The work described in this document has been conducted within the Research & Innovation action TAPPS (project no. 645119), started in January 2015, and co-funded by the European Commission under the Information and Communication Technologies (ICT) theme of the H2020 framework programme (H2020-ICT-2014-1).

©Copyright by the TAPPS Consortium.
<table>
<thead>
<tr>
<th>Name</th>
<th>Organisation</th>
<th>Contact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oliver Horst</td>
<td>FORTISS</td>
<td><a href="mailto:oliver.horst@fortiss.org">oliver.horst@fortiss.org</a></td>
</tr>
<tr>
<td>Falak Sher</td>
<td>FORTISS</td>
<td><a href="mailto:falak.vira@fortiss.org">falak.vira@fortiss.org</a></td>
</tr>
<tr>
<td>Christian Prehofer</td>
<td>FORTISS</td>
<td><a href="mailto:prehofer@fortiss.org">prehofer@fortiss.org</a></td>
</tr>
<tr>
<td>Nora Koch</td>
<td>FORTISS</td>
<td><a href="mailto:nora.koch@fortiss.org">nora.koch@fortiss.org</a></td>
</tr>
<tr>
<td>Michele Paolino</td>
<td>VOSYS</td>
<td><a href="mailto:m.paolino@virtualopensystems.com">m.paolino@virtualopensystems.com</a></td>
</tr>
<tr>
<td>Alvise Rigo</td>
<td>VOSYS</td>
<td><a href="mailto:a.rigo@virtualopensystems.com">a.rigo@virtualopensystems.com</a></td>
</tr>
<tr>
<td>Daniel Raho</td>
<td>VOSYS</td>
<td><a href="mailto:s.raho@virtualopensystems.com">s.raho@virtualopensystems.com</a></td>
</tr>
<tr>
<td>Riccardo Dodi</td>
<td>FCSR</td>
<td><a href="mailto:dodi.riccardo@hsr.it">dodi.riccardo@hsr.it</a></td>
</tr>
<tr>
<td>Sauro Vicini</td>
<td>FCSR</td>
<td><a href="mailto:vicini.sauro@hsr.it">vicini.sauro@hsr.it</a></td>
</tr>
<tr>
<td>Jiseon Moon</td>
<td>FCSR</td>
<td><a href="mailto:moon.jiseon@hsr.it">moon.jiseon@hsr.it</a></td>
</tr>
<tr>
<td>Eleonora Montanari</td>
<td>EMC</td>
<td><a href="mailto:emontanari@energicasuperbike.com">emontanari@energicasuperbike.com</a></td>
</tr>
<tr>
<td>Giovanni Gherardi</td>
<td>EMC</td>
<td><a href="mailto:ggherardi@energicasuperbike.com">ggherardi@energicasuperbike.com</a></td>
</tr>
<tr>
<td>Christian Reinisch</td>
<td>TTTech</td>
<td><a href="mailto:christian.reinisch@tttech.com">christian.reinisch@tttech.com</a></td>
</tr>
<tr>
<td>Arjan Geven</td>
<td>TTTech</td>
<td><a href="mailto:arjan.geven@tttech.com">arjan.geven@tttech.com</a></td>
</tr>
<tr>
<td>George Kornaros</td>
<td>TEI</td>
<td><a href="mailto:kornaros@ie.teicrete.gr">kornaros@ie.teicrete.gr</a></td>
</tr>
<tr>
<td>Marcello Coppola</td>
<td>STGNB 2 SA</td>
<td><a href="mailto:marcello.coppola@st.com">marcello.coppola@st.com</a></td>
</tr>
</tbody>
</table>
# Table of Contents

1. **INTRODUCTION** .................................................................................................................. 6  
   1.1. Structure of the document .............................................................................................. 6  
   1.2. Relation to other project work ...................................................................................... 7  

2. **ARCHITECTURE CONCEPT & DESIGN** ............................................................................. 8  
   2.1. TAPPS Architecture Key Features ............................................................................. 10  
      2.1.1. Execution Environments and Apps Platform ......................................................... 11  
      2.1.2. Trusted & Real-time Resource Management ......................................................... 12  
      2.1.3. Trusted Interconnection ....................................................................................... 13  
      2.1.4. Trusted Inter-EE & Inter-App Communication ..................................................... 13  
   2.2. Execution Environments Definition and Differentiation .................................................. 14  
      2.2.1. Rich Execution Environment (TAPPS REE) .......................................................... 14  
      2.2.2. Trusted Execution Environment (TAPPS TEE) ...................................................... 14  
      2.2.3. Critical Execution Environment (TAPPS CEE) ...................................................... 15  

3. **ARCHITECTURE COMPONENTS & METHODS** ................................................................. 16  
   3.1. Hardware Components .................................................................................................. 16  
      3.1.1. Secure CAN Controller ....................................................................................... 16  
      3.1.2. Deterministic Ethernet ......................................................................................... 18  
      3.1.3. Secure Network on Chip (NoC) ............................................................................ 23  
      3.1.4. TrustZone Concept ............................................................................................. 24  
   3.2. Operating Systems ......................................................................................................... 26  
      3.2.1. Android ................................................................................................................ 26  
      3.2.2. Linux Embedded ................................................................................................... 27  
      3.2.3. FreeRTOS ............................................................................................................ 27  
      3.2.4. KVM Hypervisor .................................................................................................. 27  
   3.3. Toolchain and App Store ............................................................................................... 28  
      3.3.1. Trusted Toolchain ............................................................................................... 28  
      3.3.2. Application Container ......................................................................................... 29  
      3.3.3. App Store ............................................................................................................. 29  
   3.4. Application Services ...................................................................................................... 30  
      3.4.1. Inter-Execution-Environment Communication ..................................................... 30  
      3.4.2. Safety Integration Layer ....................................................................................... 34  

4. **ARCHITECTURE INTERFACES** ......................................................................................... 37  
   4.1. App Store Interfaces ...................................................................................................... 37  
   4.2. Critical Apps ⇔ CAN .................................................................................................... 37  
   4.3. Critical Apps ⇔ TT/Ethernet ........................................................................................ 38  
      4.3.1. Deterministic Ethernet Configuration ................................................................. 39  

