to the next package. Otherwise, the next package ID has the "none" value.

Upon receiving a package ID response from the smart contract in step (b) on Fig. 7 that was other than "none" the vehicle can request the metadata based on the package ID in step (c). Smart contract responds with the firmware metadata, the key, signature, and status in step (d). If the package data is verified by the vehicle, it requests firmware data from IPFS in step (e). In step (f) IPFS responds with the firmware data based on the request from the vehicle. If the vehicle processes the update package in step (g), it compiles a verification for the manufacturer and call the smart contract to update the package *verification* field in step (h). Smart contract sends a *packageProcessed* event in step (i). The manufacturer listens to the smart contract event in step (j). The manufacturer can have an overview of the update processes of a vehicle and react to any failed firmware update. In step (k) the manufacturer can now choose to initiate the update process for the DT of the vehicle. Step (l) states that the vehicle DT repeats steps (a) - (g) similar to the vehicle to update its own firmware. DT omits the steps after (g) as they are only necessary for the physical vehicle and initiating the update sequence for the twin.

The same update process can be applied to the DT instead of the physical vehicle. This would omit steps after (g) and replace the vehicle with the DT. The IPFS and blockchain, in this case, would be in a separate environment created for the DT setup in order to analyze the process before applying the update to the physical vehicle. Similarly, real data on the blockchain and IPFS can be used to analyze the process in a DT setup.

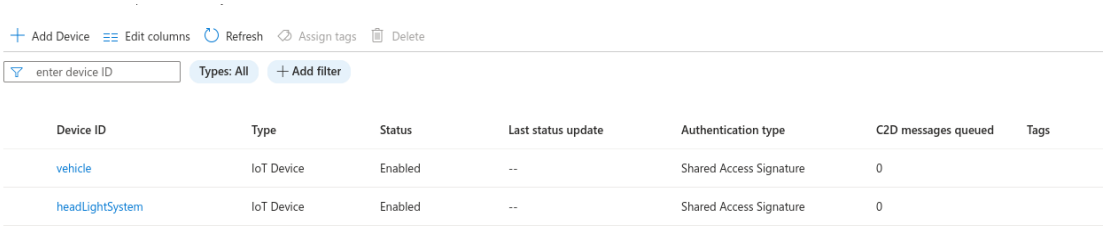

Figure 8. Vehicle and headLightSystem devices in Azure IoT Hub.

5.4 Implementation

In order to demonstrate the proposed OTA firmware update process, two use cases have been considered. Firstly, the process of updating the physical vehicle. This process is based on the description in Section 5.3.3 in which the vehicle receives an update and the manufacturer updates the DT. Secondly, a process where the vehicle in Fig. 7 is replaced with vehicle DT and steps after (g) are omitted. The second use case showcases the example of the update process where the physical vehicle is not considered and mainly for update testing purposes. A simple scenario for both use cases is going to be the same: A vehicle that has a secondary ECU which is a headlight system. The headlight system ECU is going to produce different beam intensities based on the simulated "firmware installation" and ambient light value. In the next subsections, we are going to discuss the Ethereum, IPFS (Section 5.4.1) and DT (Section 5.4.2) setup. Finally, console applications are going to be described in Section 5.4.3.

5.4.1 Ethereum and IPFS

The proposed solution relies on IPFS and Ethereum blockchain components. A local IPFS node was used to ease the testing and development. The local environment was considered sufficient to showcase the firmware update process. The IPFS node is created using docker "ipfs/kubo:latest"[5] image. The IPFS node startup script is shown in Listing 2. This script serves as the entry point: the default executable for the container. The node is assigned a unique swarm key on lines 2-4 in Listing 2 to create a separate network for the implementation. The swarm key is taken from the environment values. The API port 5001 and gateway port 8080 on lines 9 and 10 in Listing 2 are configured in the script and later exposed in docker. The script configures the IPFS node to allow PUT, POST, and GET methods from any origin on lines 13 and 14 in Listing 2. This allows the manufacturer and vehicle gateway ECU to create a connection to the node and use the HTTP RPC API of the IPFS node. The node is started on line 15 in Listing 2.

[5] `https://hub.docker.com/layers/ipfs/kubo/latest/images/sha256-1b1751061941d0d41bc034075435c9e48e912c6322558e329305b4b30fb65eac?context=explore`

```
!/bin/sh
echo "/key/swarm/psk/1.0.0/" > /data/ipfs/swarm.key
echo "/base16/" >> /data/ipfs/swarm.key
echo $SWARM_KEY >> /data/ipfs/swarm.key
if [ ! -f /data/ipfs/config ]; then
    ipfs init

    ipfs config Routing.Type dhtclient
    ipfs config Addresses.API /ip4/0.0.0.0/tcp/5001
    ipfs config Addresses.Gateway /ip4/0.0.0.0/tcp/8080
fi
ipfs bootstrap rm --all
ipfs config --json API.HTTPHeaders.Access-Control-Allow-Origin '["*"]
    '
ipfs config --json API.HTTPHeaders.Access-Control-Allow-Methods '["
    PUT", "GET", "POST"]'
exec ipfs daemon
```

Listing 2. IPFS docker image configuration startup script.

Ethereum blockchain implementation is based on Truffle Suite's Ganache[6]. Ganache creates a private Ethereum blockchain for local development and testing and has a graphical interface (Fig. 9). This is sufficient to implement the proposed solution to demonstrate the firmware update process as Ganache supports the features necessary for this demonstration and evaluation. These features include creating and using smart contract functions and listening to events of the smart contracts. Ganache sets up 10 accounts by default with 100 ETH initially as seen in Fig. 9. These accounts are used as the blockchain accounts for the manufacturer and the vehicle. Ganache exposes the network's RPC server endpoint which can be used to make connections to the node. The blocks in Ganache are auto-mined.

5.4.2 Digital Twin

The DT environment consists of a vehicle DT called *vehicle* and the headlight system ECU DT called *headLightSystem*. The implementation environment is realized using Microsoft's Azure platform and a screenshot of the nodes can be seen in Fig. 10 on the left. The *vehicle* has two properties: *lastUpdateState* and *version*. The *version* is the package ID value from the firmware update which shows the current version of the installed firmware package. Based on the status, the boolean *lastUpdateState* shows the last processed update state as true or false, meaning successful update or failed. The vehicle has a relation to the *headLightSystem*. Headlight system DT *headLightSystem* has 3 properties as pictured on Fig. 10 on the right: *ambientLight*, *beamIntensity* and *version*. These values are going to change based on updates from the console applications. The

[6] https://archive.trufflesuite.com/ganache/

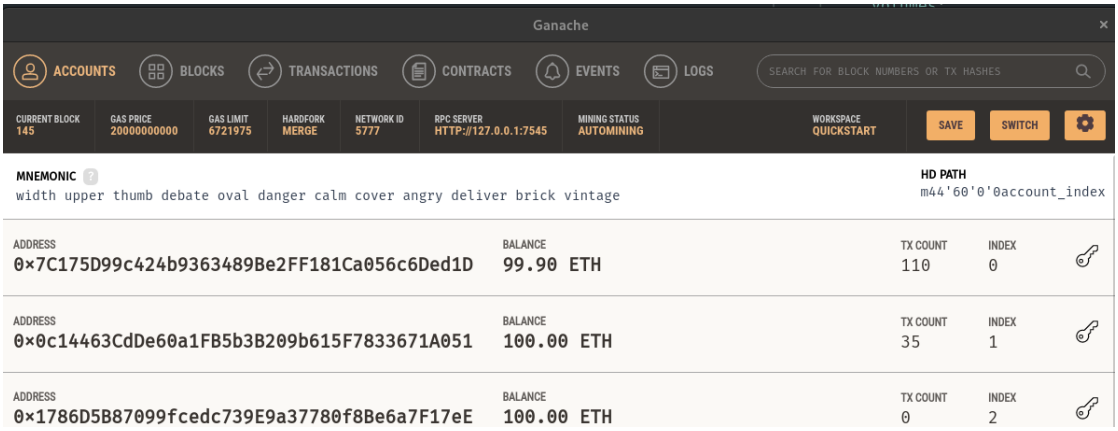

Figure 9. Screenshot of Ganache graphical interface which is used as the Ethereum blockchain for proposed solution implementation.

DT instances are defined using the Digital Twins Definition Language (DTDL). The definition is shown in Listing 5 where the headlight system DT is defined in lines 1-11 and the vehicle is defined on lines 12-22 with the previously mentioned properties. DT instances *vehicle* and *headLightSystem* are created out of their respective definitions.

```
1 import { Mqtt } from 'azure-iot-device-mqtt';
2 import { Client, Message } from 'azure-iot-device';
3 const client: Client = Client.fromConnectionString(
    deviceConnectionString, Mqtt);
4 const message: Message = new Message(JSON.stringify(telemetry));
5 client.sendEvent(message);
```

Listing 3. IoT device message event sending script to send messages to IoT devices in Azure.

Similarly, devices for vehicle DT and headLightSystem DT are created in Azure IoT Hub. The devices can be seen on the screenshot in the list on Fig. 11. The connections to these devices are going to be used to send messages from the console applications to the IoT Hub (Listing 3). The *deviceConnectionString* in Listing 3 on line 3 is the primary connection string of the device in IoT Hub found under device identity in Azure back office. The *telemetry* value on line 4 in Listing 3 represents the message being sent to the IoT Hub in JSON format. Two types of messages are going to be sent: a message that contains new values for *vehicle* DT properties and a message that contains values for the *headLightSystem* DT properties. The message is being set on line 5 in Listing 3. Sending the IoT device messages creates events in the Event Hub. Using the connection data from the endpoint configuration, messages can be consumed by Azure Function App.

