HAL open science

Fenrir: Blockchain-based Inter-company App-Store for the Automotive Industry

David Fernández Blanco, Frédéric Le Mouël, Trista Lin

To cite this version:

David Fernández Blanco, Frédéric Le Mouël, Trista Lin. Fenrir: Blockchain-based Inter-company App-Store for the Automotive Industry. IEEE Access, 2022, 10, pp.122933-122953. 10.1109/ACCESS.2022.3223130 . hal-03938624

HAL Id: hal-03938624
https://hal.inria.fr/hal-03938624

Submitted on 13 Jan 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2022.DOI

Fenrir: Blockchain-based Inter-company App-Store for the Automotive Industry

DAVID FERNÁNDEZ BLANCO[1,2], FRÉDÉRIC LE MOUËL[1], AND TRISTA LIN[2]
[1]Univ Lyon, INSA Lyon, Inria, CITI, EA3720, 69621,Villeurbanne, France. (email: firstname.midlename-lastname@insa-lyon.fr)
[2]STELLANTIS, 78140, Velizy-Villacoublay, France. Email: firstname.MidlenameLastname@stellantis.com

Corresponding author: David Fernández Blanco (e-mail: david.fernandez-blanco@insa-lyon.fr).

The research leading to the results presented in this study was supported by Stellantis under the collaborative framework OpenLab VAT@Lyon, involving STELLANTIS and CITI Lab (ANRT contract n°2020/1415).

ABSTRACT From a software evolution perspective, more actors are integrating the in-vehicle software development cycle. In this process, software deployment mechanisms must include more complex techniques to meet the software verification and traceability levels required by industry safety and security constraints. In this context, we propose Fenrir, a public inter-automaker blockchain-based application store framework in which each automaker retains software installability control. This application store also aims to ensure traceability and security, while also keeping the solution light in terms of both energy consumption and computing requirements, to be used in constrained environments. We implemented Fenrir in a heterogeneous architecture composed of both on-board (bearing an ARM Cortex-A53 chipset, already deployed in cars) and off-board (Amazon EC2) nodes for a realistic automotive use-case scenario, in which we evaluated its performance and energy consumption. We demonstrate that the overheads added by our solution for an entire software deployment pipeline—comprising both deployment and usage of already deployed software packages—depends mainly on the verification mechanism, whose impact is not significant, i.e., 3.8% for the worst-case scenario and 0.3% for a typical scenario.

INDEX TERMS Automotive, Application Store, Blockchain, Distributed Systems, Multi-provider, Software Dependency Management, Software Deployment.

I. INTRODUCTION

IN recent decades, Information and Communication Technologies (ICT) have dominated the transformation of the automotive world, progressively integrating it into Smart City ecosystems. This transformation has considerably increased vehicles' connectivity, allowing new services such as autonomous driving services, connected mobile applications, and stolen vehicle tracking software to reach the market. As a result of this trend, increasingly disruptive innovations from many new actors will continue to appear even more rapidly in the coming years, thus making embedded software increasingly dynamic and varied. This trend also motivates the progressive transformation of vehicles into mobile and interconnected cloud nodes, allowing passengers' services and data to follow them everywhere [1], [2], [3].

However, such a rapid pace of innovation pace has dangerously increased system and software complexity, meaning software development and integration errors are responsible for 50% of all vehicle recalls [4]. These errors are mostly caused by inter-services' undetected incompatibilities, thus, software version management, traceability, and maintenance are now critical issues for the automotive sector [5], [6].

If these errors were patchable through simple software or configuration modifications, this would enable remote Over-the-Air (OTA) diagnosis and updates instead of traditional manual garage updates, thereby saving millions of dollars, minimizing repair delays, and reducing the environmental impact of update campaigns [7]. Besides software patches, this mechanism also provides new business opportunities, making flexible software service pay-as-you-go subscriptions a reality for the automotive sector. Nevertheless, the security and safety vulnerabilities of these fully autonomous software deployment systems also pose significant risks to vehicle safety and performance, potentially endangering their passengers'. For example, the 2002 Volkswagen and 2015 FCA [8] cases could have been easily prevented through more exhaustive software integrity verification mechanisms. Similarly, the 2005 BMW and 2020 Ford [9] issues could have been resolved with fewer casualties with a proper software delivery pipeline. These risks will be even greater consider-

ing the near-future evolution toward more decentralized and collaborative development environments with the constant inclusion of new actors.

Although the scientific challenges of remotely updating or installing software have been widely addressed in existing literature [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], to the best of our knowledge, no studies have combined remote software deployment, automotive constraints, and the multi-provider characteristics of a multi-automaker application store, nor have software inter-dependencies in been considered in this context. In addition, the contributions above have evaluated only the security and performance of the proposed mechanisms, ignoring energy, computational, and storage demands, which currently represent some of the most challenging on-board constraints.

In this paper, we present Fenrir, a novel and highly secure application store framework for vehicles. To the best of our knowledge, while other state-of-the-art proposals focus on updating on-board services in a mono-automaker environment, Fenrir is the first to address not only updates but also distributing new applications in a highly-collaborative multi-automaker application store. Our work enhances the security of existing solutions via a hybrid public/private blockchain-based mechanism to ensure software integrity and authentication through the software deployment process, either in standard Vehicle-to-Cloud(V2C) or Vehicle-to-Vehicle(V2V) approaches. Thus, in this framework, all new suppliers can publish their new applications, however, control over which applications can be installed for a specific vehicle model is preserved by the vehicle's manufacturer who, at least for now, remains accountable for problems that software may cause in their vehicle's fleet. Finally, Fenrir also contains a newly proposed mechanism to handle inter-software dependencies before downloading them into each vehicle to optimize the distribution pipeline and reduce energy and computation costs. In summary, in this paper, we exhaustively present Fenrir, a blockchain-based multi-automaker application store framework with cloud-offloaded dependency management, which includes the following scientific contributions:

- A new model for privately managed public blockchains for software storage and distribution for multiple companies with heterogeneous hardware and software constraints (§V-A).
- Energy, computation, and storage optimization for proofs, verification, and pruning mechanisms in highly participative private blockchains for safety-critical software storage in highly constrained systems (§V-B, §V-C & §V-E).
- A cloud-assisted distributed software dependency management mechanism directly integrated into the blockchain to optimize storage needs and maintain an in-vehicle backup and up-to-date global software image (§V-D).
- A realistic system implementation and an industry-inspired testbed to measure the solution's performance and viability for current OTA update traffic and future V2V update campaigns (§III & §VI).

The remainder of this paper is organized as follows: Section II presents the study's automotive sector context and technological background. Subsequently, Section III details the use case upon which the development of Fenrir was based. After that, Section IV describes all the contributions noted above through a detailed description of Fenrir, Section V evaluates its performance and Section VI gives an extensive discussion. Finally, Section VII surveys the existing state of the art and Section VIII presents the conclusions of our results and proposed future work.

II. THEORETICAL BACKGROUND

A. AUTOMOTIVE SOFTWARE ARCHITECTURE

Electric and Electronic (E/E) architecture has evolved significantly in recent years, marking a transition from a previously fully distributed architecture that was almost wholly composed of mono-functional micro-controllers. These micro-controllers were connected through methods such as Controller Area Network (CAN), Local Interconnect Network (LIN), FlexRay, and Universal Serial Bus (USB). Newer approaches use actual domain vehicle architecture, comprising fewer, more powerful microprocessors connected primarily via Ethernet. Thus, considering recent technology evolution, there is a clear tendency toward E/E architecture [7] reconfiguring the older, highly constrained electronic control units (ECUs) into new, higher-end devices with greater computational and storage capabilities. This tendency has helped to reduce the price, weight, space, and complexity of these systems. On this basis, systems will most likely evolve to Central Computer Architectures (CCA) or Zonal Architecture (ZoA), in which, respectively, one or few higher-end ECUs encompass all the functions running in each subcluster or even the whole vehicle. These new architectures, linked to virtualization and advances in networking, will yield many possibilities for dynamically allocated software, performant OTA update mechanisms, and Service-Oriented Architecture conceptions [7], [20], [21], [22]. Thus, we used these near-future architectures as a basis for developing our proposed solution in this study.

B. AUTOMOTIVE SOFTWARE DEVELOPMENT CYCLE

The automotive industry is a highly heterogeneous and participative environment in which automakers act as integrators of different pieces of software developed by different suppliers. To facilitate software update delivery by all the relevant actors, the automakers maintain an application repository that hosts update packages. These updates can then be downloaded by dealerships and installed through OBD2 in a garage setting or exclusively through telematics and infotainment ECUs. Such updates can, either be directly installed by cars remotely or installed by users via the USB port. Furthermore, as shown in Fig. 1, Tier 1, 2, and 3 suppliers often sell their solutions to other suppliers and multiple automakers, who can then also deploy this software

to some vehicles in their fleets. Note that Tier 1 suppliers are direct suppliers of automakers, Tier 2 are suppliers or subcontractors for Tier 1 suppliers and Tier 3 are suppliers or subcontractors for Tier 2 suppliers. Furthermore, it is common for Tier 3 suppliers to work closer to the hardware than Tier 2s and so on. Thus, sharing a common distributed software hub between all automakers and actors will help standardize and secure both the software packages and the associated delivery process.

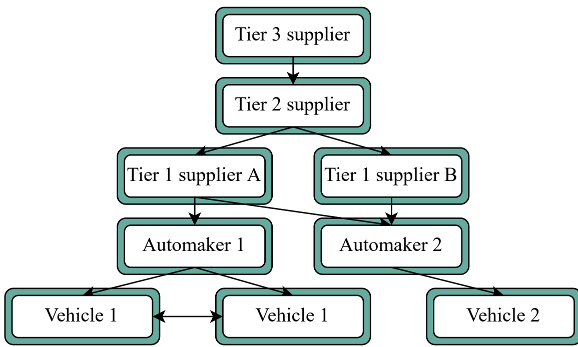


FIGURE 1: Automotive Software Ecosystem: Suppliers and Relationships.

C. AUTOMOTIVE OTA UPDATE FRAMEWORKS

As the first OTA update frameworks reach the market, we can identify the design patterns most automakers apply. These frameworks are usually composed of four different services distributed through the different high-end ECUs of the architecture. Fig. 2 shows an example of OTA update framework mapping over ZoA reference architectures. These services are: (1) the Downloader, responsible for retrieving the Vehicle Software Package (VSP) containing the update from the Cloud software storage platform, (2) the Orchestrator, in charge of distributing updates to the node that will perform the installation, (3) the Installer, responsible for installing and testing each software component, and (4) the Authorization Manager, in charge of verifying the authenticity of the received package.

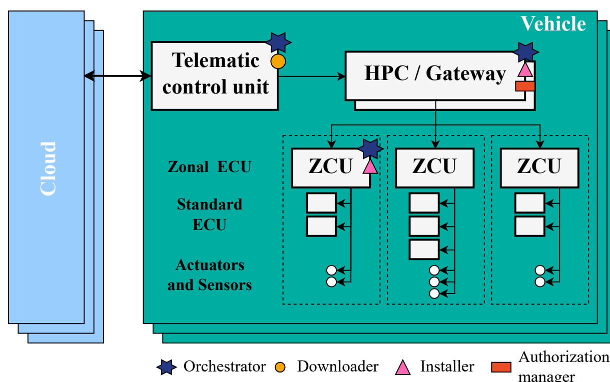


FIGURE 2: OTA Update Framework service mapping over ZOA.

Furthermore, Fig. 2 shows the different kinds of in-vehicle node profiles (i.e., Zonal ECUs, Standard ECUs, Actuators and Sensors) and their network locations, with the Zonal ECUs more central in the network and in charge of managing a cluster of Standard ECUs, Actuators and Sensors. These three node profiles do not only differ in their network location, but also in the quantity of resources they bear. While Zonal and Standard ECUs are normally deployed using MPUs, since Zonal ECUs are more resourceful than Standard ECUs, Actuators and Sensors are normally conceived using highly restrained MCUs. Thus, updates can take many forms depending on their final target node and purpose. Here, we highlight four. (1) Considering lighter updates, the *Application Configuration Updates Over-The-Air* (AOTA) are composed of, as indicated by their name, a set of runtime/post-install parameter changes. These parameters include examples such as driver profile changes, deep learning algorithm optimizations, or parameter updates for regulatory compliance. These updates strongly condition the vehicle's behavior without requiring any software change; thus, these update types do not require the relevant software components to shut down, only the vehicle to stop. Another key update type is (2) *Firmware updates Over-The-Air* (FOTA), which involves installing or updating the main system software that controls the underlying hardware. Thus, to achieve these updates, a complete restart and re-flash of the ECU are required. After the update, the software must be tested entirely and, if there are any errors, switched back to the previous firmware version through techniques such as dual banking. This form of updates will be used for the Actuators and Sensors. A third update type is (3) *Software updates Over-The-Air* (SOTA), involving the installation of application components. These updates can be whole (e.g., a full software install) or partial, also known as Δ software updates (ΔSOTA). Note that the size of the partial updates is typically close to the aforementioned FOTA packages. For both SOTA and ΔSOTA, the software install process must be performed with the vehicle shut down; the process must also be tested afterward and allowed to roll back if the system does not operate properly following the update. SOTA usually take place over Unix-like systems, typically in infotainment or telematics ECUs. Finally, update type (4) comprises *Media file updates Over-The-Air* (MOTA) packages, which include some multi-media files such as Global Navigation Satellite Systems (GNSS) maps, custom images, sounds, or videos for the In-Vehicle Infotainment (IVI). These updates are considerably heavier than those described above, thus, they are presently not possible as OTA updates. In this paper, we divide these files into chunks that can then be reconstructed on board to preserve homogeneity through the data structures. Note that these last three forms will be used for updating the Zonal and Standard ECUs but not for the Actuators and Sensors.