5. **ARCHITECTURE UTILIZATION IN USE CASES** .............................................................. 42  
   5.1. Energica Superbike ....................................................................................................... 42  
   5.2. Health Trolley .............................................................................................................. 45
List of Figures

Figure 1: High-level view on the TAPPS device software architecture and development toolchain ........................................ 9
Figure 2: Illustration of the TAPPS network architecture ........................................................................................................ 10
Figure 3: CAN packet format (extended packet format is outlined) ........................................................................................ 17
Figure 4: Embedding firewalling in CAN-bus infrastructure ................................................................................................. 18
Figure 5: TTEthernet traffic types and relation to other protocols .......................................................................................... 19
Figure 6: Example schedule as output of the TTE toolchain .................................................................................................. 20
Figure 7: Emergence of TT Ethernet towards Deterministic Ethernet ......................................................................................... 20
Figure 8: Distributed TAPPS architecture ............................................................................................................................. 22
Figure 9: NoC supporting protection domains ....................................................................................................................... 24
Figure 10: ARM TrustZone secure and non-secure worlds ....................................................................................................... 25
Figure 11: The GlobalPlatform TEE architecture .................................................................................................................. 31
Figure 12: GlobalPlatform secure boot (source: TEE System architecture specification 1.0) ................................................. 32
Figure 13: Interaction between TEE and CEE .......................................................................................................................... 33
Figure 14: REE interaction with the CEE ................................................................................................................................. 34
Figure 15: Installation process for trusted and critical apps. .................................................................................................. 35
Figure 16: TT Ethernet network access .................................................................................................................................... 39
Figure 17: TT Ethernet configuration toolchain ...................................................................................................................... 40
Figure 18: Display sharing organization ................................................................................................................................... 43
Figure 19: Display navigation .................................................................................................................................................... 44
Figure 20: Draft of the smart trolley UI .................................................................................................................................... 47

List of Tables

Table 1: Functionality of the TT Ethernet controller .................................................................................................................. 22
Table 2: Per execution environment application examples in the Energica superbike use case ............................................ 42
Table 3: Classification of applications into execution environments in accordance to their requirements .............................................................................................................................................................................. 45
Table 4: Per execution environment application examples in the health trolley use case .......................................................... 47
1. Introduction

The focus of this document is to present the first version of the overall system architecture that is implemented within the TAPPS project. A high-level overview of all components that are integrated into the architecture is given, and a complementing in depth description of each of these components can be found in proceeding sections.

The TAPPS architecture provides several independent layers of security. The main characteristic security features are: (1) Computing and network virtualization based on novel, flexible hardware security mechanisms, while maintaining stringent real-time constraints in Cyber-Physical Systems (CPS) and their internal networks, (2) fine-grained access control to resources of the smart cyber-physical device to ensure safety and privacy, and (3) formally verified applications (apps) to ensure correct and secure behavior.

For the last feature, we plan an end-to-end solution for development and deployment of trusted apps. The implementation will consist of (1) an application store for management and for deployment of CPS apps, supporting different execution environments, and (2) a model-based development toolchain for designing and implementing trusted apps including APIs and verification tools. The toolchain design will follow and extend existing standards.

The TAPPS architecture is the product of the collective input from all partners involved in the project. A number of collaborative discussions have taken place amongst the project partners to ensure that each facet of the project has been assessed thoroughly, and to confirm that each aspect of the project will fulfill its purpose. The critical interfaces between architecture components are also detailed within this document, to ensure that all components will interoperate with each other correctly.

This document also shows how the proposed architecture provides the required functionality for the use cases planned within the scope of this project in the health and e-vehicle domain. Thus, instilling further confidence that this architecture will be sufficient to meet a range of user requirements from different application domains.

1.1. Structure of the document

This deliverable is structured as follows: Section 2 gives an overview of the complete system architecture; Section 3 provides then a more in depth description of each of the components mentioned in the overview; Section 4 describes how the most important components within the architecture will interface with one another; Section 5 presents examples of how the architecture will be integrated into the given real world use cases; and Section 6 concludes the document.

The Annexes A to C include references to other projects, acronyms used in this document and relation to requirements in the Deliverable D2.1.
1.2. Relation to other project work

This architecture described in this document is related to the requirements identified during the first months of project work. The requirements were described in deliverable D2.1 and are referred in this deliverable by their ID numbers. A small number of requirements are not covered by the current architecture. These requirements are separately listed in Annex D. They will be further discussed before a revised version of the architecture is published.
2. Architecture Concept & Design

The TAPPS project aims at open CPS systems, which can be extended during operation by downloading apps. These apps may interact or interfere with safety critical functions, which is not possible today due to security and safety reasons. The main approach in TAPPS is to develop dedicated Execution Environments (EEs) for different kinds of CPS apps.

In order to provide strong isolation or EEs as well as protection or resources and critical APIs, we utilize the following security mechanisms:

- **Protection of computing and network resources** based on novel, flexible security mechanism such as container separation and virtualization, while supporting real-time behavior for CPS devices.

- **Fine-grained access control** to the physical resources of the smart cyber-physical device to ensure safety and privacy.

- **Dedicated, model-based toolchain, which permits verified apps on the model level** to ensure correct and secure behavior.

- **Secure boot and installation process**.

Using the above techniques, we can lift the hardware-based security mechanisms, both for networking and computing, to the apps level. We derive three EEs to cater for specific needs in terms of security and real-time support. This enables the architecture to provide a high security encapsulation, even though real-time requirements may provide challenges to some security mechanisms, e.g. virtualization. In combination with different supported networks, we provide and validate an end-to-end solution for development and deployment of trusted apps, including an app store for these apps.

Thus, TAPPS provides a flexible trusted framework for different domains and scenarios. The goal of the TAPPS project is to validate the above novel security solutions for CPS devices in the health and e-vehicle domain.

TAPPS focuses on the above, key security mechanisms and their combinational usage. We follow the principle of several, independent layers of security. This greatly reduces the overall attack surface, and draws a successful attack highly unlikely (see [27], Deliverable D2.1 [15]: ID[24]). The architecture relies on the security of the used systems and platforms. We do not aim at a complete security and threat analysis for all specific possible usage scenarios, which is beyond the scope of the project.