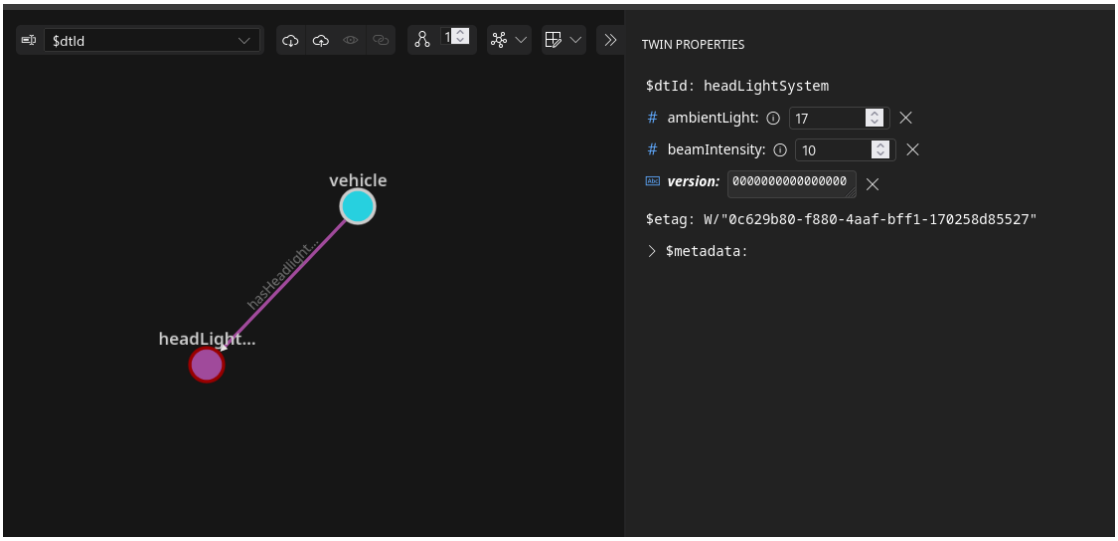

Figure 10. DT instances in Microsoft Azure Digital Twin Explorer with selected vehicle DT.

Azure Function App function is created to consume the message from Event Hub. The function is written in TypeScript and Visual Studio Code[7] code editor with Azure Functions extension installed. The function definition is shown in Listing 9. The function accepts Azure Event Hub messages and acts on messages that are related to IoT devices *vehicle* and *headLightSystem*. These messages are defined as types on lines 14 - 23 in Listing 9. The device context information is derived from the context argument on line 31 in Listing 9. Based on the device from the context, a type of either *headLightSystem* or *vehicle* is assigned to the message on line 35 or 56 based on the if statements on lines 34 or 55 in Listing 9. The values from the message, corresponding to each type, is extracted and mapped for the DT as the *replace* value function. Lastly, the associated DT is updated, as can be seen on lines 52 or 69 in Listing 9. The ID of the device in Azure IoT Hub and DT are the same. This way, the function can derive the device ID from the context and assign the value to the associated DT directly.

The full flow of updating the DT is shown in Fig. 12. The console application sends the message with either *vehicle* or *headLightSystem* properties data using the respective IoT Hub device connection. Azure IoT Hub has a default endpoint that directs the messages to Azure Event Hub. The function app then consumes the messages and updates the DT based to the message and context.

[7] `https://code.visualstudio.com/`

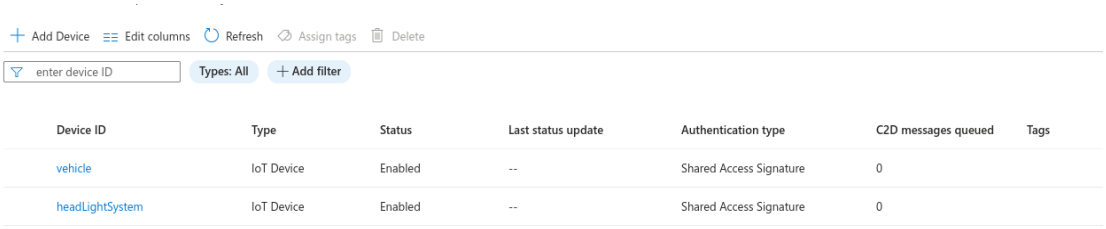

Figure 11. Vehicle and headLightSystem devices in Azure IoT Hub.

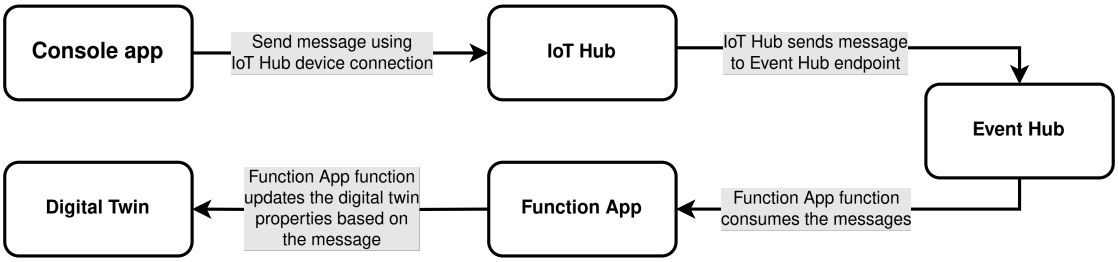

Figure 12. The process of updating DT properties from the console applications.

5.4.3 Console Applications

The console applications are used to simulate the vehicle ECU, vehicle gateway ECU, manufacturer, vendor, and the DT of the vehicle. The main function of the console applications is to showcase the update process and provide the ability to evaluate the process in a practical manner. The console application is part of a single project. The project link is available in Appenix I. The modules are generally divided into the module of the simulated entity and the script that uses the module instance and functions for the simulation. Console application scripts and modules are written in TypeScript. The project is compiled using "npx tsc". This produces compiled scripts and modules in the "dist/" folder of the project. Each script is executed separately, utilizing Node.js v20.11.1 as the execution environment. Node.js was chosen for the console applications because it supports Ethereum smart contract functionality using Ether.js library[8]. This setup allows for individual script testing and operation. The development and execution environment is a 64-bit Linux Fedora 36 workstation with 32 GiB of RAM and AMD Ryzen 5 3600X 12 thread processor.

Initially, the private and public keys are generated using the ssh-keygen tool using the command: "ssh-keygen -t rsa -b 2048 -m PEM". Command "ssh-keygen -f id_rsa.pub -e -m PEM > publickey.pem" is used to generate the public key in PEM format. Program

[8] https://docs.ethers.org/v6/

ssh-keygen[9] is an authentication key generation, management, and conversion utility. This process is repeated for the vendor, manufacturer, and the vehicle's ECUs. The values are stored for each entity in the ".env" file of the console application at the root of the project. The vendor public key is shared with the vehicle ECU. The manufacturer's public key is shared with the gateway ECU. The gateway ECU public key is shared with the manufacturer. The ECU public key is shared with the vendor. The private keys of the blockchain account for the vehicle and manufacturer are known only to the according entity and the public keys are known publicly for the manufacturer and vehicle.

The vendor module "src/vendor.ts" is responsible for creating the metadata for the vehicle ECU. The module has a single function *encryptFirmware* that accepts 3 arguments: ECU public key, its own private key, and hexadecimal encoded firmware data. The vendor script "src/vendor_script.ts" calls the vendor module function to generate and log the metadata, which is JSON formatted for convenience. The vendor's responsibility within the update flow is described in Section 5.3.3.

```
firmwareSim(num) {
  // Firmware is a hex encoded value which is parsed into integer.
  const firmwareNum = parseInt(this.firmware, 16)

  // Simulates the firmware, where num is the ambient light value and
  // the return value is the beamIntensity.
  if (firmwareNum > num) {
    return 10;
  } else {
    return 15;
  }
}
```

Listing 4. Firmware simulation function of the ECU.

The ECU module "src/ecu.ts" is the recipient of the firmware data that the vendor generated. The ECU module is initialized by the gateway ECU script "src/gateway_ecu.ts" to simulate the vehicle ECU and is also used as the DT representation separately. The ECU module has the following functionality. The *constructor* accepts the initial firmware version, the vendor's public key, its own private and public keys, and its ID. The initial firmware is hard-coded and set to "aa". This function is used to create the instance of the ECU. The *update* function is called in order to update the ECU instance. It accepts hex-encoded arguments which include the firmware, vendor-signed firmware signature, nonce, symmetric key used for encryption of the firmware, and authentication tag. The function tries to verify the firmware and check its integrity and authenticity. After verification, it decrypts the firmware and "installs" (updates the firmware variable of the module) and updates the version variable. The function *getVersion* allows to request the

[9] https://linux.die.net/man/1/ssh-keygen

current version of the firmware from the ECU. The function *firmwareSim* is a simple ECU firmware simulation function that takes an integer as the argument and compares it to the firmware variable. The code is shown in Listing 4. Since the firmware is a hex-encoded value, the firmware is parsed into an integer on line 3 in Listing 4. The function checks if the given number is lower than the parsed integer of firmware value on line 7 in Listing 4. For our use case of *headLightSystem*, the argument represents the value of *ambientLight* and the result of the function is the *beamIntensity*.

```
{
"@id": "dtmi:example:HeadlightSystem;1",
 "@type": "Interface",
 "displayName": "Headlight System",
 "contents": [
    {"@type":"Property","name":"ambientLight","schema":"integer"},
    {"@type":"Property","name":"beamIntensity","schema":"integer"},
    {"@type":"Property","name":"version","schema":"string" }
 ],
 "@context": "dtmi:dtdl:context;2"
}
{
 "@id": "dtmi:example:Vehicle;1",
 "@type": "Interface",
 "displayName": "Vehicle",
 "contents": [
    {"@type": "Relationship","name": "hasHeadlightSystem","target": "
       dtmi:example:HeadlightSystem;1"},
    {"@type": "Property","name": "version","schema": "string"},
    {"@type": "Property","name": "lastUpdateState","schema": "boolean
       "}
 ],
 "@context": "dtmi:dtdl:context;2"
}
```

Listing 5. DT definitions in Digital twins definition language used to create DTs in Azure.

Gateway ECU module "src/gateway_ecu.ts" is the central firmware update process mediator of the vehicle. It holds the state of the current firmware package ID, the secondary ECUs, and the smart contract information. The *constructor* of the module accepts all initialization information for the update process to be possible. It accepts the manufacturer's public key, its own private and public keys, the smart contract address, its own blockchain private key, the Ethereum provider instance, the Application Binary Interface (ABI) of the smart contract, secondary ECU instances, and the vehicle ID. The *getUpdate* function is used to retrieve the initial or next package ID from the smart contract. It uses the smart contract functions *getInitial* and *getNextPackageId*. The

function *getPackage* retrieves the package information from smart contract using smart contract's *getPackage* function. The function *parsePackage* parses package data from *getPackage* function. It also verifies the firmware metadata package signature and checks if the update is in PENDING or PROCESS status to decide whether to continue with the update. Finally, returns the decrypted metadata information. The function *checkVehicleId* checks if the vehicle ID in the metadata corresponds to the gateway ECU vehicle ID. The function *checkPackageId* checks if the metadata package ID field corresponds to the requested package ID from the smart contract. The function *checkPreviousPackage* checks if the metadata previous package ID corresponds with the currently installed package ID of the gateway ECU. The function *updateEcus* accepts the secondary ECU firmware metadata from the smart contract firmware metadata package and updates the secondary ECUs based on the information. Additionally, checks if the secondary ECU ID exists within the gateway ECU instance and downloads the firmware from IPFS based on secondary ECU metadata. The function *updateEcus* uses the secondary ecu *update* function to update the secondary ECU's firmware. The function *setPackageId* updates the package ID variable in the gateway ECU if the update is successful. The function *packageProcessed* calls the smart contract function *packageProcessed* after processing the firmware metadata package.