D. BLOCKCHAIN BASIC CONCEPTS AND OBJECTIVES

The essential concept behind blockchain technology emerged in the late 1980s [23] with the introduction of Paxos [24], which enabled an agreement to be reached over a result/the state of a machine in a network of computers where the computers or network itself may be unreliable. However,

the blockchain concept was not invented until 2008 when Satoshi Nakamoto, alias of the anonymous creator, published Bitcoin: A Peer-to-Peer Electronic Cash System [25]. In that paper, Nakamoto describes an alternative to the existing banking system, proposing a calculus-based consensus model allowing storage and distribution of a global transaction book without requiring a centralized, corruptible management authority. Thus, blockchain is built over Nakamoto's three pillars: (1) Decentralization—the blockchain shall be distributed between all the involved actors and act autonomously, without a central governing unit, (2) Transparency—the blockchain shall keep all anonymized transactions public and accessible by anyone on the network, and (3) Immutability—once a transaction has been published to the blockchain, it can never be altered.

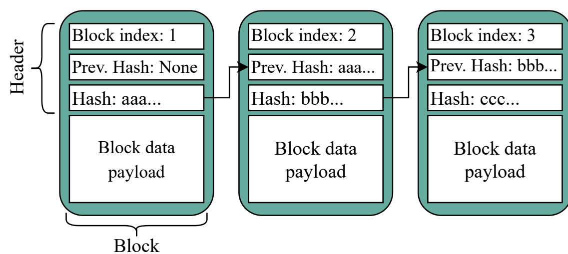


FIGURE 3: Blockchain concept scheme.

In terms of the technical details of the blockchain's storage structure, as shown in Fig. 3, a blockchain is a type of linked list including an extra mechanism to ensure its immutability. Thus, the data structure of the list elements comprises: (1) the block index, i.e., a number indicating the block's position in the chain, (2) the hash of the previous block's information, (3) the block data payload, and (4) the hash of the current block. Note that the hashing mechanism, which is a mathematical function allowing the mapping of data of arbitrary size to unique fixed-size values, is the most crucial part of the blockchain and allows the existence of an incorruptible chain of unmodifiable data, since modifying the input used to calculate the hash would not produce the same hash result. Although the blockchain was originally invented to eliminate the use of a central infrastructure manager, over time, blockchain technology has also been applied to private networks to increase the security of their storage layers. In addition, the hashing and integrity mechanisms, also known as authentication proofs, have evolved to decrease their computational and energy costs, with new options available to replace the classic computational Proofs-of-Work such as the Proof-of-Stake (based on the participants' interest) or the Proof-of-Authority (based on the trustworthiness of a set of nodes).

III. USE CASE SCENARIO

For the studied use case scenario, we base our design on a realistic micro-service partitioning and inter-service dependency use case for active driving monitoring and profiling, as presented in [26]. Furthermore, we add a lower-level mapping to this previously presented model through the automotive software development cycle (§II-B) to illustrate the complexity of the interactions between actors. Fig. 4 depicts then the inter-software dependencies presented on this study and the supplier ownership for each of the different services.

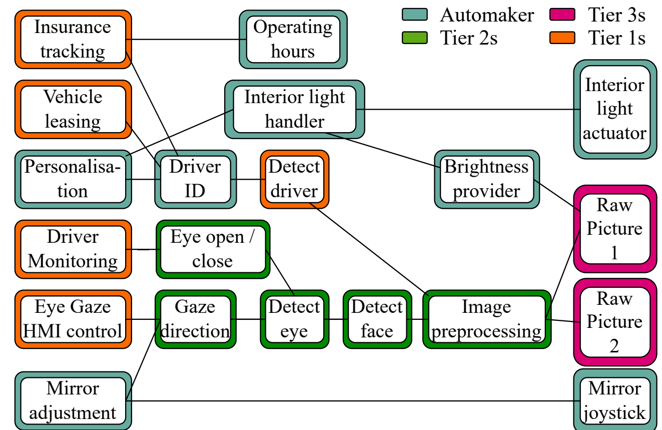


FIGURE 4: Micro-service partitioning, software inter-dependencies and supplier ecosystem mapping scenario.

Thus, to fully illustrate all the different possibilities, we provide four software ownership examples:

1) Raw pictures 1 & 2: An example of a software component implemented close to the hardware and thus probably developed by the hardware manufacturer, which, in this case, is the camera's manufacturing company.
2) Insurance tracking: A high-level software application developed by an upcoming actor through the application store to add a new scenario of interest; in this case, this scenario is real-time management of insurance policies. This application must be proposed by multiple insurance company competitors, thus offering a wide selection of options for users, and be potentially common to all automobile brands and software architectures.
3) Image pre-processing: An example of software developed by different suppliers offering potentially different functionalities. These services must ensure full compatibility with different vehicles and their underlying software to widen their selling possibilities over the same service platform proposed by concurrent suppliers. Thus, users and automakers can select which services they wish to install based on criteria other than the technical specifications, e.g., the service's pricing, preferences, or the quality of its interface.
4) Mirror adjustment: A software type mainly provided by each of the different vehicle manufacturers due to its close relationship with the vehicle's design and unsuitability for reuse by other automakers. Within a manufacturer's range, these applications may also be model specific.

In our opinion, these examples illustrate the different development profiles likely to be found in a new multi-automaker application store. These use-cases also help to

underline the increasing diversity of applications and the difficulty of ensuring compatibility between services without implementing dependency management mechanisms. Additionally, these examples illustrate the need to develop a mechanism allowing the integrity and authenticity of software originating from different developers to be preserved, as well as a mechanism allowing automakers to protect their vehicles from malicious, corrupted, or even incompatible applications, despite user requests.

Note that, all these services were conceived in collaboration with our industrial partner at STELLANTIS and follow the example proposed in [26] at the 2022 Automotive Computing Conference (ACC). For industrial property reasons, we cant provide the legacy source codes. However, despite the lack of details on the service implementations, all the services meta-data (inter-dependencies, sizes, etc.) needed by the different Fenrir sub-mechanisms are widely described when necessary on the paper central section (cf., §V) and the appendix (cf., IX-A), and discussed in the evaluation section (cf., §VI).

IV. RELATED WORKS

A. A STUDY OF SECURITY AND DATA AUTHENTICITY ON THE OTA UPDATE FRAMEWORK PROPOSITIONS

In this subsection, we discuss different OTA update mechanisms and classify them according to the tools used to ensure update package security and authenticity.

The first group of solutions is those that guarantee the authenticity of data using symmetric encryption, asymmetric encryption, or both. In these solutions, the integrity of the data is reliant upon decryption. Representative examples of only-symmetric key-based OTA frameworks include Mahmud et al. [10] and Mansour et al. [11]. In both studies, a secure software update framework is described based on sharing an initial set of link keys among automakers, vehicles, and software suppliers, which is then used for encrypting both software and communications. In addition, they propose other mechanisms to enhance security and complex transmission traceability, such as time-hopping randomization [10], or detecting potential errors and malicious behavior, such as remote diagnostic tooling [11]. However, even if the computing requirements needed to perform the encryption are low, as the set of keys is directly included in the vehicle by the automakers, the impact of a key being compromised, thus making it impossible to link a message to its sender directly, poses a significant potential threat that does not match the requirements of safety- and security-critical frameworks. In addition, neither implements a content integrity verification mechanism, thus making it impossible to detect maliciously modified packages if an authorized key is compromised.

Thus, to solve the problems linked to the risk of compromising a static set of symmetric keys and to improve message traceability, Steger et al. [12] propose a solution in which an asymmetric key is used to secure unicast communication, in addition to a symmetric multicast key from the service center to several cars, thus enabling parallel updates. In this case, the only keys that will be shared are the public keys, making identity faking difficult. However, this solution is again at potential risk of key compromise since action triggering is centralized and there is no distributed network to ensure the software package's authenticity.

Similarly, the approach proposed by Mayilsamy et al. [15] involves combining asymmetric encryption and a well-known cryptography field, steganography. Their study proposes a solution to integrate software files encrypted by an asymmetric encryption algorithm (RSA in this case) hidden along the edge region of the update's cover image using steganography. This self-verifiable stego-image would then be adequate for safe storage and transmission. However, even though the compromise of long RSA keys (2048 bit in this case) remains an open challenge, and the costs and storage needs associated with using stego-images are also unsuitable for the highly restrained nodes of the automotive industry.

Further considering the integrity of package content instead of heuristic solutions for security during the transmission, we now discuss hash-based solutions. Based on this technique, Nilsson et al. [13] propose a secure OTA firmware update protocol for connected vehicles based on dividing software and then hashing and encrypting each chunk. However, this division and hashing process appears inefficient in terms of computing and energy consumption compared to using software packages as a whole. Oka et al. [14] propose an alternative infrastructure in which a trusted portal calculates the hash of the whole software package and places it at the end of the message so the receiver can check the message's veracity. However, in both of these approaches, having a centralized authority in charge of distributing keys is highly vulnerable to targeted, single point of failure, and identity usurpation attacks. Within the same scope as [14], we can consider the proposed approach of Karthik et al. [16]. This solution secures key storage at a lower level by using Secure Hardware Module technology to handle key management. Their study also proposes an OTA framework that distributes updated software to ECUs in the form of images (containing collections of code and data) and metadata (containing image-related files such as the size of the file, the image hash, creation date, author, etc.). In addition, Karthik et al. suggest introducing different keys introduced in the hardware to verify the encryption of different files. However, this solution is vulnerable to rollback attacks due to a lack of proper verification mechanisms during software update installation. This system also suffers from the same issue as those in the previously discussed approaches of being centralized rather than distributed.

Through optimizing hashing solutions by adding distributed middleware and some new optimization mechanisms, some studies propose blockchain-based solutions. In the blockchain, software packages are linked to each other immutably (cf. Fig. 3) and spread through the different nodes integrating the system; in this case, it is always possible to use a simple majority vote approach to detect malicious or erratic introductions. Thus, this technology guarantees data

integrity, complete traceability, and higher consistency and security than the aforementioned techniques. However, there are also some drawbacks—for example, blockchain-based solutions do not allow, or barely allow, modification of past data, rely on the secrecy of the private user keys, require large amounts of storage and significant computational resources (for the Proof of Work and other cryptographic mechanisms), and remain uncertain with relation to future legislation and regulations. Nonetheless, in recent years, increasing efforts have been expended to develop new techniques to reduce the resource consumption of this solution and allow its deployment in the IoT world, such as pruning and new, less resource-consuming proofs. In the automotive sector, multiple papers have proposed blockchain-based OTA update frameworks. However, most of these, such as Steger et al. [17], are based on the Proof-of-Work mechanism, while others such as Witanto et al. [19] and Mtetwa et al. [27] focus on other parts of the implementation such as peer-to-peer exchange or the transmission details, respectively. However, the resource consumption constraints of the embedded vehicle architecture suggest that Proof-Of-Work algorithms are unsuitable for such systems, thus indicating a key limitation of these propositions. Additionally, all the cited blockchain solutions follow the classic public blockchain schemes, which we consider unpractical in our software development context (cf. §II-B), in which the blockchain publishers are companies that need to use this chain as a means to sell their software.

Consequently, and as shown in Table 1, blockchains appear to be the most appropriate solutions in terms of security and data integrity preservation, for both distributed and centralized systems, despite the resource consumption increase. This is mainly due to all the mechanisms added to ensure the traceability and the immutability of the data. To the best of our knowledge, no previous study has proposed hybrid public/private blockchain networks, inter-automaker collaborative stores, low-power-consumption authentication proofs, or software versioning or dependency management tools, which, in our view, would improve the resource efficiency and security of the OTA update frameworks. Therefore, the background study presented in this section justifies the choices proposed in our study (cf., V), which we believe to offer an improved collaborative software deployment framework for the automotive context. Furthermore, note that this migration through blockchain-based solutions to enhance the system traceability and data integrity has also started in other related sectors such as eHealth [28], [29] and Industry 4.0 [30].