Figure 1 and Figure 2 illustrate the overall architecture. Figure 1 details the software architecture of a TAPPS device with all its components, as well as the development toolchain and services. Figure 2 focuses on the intended network architecture for the TAPPS devices illustrated in Figure 1. For the description of the components, services and methodologies of the software and network architecture the reader is referred to the remainder of this section and the next two sections of the deliverable.
Figure 1: High-level view on the TAPPS device software architecture and development toolchain
2.1. **TAPPS Architecture Key Features**

The architecture we propose within the scope of the project addresses all necessary layers from hardware over software to an app store ensuring security and full real-time support for the applications. An overview of the software architecture is presented in Figure 1; the network architecture is given in Figure 2. These figures show the three execution environments and the main security features of the connected TAPPS architecture. The security features are explained below; the EEs are introduced in the next section.

For ensuring safe execution of CPS apps, we focus on four key features. The following list gives an overview of these main features of the TAPPS architecture, plus the specific means to reach this objective (shown as sub-items).

1) **Execution Environments and Apps Platform**
   1) Trusted Development / Model-based Toolchain
   2) Trusted Boot
   3) Trusted Installation
   4) Virtualization

2) **Trusted and Real-time Resource Management**
   1) Safety Integration Layer
      Resource and Timing Management
   2) Hardware mapping:
i. Network on Chip (NoC)
ii. ARM TrustZone

3) Trusted Interconnection
   1) Control via Safety Integration Layer (SIL)
   2) Secure CAN
   3) Deterministic Ethernet

4) Trusted Inter-EE & Inter-App Communication
   1) Global Platform’s TEE Concept
   2) Secure and Configurable Network on Chip (NoC)

The first pillar of the TAPPS architecture is the basic concept of trusted execution platforms. While other virtual execution platforms (e.g. Java) provide full separation of apps, we propose a multi-faceted approach consisting of isolated execution environments, a trusted toolchain, and a trusted install and boot process. This optimally exploits the given hardware capabilities like the virtualization and ARM TrustZone features, and provides a secure end-to-end solution from the development until the usage of an app. The concept ensures that apps are intensively checked and verified by the toolchain, e.g., with the help of a model checker or a trusted third party, before they are submitted to the app store. Therefore, runtime checks on those apps can be reduced to a minimum, if required to meet real-time requirements for example.

The second pillar consists of the trusted and real-time resource management. This resource management performs the runtime checks and provides a provisioning of the system with respect to resource utilization and timing constraints of individual apps. The Safety Integration Layer (SIL) is in charge of enforcing the timing and resource requirements of the different apps and guaranteeing an overall sane system schedule. This is achieved through configuring the hardware partitions and resource assignments within the Network on Chip (NoC) and the ARM TrustZone worlds.

To integrate TAPPS-based systems in larger cyber-physical systems, the TAPPS architecture considers two different trusted interconnections as the third pillar of the architecture. A secure CAN bus derive, the sCAN, and a deterministic Ethernet, the TTEthernet. Both aim to integrate the communication of multiple apps of different security levels on a single bus, while still maintaining the requested real-time behavior. As above, the SIL is in charge of the resource control.

Finally, a reliable and secure communication between apps and EEs is established that ensures a trusted interaction between apps of different security levels.

### 2.1.1. Execution Environments and Apps Platform

The basic concept of trusted apps relies on the combination of two trust chains. Firstly, a trusted boot mechanism exploits the security on hardware level and ensures correct booting and installation of the layers of the TAPPS architecture, including the hypervisor, execution environments (EEs) and the app installer. Secondly, the installation of apps is done via a trusted app store, based on a trusted toolchain, which can verify and sign the trusted apps – if they comply with the required properties. Furthermore, virtualization and the ARM TrustZone technology is used
to separate apps from the system layers below. To provide specific runtime environments for the different apps and their requirements, the TAPPS architecture utilizes three different EEs (see Figure 1). A definition and discussion of these EEs is given in Section 2.2.

1) Trusted Development / Model-based Toolchain
   Trusted applications are developed in a model-based toolchain that comprises the modeling of the applications using a state-machines-based approach, automatic code generation and a verification process based on model checking principles. For further details, see Section 3.3.1.

2) Trusted Boot
   One root of the TAPPS trust chains is the boot procedure. In fact, when the machine boots, the security configuration of the system is not yet in place and as a result, the system is vulnerable to attacks. For this reason, TAPPS bootloader aims to leverage security best practices and standardization efforts (e.g., ARM Secure Boot [22]).

3) Trusted Installation
   The second TAPPS chain of trust originates in the TAPPS app store, which attests a) the integrity of the used toolchain on the developer site, b) the deployment from the developer to the app store, as well as c) the deployment from the app store to the TAPPS device. On the device itself, a trusted installation process, described in Section 3.4.1, guarantees in conjunction with the SIL the integrity of the overall system. The trusted installation, therefore, forms the junction point of the two TAPPS trust chains.

2.1.2. Trusted & Real-time Resource Management

In order to ensure the proper cooperation of different apps and EEs with the resources on the platform, we use a safety integration layer (SIL) (see Figure 1). This layer takes care of the memory management, CPU scheduling priorities and deadlines, as needed for real-time properties and separation of apps and EEs. Thus, it maps these resources to the operating systems and hardware capabilities, as well as the hardware partitions of the NoC.

1) Safety Integration Layer
   The central controller of the resource and timing management in each TAPPS device is the Safety Integration Layer (SIL). It is responsible for guaranteeing the sanity of the overall system schedule with respect to the requested properties of each application. Important from this point of view are in particular real-time properties and resource usage constraints. Section 3.4.2 elaborates on the responsibilities and internals of the SIL.

2) Hardware Mapping
   To increase the security of the TAPPS architecture, resource constraints and access permissions are mapped to the hardware whenever possible. This mapping is especially supported by the configurable network on chip (NoC), as described in Section 3.1.3, and its secure partitions. Also standard hardware features, like the ARM TrustZone concept and memory management are exploited.
2.1.3. **Trusted Interconnection**

TAPPS devices are designed as a part of a larger cyber physical system (CPS) with several nodes that are connected through multiple networks, as shown in Figure 2. To guarantee a secure interconnection within a distributed CPS, we consider two secure networks within TAPPS: a secure CAN, the sCAN, and a deterministic Ethernet, the TT/Ethernet. Both networks provide a trusted interconnect that enables apps from different security levels and with different timing requirements to communicate safely over a single network. Both trusted peripherals are configured and provisioned by the SIL.