Gateway ECU script "src/gateway_ecu_script.ts" acts as the simulation of the physical vehicle. It initializes the gateway ECU and its secondary ECU. Then checks for an update multiple times in a minute. The flow of the execution is based on Section 5.3.3 and realizes the vehicle steps. First, it checks for a new package ID using *GatewayEcu.getUpdate*. The update process continues if the response includes a new package, otherwise restarts. Next, get the package using *GatewayEcu.getPackage*. Parses the package to get the metadata *GatewayEcu.parsePackage*. If the parsing fails, it restarts the process. Runs the package checks using *GatewayEcu.checkPackageId*, *GatewayEcu.checkPreviousPackage* and *GatewayEcu.checkVehicleId*. Updates the ECUs based on the firmware update package using *GatewayEcu.updateEcus*. Stores the processing of the firmware update using *GatewayEcu.packageProcessed*.

The manufacturer "src/manufacturer.ts" is responsible for creating the firmware metadata package with *createMetadata* function. The function *addToBlockchain* adds the metadata to the smart contract and function *uploadFirmware* adds the firmware data to IPFS. This functionality is called by the "src/manufacturer_script.ts" which automates the process by accepting the output from the vendor script "src/vendor_script.ts". The module for a standalone DT instance of the vehicle "src/gateway_ecu_dt_script.ts" is used for the simulation of firmware updates on the DT before the physical vehicle. This is used for firmware update simulation to verify the process and test the firmware in a virtualized environment before actual deployment. The IPFS client is realized in IpfsClient "src/ipfs_client.ts" and is a helper for adding and retrieving firmware from IPFS using the node detailed in Section 5.4.1. The SymCrypt "src/sym_crypt.ts" module

is used by other modules as a helper for cryptographic functions. Provides functions for symmetric encryption and decryption, encryption with public key and decryption with private key as well as signing data with private key and verification of signed data. The DT simulation script "src/app.ts" acts as the manufacturer, which listens to the smart contract event and updates the DT instance based on the firmware update. It realizes the steps (j) - (l) discussed in Section 5.3.3. If the update process failed for the simulated (physical) vehicle, the update is sent to the DT, where the *lastUpdateState* is set to false. In case the update succeeds, the script executes steps step (l) discussed in Section 5.3.3 for the DT similar to the gateway ECU script execution.

5.5 Summary

This section describes the components of the system and their purpose. The design goals of the proposed system were given based on findings from Section 4 and Section 3. We described the architecture of the proposed OTA firmware update solution and detailed the functional components. A high-level overview of the firmware update flow was given. Finally, we reviewed the implementation details where all the previously described components were used and implemented as console applications. Additionally, the functionality required to update the DT instances in Azure was discussed. The architecture of the OTA firmware update process and its implementation discussed in this section is used in the evaluation in Section 6. Evaluation consists of evaluating the components of the implementation and scenarios based on the console applications.

6 Evaluation

In this section, we focus on design-science guideline **DG.3** from Table 1 and aim to answer $\mathbf{RQ_4}$: **What are the performance, robustness, and security outcomes of implementing a blockchain and DT-based OTA firmware update system?**. We evaluate the proposed solution in three parts. In Section 6.1 the proposed smart contract is evaluated with a static analysis tool slither. Secondly, unit tests are created to test the smart contract business logic. The cost of operations of the smart contract is given in Section 6.2 based on different scenarios. The implementation of the proposed architecture for the OTA firmware update process is evaluated based on a scenario of successful recovery from a failed update in Section 6.3. Furthermore, the scenario, implemented as a console application, is compared to the design goals stated in Section 5.2.

The evaluation criteria used in this section are based on security and performance. The criteria for security is based on the design goals stated in Section 5.2. We evaluate the proposed solution based on firmware integrity, authenticity, and authentication. Checks are done to ensure that the correct package was received. Moreover, traceability and records of the update transactions, monitoring and reliability, and firmware update

validation are evaluated. The performance is evaluated based on the gas consumption of the proposed solution.

6.1 Smart Contract Analysis

The proposed smart contract is evaluated using Slither[10] and unit testing using Hardhat[11] Ethereum development environment. Slither is a framework written in Python3 that performs static analysis on the contract and runs a suite of vulnerability detectors. The Slither static analysis tool helps in analyzing security vulnerabilities and code quality issues in smart contracts written in Solidity. In order to run Slither, a Python virtual environment is created using "python -m venv ~/python-slither". Slither is installed using pip "pip install slither-analyzer". Finally, slither is executed on the hardhat project containing the smart contract source code "slither .".

```
1 INFO:Detectors:
2 Firmware.constructor(address).vehicle_ (contracts/Firmware.sol#28)
    lacks a zero-check on :
3     - vehicle = vehicle_ (contracts/Firmware.sol#30)
4 INFO:Slither:. analyzed (1 contracts with 94 detectors), 1 result(s)
    found
```

Listing 6. Smart contract static analysis tool Slither analysis results.

The slither analyzer shows a single detected issue as shown in Listing 6. The log shows that there is no zero-check on the input of the vehicle address in the constructor. This is not considered an issue as the initiation of the contract is performed by the manufacturer. The manufacturer can make sure that the construction of the contract receives the correct public address of the vehicle. This function is not reusable after the initial deployment of the contract and thus, cannot be modified and is considered not an issue.

Hardhat enables to write automated tests with integration of ethers.js[12] to interact with the Ethereum contract and Mocha[13] to run the tests. Slither analyzed the contract for vulnerabilities and did a static analysis of the contract. The unit tests evaluate the business logic. The smart contract provides the functionality of authenticated requests from the manufacturer and vehicle. The authenticated function calls and Ethereum blockchain provide integrity of the data stored on the smart contract. In order to ensure the correct functionality of the smart contract to achieve integrity and authenticity, authentication unit tests are created. The contract is deployed for each unit test using the created deploy

[10] https://github.com/crytic/slither
[11] https://hardhat.org/
[12] https://docs.ethers.org/v6/
[13] https://mochajs.org/

function. The function returns the deployed contract, the owner (manufacturer), the vehicle address for which the contract was created, and an additional vehicle address for testing. The test cases are listed below. All function names mentioned in the list are defined in the contract. The contract tests can be found in the hardhat project[14]. The following list describes the unit test components:

- "deploy": This is a deployment helper function used in all test cases.
- "initial package set to 'none'": Executes the *getInitial* function to check if the initial package id is set to the 'none' value.
- "initial package retrievable only by the owner, vehicle": Checks if the function can only be executed by the owner or vehicle the contract was deployed for.
- "owner able to add package": Checks if the owner can execute the *addPackage* function successfully.
- "only owner able to add package": Checks if only the owner is able to add the new package to the contract. If any other party calls the function, it returns "unauthorized".
- "allow unique package ID only": Tries to add a second package with the same ID. Tests if the function execution is reverted with the error "packageID already exists".
- "packageProcessed only allowed by the vehicle": Only the vehicle can execute the *packageProcessed* function. The test returns "unauthorized" for the owner or other executors.
- "packageStatus return package status": Checks if the *packageStatus* function actually returns the requested package status. Both PROCESSED and PENDING statuses are tested.
- "returns nextPackageID if there is a new package": If a new package has been added and the previous package has the next package ID set, the execution of *getNextPackageId* returns the ID of the next package.
- "nextPackageId none if no new package": Function *nextPackageId* returns the value "none" if there are no new packages for the requested package ID.
- "package can be inserted and old omitted": The packages are structured as a linked list, where the previous package has the next package ID. This function tests, if a

[14] `https://github.com/edgarmiadzieles/thesis_hardhat`

new package can be added while replacing an older package without deleting it. Three packages are created with each pointing to the next. The middle package is then replaced by executing *addPackage*, which replaces the first package's next ID with the currently added package, and the current package points to the last package.

- "only owner allowed to retrieve package verification": Checks if the owner of the contract is the only party who can execute this function.
- "getPackage returns package information": Checks if requesting *getPackage* by a package ID returns the correct package data.
- "getPackage allowed by owner and vehicle only": Tests if the function *getPackage* can only be executed by the owner or the vehicle the contract is assigned for.

The smart contract tests are executed by running "npx hardhat test". Failed tests would mean that the business logic does not perform as expected and there might be an issue when the smart contract is deployed for the vehicle. The test execution succeeded in all 14 cases as shown in Listing 7. Based on the unit tests the contract allows only authenticated function calls for functions that store data on the smart contract. The ordering functionality of firmware update packages is achieved.

```
    OK initial package set to 'none' (602ms)
    OK initial package retrievable only by owner, vehicle
    OK owner able to add package
    OK only owner able to add package
    OK allow unique package ID only
    OK allow unique package ID only
    OK packageProcessed only allowed by vehicle (43ms)
    OK getPackageStatus return package status
    OK returns nextPackageId if there is a new package
    OK nextPackageId none if no new package
    OK package can be insterted and old ommitted (51ms)
    OK only owner allowed to retrieve package verification
    OK getPackage returns package information
    OK getPackage allowed by owner and vehicle only
  14 passing (978ms)
```

Listing 7. Hardhat firmware smart contract unit testing results.