B. A WALKTHROUGH OF THE DEPENDENCY MANAGEMENT MECHANISMS

As software philosophy evolves from application-based to service-based, software reuse and sharing will be enhanced. However, as the volume of software in vehicles increases, so too does the number of developers in the ecosystem. Thus, managing the dependencies between these new software packages is becoming increasingly complex and requires careful attention. In this subsection, we will provide an overview of different dependency management solutions and classify them into two categories: (1) those focused on service-library dependencies and (2) those aiming to deal with inter-service dependencies.

The solutions dealing with software library dependencies include [31] and [32]. The proposal in [31] aims to develop a highly granular dependency network that goes beyond library packages and generates a versioned ecosystem-level call graph to maintain an actualized network and set of de-

TABLE 1: Comparison of the Over-The-Air frameworks found in the state-of-the-art.

		Integrity & authenticity	For trust-less dynamic environments	Secure	Resource consumption	Traceability	On-board storage needs
Symmetric or Asymmetric key based (CA-like)	S. Mahmud *et al.* [10]	Very low	No	Very low	Low	No	Very Low
	K. Mansour *et al.* [11]	Very low	No	Very low	Low	∼	Very Low
	M. Steger *et al.* [12]	Very low	No	Low	Low	No	Very Low
	K. Mayilsamy *et al.* [15]	Medium	∼	Medium	High	∼	High
Hash-based	D. K. Nilsson *et al.* [13]	Medium	No	Medium	High	∼	Medium
	D. Oka *et al.* [14]	Medium	No	Medium	Medium	∼	Medium
	T. K. Kuppusamy *et al.* [16]	Medium	∼	High	Medium	No	Medium
Blockchain-based	M. Steger *et al.* [17]	High	Yes	High	Very High	Yes	Very High
	E. N. Witanto *et al.* [19]	High	Yes	Very High	Very High	Yes	High
	N. S. Mtetwa *et al.* [27]	High	Yes	High	Very High	Yes	High
	Fenrir	Very High	Yes	High	Medium	Yes	Medium

∼ : More or less compliant.
Note that this comparison is theoretical and done analyzing the papers and claims.

pendencies. In contrast, the approach in [32] describes the dependency management mechanism of Gradle, a well-known build automation tool that works in a declarative fashion in which the developer sets its dependencies and versioning; as a result, there is no need to calculate dependencies between libraries for higher compatible versions. These solutions are not precisely focused on the same scope as that of the present study as we aim to handle dependencies between pre-built services and, thus, do not require an external library.

However, studies [33], [34], [35], [36] seek solutions to the same problem presented in our study. Among these solutions, [36] is notable for containing a fundamental definition of software components, in which a component is equivalent to a mono-functional service with a set of dependencies on other components. In this solution, the dependency calculation mechanism requires a description of the system software architecture, the resources of the system, and the software components to generate a dependency set. This dependency set comprises mandatory dependencies, firm requirements without which installation is impossible, optional dependencies, and negative dependencies, indicating a conflict forbidding the software installation. In this solution, the dependency calculation process comprises two phases, as follows. (1) The installability phase: before authorizing the installation of a component, checks are performed to ensure it is not forbidden, the services it requires are available, and it does not provide forbidden services. (2) The installation phase: once the component is proved to be installable, the effects of its installation on the system are calculated. These effects comprise newly available services, new forbidden services, new forbidden components, and new dependencies (represented by a dependency graph).

Considering a similar scope, study [33] presents a four-phase process composed of: (1) collecting historical service network flow data set and pre-processing it, (2) computing dependencies and constructing dependency matrices, (3) compiling the constructed matrices into a graph database, and (4) mining this graph database to identify service dependencies. In study [34], the authors use declarative contracts to calculate the dependency graphs in a controlled environment, while in [35], an implementation is proposed for a dependency management mechanism for fully private blockchains to track service logs and collaboratively calculate unknown service dependencies. However, while these solutions are close to our aims, the system constraints and challenges they try to solve differ markedly. In our case, the installation environment is managed by the different automakers; there is no need to guess the dependencies between software because the automakers must declare these for the risk assessment analysis. Thus, our approach would be closest to the one described in [36] but with a design adapted to blockchain properties and additional cloud reinforcement for the dependency calculation in pruned blockchains (cf. §V-D & V-E).

V. FENRIR FRAMEWORK DESCRIPTION

Fenrir is a multi-automaker application store framework built on a hybrid public/private blockchain-based application storage layer. This framework focuses on software package authenticity and integrity throughout the entire update pipeline (i.e., software transmission, storage, installation, and configuration steps) and manages inter-service dependencies between software packages before they are downloaded to the on-board architecture via V2C or V2V approaches, with constant consideration of the energy consumption and computing and storage constraints of the embedded vehicle systems.

Fenrir implements the public/private characteristics of the blockchain model by first defining roles for the different mode types in the network. Different permissions levels are then assigned to each role (§V-B). This approach ensures that the application store is public-to-publish, however, the software eligibility control for each vehicle model remains with its manufacturer since these manufacturers are legally accountable for failures on their devices. The automaker's central node network contains a complementary distributed application green-list layer that holds the authorizations for each vehicle model (§V-C). In addition, to reduce the blockchain storage model's energy consumption impacts, Fenrir proposes a key-management layer and multi-stage Proof-of-Authority mechanism to ensure software authenticity. This can be achieved given that the system's control is based on a defined set of trusted nodes. This key-management layer (§V-B & V-E) is distributed among the different automakers and manages the addition of new automakers and actors following a democratic vote model that we describe below.

In the remainder of this paper, we decompose the Fenrir framework into four largely independent sub-features, as discussed in the following subsections:

- Identity/Role management: a service managing key distribution, role management, permissions, and the introduction of new nodes in the system (§V-B).
- Software package authentication: this mechanism is key to ensuring software integrity and traceability through the update process. This mechanism details the relationships between data structures and their behavior throughout the software delivery process (§V-C).
- Distributed inter-service dependency mgmt.: before downloading software packages to the vehicle, either via V2V or V2C, application version control is performed to minimize redundant data transmission (§V-D).
- Light and resilient Software Package Storage: considering automotive systems' storage limitations, Fenrir proposes a new three-level pruning and backup model to ensure data recomposition at low energy/computational cost (§V-E).

Finally, we conclude this section by describing a typical end-to-end software deployment cycle (§V-F).

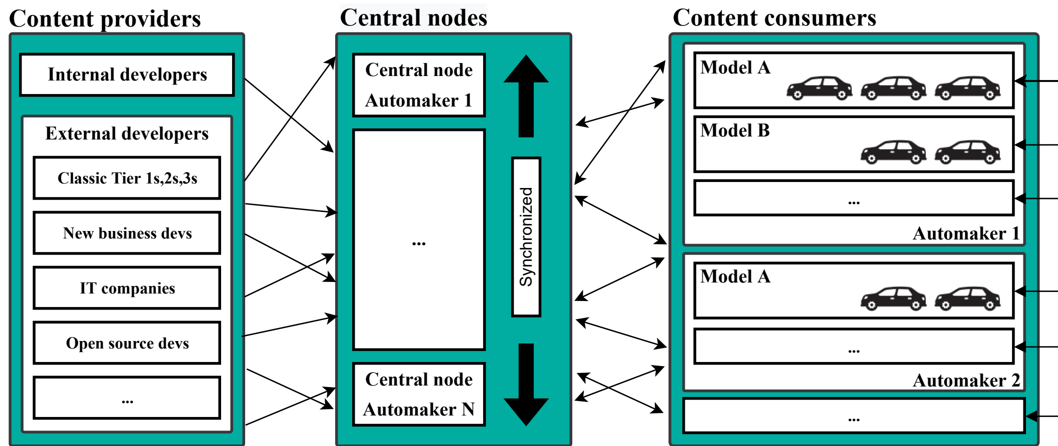


FIGURE 5: Fenrir high-level system architecture.

A. FENRIR BASICS

In Fenrir, we define the Software Application Package, (SAPckg) as an ensemble of a Software Package Structure (SPS) and its associated Software Transaction Block (STB) published into the blockchain, whose formal definition is illustrated in Fig. 8. This SAPckg is first submitted to the Pending Submission Repository (PSR), a distributed temporary repository in which application proposals are stored until at least one automaker is interested. Once the temporary SAPckg has sufficient endorsers, it will be pushed into the Main Storage Chain (MSC), its endorsements will be assigned to the Distributed Green List Ledger (DGL), and the software block can then be distributed to authorized vehicles. More details about this process are presented in §V-C. As our framework is initially conceived for an SOA/Micro-service architecture, each SAPckg shall contain exactly one service identified by a tuple (service identifier, version) and the list of dependency relationships with the other services in the infrastructure described in the *Requirements* field of the STB. Thus, each service will preserve its unique identifier irrespective of its version and, once a tuple has been published, only tuples with higher versions can be published so the system will never contain two identical tuples.

Considering the definition of inter-service compatibility in Fenrir, two services are deemed compatible when they do not have any declared mismatching dependencies between them or any of their sub-dependencies, thus allowing them to safely cohabitate in the same environment. Two versions of the same software are always considered incompatible; in addition, Fenrir's *STB-Req.* field may contain not only the definition of a given service's dependencies but also the list of service incompatibilities. The management of inter-service compatibility conflicts and prioritization when calculating the Global Software Image (GSI) and Backup Software Image (BSI) is described by the Distributed Inter-Service Dependency Management Mechanism detailed in §V-D.

Having defined the software package structures and inter-dependency model, we further expand on the architecture definition. Fig. 5 shows the different actors participating in the Fenrir software deployment cycle based on the role profile definition in §V-B. Note that all the aforementioned control structures (i.e., the PSR, DGL, and MSC) are fully hosted and synchronized among the central nodes, which will guarantee the access and propagation of different blocks to other nodes depending on their roles and needs.

B. IDENTITY / ROLE MANAGEMENT

As shown in Fig. 5, Fenrir comprises three layers: (a) the content providers, (b) the content consumers, and (c) the central nodes. Each layer has different rights and interests;

	Read & Store MSC	Application Submission	Application Authorization	Green-list Access	Dependency calculation	SAPckg verification	Software spread	Correct nodes
Internal developer	~ (Read only)	✓	✓	X	X	✓	X	X
External developer	X	✓	X	X	X	X	X	X
Content consumer	~ (Partial storage)	X	X	X	~ (with local information)	✓	✓	✓
Central node	✓	X	X	✓	✓	✓	✓	✓

FIGURE 6: Summary of Fenrir's roles and permissions.

Fig. 6 summarizes the rights of each role.

(a) Content providers. This first layer is the only one composed simultaneously of workers from both the automakers and suppliers (cf. §III). Thus, it is necessary to split it into two different roles, as follows. (1) Internal Developers, whose principal role is to integrate and test the incoming suppliers' proposed software and manage software installability (i.e., the software authorization green list described in §V-C). However, as there is also potentially an internal software development department within each automaker, this role also allows new software proposals to be produced and published in the chain. (2) External Developers, whose role is exclusively to develop new software products for sale either to automakers or to end users. While those from the first role can access the source code of any software published, the second role type will only have access rights to applications developed by their company unless authorized by the other concerned company.

(b) Content Consumers. This layer comprises all the vehicles of the different automakers. As vehicles are the target of the software deployment process, they will maintain a partial copy of the blockchain, which is generated by the central nodes per the dependency management mechanism described in §V-D and the pruning mechanism in §V-E. However, while new software applications must be always retrieved from the blockchain's central nodes for subscription control, updates for already-installed software can be retrieved directly from other vehicles to accelerate update campaigns.

(c) Central nodes. This layer orchestrates the interactions between all the previous roles. Additionally, it is responsible for maintaining a distributed and synchronized copy of the main chain, SPS repository, software submission queue, authorization management key repository, and the distributed green list. These nodes also help to correct other corrupted nodes, handle the distribution of different automakers' green lists, consider the dependencies between software blocks and automakers' preferences to generate the GSI and BSI, store and propagate key-related requests, and control the update submission process.

Henceforth, the ensemble of Internal Developers, Content Consumers, and Central nodes will be referred to as automaker nodes. Thus, being the *digital* identity of all the automaker nodes that can be matched to their real identity, Fenrir bases their proof of authenticity on this, proposing an identity and time-based key management mechanism. Fig. 7 illustrates the detailed key object structure description. The key authorization manager service ensures these keys' addition, verification, and replication; these steps are complete on the central nodes and partial on those of the content consumers. As these three processes are independent, we will detail them separately in the following paragraphs.

First, we highlight the cases in which a new key may need to be added to the distributed key management service hosted by the central nodes. For the automaker staff keys, there are

Key object definition

A key must be given to anyone wishing to participate in the Fenrir network, whichever their role is. Keys are used for the proof of authority that allows the software integrity to be verified.

Time-related fields

Field	Description
Creation date	The administrative date upon which this key was introduced in the system. Used to verify the validity of the endorsers when adding the key.
Start date	Start effective date from which time the key can be used.
End date	Effective end date from which the key will no longer be valid.