1) Control via Safety Integration Layer
   As mentioned before, the SIL is responsible for controlling and managing the access to the peripherals attached to the TAPPS device. This also includes the trusted interconnects. Here, the responsibility of the SIL is especially the individual configuration and assignment of virtual channels per app or execution environment.

2) Secure CAN
   The first trusted interconnection, considered in TAPPS, is a secure derivate of the CAN bus, the sCAN. sCAN isolates and secures the communication of different apps on the network with the help of firewalls. Two types of firewalls are evaluated, those contained within nodes and those contained in the network itself. Details on the sCAN approach are given in Section 3.1.1.

3) Deterministic Ethernet
   A second trusted interconnect for TAPPS devices is provided by TT/Ethernet, a deterministic Ethernet. TT/Ethernet enables apps to reserve virtual links on the common network with well defined properties that are enforced by the network itself. For this purpose, TT/Ethernet features a strong temporal isolation and a distributed clock synchronization in hardware. The further discussion of TT/Ethernet takes place in Section 3.1.2.

2.1.4. **Trusted Inter-EE & Inter-App Communication**

The trusted communication between the different execution environments, as well as in-between critical apps are the main communication aspects to be considered in TAPPS. The first one is enabled by an extended version of Global Platform’s TEE API ([20],[21]): the second one targets a data centric communication approach with plug and play capabilities to allow for a variety of trusted services for other apps in the same or different EEs.

1) Global Platform’s TEE Concept
   The TEE API by Global Platform ([20],[21]) provides a concept for a secure delegation of functions between a trusted and an untrusted execution environment. This concept is adapted and extended to fit the TAPPS architecture with its three execution environments. A discussion of the general work by Global Platform is provided in Section 3.4.1.

2) Secure and Configurable Network on Chip (NoC)
   The trusted inter-EE communication is supported in hardware by the secure and configurable network on chip (NoC). The NoC allows a fine granularly partitioning of the available hardware resources and puts an effective access control into place. Again, a firewalling concept is
evaluated to harden the NoC even further. Details on that concept are presented in Section 3.1.3.

2.2. **Execution Environments Definition and Differentiation**

The TAPPS project considers that for the security features and layers described above for trusted applications, different execution environments are needed (see Deliverable D2.1 [15]: ID[49], ID[50] and ID[51]).

We distinguish three types of applications: untrusted (U-Apps), trusted (T-Apps) and critical (C-Apps) (Deliverable D2.1: Section 2.1) and propose therefore an architecture with **three different execution environments**: Rich Execution Environment (REE), Trusted Execution Environment (TEE) and Critical Execution Environment (CEE), that are briefly described in the following (see Figure 1).

**2.2.1. Rich Execution Environment (TAPPS REE)**

The TAPPS Rich Execution Environment (or TAPPS REE) is security wise, the least critical environment of the projects’ architecture. It is hosted by a feature rich operating system, e.g., Android or Linux, and is isolated by the KVM hypervisor only.

The rich execution environment shall provide an environment that is suitable for hosting untrusted apps (U-Apps) like those known from current smartphone eco systems. Applications of this class are, e.g., games, fun apps, or social apps. The rich execution environment neither isolates the different U-Apps from each other, nor protects them from any other interference. Hence, the rich execution environment provides the same capabilities as standard smartphone operating systems do today. However, U-Apps do not have access to all types of peripherals, but only to untrusted peripherals like for example WiFi and Bluetooth adapters. An additional read-only access to peripherals can be granted by device mirroring through one of the more privileged EEs, as explained in Section 3.4.1.

**2.2.2. Trusted Execution Environment (TAPPS TEE)**

The TAPPS Trusted Execution Environment (TAPPS TEE) is the architecture compartment which runs user applications that interact with the Critical Execution Environment (TAPPS CEE, see Section 2.2.3).

One or more TEE can be run concurrently in the TAPPS platform, by means of the KVM hypervisor. Each trusted app (T-App) is enclosed into its own hypervisor partition, running on top of its own virtual machine (VM). T-Apps are the only entities in the system which are allowed to collaborate with critical applications (C-Apps) running in the Critical Real-time Execution Environment. This interaction is described in Section 3.4.1. The TEE is hosted by a hardened, but still feature rich operating system, like e.g., Automotive Grade Linux [35].

Ideally, T-Apps are developed with the model-driven toolchain of the TAPPS project; however, also other development approaches will be supported. Another option is to let a trusted third party review and sign the T-Apps to ensure their trustworthiness.
2.2.3. **Critical Execution Environment (TAPPS CEE)**

The Critical Real-Time execution environment (or TAPPS CEE) is the third, and security/safety wise the most important, execution environment of the TAPPS architecture. The TAPPS CEE is designed to run critical soft real-time applications (C-Apps) and is hosted in the Secure World of the ARM TrustZone architecture (cf. Section 3.1.4). The operating system that controls this execution environment shall be small, real time, lean and safe like, e.g., FreeRTOS [36]. To allow the execution of several individual C-Apps without interference and to coordinate accesses to the peripherals from application of different trust levels / criticality, in the TAPPS CEE environment, the Safety Integration Layer (SIL) is put in place. This component coordinates the different access from the TEE Server, as well as the TEE Internal API to the resources of the operating system and the processor (peripherals). Furthermore, the SIL is responsible for a reliable and predictable communication among C-Apps. All applications developed for the critical real-time execution environment shall be developed with the model-driven toolchain, which adds an additional safety layer and the possibility of an objective validation of certain properties with model checking tools.
3. Architecture Components & Methods

The TAPPS architecture relies on a set of hardware and software components, which are used to implement a secure framework for running trusted applications. The following sections briefly describe (1) the networking components like sCAN and TTEthernet, (2) the relevant operating systems, (3) the toolchain and the application container, and (4) the application services.

In the following sections, we detail the novel hardware components, as needed for secure networking inside and outside the TAPPS device. This is followed by the operating systems, toolchain and services.

The focus is not only on the description of the components, but also on the processes supported by them for developing, storing, installing and running apps, as well as on the methods and techniques used for producing and running trusted apps. Furthermore, model checking methods are used for achieving trustworthiness in the production; and virtualization, isolation and controlled communication methods for ensuring trustworthiness at run-time. Regarding data security methods like encryption are used, and authorization and authentication among others are implemented for providing appropriate access control.