6.2 Smart Contract Cost Estimation

In this section, smart contract functionality and deployment costs are evaluated. The computational and storage computation in Ethereum smart contract functions and deployment requires gas. Gas is used as a measurement of computation required to execute

a transaction. The actual cost of the execution is calculated in Ether. However, the Ethereum native currency, Ether, is usually too large of a unit to calculate the cost of a transaction. Ethereum uses denominations, where, for example, the unit Wei, where 1 Ether is equivalent to 1e18 Wei. Similarly, 1 Ether is equivalent to 1e9 Gwei. We use Gwei as the unit for the cost calculations. With EIP-1559[15], Ethereum introduced a base fee which changes per block alongside a priority fee (tip) for miners. The higher the priority, the faster the faster the transaction is expected to be processed. Additionally, the gas required can vary greatly depending on the input size and data already stored in memory. Adding additional data to the memory increases the gas consumption [54].

The cost of executing smart contract functions and deployment can be very volatile. Ethereum gas price changes based on the congestion of the network. Meaning transactions can be significantly cheaper when the network is less congested. To calculate the gas, we are using the tool hardhat-gas-reporter[16]. This tool calculates the gas cost based on given test cases. The test cases have been added for each function that uses gas in the smart contract. Functions that are of type pure or view do not modify the state and do not cost any gas. The calculation to estimate the cost in Gwei is ((base fee + priority fee) * gas). We use 20 Gwei as the unit of gas cost. The only functions that require gas, are *addPackage* and *packageProcessed*. Deployment of the contract also requires gas. Other functions are view or pure functions and do not require any gas.

The gas amount for deployment of the contract does not vary and requires a fixed amount of gas. The gas required to deploy the contract is 1291897. The Gwei cost is 25837940. Gas amount of *addPackage* and *packageProcessed* can vary greatly on the state of the contract and the input data length. Since *packageProcessed* depends on a package being added to the contract, these cases are resolved by adding the package and calling *packageProcessed* in every test case. The cases for gas estimation for *packageProcessed* and *addPackage* have been split into two categories of which each has 3 cases. The categories are "minimal" and "average". The "minimal" category considers a package for a single ECU. The *verification* field value is a 512-byte length hexadecimal in all cases. The "average" category considers a package with data for 3 ECU's. Each category is divided into 3 cases. Case (1) considers the gas cost for the initial package upload. Case (2) estimates the gas cost for 10 packages. Case (3) estimates the gas for 100 packages. The gas costs are given as an average.

The results from Table 9 show, that the cost between cases in both categories varies very little. Adding additional packages to the contract does not produce a significant cost increase considering adding a single package up to 100. Adding additional ECU metadata to the package increases the cost significantly, as the metadata for a single ECU amounts for the most of data within the package. Testing the gas consumption for verification data shows, that the cost of the verification for each case and category

[15] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1559.md

[16] https://www.npmjs.com/package/hardhat-gas-reporter

Table 9. Gas cost estimation results for different scenarios and use cases of the firmware update Ethereum smart contract.

Category, Case	addPackage Gas/Gwei	packageProcessed Gas/Gwei
minimal, (1)	1505532 / 30110640	229358 / 4587160
minimal, (2)	1507837 / 30156740	229354 / 4587080
minimal, (3)	1508072 / 30161440	229357 / 4587140
average, (1)	3118247 / 62364940	229358 / 4587160
average, (2)	3120557 / 62411140	229357 / 4587140
average, (3)	3120786 / 62415720	229356 / 4587120

is almost fixed. If the verification has a fixed size the memory consumption gas cost is almost irrelevant in our cases. However, this varies based on the verification data size.

The gas estimation shows that it is relatively costly to update even a single ECU with the proposed format of smart contract. The cost rises significantly if the multiple ECUs are to be updated within a single firmware package. This cost might become unmanageable if the fleet of vehicles is large. Since the cost of gas rises when the network is congested, it becomes even more costly to issue updates. This can, to some extent, be mitigated if the firmware package is moved to the IPFS network or other decentralized storage solutions keeping only the location of the firmware data on the blockchain. The same can be true for the verification data. In our testing we used a fixed-length verification, however, depending on the case, this verification might consume more or less storage on the blockchain. Similarly to the firmware package, the verification could be moved to another storage technology and use the field to provide the location of the verification.

6.3 OTA Firmware Update Scenario

The evaluation of the console application considers a scenario in which the vehicle undergoes three firmware update processes in Section 6.3.1. The first process is successful, the second fails and the third one recovers from the failed state. During the updates, the DT is monitored and updated based on the smart contract state of the physical simulated vehicle updates. The second evaluation involves the process of updating the DT in Section 6.4. The first update succeeds, the second fails and the third succeeds again.

6.3.1 Scenario Description

The evaluation for the console application is conducted based on the setup discussed in Section 5.4. The scenario involves steps (a) - (l) from the Section 5.3.3. In this scenario we simulate all entities and steps of the firmware update process from start to finish. The

scenario showcases the vehicle receiving and successfully processing the firmware update after which the DT is updated. The next firmware update fails. Then third replaced package firmware update resolves the previously failed state to demonstrate the ability to replace existing packages on the smart contract by introducing new ordering. The console application scripts are executed to create the packages, add them to the smart contract. The simulation scripts gateway ECU, manufacturer, and DT are listening to the events on the smart contract. The video of the scenario is available in Appendix II. The order of execution is stated below:

- Smart contract is deployed on the local Ganache Ethereum blockchain. The smart contract address is shared between all used scripts.
- The vehicle simulation can now start polling the update events by running script "dist/gateway_ecu_script.js" with required arguments, which includes the smart contract address. The script starts polling the smart contract for the initial update. The script is referred to as *vehicle simulation* from this point.
- The DT simulation script "dist/app.js" is started. It listens to the smart contract firmware update processed events and sends *headLightSystem* messages to the device in IoT Hub which eventually updates the DT properties based on the process described in Section 5.4.2. The DT simulation script is referred to as *DT simulation* from now on.
- Vendor script "dist/vendor_script.js" creates the sequential ECU firmware metadata. ECU firmware metadata from vendor script is used to create the full firmware metadata by the manufacturer script "dist/manufacturer_script.js". The script execution adds the firmware to IPFS and updates smart contract packages.

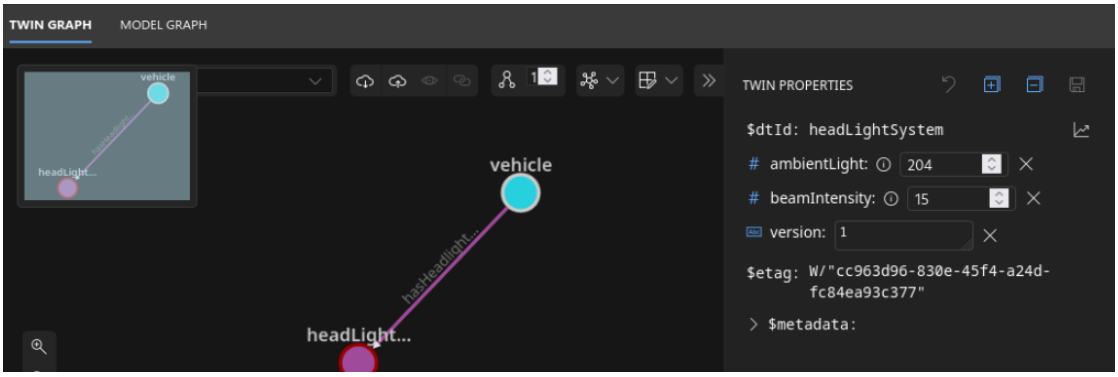

Figure 13. Azure DT instance value changes based on DT simulation messages.

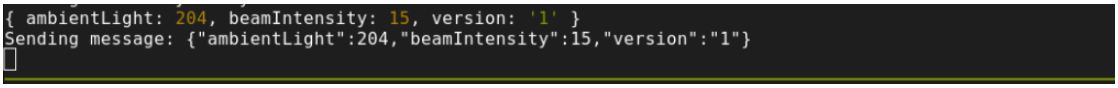

Figure 14. Vehicle simulation telemetry messages sent to Azure IoT devices to update DTs as console output.

Initially, the messages, produced by the *DT simulation* are updating the DT instance in Azure. The properties of the *headLightSystem* and *vehicle* are changed accordingly Fig. 13. The version of the ECU, *ambientLight* and beamIntensity are updating according to console log messages of the *DT simulation* (Fig. 14). The first package addition by the manufacturer to the smart contract results in the *vehicle simulation* to start the process of updating its firmware. The process is shown in Fig. 17. The vehicle simulation fetches the package from the contract in step (a) on Fig. 17. It proceeds to validate and decrypt the process in step (b) on Fig. 17. It compares the package ID from the received metadata to the requested package ID in step (c) on Fig. 17. The previous version is checked to confirm that the package is the next in order in step (d) on Fig. 17. Vehicle simulation checks the vehicle ID to confirm that this package is meant for the vehicle in step (e) on Fig. 17. It proceeds to download the firmware from IPFS for ECU in step (f) in Fig. 17. ECU verifies the firmware in step (g) on Fig. 17. ECU update is called in step (h) on Fig. 17. The ECU processes the firmware. In the case of the first package of our scenario, the package is successful, and *processVerification* is called in the smart contract in step (i) on Fig. 17. The smart contract *processVerification* function sends an event which is captured by the *DT simulation*. In the case of the first package, the update was successful. *DT simulation* performs (a) - (h) steps from Fig. 17 to install the firmware and update the Azure DT instance.

The second package contains modified metadata for the ECU. The manufacturer adds the package to the smart contract which is then requested by the *vehicle simulation* using the smart contract *getNextPackage* function. The ECU encounters an error in step (g) on Fig. 17 when trying to verify the signature of the data. The installation is aborted and the firmware is not applied. Step (h) is omitted. The verification reports a failed update state in the smart contract in step (i) in Fig. 17. The *DT simulation* does not initiate the full update process but report the last state of the update of the vehicle. The Azure DT vehicle instance reports a false *lastUpdateState* for the update as shown in Fig. 15.

The manufacturer issues a corrected firmware update package and updates the smart contract. The *vehicle simulation* requests the next id of the last successful update and repeats steps (a) - (i) from Fig. 17. The *DT simulation* now proceeds in turn with steps (a) - (h) from Fig. 17 and updates the Azure DT instances reporting a successful update as shown in Fig. 16. The firmware package of the ECU in this scenario consisted of 32 bytes of hexadecimal formatted version value concatenated with the hexadecimal

bytes of the firmware. The simulation of the ECU firmware is described in Section 5.4.3. The *DT simulation* generates a random number between 0 and 500 and executes it as an argument in the ECU function *firmwareSim*. The function returns the new value for *beamIntensity* by comparing the parsed integer value of firmware and the given randomly generated input. This allows us to check if the firmware was installed and if the behavior is expected.