Identity-related fields

Field	Description
Company ID	The company ID to whom this developer belongs. This attribute is null for any non- automaker developer, since Fenrir aims to maintain the free-to-submit philosophy. However, the company ID must be valid for any non-external developer.
Role ID	This field can take a value from the four described before; however, if the desired key does not have a non-null correct Company ID, it cannot take any role other than *External Developer.*
Real life data	In this field, real life data are placed to allow the real identity of the developer to be identified and traced for legal purposes.

Application authentication fields

Field	Description
Public Key	The key allowing to decryption of user signatures.
Key ID	Reference number for administrative purposes.

Key authentication fields

Field	Description
Key Object Hash	Hash of all the aforementioned parameters.
Endorser signature list	List containing the Key IDs of the request validators and signature of the *Key Object Hash* to ensure their veracity.

FIGURE 7: Key object formal definition.

two distinct scenarios. Scenario (1) is when an automaker, having already joined Fenrir, wants to add a new key or set of keys for its workers, and scenario (2) is when a new Automaker wants to join Fenrir for the first time. For the common use case, i.e., scenario (1), a key object must be generated. This request must then be endorsed by a set of workers who have a registered key within the same company—five such endorsers are required by default. However, whenever a new automaker wishes to join the network, most of the network must agree to this request. Therefore, the first key generated for a new automaker must be endorsed by at least half of the existing automakers in the network. Subsequently, adding the first keys of this automaker (until it reaches the five keys threshold) will exceptionally only require the endorsement of its first key. However, this mechanism relies on collaboration between automakers, thus, some automakers may potentially deliberately not approve or slow down new joiners to maintain exclusivity or prevent competitors from gaining an advantage. To mitigate this problem, contractual

measures may need to be established from the outset of the system. On the other hand, keys for external developers will not be added to the key management service until an automaker endorses their application. Note that Fenrir does not require specific vehicle keys and will instead use those assigned during the vehicle's construction.

Whenever a new key is added to a central node or a vehicle's local storage, complete verification of the key object structure is required. To do so, the authorization manager will check the key's chain-of-trust recursively until arriving at an already-verified set of keys. To verify each key, it will then match the creation dates, authorizations, and company IDs and check the integrity of the signature match. Notably, the key verification is not necessary for external developers, but the verification of the endorsements lists, since their keys will only be stored once an automaker endorses their application. The full endorsement process is detailed in §V-C.

Finally, as Fenrir is conceived as a distributed system, once a key is added to any central node, and before the addition is made fully effective, the request will spread the key to the other central nodes on a two-phase Paxos-like consensus basis, comprising first a promise phase, in which the key object is verified by all the nodes, and a commit phase, in which the nodes agree to add the given key to their internal key storage repository. It is only at this point that the key addition is effective. Note that the spreading process is the same for SAPckg addition, key addition, and green list endorsement additions.

C. SOFTWARE PACKAGE AUTHENTICATION

In this subsection, we complete the SAPckg definition and expand on its authenticity and integrity verification, which are key aspects of using blockchain-based solutions to build a software distribution framework (§IV).

1) SAPckg format and verification mechanisms

As noted above, Fenrir's SAPckg is formed from an SPS and its associated STB, mostly like a smart contract, which is stored in the MSC. Fig. 8 provides a detailed definition of these structures. Note that the temporary SAPckg data structure stored in the PSR waiting to be approved by an Automaker is highly structurally similar to the classic SAPCkg, changing only the STB ID to a temporary one and removing the *Authenticity and smart contract verification* fields. Thus, to verify the SAPckg authenticity and integrity, Fenrir proposes a simple three-step verification process: (1) STB validation, performed by comparing the hashes and the validity of the signatures and green-list endorsements (§V-C2) from the precedent, actual and posterior STBs, (2) ensuring that the SAPCkg ID link matches both the STB and SPS and (3) verifying the SAPCkg ID signature consistency. However, to prevent truncated chain attacks, the SAPckg will not be considered final and, thus, will not be used unless its STB is at least six blocks deep, as in other classic Blockchains, with at least three different additional publishers between these last six blocks (i.e., the one from the block being verified and at least three other publishers).

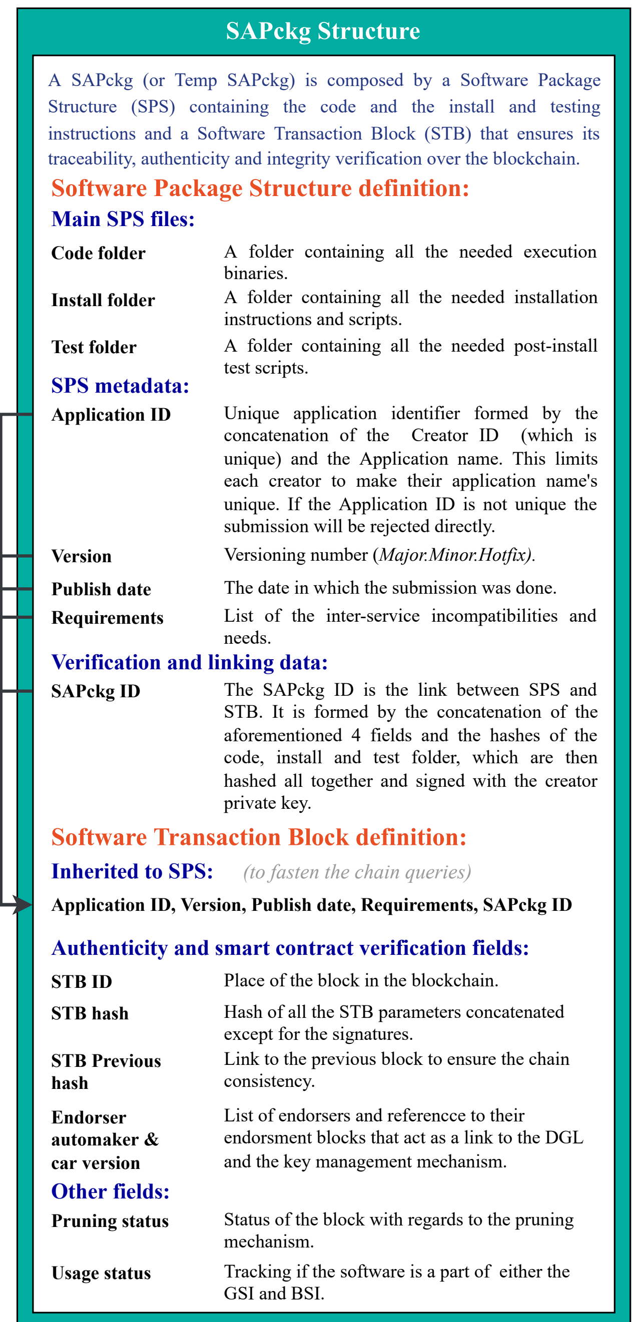


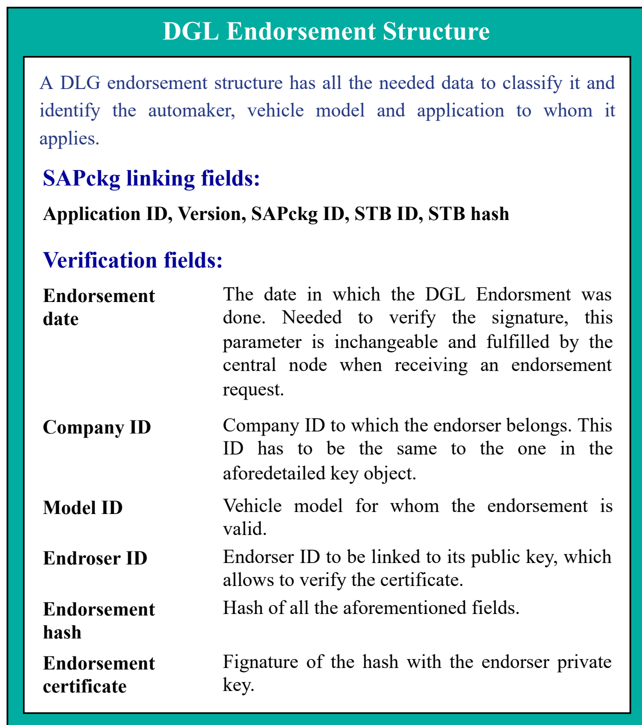
FIGURE 8: SAPckg formal definition.

Fenrir also adds different verification levels (critical, medium, or, none) to improve performance by reducing the verification's security level as required. Thus, critical verification would check the whole ensemble of hashes, medium verification all the hashes but the file hashes, and "none" would verify only the links between the SPS and STB. Thus, the first time a block is added, the verification level will be critical, however, once inside the secure vault, the verification level is reduced to medium to improve performance unless the system suspects malicious behavior, in which case it will

switch back to critical verification.

2) The Distributed Green List Ledger (DGL)

As a means to preserve automakers' authorization control over what can and cannot be installed in the vehicles individually, Fenrir benefits from the aforementioned identity-based and role-based authority proofs to build an auxiliary software authorization chain, the DGL. The DGL comprises an authorization transaction graph in which the details of the endorsements (whose structure is defined in Fig. 9) for a given software package are stored. To keep the MSC clean, Fenrir also uses the DGL and endorsements to regulate the addition of SAPckgs to the definitive chain, from which they will never be erased, requiring a temporary SAPckg endorsed by at least one automaker to be added to the MSC. Endorsements from other companies can also occur after adding the block to the chain. However, even though these two options appear to be the same process, their behavioral mechanisms differ.



DGL Endorsement Structure

A DLG endorsement structure has all the needed data to classify it and identify the automaker, vehicle model and application to whom it applies.

SAPckg linking fields:

Application ID, Version, SAPckg ID, STB ID, STB hash

Verification fields:

Field	Description
Endorsement date	The date in which the DGL Endorsment was done. Needed to verify the signature, this parameter is inchangeable and fulfilled by the central node when receiving an endorsement request.
Company ID	Company ID to which the endorser belongs. This ID has to be the same to the one in the aforedetailed key object.
Model ID	Vehicle model for whom the endorsement is valid.
Endroser ID	Endorser ID to be linked to its public key, which allows to verify the certificate.
Endorsement hash	Hash of all the aforementioned fields.
Endorsement certificate	Fignature of the hash with the endorser private key.

FIGURE 9: Endorsements formal definition.

While endorsing a SAPckg already in the network only requires generating some endorsements (five by default), endorsing and adding an STB off the chain to the MSC is slightly more complex. An automaker seeking to add a temporary application proposition to the MSC must endorse it (in addition to five further endorsements by default); they must then prepare the final SAPckg including the missing information and, once this is done, ask the development company to re-sign the final block. Even though this mechanism might appear tedious, by including it, Fenrir can considerably reduce the MSC size and decorrelate the installability of a given service from its STB, which can then change with hot-fixes, contract evolutions, or legislative changes, revoking the previous endorsement if the automaker no longer wishes to install this application. Note that if all endorsers stop supporting a certain SAPckg, it will not be removed from the final chain. A global overview of the entire pipeline is presented in §V-F.

Finally, as shown in Fig. 10, given the previously described secure key-authorization and block verification mechanisms, the integrity of the endorsement can be readily traced without needing to link the endorsements. Thus, when verifying and retrieving the software certificates, our approach is based on the indexing patterns of the databases, with the DGL organized as a four-leveled graph instead of a classical list [37]. Furthermore, we simplified the indexing model by picking a classic automotive semantic classification—automaker, car model, and software block ID (cf. Fig. 10).

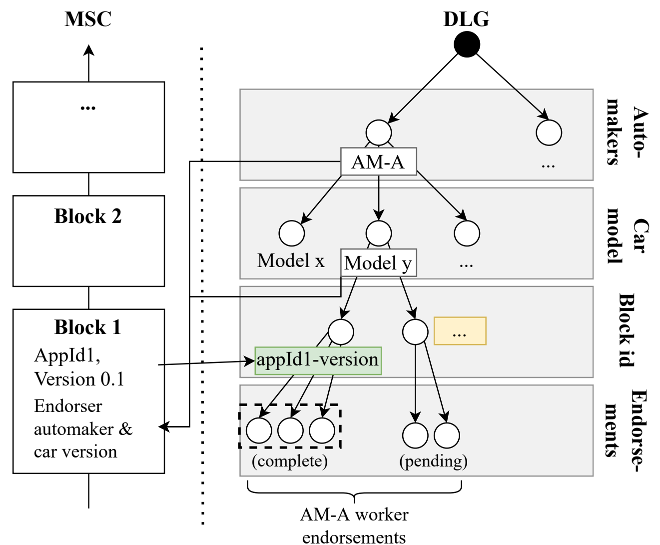


FIGURE 10: DGL interactions with the MSC.

D. DISTRIBUTED INTER-SERVICE DEPENDENCY MANAGEMENT

To prevent software incompatibilities and given the high computational, storage, and energy costs of transmitting an update, Fenrir implements an inter-dependency calculation mechanism. This mechanism is distributed through the Central Nodes (or the updater vehicle in the case of V2V updates), referred to hereafter as updater nodes. These nodes pre-calculate the software dependencies before sending the global software image to the vehicles to be updated, which are hereafter referred to as receiver nodes.