3.1. Hardware Components

In order to be sure that cyber physical systems (CPSs) are secure it is important to build in distributed hardware security mechanisms within the individual CPS nodes. In the TAPPS project, in compliance with the requirements described in Deliverable D2.1 [15] (ID[52], and ID[53]), we utilize a processor from the latest ARM Cortex A series, namely the A53 or the A57. These processors come with support for the ARM TrustZone feature (cf. Section 3.1.4), to fulfill requirement ID[52], and are extended on-chip and off-chip hardware security extensions as described in the Section 3.1.1 till Section 3.1.3 to meet the requirements ID[54-67].

3.1.1. Secure CAN Controller

In the direction of TAPPS objectives the TEE acts as security controller, which is responsible for validating the trust level of a virtual appliance and for controlling the communication between virtual appliances and other target devices. In order to support controlled transfer of information among trusted CPS devices we introduce innovative architectural features in the communication infrastructure. Following the set of requirements (ID[55-74], and ID[15], ID[35]) presented in deliverable D2.1 [15], and in order to employ trusted communication via for instance securing of CAN bus communication, encryption of transmission can be utilized.

Due to the particular constraints of automotive bus communication systems (computing power, capacity, and timing), a combination of symmetric and asymmetric encryption can meet the requirements on adequate security and high performance. New trusted CPS devices added in a system can adopt hardware and software enhancements introduced herein, while systems with legacy devices that do not provide open access to its internals (hardware and software components) can be addressed through additional external hardware firewalling component that is transparent to these legacy devices. In several domains, so-called ECU (Electronic Control Unit) are used as specific forms of CPS devices, as used below.
Both, either packet filtering and/or fast and efficient symmetric encryption can secure the bus-internal broadcast communication. In the latter case, all controllers of a local bus system share the same, periodically updated, symmetric key to encrypt their bus-internal communication. Asymmetric encryption can be used to handle the necessary secure key distribution. It provides the acquisition of the symmetric key for newly added authorized controllers and carries out the periodic symmetric key update, as well as the required authentication process. One option to handle complexity is to employ hardware accelerator engines. To limit the latency impact on the real-time communication, symmetric cryptography is proposed in the literature [23, 24], since it is computationally simpler.

Symmetric encryption requires that all participants of a protected communication know the secret keys a priori. This opens a new attack vector, as keys are often pre-programmed into ECUs and valid for the lifetime of the ECU. On top, in TAPPS architecture we intend to advance beyond using a hardware module dedicated to cryptographic operations, such as the Hardware Security Module (HSM) [25] from the EVITA project (to free the ECUs computational capacities), to packet authentication on the basis of using trusted CANbus Identifiers and firewalling methods.

A CAN packet (shown in Figure 3) does not include addresses in the traditional sense and instead supports a publish-and-subscribe communication model. The CAN ID header is used to indicate the packet type, and each packet is both physically and logically broadcast to all nodes, which then decide for themselves whether to process the packets.

![Figure 3: CAN packet format (extended packet format is outlined)](image)

CAN packets contain no authenticator fields, or even any source identifier fields, meaning that any component can indistinguishably send a packet to any other component. This means that any single compromised component can be used to control all of the other components on that bus, provided those components themselves do not implement defenses. CAN 2.0a supports 11 bit IDs, while CAN 2.0b supports 29 bit IDs. In the communication process it is necessary to ensure that the data, transactions, communications are genuine and additionally to validate that both parties involved are who they claim they are.

The objective of ensuring trusted CAN communication is to provide guarantees to Electronic Control Units (ECUs) that critical information transmitted to them is based on assurances about integrity and authenticity in terms of origin, content, and time. In the TAPPS scope, the method can consist of: (1) ECU authentication and (2) stream authorization. In the first step, each ECU authenticates against a central security module. This is performed e.g. when the vehicle or the trolley is not operating and real-time behavior is of limited importance.
In the second step, during operation, every message stream is authorized and each CANbus ID between any two nodes that communicate is changing in time on the basis of their private algorithm.

The main principle is to program a CAN interface with a number of acceptance codes. These codes tell the interface to listen to messages with a CANbus ID matching any of the acceptance codes. It is also possible to specify acceptance masks, allowing a single acceptance code to match multiple IDs. By linking keys to these (acceptance code, acceptance mask) sets, a group key setup can be achieved. A group of related messages $G_i$ can be defined as the collection of all messages with IDs matching the pair (acceptance code, acceptance mask). We reserve an amount of bits to change using a hashing function in a programmable way, so that senders and receivers are tied and synchronized together.

In addition, CAN Firewalling and bridging functions will be employed with dynamic re-programming of the rules. As shown in Figure 4, using this architecture in TAPPS indicates that any communication on both directions (gateway-to-ECUs and ECUs-to-gateway) is filtered by the rules that are configured by the Safety integration layer.

Both on-chip (at NoC level) and off-chip (CANbus communication) firewalling mechanisms need to support authorization control when filter rules are configured. Harmonized policies can provide a coherent view of CPSs filter rules across the platform.

### 3.1.2. Deterministic Ethernet

Since TAPPS targets distributed, safety-critical CPS applications, the communication between these distributed parts of applications is of crucial importance. Deterministic Ethernet provides the access to trusted network components for applications. This access through deterministic Ethernet allows for an additional level of distributed applications to operate on the same trusted environment as local application and thus achieve the same level of trust, whilst guaranteeing real-time communication properties between the CPS nodes (Requirements ID[81], ID[82], ID[88], ID[89] of Deliverable D2.1 [15]).

Deterministic Ethernet properties help to facilitating core properties of distributed CPS environments: Safe access to TAPPS peripherals and computing resources connected through Ethernet), partitioning of network resources, and finally support for distributed TAPPS applications.
TTEthernet, Deterministic Ethernet and Time-Sensitive Networking

TTEthernet (SAE AS6802) [1] is a scalable, open real-time Ethernet platform used for safety-related applications primarily in transportation industries and industrial automation. TTEthernet extends classic Ethernet functionalities to provide more flexibility, modularity and scalability in Ethernet-based systems. It is compatible to IEEE 802.3 Ethernet and integrates transparently with Ethernet network components.