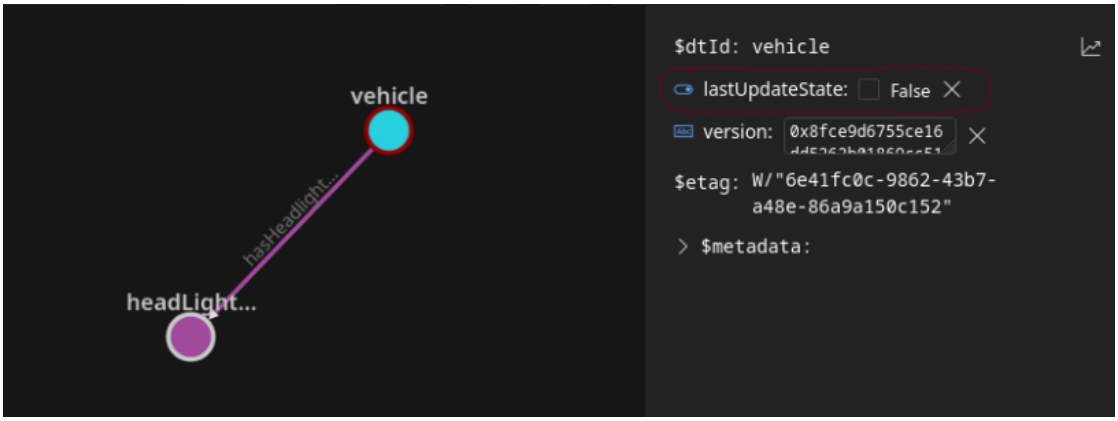

Figure 15. Vehicle Azure DT state after unsuccessful update shows *lastUpdateState* as false.

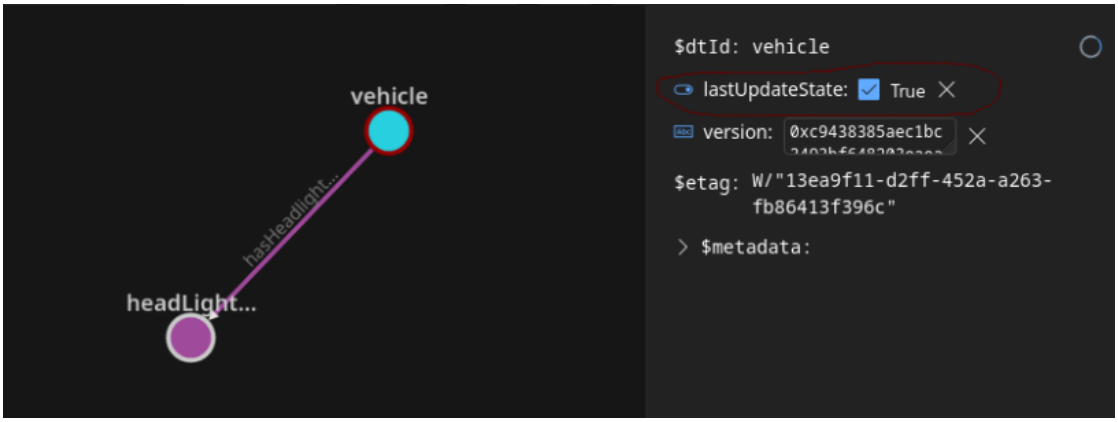

Figure 16. Vehicle Azure DT state after successful update shows *lastUpdateState* as true.

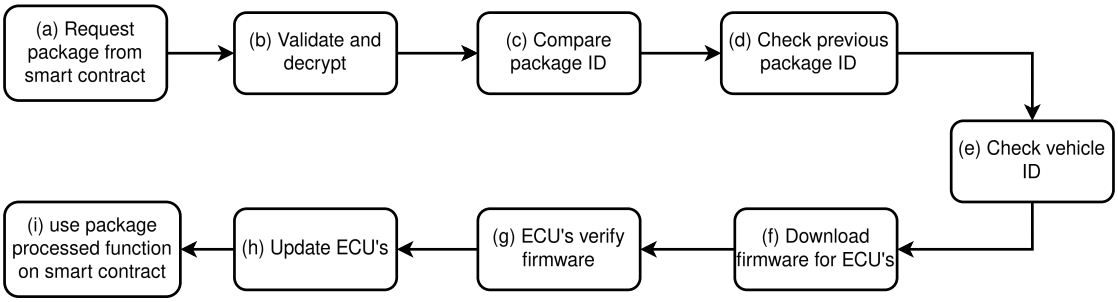

Figure 17. Full flow of successful vehicle simulation OTA firmware update process.

6.3.2 Scenario Evaluation

The simulation in the previous section successfully completed the update for the *vehicle simulation* and set the verification on the smart contract. The manufacturer was able to receive the notification from the blockchain and pass it to *DT simulation*. The *DT simulation* then proceeded with the update. The updated data was shown in the Azure DT instance providing oversight of the vehicle and its functionality. In case of the failed update, the DT was showing a failed state, and a new package was issued for the *vehicle simulation*. The failed package on the smart contract was successfully replaced and the *vehicle simulation* could successfully update its firmware. The process was repeated for *DT simulation* and the result was shown in the Azure DT instance.

The simulation shows that the update process is traceable and the DT provides oversight of the update status. The *DT simulation* simulates the update by performing the same steps of the firmware update as the vehicle, providing assurance of the vehicle's successful update state, and the DT accurately represents the vehicle. The list below shows the implementation scenario comparison with evaluation criteria based on design goals stated in Section 5.2:

- **Authenticity**: According to the contract, as described in Section 6.1, only the manufacturer can add the firmware. In case a malicious package was sent to the vehicle, the signature of the package is verified. ECUs can verify the signature of the firmware based on the metadata provided.
- **Integrity**: Based on the signature and authentication tag of the firmware package, the vehicle can make sure that the firmware package has not been tampered with. The firmware signature for the ECUs is checked with the same method.
- **Authentication**: The firmware package and firmware symmetric encryption keys can only be decrypted by the vehicle private key or the ECU private key accordingly.

- **Correct package**: Comparison of the requested package ID and package ID from the metadata verifies, that the package ID requested corresponds with the package ID received. Comparing the previous package ID from the metadata provides assurance, that the package received is the next in-order update package. This mitigates the possibility of receiving a valid package that is out of order within the smart contract. Checking the vehicle ID within the firmware metadata assures that the package was indeed intended for the vehicle requesting the update.

- **Availability**: The decentralized nature of IPFS and blockchain provides high availability for firmware requests. It also potentially mitigates issues, where the server (in case of client-server architecture) is unreachable when the vehicle wants to perform an update. However, the vehicle is limited to the requests to trusted blockchain and IPFS nodes. This can potentially create a bottleneck if multiple vehicles are trying to reach the same trusted nodes.

- **Update status from ECU**: In the current simulation, the ECU verifies the firmware before performing the update. However, the case of a faulty installation of the firmware and rollbacks was not considered.

6.4 Firmware Update Simulation With Digital Twin

DT enables testing the firmware in a virtualized environment before the firmware is deployed for the physical vehicle. This can potentially eliminate faults in the firmware update process for the physical vehicle and increase the reliability of the update. This scenario simulates steps (a) - (g) from the Section 5.3.3. The vehicle in this process is replaced with the vehicle DT. The scenario uses a similar OTA firmware update flow as the Section 6.3. The first firmware package succeeds. The second firmware package fails due to ECU not being able to verify the firmware. Third, the replaced package is successfully updated the simulated vehicle. The video of the process is available in Appendix II.

A new smart contract is deployed on the local Ganache Ethereum blockchain. The address is shared between all used scripts. The vehicle DT simulation is realized in script "dist/gateway_ecu_dt_script.js". This script queries the smart contract for new updates. Additionally, it sends simulated telemetry messages to the devices in Azure. The properties of the *headLightSystem* are changed according to the messages. The manufacturer script "dist/manufacturer_script.js" is again used to add the firmware to IPFS and firmware packages to the smart contract. The vehicle DT script polls the smart contract for new packages periodically. The firmware update process follows steps (a) - (i) in Fig. 17. The first firmware update succeeds and the DT is updated accordingly. The second firmware update data is faulty and the secondary ECU update results in unverified firmware. This state is updated on the DT in Azure. The third package replaces the

second package and the vehicle DT firmware update succeeds. This process of testing the firmware updates in a virtualized environment allows the manufacturer to test the firmware update process safely. It gives assurance of the reliability of the update and the update process in general for the physical vehicle. Given that the firmware update successfully works in the virtualized environment, the manufacturer can proceed to deploy the firmware update for the physical vehicle.

6.5 Summary

In this section, we have conducted a static analysis of the smart contract of the proposed solution using Slither. Slither found one issue stating that the constructor of the smart contract lacks a zero check on the input argument. This is regarded as not an issue since the manufacturer is deploying the contracts and can check the argument before deployment. Additionally, unit tests were created to check the functional validity of the smart contract. The unit tests verify that the smart contract functions that modify its state can only be used by the vehicle or manufacturer and that the ordering of the packages is correct. A performance evaluation was conducted to test the cost of the smart contract functions and deployment. The tests were divided into separate scenarios and cases. The results show that maintenance and deployment of the smart contract are costly, even when updating a single ECU. The cost rises significantly if there is a big fleet of vehicles and amount of update metadata. The last evaluation consisted of two scenarios using console applications. In the first scenario, the vehicle console application receives updates from the manufacturer, and the DT is updated according to the Ethereum blockchain state. It shows that the proposed solution allows recovery from a faulty update by issuing new firmware metadata on the Ethereum blockchain. The firmware update process is repeated for the DT which also shows the faulty state in case of a failed update. The authenticity, integrity, and correct package can be verified by the vehicle by installing authenticated firmware. The vehicle can verify that the firmware package is the one that the vehicle requested and can receive update statuses from the ECU. The scenario was repeated that included only DT. This was an evaluation to show that the firmware can be tested on the DT before the actual firmware update on the physical vehicle, increasing the reliability of the firmware update. The results are discussed further in Section 7 bringing out the limitations and challenges of the proposed solution.