To prevent software incompatibilities and considering the high computational, storage, and energy costs of transmitting an update, Fenrir implements an inter-dependency calculation mechanism distributed through the Central Nodes (or the updater vehicle in the case of V2V updates), referred to hereafter as updater nodes, which pre-calculate the software

dependencies before sending the global software image to the vehicles to be updated, hereafter referred to as receiver nodes.

This process takes place once the SAPckg is already in the MSC and represents one of the last phases of the software deployment process. Whenever a vehicle wishes to check for new software availability, it will send its vehicle manifest, i.e., a file containing the vehicle's identity and a list of currently installed software, to check for updates and new desired software. After receiving this manifest, the updater node will then check inter-software dependencies and incompatibilities and generate a global dependency graph to determine the order to preserve when determining the different software versions. Note that this first step is only performed whenever the graph in the updater cache does not contain all the requested software.

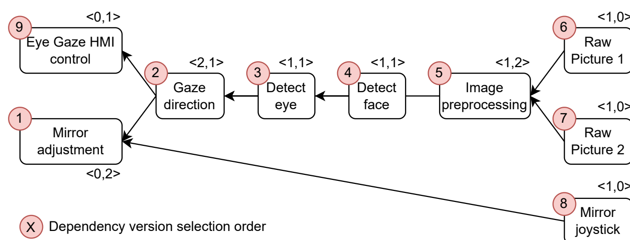


FIGURE 11: Dependency calculation example based on the previously presented use case figure.

As shown in Fig. 11, the global dependency graph algorithm will consider all the inter-software dependencies on the graph and generate a tuple $< X, Y >$ for each of the applications, in which X is the number of software packages for which this software is a dependency and Y is the number of dependencies of this software (e.g., $< 2, 1 >$ for Gaze Direction or $< 0, 2 >$ for Mirror Adjustment in Fig. 11). The algorithm will then calculate the global software image to send to the vehicle with this list of tuples. This algorithm is iterative and is performed as follows:

1) The algorithm selects the node (between those not already selected before) with the lower X and higher Y and selects the most possibly updated version, given that one version cannot be selected if it creates dependency problems with other software in the graph or if it contains a dependency to other software not included in the graph.
2) The algorithm will select the most up-to-date version, in case it is compatible with more than one, for all the different dependencies, starting for thus with higher Y, then X and, ultimately, alphabetic order. This process will continue iteratively for the dependencies of the dependencies until a point is reached at which there are no further dependencies.

However, consider a scenario in which the process does not allow for the creation of a fully compatible set that contains all the software in the list. In that case, the algorithm will then mark the software that has a compatibility issue and treat it at the end. It will then continue the process with the most updated version of the next package with lower X and higher Y. If the algorithm reaches a point where all the software packages have been tested unsuccessfully, it will restart the process and select the second most up-to-date version instead. This process will then be repeated until either a stable version is found or all the software blocks are marked, in which case the request will be impossible to resolve and no global image will be sent to the vehicle. Instead, a message will be provided in this case detailing the services creating the incompatibilities. This process accompanies the software compatibility definition described in §V-A.

E. LIGHT AND RESILIENT SOFTWARE PACKAGE STORAGE

Considering that vehicles have intermittent network access and long inactivity periods and given that, most of the time, vehicles are parked in underground facilities without network access, being able to correctly boot from a previous software version without needing to interact with the Cloud or other Vehicles is mandatory. Thus, vehicles need to possess a software image backup to boot without requesting extra information in case of corruption. To generate this backup, Fenrir first cleans up the chain and keeps the strictly necessary information to verify and install the GSI and BSI; this process is classically known as pruning.

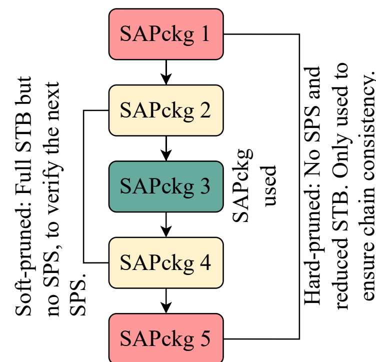


FIGURE 12: Pruning example.

Fenrir's pruning mechanism begins by classifying all the software in the chain into three categories: not pruned, soft-pruned, and hard-pruned. As shown in Fig 12, not pruned blocks are the software applications that the vehicle will use for the GSI and BSI, soft-pruned are those used to verify the usable blocks, and hard-pruned are those used to maintain the chain consistency. Then, to tag the chain blocks, Fenrir starts defining those directly used, which will not be pruned. Afterward, it will tag any block next to a not pruned block as soft-pruned and the remainder as hard- pruned. §VI shows the advantages of pruning blocks to reduce the chain storage needs, which is essential for the vehicles' operation. After pruning, Fenrir proceeds by compressing and saving the backup over the previous backup. However, this process has a considerable impact and thus cannot be performed each time

a software package is added. A detailed study of the overhead and optimal frequency of this process is presented in §VI.

F. SOFTWARE AUTHENTICITY & TRACEABILITY DISCUSSION

At this point, we have presented all the different mechanisms that allow authenticity to be preserved throughout the software distribution pipeline, however, these must now be connected. This process begins when an application's development is complete and a company wants to publish this application to Fenrir. The developer will start by generating their application proposal, in which they create a temporary SAPckg (as detailed in §V-C). As noted above, this SAPckg is like the final SAPckg but without the STB ID, hashes, or relation with the DGL. This package is then submitted to the PSR. Once in the PSR, as presented in §V-C2, the automaker can then endorse their submission request, generate the final SAPckg, and push it to the MSC. Once in the MSC, other automakers can endorse the SAPckg and authorize its installation into their vehicles. Subsequently, whenever a vehicle requests new updates and software packages, this block will form the vehicle's GSI and BSI (cf. §V-D). Once the software and set of keys allowing the package to be verified are downloaded to the vehicle, it will store them safely, update its backup, and prune redundant information from its local chain (§V-E).

Typically, blockchain-based solutions provide a high level of anonymity for both users and their transactions, which strengthens the traceability of malicious behaviors [38], [39], [40]. However, in Fenrir, the migration from proof-of-work to an identity-based proof-of-authority, allows us to link a software block to the identity of its producer and the identity of its acceptors within each of the Automakers, as suggested previously in the literature [41], [42]. In this way, we can easily provide a method for tracing the accountability whenever a malicious behavior occurs. Furthermore, thanks to the dependency-management mechanism, we can easily trace the software subset installed, thereby helping debug teams to find and correct threats.

VI. EVALUATION

A. EXPERIMENTAL SETUP

To ensure that the simulations were as realistic as possible, we implemented the central nodes in three automotive cloud-like nodes in Amazon EC2 instances (t2.micro – Ubuntu20.04) and the vehicles using two Raspberry Pi 3b v1.2 units in order to be able to test both V2C and V2V interactions. Note that we chose to use Raspberry Pi 3b because of its proximity to the automotive solutions deployed (such as NXP S32 G) with regards to its chipset ARM Cortex-A53. However, since the network between the nodes was not representative of the vehicle characteristics, we elected to remove the ping time between nodes from the results. Thus, the results presented in this section purely evaluate the performance of the mechanism itself rather than the communication channel.

Furthermore, to improve the readability of the data, as shown in Fig. 13a, we do not illustrate the whole data set but only the best-fit second-degree polynomial trend lines. Each data point was measured until its distribution followed a normal law with a standard deviation lower than 5%, allowing the mean for each to be considered a representative summary.

B. ENERGY CONSUMPTION MODEL DESCRIPTION

As energy consumption is crucial in the automotive sector, particularly for non-electric vehicles, we studied the energy consumption of the Raspberry Pi 3b for the different mechanisms in our proposed system. We started by measuring the real consumption associated; however, given the proximity of the first tests energy consumption measures to the consumption model proposed by PowerPC [43], we chose to simplify the test set-up by basing our work directly on this model. In this model, the energy consumption is estimated as:

$$P_{u,d,if}(W) = P_{idle} + P_{CPU}(u) + \sum_{if} P_{Nw,idle} + P_{Nw,dl/ul,d}(r) \quad (1)$$

Where u is the RPI %CPU, d is the data sent or received (in MB/s), and if is the interface (Wi-Fi, Bluetooth, Ethernet, etc.). In our case, all scenarios were tested with LTE-M (which can be taken as 1.54 times the energy consumption of Wi-Fi [44]), thus $P_{idle} = 1.5778W$, $P_{CPU}(u) = 0.181W \cdot u$, $P_{Ltem,idle} = 1.45068W$, $P_{Ltem,download,d} = 1.54 \cdot (0.057W + 4.813e^{-3} * d)$ and $P_{Ltem,upload,d} = 1.54 * (0.064W + 4.813e^{-3} * d)$. Note that for the data volumes handled in this work, we can neglect the variable part of the data transmission and approximate $P_{Ltem,download,d} = 0,08778W$ and $P_{Ltem,upload,d} = 0,09856W$. However, in this test session, we assume that the connection between on- and off-board components (or V2V) is direct and through LTE-M. Thus, we do not consider all gateway to cloud energetic communication costs as these may change considerably depending on the company's implementation choices.

C. BENCHMARK SPECIFICATION JUSTIFICATION

The data set used in our experiments consist of a realistic ensemble of keys (RSA-2048-based) and SAPckgs following the inter-software dependenciesand characteristics described in §III and detailed in appendix A (cf., §IX-A). We test all the different SAPckg profiles described (i.e., AOTA, FOTA, SOTA, and MOTA), which limits the number of simultaneous SAPCkgs to 200 the (limit imposed by the MOTA tests). The size of these packages is around 1 kB for configuration updates (AOTA), 1.133 MB for firmware or delta software updates (FOTA or ∆SOTA), 10.531 MB for full software package updates (SOTA), and 33.5 MB for media updates (MOTA). However, to be able to explore the limits of the solution and its behavior in unexpected situations more unnatural dispositions (i.e., if all the packages in the chain haveare from the same size), we add - when needed -

unnecessary mock files to enlarge the package size. Note that, even though nowadays some update packages are commonly GB-scale, with the emergence of OTA update frame-works, we assume that huge media update packages such as GPS updates will be adapted to the system constraints with more periodic, smaller OTA updates. Consequently, we do not exceed 500 simultaneous keys as this number is sufficient to verify 200 packages. These packages and keys allow us to test the authenticity of the full software deployment pipeline and evaluate the behavior and overheads of each of the aforementioned mechanisms.

D. EVALUATION OF THE KEY MANAGEMENT MECHANISM

In this section, we investigate the performance of the key management mechanisms to prove their adequacy for both on-board vehicle systems and vehicular Cloud platforms, i.e., the off-board systems.

Fig. 13a and 14 show the result of the function execution performance while adding and deleting keys, in addition to its evolution with the number of keys already present in the vehicle. The displayed performance values may appear slow given current computational power (i.e., the time to add/delete a key oscillates between 1.1 to 1.5 seconds on-board and between 0.75 to 0.875 seconds off-board). However, as noted above, the key management mechanism is the layer that the security of our Proof-of-Authority mechanism relies upon. Modification of the key set will only occur when a passenger chooses to install a new application originating from an unknown developer, the impact is minimal given the security and traceability advantages that it brings to our solution.