TTEthernet based networks enable the seamless communication of all kinds of applications via Ethernet. Conventional PCs, web and office devices, multimedia systems, real-time systems and safety-critical systems are to use the same network. One single network that is completely compatible with the IEEE Ethernet 802.3 standards is suited for data transmission among different applications with various requirements, e.g. satisfying different criticality requirements and fail-safe or even fail-operational behavior. Figure 5 gives an overview of the different communication types of TTEthernet.

Time-triggered (TT) traffic has two important pre-requisites: the need for a global notion of time in the network and the availability of schedules that organize the communication in the time domain, i.e. providing time partitioning on the network. For these reasons, switches in TTEthernet take over the central role of organizing the data communication. TT messages are routed in the switch according to a predefined schedule with as little delay as possible. Precise planning at the time of system design precludes resource conflicts at runtime. TT messages have the highest priority level. If the planned transmission time of one of these messages arrives, this message is immediately transmitted. Due to the predefined transmission of the message, the switch ensures that the medium is free at the time of transmission and delays are precluded. A communication plan for a given (communication) problem statement is termed a schedule. An example schedule is shown in Figure 6.
Over the last years TTTech’s TTEthernet technology has been updated to implement not only TTEthernet specific extensions, but also to implement the Audio Video Bridging (AVB) and Time-Sensitive Networking (TSN) features. AVB and TSN both add quality of service (QoS) layers to standard Ethernet, with a focus on audio and video data and real-time control data, respectively. TSN is the successor of AVB, and the IEEE 802.1 working group on AVB has continued its activities under the name new TSN name. The umbrella name for these technologies is “Deterministic Ethernet”; as the different protocols and their QoS guarantees can provide upper bounds on the network latency and jitter and are as such deterministic in their behavior when certain conditions are met.

Deterministic Ethernet (or DE) is an evolution step of TTEthernet, which shall be considered in the TAPPS project, and solutions will be provided based on this technology.

Time-Sensitive Networking (TSN) activity focuses on the standardization of Deterministic Ethernet in an open IEEE standard that provides synchronization and time-based shaper properties in order to merge advantages of AVB and TTEthernet networks in an integrated and open standard. While the AVB standardization has been completed and AVB products are available on the market, the IEEE standardization group has proceeded towards improving the IEEE 802.1 standards with even more real-time characteristics. Furthermore, driven by the industrial and automotive industries, also robustness and fault-tolerance properties are now being standardized within the IEEE. To emphasize the completion of the AVB technology and as a clear separation from the ongoing work, the IEEE 802.1 AVB task group has renamed itself to IEEE 802.1 Time-Sensitive Networking (TSN) task group.
Deterministic Ethernet in TAPPS

In the TAPPS project, we classify the different flavors of real-time and/or mixed-criticality Ethernet under the general heading “Deterministic Ethernet”. In this document, we differentiate between the different terms in the following way:

- **Deterministic Ethernet**: backbone technology that handles many different kinds of traffic classes, with the highest classes providing fully deterministic behavior.
- **TTEthernet**: Specific technology for providing a subset of the complete deterministic Ethernet functionality.

For this reason, the backbone technology, i.e. the switching fabric that connects the different nodes, performing traffic policing, different types of time synchronization, is referred to as Deterministic Ethernet backbone, whereas the end-points or nodes are typically sub-classes, such as TTEthernet, TSN or AFDX nodes.

Within the project, the goal is to implement the network for use in the TAPPS context, facilitating the following core properties:

- **Safe access to peripherals** – Utilizing existing safety properties of the Deterministic Ethernet concept, the access to peripherals by TAPPS applications can be completely controlled with respect to the real-time behavior of the system. This applies to e.g. smart sensors and actuators connected through Ethernet, computing resources and other I/Os.
- **Partitioning of network resources** – partition the network to allow complete segregation of trusted and untrusted traffic safely in the Deterministic Ethernet backbone.
- **Support for trusted distributed applications** - Since most CPS systems do not operate in isolation, a distributed approach is a necessity for the deployment of the TAPPS concepts in real-time scenarios.

The overall architecture for distributed TAPPS devices (nodes) is displayed Figure 8.

The core of the distributed architecture resides in the Critical Execution Environment (CEE) due to the typically (safety-)critical nature of the real-time communication, which can only be handled in real-time by the CEE in the real-time OS. This implies also that the driver handling access to the TTEthernet controller resides in this environment.

In this architecture, the TTEthernet controller provides different types of ports to the RTOS, which map to the different virtual links for time-triggered and rate-constrained traffic that may exist in the network. Furthermore, a third type of ports is provided for best-effort traffic. In order to provide more flexibility to the higher layers, these can be configured to be accessible directly (i.e. the OS provides complete Ethernet frames, or the OS provides the payload and destination separately and the controller maps this to complete Ethernet frames).
Based on the configuration of the controller and the network, the outgoing TT and RC frames are sent according to their respective schedules and BAGs, whereas best-effort frames are sent whenever there is no other frames of higher priority queued. The TTEthernet controller handles incoming frames serially. The switches in the network perform the shaping and policing of the incoming traffic; it can be safely assumed that this traffic is shaped correctly. The controller performs integrity and timing correctness verification on incoming frames but no explicit prioritization. Incoming frames are forwarded to the RTOS by their respective virtual links and can be retrieved through the same concept of ports.

In order to provide complete freedom from interference, the internal memory of the TTEthernet controller is to be partitioned into separate partitions for each of the environments in the TAPPS node that requires access to the TTEthernet network.