7 Discussion

This section provides additional discussion in the context of design-science design evaluation guideline **DG.3** from Table 1. This work reviews the OTA firmware updates for vehicles in the context of the IoV. The shift of vehicle ECUs becoming more software-defined has introduced the ability for software updates for individual ECUs. Moreover,

connected vehicles have introduced the capability of vehicles receiving updates over the air without manual intervention. This poses new security challenges for secure firmware updates. One of the issues highlighted is the dominating server-client architecture of firmware updates. The problems regarding the architecture are network congestion and downtime. Moreover, the unique connectivity environments in an IoV system might benefit from a firmware update framework, where the firmware updates can be decentralized and potentially closer to the end vehicle receiving the update. This thesis addresses these issues by considering the Ethereum blockchain and distributed storage system IPFS in order to mitigate these issues. Additionally, it is important to develop a good oversight and monitoring system for the vehicles receiving updates. Specifically, in case the vehicle setups might be various. This thesis discusses the usage of DTs to overcome this issue. DTs can enable monitoring of a system that is a virtual copy of the physical entity. The firmware updates can be simulated in a digitalized environment without the intervention of the physical entity. In this section, the results of the evaluation of the proposed framework are discussed related to the research questions in Section 7.1. The limitations and challenges of the system are discussed in Section 7.2. Finally, potential future work is discussed in Section 7.3

7.1 Answer to Research Questions

This thesis posed 5 research questions and 2 sub-questions. The questions and their related sections are shown in Table 10. The thesis started by asking the main question **How to build a DT and blockchain-enabled firmware update system for the Internet of Vehicles?**. In order to answer this question, three additional questions were formed. Question $\mathbf{RQ_1}$ (Table 10) focuses on the current advancements in state-of-the-art OTA firmware updates in and IoT environment in Section 4. This question was divided into 2 sub-questions in Section 4: $\mathbf{RQ_{1.1}}$ and $\mathbf{RQ_{1.2}}$ (Table 10). We posed the question to review if blockchain and DT technologies have been proposed for firmware updates. Additionally, what are the benefits of using these technologies for firwmare updates. Section 4 examines the latest research to give an overview of the latest literature on blockchain and DT-enabled firmware updates in an IoT environment. It extracts the relevant features and use cases of the technology in question and provides a summary of the reviewed literature. Blockchain technology was mainly used to achieve firmware authenticity, integrity, high availability, and storage of firmware update records. DTs mainly provided and were used for monitoring and testing features.

In order to scope the thesis for the use case of IoV, it was necessary to understand the conditions for firmware updates in the context of IoV and ITS. Section 3 focuses on the question $\mathbf{RQ_2}$ (Table 10). It provides an overview of the unique networking conditions of an IoV environment and the current state of the vehicle OTA firmware updates. Additionally, it extracts the requirements for OTA firmware updates for vehicles. The review showed that OTA firmware updates in an IoV system benefit from decentralized

solutions to omit a single point of failure. Moreover, a decentralized solution brings the update process closer to the vehicle. Furthermore, the requirements were found for an OTA firmware update solution for vehicles.

Section 4 and Section 3 were used as the basis for creating the proposed solution in Section 5. The aforementioned sections answered the questions $\mathbf{RQ_1}$ and $\mathbf{RQ_2}$ to form the basis for answering question $\mathbf{RQ_3}$ in Section 5. The firmware updates in IoV context required a robust and secure firmware update mechanism that ensured firmware integrity, high availability, authenticity, authentication and ability to monitor and test the firmware. The proposed solution was created using Ethereum smart contract and IPFS for firmware delivery and storage to aim for a secure and decentralized firmware delivery mechanism. We answered $\mathbf{RQ_4}$ by evaluating the proposed solution architecture by testing the smart contract with unit tests and static analysis tool in Section 6. The smart contract gas cost evaluation shows that the operations of the proposed solution are too costly. Console applications simulated a scenarios for vehicle and DT OTA firmware update. The simulation scripts showcased the integrity, authenticity, authentication, correct update package order, availability and overall function of the process update flow. Additionally, testing the firmware update pre-deployment and monitoring of the update process was successfully presented.

Table 10. Research questions and related sections.

RQs	Question	Section
$\mathbf{RQ_1}$	What are the latest advancements contributing to the current state-of-the-art in firmware updates within IoT systems?	Section 4: Systematic Literature Review
$\mathbf{RQ_{1.1}}$	How does blockchain contribute to the firmware updates in the IoT systems?	Section 4.5.1: How Does Blockchain Contribute to the Firmware Updates in the IoT Systems?
$\mathbf{RQ_{1.2}}$	How does DT contribute to the firmware updates in the IoT systems?	Section 4.5.2: How Does DT Contribute to the Firmware Updates in the IoT Systems?
$\mathbf{RQ_2}$	How can OTA firmware updates be adjusted to meet the unique connectivity and security challenges present in the IoV ecosystem?	Section 3: Use Case: IoV
$\mathbf{RQ_3}$	How to implement blockchain and DT-based OTA firmware update?	Section 5: Proposed Solution
$\mathbf{RQ_4}$	What are the performance, robustness, and security outcomes of implementing a blockchain and DT-based OTA firmware update system?	Section 6: Evaluation

7.2 Limitations and Challenges

The proposed solution was based on the findings from the conducted SLR. There were limitations to conducting SLR. The search queries and results generalized the term "firmware" by including "software" updates. Otherwise, the results would produce significantly less results. Furthermore, the search term "firmware or software update" could sometimes produce results related to device management with topics being searched for. Another limitation is related to limited studies found related to firmware updates and DTs. Papers included mentioned DTs, however, the technologies used for implementation or evaluation purposes can be considered to not fit the evolving definition of DT fully. Some papers mentioned DTs, however did not specify a concrete DT technology. The blockchain update frameworks in IoT environments did not mention a concrete consensus mechanism and in some cases, the consensus mechanism had to be derived. Similarly, a selection of works focused mainly on the architecture and unspecified blockchain technology. Specific blockchain technology was in some cases mentioned as the technology that would fit the purpose of the proposed architecture. Ethereum is considered a public-type blockchain. However, only some reported the environment specifically related to the private or public blockchain used so the publicity of the blockchain was again specified based on the general specification of Ethereum.

The proposed solution and implementation have multiple limitations. Firstly, smart contracts need to be created for and used by every vehicle. According to the gas cost evaluation in Section 6.2, the gas cost was relatively high. This would mean significant costs for the manufacturer. Additionally, vehicles need to have enough Ethereum funds to operate. The cost could be lowered by storing less data on the smart contract and moving it to a different storage like IPFS. The proposed solution also relies on trusted blockchain nodes. This means that the vehicle must have at least (but not limited to) one available trusted blockchain node. If multiple vehicles try to access the node simultaneously it might introduce too much load in terms of computation or network congestion for the node. Having multiple trusted nodes is in this case a must. The same applies to the IPFS nodes. However, the latter could be partially addressed by the vehicle acting as the node itself storing only limited necessary data. This could also improve the availability of files between the vehicles if every vehicle has IPFS node functionality. Another issue with decentralized solutions is giving control of the firmware update process to other involved parties. Issues with the Ethereum blockchain or IPFS like congestion or gas price increase affects the overall functionality of the firmware updates. Resolving these issues depends more on the participants using the technologies than the manufacturer of the vehicles. Additionally, the privacy of the vehicles was less considered in the proposed solution. The proposed solution might be considered only pseudonymous.

The proposed system relies on relatively computation-heavy encryption schemes. This would not work for resource-restrained ECUs within the vehicle. Computationally intensive operations can be replaced with a lighter encryption scheme or be offloaded to

more capable ECUs. This would depend on the different ECUs on the vehicle. Encrypting individual firmware with different symmetric keys means that the encrypted firmware file cannot be shared among vehicles with similar ECUs. Encrypting the firmware file with the same keys might expose the firmware if only one of the ECUs is compromised in a way, where the symmetric key is derived. Both versions have their benefits and drawbacks. One would improve the security ensuring no encrypted firmware file would be similar and in another case, the storage requirements would be significantly lower.

Another challenge with the proposed solution is keeping the DT in sync. Deriving the firmware status from the blockchain might produce some delays. In case the DTs receive real-time data from different sources, which means race conditions have to be taken into account. One of the solutions might be to always include the firmware version of the vehicle with the data being given to the DT or the application governing the twins. This would ensure that the messages are ordered correctly regarding the versioning. If the DT shares data with the vehicle, a similar issue might occur, where the vehicle already might have updated the version but receives outdated version data from the twin. Furthermore, DTs in the implementation hold the same encryption keys as the vehicle, which might be a security issue if the DTs were to be compromised. Storing different keys for the DTs would potentially mean a different smart contract and firmware storage and encryption solution.

The proposed solution was only tested locally using auto-mined transactions. Introducing the framework real network still needs to be tested. Moreover, the console applications only accounted for limited scenarios. Console applications were meant to showcase and test the proposed solution process. Scenarios and edge cases would be various in the production environment and would have to be taken into account. Real-world scenarios introduce many challenges related to networking and vehicle conditions. For example, it would be beneficial to use trusted IPFS and Ethereum nodes that are closer to the vehicle. This would mean deploying multiple nodes in many different locations.

7.3 Future Work

The proposed system relies on the Ethereum blockchain and IPFS for storage and firmware delivery. The proposed solution requires gas to operate. Minimizing the data stored on the smart-contract can reduce this limitation, however not entirely. Other blockchain frameworks that have similar smart contract capabilities can be considered that do not have similar limitation can be considered to avoid gas costs completely. Regarding IPFS, the vehicle could act as a IPFS node and download limited files to avoid the necessity of a trusted node. Additionally, other storage solutions like BitTorrent can be considered for decentralized storage solutions. A robust data delivery mechanism for versioning and real-time updates could be considered for DTs. The current solution produces delays in keeping the DTs up to date because the state is received from the blockchain. In the current solution, the DTs require the encryption keys of the ECUs of

the physical vehicle, potentially DTs could rely on encryption keys other than the real vehicle to avoid sharing the encryption details. Different, less computationally intensive encryption schemes or offloading the decryption process to a trusted ECU can be tested on the proposed solution to overcome the issues of resource-constrained devices. Finally, the proposed solution or variants could be tested in networking conditions that resemble more closely real scenarios.