However, a comparison between block addition with and without backup highlights that backups have a marked impact. As shown in Fig. 13b and 13c, the backup update increases the CPU load of the OTA master node to almost 75%, significantly impacting the node energy consumption. However, the backup has less impact on the execution performance, with an execution time increase consumption. However, the backup has less impact on the execution per-

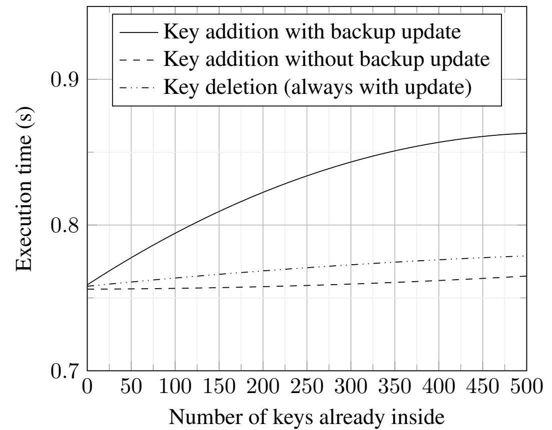


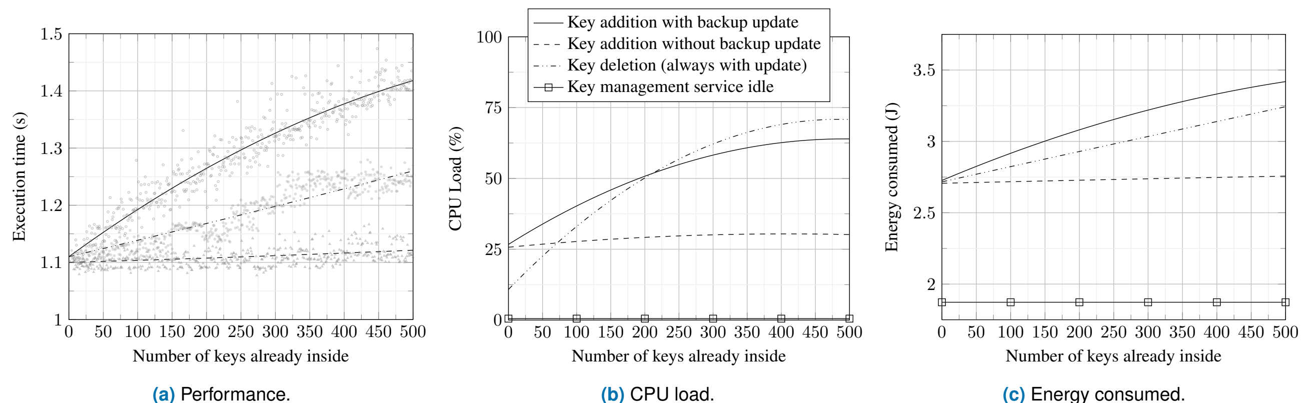
FIGURE 14: Performance of the off-board key management mechanisms.

formance, with an execution time increase of 25% recorded for 500 keys. As this layer is the center of the integrity and authenticity control and, as noted in §V-F, this mechanism will occur only when adding new updates and pieces of software with previously unknown keys, which will not occur frequently, the impact of this mechanism is negligible compared to the run time of the full install mechanism, which typically takes nowadays around 30 minutes in most of the constructors. Thus, despite the performance overhead, the key backup update will always be performed after adding a set of keys in a software deployment cycle. However, in terms of backup generation in the off-board nodes, this mechanism must be activated and performed each time a new key is added as these resources are even more negligible at the computational scale of cloud computing nodes.

E. EVALUATION OF THE SOFTWARE MANAGEMENT LAYER.

1) On-board software verification mechanism evaluation

Before investigating the addition of blocks to the chain, we first focus on the performance of the software verification mechanism described in §V-C. Fig. 16 depicts the impact of the update package size on the verification execution time. As anticipated, increasing the package size also increases



(a) Performance. (b) CPU load. (c) Energy consumed.

FIGURE 13: Evaluation of the On-board key management mechanisms.

the time to calculate the hashes and signatures needed to verify the SAPckg. However, as shown in Fig. 16, the number of blocks already inside the MSC does not influence the verification mechanism performance.

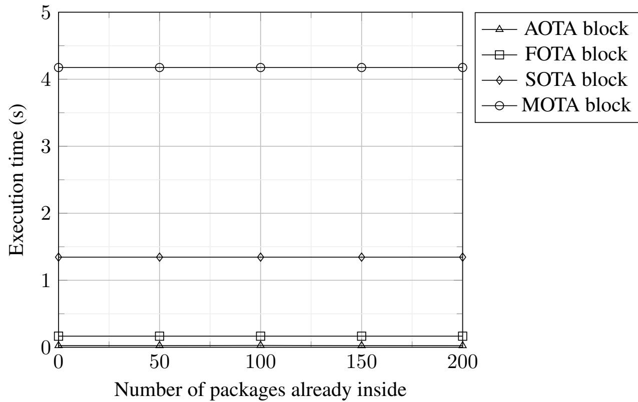


FIGURE 16: Evaluation of the *critical* verification mechanism when adding more blocks to the MSC.

We now consider the performance of the verification mechanism in response to changing its verification level (i.e., Critical, Medium, and none) as presented in §V-C, Fig. 17 shows that the size of the SPS only impacts the critical verification as this level is the only one in which the hash of the code files is calculated. It illustrates once again that the impact of the blockchain mechanism itself is negligible; all of the overhead is due to the hashing verification itself, which is mandatory to preserve package integrity irrespective of the underlying storage structure. In addition, from Fig. 17, we can also confirm the system's suitability for real-time applications given the different verification levels and package sizes. Thus, in a fully dynamic SOA embedded architecture, AOTA, FOTA, SOTA, and MOTA verification can be performed in real-time, with the services chosen at will to match the user's needs. However, for the addition of new FOTA, SOTA, and MOTA packages to the MSC for the first time, the vehicle will require a certain calculation time, adding a delay between the vehicle receiving the package and when it will use it. This could potentially limit future shared driving possibilities, as the services must be pre-submitted to the MSC before using them. However, since Fenrir is based on the concept of global software compatible images that give the orchestrator a range of software applications that it can use, any desired application will already be in the GSI and, thus, already within the vehicle's local MSC.

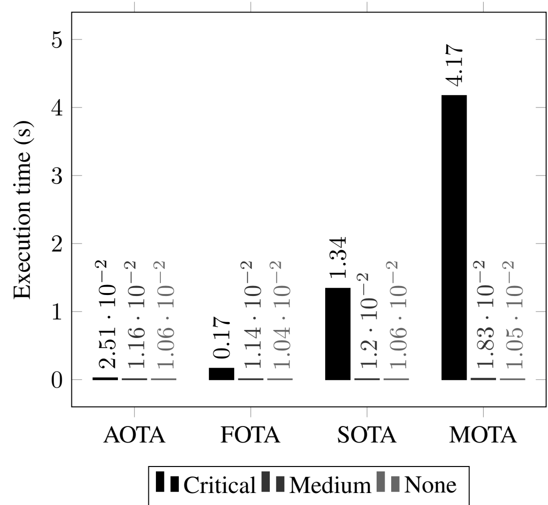


FIGURE 17: Evolution of the software verification mechanism with different *critical* levels.

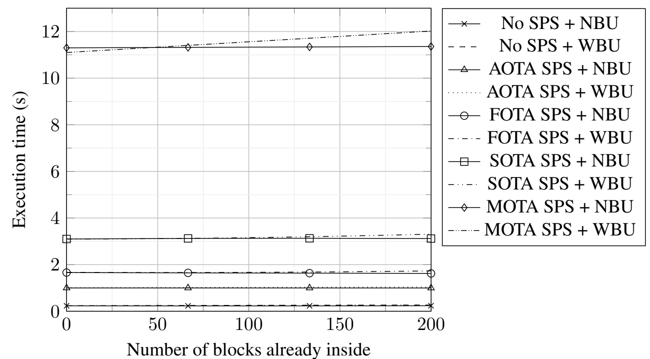
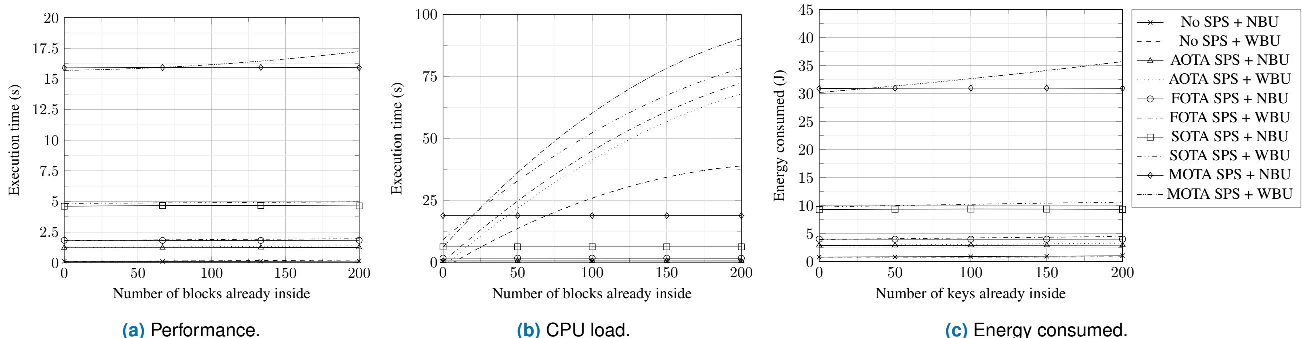


FIGURE 18: Performance of the off-board block addition mechanisms.

2) Software addition mechanism performance evaluation

In terms of the performance of the software addition mechanism, as shown in Fig. 15a and 18, the time to add a block of any kind is within acceptable levels for any package type for



(a) Performance. (b) CPU load. (c) Energy consumed.

FIGURE 15: Performance of the on-board block addition mechanisms. SMALL = 956 Bytes, MED = 1.133 MegaBytes, BIG = 10 MegaBytes, HUGE = 33.5 MegaBytes. (WBU = with backup update and NBU = No backup update)

both off-board and on-board scenarios and does not increase with the number of blocks already inside, as long as the block backup is not updated after the addition. Thus, once again, the backup will be generated once all the GSI software is stored in the MSC rather than when the software is added to keep the overhead low. However, in the central nodes, the backup will be performed each time a new block is added to the MSC, as the resources on the Cloud side are readily scalable, and the MSC of the central nodes is the reference for all other nodes in the infrastructure. Other techniques, such as creating the backup from sub- backups, could be implemented to reduce the impact of creating a full backup each time.

3) Blockchain pruning mechanism evaluation

Blockchain pruning mechanism evaluation: In this subsection, we study the impact of the pruning mechanism on system performance depending on the number of nodes to be preserved. Fig. 19 illustrates the impact of the number of blocks to be saved on the backup generation. This study also includes the pruning calculus, whose impact is negligible compared to the whole backup generation mechanism. Fig. 19 shows that the impact on the backup generation is significant (over 75% performance improvement on the 80% pruned example). This mechanism will only become more effective as the number of blocks in the chain increases. The pruning mechanism also greatly impacts the package size, reducing the SAPckg size to almost the size of the STBs (a few kB) for the pruned blocks, representing an enormous reduction for SOTA, MOTA, and even FOTA SAPckgs.

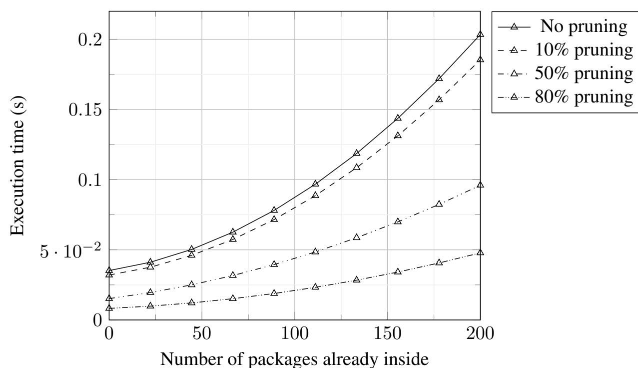


FIGURE 19: Impact of pruning mechanism with 20% of AOTA, 70% of FOTA or ΔSOTA, 5% of SOTA, and 5% of MOTA.

4) Block retrieval mechanism evaluation

Finally, as the last block management-related mechanism, we consider the time required to retrieve a block from the blockchain. Specifically, this is the time that another ECU, usually the OTA orchestrator, will take to safely retrieve a block from the MSC and for it to be ready to install. As shown in Fig. 20, the time for this operation is low because the verification level used whenever retrieving a block from the local MSC is only *medium* instead of *critical*. In addition, the communication impact is negligible as this action occurs between two ECUs connected directly via Ethernet.

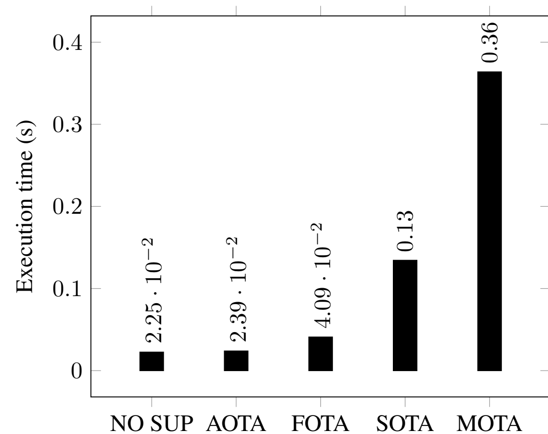


FIGURE 20: Influence of the SUP size when retrieving a block.

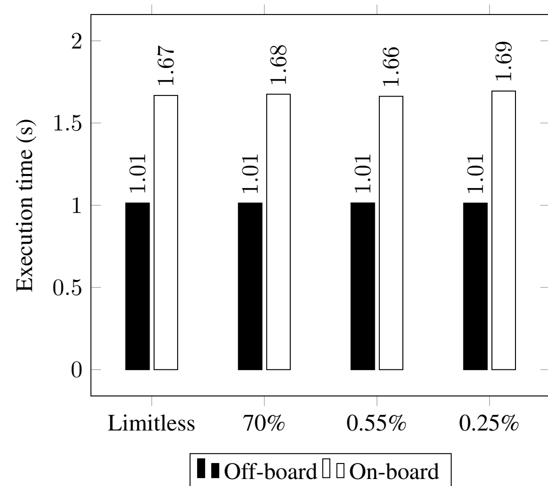


FIGURE 21: Influence of the % of install limitations to calculate the global software image.