The following table summarizes the functionality that is handled by the TTEthernet controller in the TAPPS node as an interface between the RTOS and the Deterministic Ethernet backbone and cover the requirements on distributed CPS applications:

<table>
<thead>
<tr>
<th>Category</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sending of messages (TX)</td>
<td>Translate the data that is to be sent from the different partitions to respective Virtual Links in the network. Respect the communication schedule for the sending of time-triggered (TT) messages and send rate-constrained (RC) and best-effort (BE) messages whilst respecting</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>bandwidth constraints of the outgoing link.</td>
</tr>
<tr>
<td></td>
<td>Synchronize the sending of the same message in parallel in up to three parallel physical ports (when redundant physical paths are available)</td>
</tr>
<tr>
<td>Receiving of messages (RX)</td>
<td>Safely and securely map received messages to the application partitions.</td>
</tr>
<tr>
<td></td>
<td>Verify received messages with regards to integrity (both content and timing)</td>
</tr>
<tr>
<td>Time synchronization</td>
<td>As synchronization master, together with other nodes, responsible for the decision on the distributed time-base between the individual nodes</td>
</tr>
<tr>
<td></td>
<td>Synchronisation of the local time to the global time provided by the network</td>
</tr>
<tr>
<td></td>
<td>Ensure the synchronisation in the start-up phase as well as the re-integration in a recovery-phase.</td>
</tr>
<tr>
<td>Diagnosis and Control</td>
<td>Provide status information about the underlying TT Ethernet network.</td>
</tr>
<tr>
<td>information</td>
<td></td>
</tr>
</tbody>
</table>

The communication between the critical and the non-critical elements of the architecture is handled in line with the non-distributed TAPPS architecture, and is thus handled by Inter-Execution-Environment communication provided by the Safety Integration Layer described in Section 3.4.1.

### 3.1.3. Secure Network on Chip (NoC)

In the scope of TAPPS we address the secure programming of the on-chip communication infrastructure developing a new technology WORMNoC on top of the current STNoC solution provided by ST that enable (write once, read many) on the configuration registers provided by the NoC. This technology allows information to be written a single time, preventing the user from accidentally or intentionally altering or erasing the configurations. In addition, we are going to support the three execution environments defined in Section 2.2 via isolation domains that provide differentiated access attributes to the different cores internal and external to the gateway and to the applications running on top.

The dynamic NoC Firewall (FW) protection framework is a combination of hardware and system software that provides efficient coarse-grained memory protection across multiple protection domains; these protection domains may involve logical memory compartments or memory-mapped devices (i.e., CAN, Ethernet, or WIFI interfaces). The novelty that TAPPS is going to combine security and safety in the on-chip communication. CPS data security implies a secure on-chip communication with a different element in the CPS device. In order to realize this novel CPS protection domains will be put in place and they can be configured only once and read by the Safety Integration Layer several times. This enables the possibility from the software via the Safety Integration Layer to track down all communication and security errors. Tracking down these errors CPS devices can discover real-time defects. In this context, the gateway considers efficient
isolation of source code and data among trusted and untrusted applications/processes in the rich execution environment.

The WORMNoC (write once, read many) technology will be developed in order to ensure the authenticity of critical data and maintain the chain of custody in on-chip network in including the firewall authorized configuration of rules. The WORMNoC allows information to be written a single time, preventing the user from accidentally or intentionally altering or erasing the configurations. Thus, rules cannot dynamically be updated in different time intervals and in different NoC Firewall instances, the accesses in the system should be handled by the same set of rules.

3.1.4. TrustZone Concept

ARM TrustZone aims to provide a software security framework that enables embedded systems protection. Instead of providing a fixed security solution, such as what it is proposed by TPM (Trusted Platform Module) chips, TrustZone provides a flexible infrastructure that can be customized according to the needs of the particular system. In order to achieve flexibility, it offers a programmable environment that guarantees confidentiality and integrity against specific attacks. In fact, in a TrustZone system the applications’ execution environment is split in Secure and Normal (or Non-Secure) Worlds, which are isolated by hardware. The former represents the place in which security assets and critical programs are protected and confined, while the Normal world is the user’s system, where common operating systems such as Linux/Android are installed and programs/mobile apps are run.
ARM TrustZone is a system-wide approach to security, tightly integrated into Cortex-A processors and extended to the system’s peripherals by the secure NoC via the compartments above described. In fact, the NoC is aware of the security state of the application currently running in the processor, and ensures that Secure World resources cannot be accessed from the Normal World. This approach allows to secure peripherals such as crypto blocks, keyboards, screens, CAN and Ethernet interfaces, in order to protect them from software attacks.

Moreover, with TrustZone, a multiprocessor is able to execute code from both the Normal and the Secure worlds, thus eliminating the need for a dedicated security core. However as required by TAPPS the secure core has to disable all L1 and L2 cache to support the realtime requirements. A multicore processor is split in two virtual processors, one corresponding to the Normal domain and another for the Secure domain. The context switch between these domains is performed via the monitor mode of the processor, resulting in a change of the running virtual processor. As a matter of fact, a security breach in this procedure could compromise the safety and the security of the Critical execution environment. On the other hand, it is also of pivotal importance for performance, as the context switch operations are adding overhead to the overall performance of the system. For these reasons, the mechanisms to enter into the monitor mode from the Normal world are tightly controlled and include the Secure Monitor Call (SMC) instruction and a subset of the hardware exception mechanisms. In this direction, the open source community currently is working on the Arm Trusted Framework [28] and OPTEE [14] respectively an open implementation of TrustZone Secure Monitor and an implementation of the GlobalPlatform API. However, the TAPPS approach goes beyond OPTEE, because it requires the execution of a full, real-time operating system in the Secure World.

To prevent the proliferation of custom solutions and to foster the growth of a secure application developer’s community, ARM designed TrustZone to be compliant with the GlobalPlatform Trusted Execution Environment (TEE) specifications, which are discussed in Section 3.4.1.
3.2. Operating Systems

The goal of the TAPPS project is to support multiple operating systems, depending on the specific EE and also on the application domain. We consider three different operating systems (OSs) in TAPPS, as representatives of their classes. Android represents a classical feature rich OS, Linux a standard embedded OS for systems with no strong real-time requirements, and FreeRTOS as a representative of the real-time OS class.

These OSs just form a selection/instance of possible OSs that would be usable with the TAPPS approach. The implementation priorities will depend on the targeted use cases and needs of the application domains. For instance, multi-apps and multi-tasking are required as in requirement ID[33], as well as different toolchain support of these operating systems ID[104]. Thus, within TAPPS we will focus on the following three operating systems. The same accounts for the hypervisor selection. KVM was chosen due to practical reasons based on the experience and the fact that KVM is open source.

3.2.1. Android

Depending on the use case open source Android can be utilized for the Rich-, as well as Trusted-Execution-Environment. Being originally developed for mobile devices, this OS is especially suitable for the trolley use case, which involves a commercial solution as dashboard, i.e. an HD touch monitor. Among the benefits of such an OS, its high level of customization may allow manufacturer and (trusted) developers to adjust system files according to specific functions.