8 Conclusion

The recent shift towards software-defined ECUs within a vehicle and the new connectivity capabilities have enabled vehicle firmware updates over the air. Additionally, IoV has presented new connectivity paradigms for vehicles, providing additional operational capabilities to ITS. Utilizing real-time traffic solutions in the context of ITS produces a large amount of spatio-temporal data. It is beneficial to offload the computation to edge servers and move the computation closer to the source. In the case of this thesis, the general client-server architecture for OTA updates could be replaced by a decentralized solution. This thesis investigated the process of OTA firmware updates in an IoV system by incorporating the Ethereum blockchain, DT technology, and IPFS. We reviewed the architecture of IoV and the current state of OTA update processes for vehicle ECUs and found the requirements for vehicular firmware OTA update processes. The SLR was conducted to find the current state-of-the-art blockchain and DT-based firmware update systems. The SLR data extraction found the design goals and capabilities of a decentralized update solution in IoT systems. Based on these findings we proposed a decentralized OTA firmware update process based on Ethereum blockchain, DTs, and IPFS. This solution was realized as a console application to showcase the update process for the simulated physical vehicle and DT instance.

The evaluation of the proposed solution shows that a smart contract can be used for the secure transfer of firmware metadata packages between the vehicle and the manufacturer. Moreover, encrypted metadata and firmware can be securely shared between authenticated parties in a decentralized system achieving firmware integrity, high availability, authenticity, authentication, and firmware update record storage. Additionally, the firmware update can be simulated on with DTs before updates to gain firmware update reliability assurance and monitoring during the update of the physical vehicle. However, the gas cost of the smart contract functions proved to be costly. The encryption schemes used and solution implementation do not consider resource-constrained ECUs. A decentralized solution based on the Ethereum blockchain produces delays in updating the DTs which might lead to race conditions if additional data is received by the DTs from other sources. This solution was only tested as a simulation and requires testing in a scenario that resembles the real-world conditions.

Consideration of future work might include developing the OTA firmware update

framework using different types of blockchains, where gas cost is a smaller issue or doesn't involve payments at all. Use cases where resource-constrained devices are involved might be considered. Full privacy, not only pseudonymous privacy, of the vehicles, can be taken under consideration. The proposed solution relies on gateways for IPFS nodes. Elimination of the concept by moving the operations in a computationally and storage-effective manner to the vehicle could increase the operational efficiency and reliability of the system. Developing and testing the applications in real-world environments should also be considered.

List of References

[1] AUTOSAR, Explanation of Adaptive Platform Design, 2022, https://www.autosar.org/fileadmin/standards/R22-11/AP/AUTOSAR_EXP_PlatformDesign.pdf (24.1.2024).

[2] AUTOSAR, Explanation of Firmware Over-The-Air, 2020, https://www.autosar.org/fileadmin/standards/R20-11/CP/AUTOSAR_EXP_FirmwareOverTheAir.pdf (24.1.2024).

[3] Bauwens, J., Ruckebusch, P., Giannoulis, S., Moerman, I., Poorter, E. D. "Over-the-Air Software Updates in the Internet of Things: An Overview of Key Principles". *IEEE Communications Magazine*, 2020, Vol. 58, No. 2, pp. 35–41.

[4] Baza, M., Nabil, M., Lasla, N., Fidan, K., Mahmoud, M., Abdallah, M. "Blockchain-based Firmware Update Scheme Tailored for Autonomous Vehicles". Marrakesh, Morocco: IEEE, 2019, pp. 1–7.

[5] Benet, J., IPFS - Content Addressed, Versioned, P2P File System, 2014, https://arxiv.org/abs/1407.3561 (24.3.2024).

[6] Bhatt, A., Karthikeyan, V. "Digital Twin Framework and its Application for Protection Functions Testing of Relays". Coimbatore, India: IEEE, 2022, pp. 682–687.

[7] Cachin, C., Architecture of the Hyperledger Blockchain Fabric, 2016, https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf (19.2.2024).

[8] Chavhan, S., Gupta, D., Chandana, B. N., Khanna, A., Rodrigues, J. J. P. C. "IoT-Based Context-Aware Intelligent Public Transport System in a Metropolitan Area". *IEEE Internet of Things Journal*, 2020, Vol. 7, No. 7, pp. 6023–6034.

[9] Contreras-Castillo, J., Zeadally, S., Guerrero-Ibanez, J. A. "Internet of Vehicles: Architecture, Protocols, and Security". *IEEE Internet of Things Journal*, 2018, Vol. 5, No. 5, pp. 3701–3709.

[10] Cui, Y., Lei, D. “Design of Highway Intelligent Transportation System Based on the Internet of Things and Artificial Intelligence”. *IEEE Access*, 2023, Vol. 11, pp. 46653–46664.

[11] Custodio, P., McBride, B., Le, T., Jackson, J., Haulmark, K., Di, J., Farnell, C., Mantooth, H. A. “Digital Twin of an ANPC inverter with integrated Design-For-Trust”. Bath, United Kingdom: IEEE, 2022, pp. 1–7.

[12] Darwish, T. S. J., Abu Bakar, K. “Fog Based Intelligent Transportation Big Data Analytics in The Internet of Vehicles Environment: Motivations, Architecture, Challenges, and Critical Issues”. *IEEE Access*, 2018, Vol. 6, pp. 15679–15701.

[13] Ghosal, A., Halder, S., Conti, M. “Secure over-the-air software update for connected vehicles”. *Computer Networks*, 2022, Vol. 218, p. 109394.

[14] Gong, S., Tcydenova, E., Jo, J., Lee, Y., Park, J. H. “Blockchain-Based Secure Device Management Framework for an Internet of Things Network in a Smart City”. *Sustainability*, 2019, Vol. 11, No. 14, p. 3889.

[15] Grieves, M., Origins of the Digital Twin Concept, 2016, `http://rgdoi.net/10.13140/RG.2.2.26367.61609` (24.3.2024).

[16] Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M. “Internet of Things (IoT): A vision, architectural elements, and future directions”. *Future Generation Computer Systems*, 2013, Vol. 29, No. 7, pp. 1645–1660.

[17] Guissouma, H., Hohl, C. P., Lesniak, F., Schindewolf, M., Becker, J., Sax, E. “Lifecycle Management of Automotive Safety-Critical Over the Air Updates: A Systems Approach”. *IEEE Access*, 2022, Vol. 10, pp. 57696–57717.

[18] Guissouma, H., Klare, H., Sax, E., Burger, E. “An Empirical Study on the Current and Future Challenges of Automotive Software Release and Configuration Management”. Prague: IEEE, 2018, pp. 298–305.

[19] Guittoum, A., Aïssaoui, F., Bolle, S., Boyer, F., De Palma, N. “Inferring Threatening IoT Dependencies using Semantic Digital Twins Toward Collaborative IoT Device Management”. Tallinn Estonia: ACM, 2023, pp. 1732–1741.

[20] Halder, S., Ghosal, A., Conti, M. “Secure over-the-air software updates in connected vehicles: A survey”. *Computer Networks*, 2020, Vol. 178, p. 107343.

[21] He, S., Ren, W., Zhu, T., Choo, K.-K. R. “BoSMoS: A Blockchain-Based Status Monitoring System for Defending Against Unauthorized Software Updating in Industrial Internet of Things”. *IEEE Internet of Things Journal*, 2020, Vol. 7, No. 2, pp. 948–959.

[22] He, X., Alqahtani, S., Gamble, R., Papa, M. “Securing Over-The-Air IoT Firmware Updates using Blockchain”. Crete Greece: ACM, 2019, pp. 164–171.

[23] Hevner, A. R., March, S. T., Park, J., Ram, S. “Design Science in Information Systems Research”. *MIS Quarterly*, 2004, Vol. 28, No. 1, pp. 75–105.

[24] IOTA Foundation, Replacing the Coordinator with a Validator Committee, `https://blog.iota.org/replacing-coordinator-with-validator-committee/` (23.3.2024).

[25] IOTA Foundation, The Coordinator - PoA Consensus, `https://wiki.iota.org/learn/protocols/coordinator/` (23.3.2024).

[26] Kitchenham, B., Charters, S., Guidelines for performing Systematic Literature Reviews in Software Engineering, 2007, `https://legacyfileshare.elsevier.com/promis_misc/525444systematicreviewsguide.pdf` (15.1.2024).

[27] Kumar, S., Tiwari, P., Zymbler, M. “Internet of Things is a revolutionary approach for future technology enhancement: a review”. *Journal of Big Data*, 2019, Vol. 6, No. 1, p. 111.

[28] Kuppusamy, T. K., DeLong, L. A., Cappos, J. “Uptane: Security and Customizability of Software Updates for Vehicles”. *IEEE Vehicular Technology Magazine*, 2018, Vol. 13, No. 1, pp. 66–73.

[29] Lee, B., Lee, J.-H. “Blockchain-based secure firmware update for embedded devices in an Internet of Things environment”. *The Journal of Supercomputing*, 2017, Vol. 73, No. 3, pp. 1152–1167.

[30] Lee, J. “Patch Transporter: Incentivized, Decentralized Software Patch System for WSN and IoT Environments”. *Sensors*, 2018, Vol. 18, No. 3, p. 574.

[31] Leiba, O., Bitton, R., Yitzchak, Y., Nadler, A., Kashi, D., Shabtai, A. “IoTPatchPool: Incentivized delivery network of IoT software updates based on proofs-of-distribution”. *Pervasive and Mobile Computing*, 2019, Vol. 58.

[32] Lin, S. “Proof of Work vs. Proof of Stake in Cryptocurrency”, 2023, Vol. 39.

[33] Mahi, M. J. N., Chaki, S., Ahmed, S., Biswas, M., Kaiser, M. S., Islam, M. S., Sookhak, M., Barros, A., Whaiduzzaman, M. “A Review on VANET Research: Perspective of Recent Emerging Technologies”. *IEEE Access*, 2022, Vol. 10, pp. 65760–65783.

[34] Mallozzi, P., Pelliccione, P., Knauss, A., Berger, C., Mohammadiha, N. “Autonomous Vehicles: State of the Art, Future Trends, and Challenges”. Cham: Springer International Publishing, 2019, pp. 347–367.

[35] Manolache, M. A., Manolache, S., Tapus, N. “Decision Making using the Blockchain Proof of Authority Consensus”. *Procedia Computer Science*, 2022, Vol. 199, pp. 580–588.

[36] Mtetwa, N. S., Tarwireyi, P., Sibeko, C. N., Abu-Mahfouz, A., Adigun, M. "Blockchain-Based Security Model for LoRaWAN Firmware Updates". *Journal of Sensor and Actuator Networks*, 2022, Vol. 11, No. 1, p. 5.