F. EVALUATION OF THE DEPENDENCY CALCULATION MECHANISM

Fig. 21 presents the results of calculating a global software image both on- and off-board. Vehicle or automaker software limitations do not influence the generation time or the CPU load (not presented in this paper but 0–5% for off-board and 20–30% for on-board). However, while 1.7 seconds is a relatively short time for executing a vehicle dependency management algorithm, this time might be too high for V2V updates on the road. Thus, to avoid overloading nearby vehicles, this mechanism will only be performed when the vehicle is stopped (traffic lights, traffic jams, parking lots, etc.); in other cases, communication will occur directly to the Cloud. However, even after updating from another vehicle, a verification request is sent to the Cloud as soon as possible to verify the hashes in the chain.

G. EVALUATION OF THE SYSTEM OVERHEAD

As described above, the Fenrir framework overhead can be divided into two phases: a first phase involving adding the incoming software blocks to the vehicle's local chain and a second phase whenever these blocks are used. However, even though these two phases are entirely independent, their performance depends mainly on the integrity verification mechanism, which is described in detail in §VI-E1. From this section, and considering the typical automotive industry update cycle of around a dozen minutes per service update (value given on the discussion with industrial experts), the overhead added by this verification mechanism can be estimated as 0.579% both in terms of performance and energy consumed for the worst-case verification scenario (i.e. MOTA package verification); thus, this overhead can be considered negligible. Combining the definition of the block addition mechanism presented in V-C, the results shown in §VI-E2, and the key addition mechanism results in §VI-D, the overhead of this mechanism can be estimated as 2.911% in a worst-case scenario in a MOTA SAPckg must be added to a whole MOTA SAPckg chain and all the keys are needed to verify this software. However, despite the increased overhead, this value is still not significant in terms of the whole system performance given the low frequency.

H. THREAT TO VALIDITY ANALYSIS

This section, focuses on the threat-to-validity approach presented in [45], [46] and, in particular, [47], which developfocuses on the verification of the validity of empirical software-based experiments. In these papers, the authors propose four main threats on which to focus when conducting a threat-to-validity analysis: *threat to conclusion validity*, which leads to an incorrect conclusion about a relationship in an observation; *threat to internal validity*, which increases the difficulty of finding causal links between variables and events; *threat to construct validity*, which focuses on how well a test measures the concept it is designed to evaluate; and, *threats to external validity*, which refers to the generalization of the results. However, from the sub-properties of each type we will not focus on those that are not applicable

TABLE 2: Threat to validity analysis.

Threat	Sub-property	Analysis
Conclusion	Statistical Validity	The tests were repeated until a standard deviation of less than 2% was achieved. The test results followed a normal distribution $N(\mu, \sigma^2)$, with μ being the mean and σ being the standard deviation, which allowed us to make use of the mean and trend lines as good representatives of the functional behavior.
	Statistical assumptions	The test sessions were planned to test each isolated mechanism of Fenrir. This way, we could easily set aside the potential correlations between the test sessions/experiments.
	Lack of expert evaluation	The design and evaluation of Fenrir were iteratively discussed with experts from both our industrial partner and laboratory members, taking their concerns into consideration to re-adjust some aspects of the design and test sessions.
	Reliability of the measures	The system was tested in multiple test sessions over several months, consistently achieving similar results.
	Reliability of the test-sessions	The test set-up was the same for all the experiments (except for the times when we tested V2V instead of V2C).
	Lack of data pre-processing	Once we analyzed the data from the first test sessions, we planned precise sessions to complete the missing data spots that could have influenced our conclusions.
Internal	Deficiency of the set-up	The set-up was isolated from other machines and only turned on for the experiments. We did not observe any abnormal signs of network or hardware under-performance.
	History	The system was tested in multiple test sessions over several months, consistently achieving similar results. The results were normalized to decrease the influence of unrelated events.
	Maturation	The time-scale over which we conducted the experiments was too short to appreciate its influence on the hardware life-cycle. Furthermore, the test set-up was re-flashed for each test; this way the environment, was completely clean and isolated from past and future experiments.
	Testing	The system was re-flashed before each experiment. In this way, there was no test correlation despite conducting it several times in a row.
	Treatment design	The material chosen was selected to match the actual automotive–cloud environment, as stated in §VI-A.
	Subject / Sample selection	The data set (cf. §VI-C) was chosen based in the literature and discussions with our industrial partner to cover future automotive use-cases.
	Incompleteness of data	Further discussions with other automotive groups would be of interest to complete our data set with their specific use-cases.
Construct	Monooperation bias	The study includes more than one independent variable and evaluates more than one mechanism.
	Monomethod bias	The different mechanisms were evaluated according to multiple metrics (i.e., energy & network cons., CPU% , performance, etc.).
	Measurement metrics	The measurement metrics were completely objective, measuring the CPU consumption and the end-to-end time directly from the board. We also measured the energy footprint directly with a multimeter.
External	Representation of the population	The data set was chosen based on the literature and the discussion with our industrial partner to cover the future automotive OTA update use-cases.
	Context of the study	The experiments were conducted over several months, between January and June, and at different times and locations, which enhanced the generalizability of the findings.

to our experiment (i.e., mortality, limitation of treatment, motivation, appropriateness of data, interaction with different treatments, treatment testing, hypothesis guessing, evaluation apprehension, and representation of the setting) and others that we are not in a position to objectively analyze (i.e., fishing for the result, ignoring relevant factors, theory definition, experimenter bias, and experimenter expectations). However, the goal of this experiment was to study how to strengthen the data integrity of the current OTA software-deployment mechanisms without elevating the system overhead; neither we - nor our partners - had a special interest in basing our solution on the blockchain, which helped us to remain objective about the advantages and flaws of our solution. Table 2 details the precise analysis for each sub-property. Each sub-property is extensively defined in [47].

VII. DISCUSSION

Despite the encouraging results, considerable progress is still required in the automotive industry before being able to move to Fenrir's application store framework. Marked changes in the development process need to occur, including addressing the granularity of SAPckg packages [48], [49], which is currently at an application level and not at a service level, and the frequency of updates, which remains relatively low compared to IT systems. Additionally, significant effort must be put into the entire standardization and software reuse process [50] between different companies to reduce software development and maintenance costs, thereby paving the way for new innovative services. In addition, to improve the software installation process, hardware and software should be decoupled , moving towards a higher-level business-oriented top-down software conception. This way, the service behavior correctness could be ensured since the development phase thanks to tools such as sandboxing or shadow mode deployment, thereby easing current on-board deployment constraints [51].

In addition to the changes required in the automotive industry, the development of Fenrir also requires some open issues to be addressed that we did not discuss in this study. First, the complete transformation of automotive software is challenging to achieve in the short term, so we must study how to integrate legacy software into the proposed new deployment pipelines during the transition stage. A promising approach would be automatic parsing and wrapping mechanisms allowing for adapting and standardizing legacy packages, such as in [52]. Second, developer-declarative inter-software dependency potentially poses a threat when increasing the number of authorized services; thus, Fenrir will require some statistical model allowing potential failures that might be unnoticed in the development phase to be traced and detected [33]. Finally, we still need to work on adapting Fenrir for shared driving services and collaborative fleet management services, whose dynamicity may represent a problem for the system security, as well as for large multimedia files that could slow down system performance considerably, effectively blocking the blockchain agent.

VIII. CONCLUSION

Software is becoming increasingly important for the automotive industry; thus, solving the ever-present software-related problems remotely has become a critical aspect and a great vulnerability threat for vehicles. In this paper, we presented Fernir: an inter-automaker blockchain-based application store framework where the instability control is preserved by the automakers. Fenrir's mechanisms allow protection from critical security attacks in the most likely scenario, in which the attackers can perform man-in-the-middle attacks but have not compromised at least a certain number of signing keys (which can vary depending on the final implementation). To our knowledge, Fenrir is the first proposed software deployment framework for automobiles that addresses inter-service dependency management to optimize resource and energy consumption through the deployment pipeline. Fenrir is also the only solution of its type to address the challenge of the heterogeneity of the automotive industry's software development cycle by conceiving a hybrid public/private approach in which multiple roles with multiple permissions can handle the interactions between actors and automakers without exposing the chain to malicious publishers. Furthermore, Fenrir offers multiple verification levels to address safety legislation while also keeping energy consumption as low as possible. Finally, we also present an evaluation of both the performance, computational demand, and energy consumption of each of the mechanisms forming Fenrir and demonstrate that the overhead added by our solution for an entire software deployment pipeline (§VI-G) consisting of both deploying and using previously deployed software depends mainly on the verification mechanism and is not significant, i.e., 3.725% for a worst-case scenario and 0.2819% for a typical scenario.

In subsequent stages of the project, further work is required on the user experience to identify the use cases and business models in which Fenrir could be used. In addition, it is also necessary to clearly state when and how V2V or V2C should be chosen to retrieve software blocks. From a more technical perspective, now that Fenrir presents a wholly safe and secure end-to-end software delivery pipeline, further work is required in the service deployment phases (i.e., the application orchestration and efficient installing) as well as on the software architecture to allow dynamic adaption to deployment requirements, control of all relevant services, and life-cycle management. Finally, setting up dynamic data-centered communication reconfiguration between highly dynamic services in this set of applications remains an exciting topic for future research.

IX. APPENDIXES

A. APPENDIX A: USE-CASE SCENARIO DETAILED SOFTWARE PROFILES

In this first appendix, we describegive further details on the test-bench services developed based in close collaboration with STELLANTIS' partners and based on in [26] with collaboration of STELLANTIS. Thus, Table 3 expose de-

TABLE 3: Use case scenario: software profile description.

	Service profile	Communication pattern	Hardware coupling	Size	Depends on	Update frequency	Reusability	Supplier
Interior light actuator	Low complexity service for passenger comfort	Streaming	Yes	AOTA - FOTA	None	Yearly	Medium (mono-company)	Automaker
Raw picture 1 & 2	Real-time, critical, low complexity service for ADAS	Streaming	Yes	AOTA - FOTA	None	Half Yearly	Reduced (mono-hardware)	Tier 3
Mirror joystick	Low complexity service for passenger comfort	Discrete / passenger interaction	Yes	AOTA - FOTA	None	Yearly	Medium (mono-company)	Automaker
Brightness provider	Low complexity service for passenger comfort	Event-based	No	AOTA - (Δ)SOTA	Raw Picture 1	Yearly	High (multi-company)	Automaker
Image preprocessing	Real-time, critical, high complexity service for ADAS	Streaming	No	AOTA - (Δ)SOTA	Raw Picture 1 & 2	Half Yearly	High (multi-company)	Tier 2
Operating hours	Low complexity service for specific use case	Periodic	∼	AOTA - (Δ)SOTA	None	Yearly	Medium (mono-company)	Automaker
Interior light handler	Low complexity service for passenger confort	Periodic / Event-based	Yes	AOTA - FOTA	Interior light actuator & Brightness provider	Yearly	Medium (mono-company)	Automaker
Detect driver	High complexity for specific use case	Streaming	No	AOTA - (Δ)SOTA	Image pre-processing	Quarterly	High (multi-company)	Tier 1
Detect face	Real-time, critical, medium complexity service for ADAS	Streaming	No	AOTA - (Δ)SOTA	Image pre-processing	Quarterly	High (multi-company)	Tier 2
Detect eye	Real-time, critical, medium complexity service for ADAS	Streaming	No	AOTA - (Δ)SOTA	Detect face	Quarterly	High (multi-company)	Tier 2
Driver ID	Medium / High complexity for specific use case	Periodic	No	AOTA - (Δ)SOTA	Detect driver	Quarterly	High (multi-company)	Automaker
Eye open / close	Real-time, critical, medium complexity service for ADAS	Streaming	No	AOTA - (Δ)SOTA	Detect Eye	Quarterly	High (multi-company)	Tier 2
Gaze direction	Real-time, critical, medium complexity service for ADAS	Streaming	No	AOTA - (Δ)SOTA	Detect Eye	Quarterly	High (multi-company)	Tier 2
Insurance tracking	Low/Medium complexity service for specific use case	Event-based	No	AOTA - (Δ)SOTA	Operating hours & Driving ID	Monthly	High (multi-company)	Tier 1
Vehicle leasing	Low/Medium complexity service for specific use case	Event-based	No	AOTA - (Δ)SOTA	Driving ID	Monthly	High (multi-company)	Tier 1
Personalisation	Medium complexity service for passenger comfort	Sporadic	∼	AOTA - (Δ)SOTA - MOTA	Driver ID	Monthly	High (multi-company)	Automaker
Driver monitoring	Real-time, critical, medium complexity service for ADAS	Event-based	No	AOTA - (Δ)SOTA	Eye open / close	Quarterly	High (multi-company)	Tier 1
Eye Gaze HMI Control	High complexity for specific use case	Event-based	No	AOTA - (Δ)SOTA - MOTA	Gaze direction	Quarterly	High (multi-company)	Tier 1
Mirror adjustment	Low complexity service for passenger comfort	Discrete / passenger interaction	Yes	AOTA - (Δ)SOTA	Gaze direction & Mirror joystick	Yearly	Medium (Within company)	Automaker

tails with regards to, in one hand, their, profile, hardware coupling and communication pattern, and, in the other hand, the service size, dependencies, update dynamicity, targeted reusability and developer accountability.