[37] Nofer, M., Gomber, P., Hinz, O., Schiereck, D. "Blockchain". *Business & Information Systems Engineering*, 2017, Vol. 59, No. 3, pp. 183–187.

[38] Paolone, G., Iachetti, D., Paesani, R., Pilotti, F., Marinelli, M., Di Felice, P. "A Holistic Overview of the Internet of Things Ecosystem". *IoT*, 2022, Vol. 3, No. 4, pp. 398–434.

[39] Paul, P. "Blockchain Technology and its Types—A Short Review". *International Journal of Applied Science and Engineering*, 2021, Vol. 9, No. 2.

[40] Qureshi, K. N., Abdullah, A. H. "A Survey on Intelligent Transportation Systems". *Middle-East Journal of Scientific Research*, 2013, Vol. 15, No. 5, pp. 629–642.

[41] Qureshi, K. N., Din, S., Jeon, G., Piccialli, F. "Internet of Vehicles: Key Technologies, Network Model, Solutions and Challenges With Future Aspects". *IEEE Transactions on Intelligent Transportation Systems*, 2021, Vol. 22, No. 3, pp. 1777–1786.

[42] Seo, J. W., Islam, A., Masuduzzaman, M., Shin, S. Y. "Blockchain-Based Secure Firmware Update Using an UAV". *Electronics*, 2023, Vol. 12, No. 10, p. 2189.

[43] Sey, C., Lei, H., Qian, W., Li, X., Fiasam, L. D., Sha, R., He, Z. "Firmblock: A Scalable Blockchain-Based Malware-Proof Firmware Update Architecture with Revocation for IoT Devices". Chengdu, China: IEEE, 2021, pp. 134–140.

[44] Sharma, N., Garg, R. D. "Real-Time IoT-Based Connected Vehicle Infrastructure for Intelligent Transportation Safety". *IEEE Transactions on Intelligent Transportation Systems*, 2023, Vol. 24, No. 8, pp. 8339–8347.

[45] Silvano, W. F., Marcelino, R. "Iota Tangle: A cryptocurrency to communicate Internet-of-Things data". *Future Generation Computer Systems*, 2020, Vol. 112, pp. 307–319.

[46] Singh, M., Fuenmayor, E., Hinchy, E., Qiao, Y., Murray, N., Devine, D. "Digital Twin: Origin to Future". *Applied System Innovation*, 2021, Vol. 4, No. 2, p. 36.

[47] Solomon, G., Zhang, P., Brooks, R., Liu, Y. "A Secure and Cost-Efficient Blockchain Facilitated IoT Software Update Framework". *IEEE Access*, 2023, Vol. 11, pp. 44879–44894.

[48] Tao, F., Xiao, B., Qi, Q., Cheng, J., Ji, P. "Digital twin modeling". *Journal of Manufacturing Systems*, 2022, Vol. 64, pp. 372–389.

[49] Tao, F., Zhang, M. "Digital Twin Shop-Floor: A New Shop-Floor Paradigm Towards Smart Manufacturing". *IEEE Access*, 2017, Vol. 5, pp. 20418–20427.

[50] Taslimasa, H., Dadkhah, S., Neto, E. C. P., Xiong, P., Ray, S., Ghorbani, A. A. "Security issues in Internet of Vehicles (IoV): A comprehensive survey". *Internet of Things*, 2023, Vol. 22, p. 100809.

[51] Tsaur, W.-J., Chang, J.-C., Chen, C.-L. "A Highly Secure IoT Firmware Update Mechanism Using Blockchain". *Sensors*, 2022, Vol. 22, No. 2, p. 530.

[52] Weiß, M., Müller, M., Dettinger, F., Jazdi, N., Weyrich, M. "Continuous Analysis and Optimization of Vehicle Software Updates using the Intelligent Digital Twin". Sinaia, Romania: IEEE, 2023, pp. 1–7.

[53] Witanto, E. N., Oktian, Y. E., Lee, S.-G., Lee, J.-H. "A Blockchain-Based OCF Firmware Update for IoT Devices". *Applied Sciences*, 2020, Vol. 10, No. 19, p. 6744.

[54] Wood, G., Ethereum: A Secure Decentralised Generalised Transaction Ledger Paris Version 705168a, 2024, `https://ethereum.github.io/yellowpaper/paper.pdf` (20.3.2024).

[55] Yao, J.-F., Yang, Y., Wang, X.-C., Zhang, X.-P. "Systematic review of digital twin technology and applications". *Visual Computing for Industry, Biomedicine, and Art*, 2023, Vol. 6, No. 1, p. 10.

[56] Yohan, A., Lo, N.-W. "FOTB: a secure blockchain-based firmware update framework for IoT environment". *International Journal of Information Security*, 2020, Vol. 19, No. 3, pp. 257–278.

[57] Zhao, Y., Liu, Y., Tian, A., Yu, Y., Du, X. "Blockchain based privacy-preserving software updates with proof-of-delivery for Internet of Things". *Journal of Parallel and Distributed Computing*, 2019, Vol. 132, pp. 141–149.

[58] Zhu, F., Lv, Y., Chen, Y., Wang, X., Xiong, G., Wang, F.-Y. "Parallel Transportation Systems: Toward IoT-Enabled Smart Urban Traffic Control and Management". *IEEE Transactions on Intelligent Transportation Systems*, 2020, Vol. 21, No. 10, pp. 4063–4071.

Appendix

I. Resources

Console applications repository
This repository holds the project that is described in Section 5.4.3. The project setup and use case is described in the "readme" file. Additionally, it holds the IPFS docker-compose definition.

Repository: `https://github.com/edgarmiadzieles/thesis`

Solidity contract hardhat project
This repository stores the smart contract described in Section 5.3.2 and unit tests described in Section 6.1

Repository: `https://github.com/edgarmiadzieles/thesis_hardhat`

II. Demo Videos

Video of evaluation scenario firmware update process for simulated vehicle and DT based on Section 6.3:
The video shows the initiation of the vehicle simulation and DT. The manufacturer script is used to add firmware update packages to IPFS and blockchain. The device messages are shown as a log in Azure Function app monitor and Azure Digital Twin explorer shows the DTs.

Video URL: `https://www.youtube.com/watch?v=YMxsj7EL0K4`

Video of evaluation scenario firmware update process for DT Section 6.4:
The video shows the initiation of the DT simulation. The manufacturer script is used to add firmware update packages to IPFS and blockchain. The device messages are shown as a log in Azure Function app monitor and Azure Digital Twin explorer shows the DTs.

Video URL: `https://www.youtube.com/watch?v=ZB0kCQmCBBc`

III. Proposed Solution Code

```
struct Package {
    bytes firmwareMetadata; // Stores the firmware metadata
    bytes key; // The encrypted symmetric encryption key
    bytes metadataSig; // Signed firmwareMetadata, by the
        manufacturer
    Status status; // Can be PROCESSED or PENDING
    bytes verification; // Filled by the vehicle after processing the
         update
    bytes32 nextId; // Next package ID
    bool set; // A way to check the existence of a package entry in
        the map
}

function addPackage(bytes memory firmwareMetadata_, bytes memory key_
    , bytes memory metadataSig_, bytes32 packageId_, bytes32
    previousId_, bytes32 nextId_) external onlyOwner packageIdUnique(
    packageId_) {
    Package memory package = Package(
        firmwareMetadata_,
        key_,
        metadataSig_,
        Status.PENDING,
        "",
        nextId_,
        true
    );
    packages[packageId_] = package;

    if (initial == none) {
        initial = packageId_;
    }

    if (previousId_ != none) {
        packages[previousId_].nextId = packageId_;
    }
}

mapping(bytes32 => Package) private packages;
```

Listing 8. Partial solidity smart contract showing main functions for package addition and storage

```
import { app, InvocationContext } from '@azure/functions';
import { DefaultAzureCredential } from "@azure/identity";
import { DigitalTwinsClient } from '@azure/digital-twins-core';

const adtInstanceUrl = process.env["AZURE_DT_INSTANCE_URL"];

if (!adtInstanceUrl) {
  console.error("Application setting \"AZURE_DT_INSTANCE_URL\" not
     set");
}

const cred = new DefaultAzureCredential();
const client = new DigitalTwinsClient(adtInstanceUrl, cred);

type headLightSystem = {
  ambientLight: number,
  beamIntensity: number,
  version: string
}

type vehicle = {
  lastUpdateState: boolean,
  version: string
}

export async function eventHubTrigger2(message: unknown, context:
    InvocationContext): Promise<void> {
    context.log('Event hub function processed message:', message);
    context.log('EnqueuedTimeUtc =', context.triggerMetadata.
       enqueuedTimeUtc);
    context.log('SequenceNumber =', context.triggerMetadata.
       sequenceNumber);
    context.log('Offset =', context.triggerMetadata.offset);

    const deviceId = context.triggerMetadata.systemProperties["iothub
       -connection-device-id"];
    context.log(deviceId);

    if (deviceId == 'headLightSystem') {
      const hls: headLightSystem = message as headLightSystem;

      const updateTwinData = [{
          op: "replace",
          path: "/ambientLight",
          value: hls.ambientLight
        },
        {
          op: "replace",
```

```
            path: "/beamIntensity",
            value: hls.beamIntensity
          },
          {
            op: "replace",
            path: "/version",
            value: hls.version
          }];
        await client.updateDigitalTwin(deviceId, updateTwinData);
    }

    if (deviceId == 'vehicle') {
        const veh: vehicle = message as vehicle;

        const updateTwinData = [{
            op: "replace",
            path: "/version",
            value: veh.version
          },
          {
            op: "replace",
            path: "/lastUpdateState",
            value: veh.lastUpdateState
          }];

        await client.updateDigitalTwin(deviceId, updateTwinData);
    }
}

app.eventHub('eventHubTrigger2', {
    connection: "eventhubConnectionString",
    eventHubName: 'iothub-name',
    cardinality: 'one',
    handler: eventHubTrigger2,
});
```

Listing 9. Azure Event Hub message consumer function. Updates the according digital twin instances in Azure

IV. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Edgar Miadzieles**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Digital Twin and Blockchain-Driven Firmware Updates for the Internet of Vehicles,

   supervised by Dr. Mubashar Iqbal.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Edgar Miadzieles
14/05/2024