REFERENCES

[1] D. Bischoff, F. A. Schiegg, D. Schuller, J. Lemke, B. Becker, and T. Meuser, "Prioritizing relevant information: Decentralized v2x resource allocation for cooperative driving," *IEEE Access*, vol. 9, pp. 135 630–135 656, 2021.

[2] Z. Tao, Q. Nie, and W. Zhang, "Research on travel behavior with car sharing under smart city conditions," *Journal of Advanced Transportation*, 2021.

[3] X. Feng, E. S. Dawam, and S. Amin, "A new digital forensics model of smart city automated vehicles," in *International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2017, pp. 274–279.

[4] Stout Risius Ros, "Stout's 2020 automotive defect & recall report is a compass of insight for navigating uncertainty," Financial report available online at: https://www.stout.com/en/news/stouts-2020-automotive-defect-recall-report-compass-insight-navigating-uncertainty, 2020.

[5] A. I. Raheem and Y. H. Rashid, "Tracking software in the automotive field: Challenges and solutions," in *Journal of Physics: Conference Series*, vol. 1804, no. 1. IOP Publishing, 2021, p. 012064.

[6] S. Maro, J.-P. Steghöfer, and M. Staron, "Software traceability in the automotive domain: Challenges and solutions," *Journal of Systems and Software*, vol. 141, pp. 85–110, 2018.

[7] R. N. Charette, "How software is eating the car," 2021, iEEE Spectrum. [Online]. Available: https://spectrum.ieee.org/software-eating-car

[8] A. Greenberg, "Hackers remotely kill a jeep on the highway—with me in it," 2015. [Online]. Available: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[9] E&T Editorial Staff, "Serious cyber-security flaws uncovered in ford and volkswagen cars," 2020. [Online]. Available: https://eandt.theiet.org/content/articles/2020/04/serious-cyber-security-flaws-uncovered-in-ford-and-volkswagen-cars-that-could-endanger-drivers/

[10] S. M. Mahmud, S. Shanker, and I. Hossain, "Secure software upload in an intelligent vehicle via wireless communication links," in *IEEE Intelligent Vehicles Symposium*, 2005, pp. 588–593.

[11] K. Mansour, W. Farag, and M. ElHelw, "Airodiag: A sophisticated tool that diagnoses and updates vehicles software over air," in *IEEE International Electric Vehicle Conference*, 2012, pp. 1–7.

[12] M. Steger, M. Karner, J. Hillebrand, W. Rom, C. Boano, and K. Römer, "Generic framework enabling secure and efficient automotive wireless sw updates," in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–8.

[13] D. K. Nilsson and U. E. Larson, "Secure firmware updates over the air in intelligent vehicles," in *IEEE International Conference on Communications Workshops*, 2008, pp. 380–384.

[14] D. K. Nilsson, L. Sun, and T. Nakajima, "A framework for self-verification

of firmware updates over the air in vehicle ecus," in *IEEE Globecom Workshops*, 2008, pp. 1–5.

[15] K. Mayilsamy, N. Ramachandran, and V. S. Raj, "An integrated approach for data security in vehicle diagnostics over internet protocol and software update over the air," *Computers & Electrical Engineering*, vol. 71, pp. 578–593, 2018.

[16] T. K. Kuppusamy, L. A. DeLong, and J. Cappos, "Uptane: Security and customizability of software updates for vehicles," *IEEE vehicular technology magazine*, vol. 13, no. 1, pp. 66–73, 2018.

[17] M. Steger, A. Dorri, S. S. Kanhere, K. Römer, R. Jurdak, and M. Karner, "Secure wireless automotive software updates using blockchains: A proof of concept," in *Advanced Microsystems for Automotive Applications*. Springer, 2017, pp. 137–149.

[18] G. Falco and J. E. Siegel, "Assuring automotive data and software integrity employing distributed hash tables and blockchain," *arXiv preprint arXiv: 2002.02780*, 2020.

[19] E. N. Witanto, Y. E. Oktian, S.-G. Lee, and J.-H. Lee, "A blockchain-based ocf firmware update for iot devices," *Applied Sciences*, vol. 10, no. 19, p. 6744, 2020.

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

[21] E. Sax, R. Reussner, H. Guissouma, and H. Klare, "A survey on the state and future of automotive software release and configuration management," Karlsruher Institut für Technologie (KIT), Tech. Rep., 2017.

[22] A. Vasenev, F. Stahl, H. Hamazaryan, Z. Ma, L. Shan, J. Kemmerich, and C. Loiseaux, "Practical security and privacy threat analysis in the automotive domain: Long term support scenario for over-the-air updates." in *International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS)*. Springer, 2019, pp. 550–555.

[23] J. Michael, A. Cohn, and J. R. Butcher, "Blockchain technology," *The Journal*, vol. 1, no. 7, 2018.

[24] L. Lamport, "The part-time parliament," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 277–317.

[25] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[26] R. Grave, "Cloud vs. embedded: where does the function belong?" in *Automotive Computing Conference (ACC)*, 2021.

[27] N. S. Mtetwa, N. Sibeko, P. Tarwireyi, and A. M. Abu-Mahfouz, "Ota firmware updates for lorawan using blockchain," in *IEEE International Multidisciplinary Information Technology and Engineering Conference (IMITEC)*, 2020, pp. 1–8.

[28] M. Hamza, A. A. Khan, and M. A. Akbar, "Toward a secure global contact tracing app for covid-19," in *International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2022, p. 453–460.

[29] S. Biswas, K. Sharif, F. Li, and S. Mohanty, "Blockchain for e-health-care systems: Easier said than done," *Computer*, vol. 53, no. 7, pp. 57–67, 2020.

[30] U. Bodkhe, S. Tanwar, K. Parekh, P. Khanpara, S. Tyagi, N. Kumar, and M. Alazab, "Blockchain for industry 4.0: A comprehensive review," *IEEE Access*, vol. 8, pp. 79 764–79 800, 2020.

[31] J. Hejderup, A. van Deursen, and G. Gousios, "Software ecosystem call graph for dependency management," in *ACM International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2018, pp. 101–104.

[32] H. K. Ikkink, *Gradle Dependency Management*. Packt Publishing Ltd, 2015.

[33] S. Slimani, T. Hamrouni, and F. B. Charrada, "Scorminer: Automated discovery of network service and application dependencies using a graph mining approach," *Procedia Computer Science*, vol. 176, pp. 985–994, 2020.

[34] N. Gerasimov, "Static typing and dependency management for soa," in *IEEE Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2018, pp. 105–107.

[35] R. Khemaissia, M. Derdour, A. Djeddai, and M. A. Ferrag, "Sdgchain: When service dependency graph meets blockchain to enhance privacy," in *Proceedings of the 2021 ACM Workshop on Security and Privacy Analytics*. Association for Computing Machinery, 2021, p. 37–43.

[36] M. Belguidoum and F. Dagnat, "Dependency management in software component deployment," *Electronic Notes in Theoretical Computer Science*, vol. 182, pp. 17–32, 2007.

[37] E. Bertino, B. C. Ooi, R. Sacks-Davis, K.-L. Tan, J. Zobel, B. Shidlovsky, and D. Andronico, *Indexing techniques for advanced database systems*. Springer Science & Business Media, 2012, vol. 8.

[38] Y. Li, G. Yang, W. Susilo, Y. Yu, M. H. Au, and D. Liu, "Traceable monero: Anonymous cryptocurrency with enhanced accountability," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 2, pp. 679–691, 2019.

[39] H. Huang, X. Chen, and J. Wang, "Blockchain-based multiple groups data sharing with anonymity and traceability," *Science China Information Sciences*, vol. 63, no. 3, pp. 1–13, 2020.

[40] T. Mitani and A. Otsuka, "Traceability in permissioned blockchain," *IEEE Access*, vol. 8, pp. 21 573–21 588, 2020.

[41] M. Möser, K. Soska, E. Heilman, K. Lee, H. Heffan, S. Srivastava, K. Hogan, J. Hennessey, A. Miller, A. Narayanan *et al.*, "An empirical analysis of traceability in the monero blockchain," in *arXiv preprint arXiv:1704.04299*, 2017.

[42] O. Bischoff and S. Seuring, "Opportunities and limitations of public blockchain-based supply chain traceability," in *Modern Supply Chain Research and Applications*. Emerald Publishing Limited, 2021.

[43] F. Kaup, P. Gottschling, and D. Hausheer, "Powerpi: Measuring and modeling the power consumption of the raspberry pi," in *IEEE Conference on Local Computer Networks*, 2014, pp. 236–243.

[44] L. Zou, A. Javed, and G.-M. Muntean, "Smart mobile device power consumption measurement for video streaming in wireless environments: Wifi vs. lte," in *IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, 2017, pp. 1–6.

[45] A. Ampatzoglou, S. Bibi, P. Avgeriou, M. Verbeek, and A. Chatzigeorgiou, "Identifying, categorizing and mitigating threats to validity in software engineering secondary studies," *Information and Software Technology*, vol. 106, pp. 201–230, 2019.

[46] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research-an initial survey." in *Seke*, 2010, pp. 374–379.

[47] D. S. Cruzes and L. ben Othmane, "Threats to validity in empirical software security research," in *Empirical research for software security*. CRC Press, 2017, pp. 275–300.

[48] S. Maro, J.-P. Steghöfer, and M. Staron, "Software traceability in the automotive domain: Challenges and solutions," *Journal of Systems and Software*, vol. 141, pp. 85–110, 2018.

[49] J. Lotz, A. Vogelsang, O. Benderius, and C. Berger, "Microservice architectures for advanced driver assistance systems: A case-study," in *IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2019, pp. 45–52.

[50] H. Vdovic, J. Babic, and V. Podobnik, "Automotive software in connected and autonomous electric vehicles: A review," *IEEE Access*, vol. 7, pp. 166 365–166 379, 2019.

[51] C. Zhao, J. S. Gill, P. Pisu, and G. Comert, "Detection of false data injection attack in connected and automated vehicles via cloud-based sandboxing," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 7, pp. 9078–9088, 2022.

[52] M. Abdellatif, R. Tighilt, N. Moha, H. Mili, G. El Boussaidi, J. Privat, and Y.-G. Guéhéneuc, "A type-sensitive service identification approach for legacy-to-soa migration," in *International Conference on Service-Oriented Computing*. Springer, 2020, pp. 476–491.

DAVID FERNÁNDEZ BLANCO received both a B.S. and a M.Eng. degree in Telecommunications Engineering from the INSA Lyon (National Institute of Applied Sciences of Lyon), France, in 2020. He holds a parallel specialization diploma in Research & Development from INSA Lyon. He is currently pursuing an Industrial Ph.D. degree in Computer Science at INSA Lyon with the collaboration of STELLANTIS.

Prior to the beginning of his Ph.D. he was a Research Engineer at CITI Laboratory for a year and an Associate Lecturer and Professor for CPE Lyon and INSA Lyon. His research interest includes the development of the new generation embedded software architectures and middleware for the automotive sector, the coordination with the cloud (Classic, Fog and Edge) and the development and design of complex distributed systems architectures, such as blockchain storage layers.

FRÉDÉRIC LE MOUËL is a Full Professor in the National Institute for Applied Sciences of Lyon (INSA Lyon, France) - a leading engineering school in France, part of the University of Lyon. He is heading the Center for Innovation in Telecommunication and Integration of Services (INSA/Inria CITI Lab) and leading the Dynamic Software and Distributed Systems for the Internet of Things research group (DynaMid Team). From 2022, Frédéric Le Mouël holds the SPIE/INSA research chair in Edge AI. He is especially animating the research topic on geo-distributing data and ressources for Federated Learning. From 2019 to 2022, Frédéric Le Mouël held the SPIE/INSA research chair in the Internet of Things. He was especially animating the research topic on IoT large-scale deployments. His main interests are distributed systems, operating systems, component and service-oriented middleware, virtual machines, programming languages and more specifically in dynamic adapting, self-coordinating and autonomic environments. He is specially studying these topics in the domain of Ambient Intelligence, Internet of Things, Home Automation and Vehicular Networks. He published more than 100 publications in international journals and conferences and is member of several conference committees.

TRISTA LIN is an onboard IT architect and technical specialist in the E/E architecture design department at Stellantis since 2018. She is responsible for TCP/IP architecture design and protocol deployment. Prior to Stellantis, she has 8 years of industry experience in wireless network simulators and 3 years of research experience in open source software development. She holds a PhD degree in computer science from INSA Lyon (National Institute of Applied Sciences of Lyon), France, and a BS in mathematics and communications engineering from National Tsing Hua University, Taiwan. Her research interests lie in IT solution adaptation for cars towards software-defined architectures and services.

• • •