For instance, different levels of interaction between apps and the OS may be implemented such as:

- Memory access and management – Untrusted and Trusted apps should have different access modes to memory slots, belonging to two different virtual machines running respectively in REE and TEE;
- Battery management – even if more than one battery pack may be present within the same use case, high-consumption apps should not interfere with the main functionality of the system, i.e. should not take away energy from more essential apps;
- GPU and UI management;
- Interactions between different apps (both belonging to the same or different EE);
- Log files management and system backup;
- Kernel customization, according to specific functions.

The version of the OS should be selected among the 5.1 Lollipop or the latest 6.0 Marshmallow, which provides – for instance – a stronger permission system, a native support for fingerprint recognition and a specific power management for idle execution. Currently we plan to utilize Android only in the second half of the project as REE and as proof-of-concept trial for the TEE. This also matches requirement ID[39] from Deliverable D2.1 [15]. One possible programming model for this scenario is the development of cross-platform HTML5 applications that can be deployed to different operating systems with the help of frameworks like Cordova [37].
3.2.2. Linux Embedded

Linux Embedded is another choice for the TAPPS TEE and REE. Conversely, from Android, specific Linux derivates are available which provide real-time capabilities and thus Linux is more suitable for demanding applications like the sportbike use case. The Linux kernel is very extensible and Silicon manufacturers are recognizing Linux as the one OS that can handle both the core elements of infotainment platforms, such as graphics and multicore support, but also new features needed for connected cars: anti-collision technology, voice-activated commands, gesture recognition, and the future requirements of in-vehicle telematics.

Linux is also compatible with Energica requirement ID[14]: boot time faster than 3 seconds (Deliverable D2.1 [15]). Code designed to run on a Linux OS is written in C language, thus compatible with MISRA C ISO26262 requirements (even if Linux itself is not directly MISRA C compliant). Possible Linux distributions to be investigated for their suitability are the Genivi Open Source Platform [38], and the Automotive Grade Linux [35].

As development framework in the superbike use case, we currently consider Qt [31] for HMI design and applications developed for the TEE. Qt has a very wide community of developer (especially developers skilled for HMI design), and allow an easy, consistent and multiplatform HMI design. Communication between UI and code is accomplished via the simple and powerful Signals and Slots mechanism. Code thus is well separated from UI that becomes easily interchangeable and styleable.

3.2.3. FreeRTOS

The TAPPS architecture executes in the CEE a real time operating system, in order to meet real-time constraints specific to CPS applications, as requested in requirement ID[75] in D2.1 [15].

FreeRTOS is an open source Real Time Operating System (RTOS), which is released with a modified version of the GPL license that permits users’ proprietary code to remain closed source while maintaining the kernel itself as open source. It supports more than 35 embedded systems processors, including ARM Cortex M, Cortex A Cortex R CPUs, etc. and has a very active community of developers. As a matter of fact, this OS combines advanced features (tick-less mode for low power application, configurable scheduler, multithread support, etc.) with the availability of free extensions (e.g., device drivers and cryptographic libraries). FreeRTOS, being designed as a microkernel, provides a small overhead and has a small footprint (in the range of 7KB). Additionally, also a safety certified version of it is available, SafeRTOS, which fulfills the regulatory requirements of IEC 61508, EN62304 and FDA 510(k). Thus, FreeRTOS fulfills all characteristics required by the requirements ID[85] and ID[87] of D2.1 [15].

3.2.4. KVM Hypervisor

The Linux Kernel Virtual Machine (KVM) [16] is a Linux kernel module, which turns the Linux kernel into a hypervisor, exploiting the ARM Virtualization Extension to create a fully-featured virtualization environment which provides hardware isolation for CPU, memory, interrupts and timers. The ARM Virtualization Extensions, in particular, allow certain instructions to trap into the hypervisor, include functionality to assist with the guests’ memory virtualization and introduce a new processor mode (i.e., the hypervisor mode) which allows each guest to have access to its own privileged process execution mode. KVM works by exposing a simple ioctl interface, through which a regular Linux process can request to be turned into a virtual machine. When using QEMU (quick Emulator [17]) /KVM, the QEMU emulator is this process, and leverages on KVM for the
Architecture and Methods for 1st Iteration

virtualization of the processor/memory and provides an extensive set of software implementations used to emulate I/O guest devices (audio, video, Ethernet, etc.).

KVM is open source and perfectly integrated into the Linux kernel. As a matter of fact, it takes advantage of the existing Linux kernel infrastructure, including the scheduler and memory management. This results in a tiny code base, if compared to other hypervisors such as XEN or VMWare ESX. The TAPPS architecture exploits KVM to isolate virtual machines with each other, and to protect the non-secure operating system from direct attacks potentially coming from the U-AppS and T-AppS.

3.3. Toolchain and App Store

In the following, we introduce the toolchain for developing apps as well as the app store to deploy apps securely and conveniently on the TAPPS devices.

3.3.1. Trusted Toolchain

In the TAPPS approach, the toolchain for developing apps is an integral part of the security concept. The main idea is that only apps developed in a specific toolchain may be permitted for the critical execution environment. This ensures by construction that apps are limited to the capabilities supported by the toolchain, hence cannot perform arbitrary access to interfaces. Furthermore, apps are easier to analyze and to verify on a model-level. For the developer, it is important that toolchains are easy to use, intuitive and well integrated. For a discussion on toolchains we refer to [31].

Trusted applications (T-Apps) are developed using a state-machine based framework. For example, using the modeling framework 4DIAC [29], applications are modeled as asynchronously interacting state-machines, and their code is generated in the C language. To ensure safe operation of applications, it is an extra layer of security and safety to verify these applications [31]. The 4DIAC tool implements the IEC 61499 standard [31] used to model applications for industrial automation. We do not aim here to utilize the complete standard, but to use the tool as a basis to implement specific parts of the standard. In addition, extensions for interfaces and platform specifics can be added to address access control, addressing requirements ID [90-98] in Deliverable D2.1. Overall, the model-based toolchain addresses the requirements ID[105], ID[106], ID[108], ID[109], [110] and ID[111].

Model checking [33] is an automated technique that, given a finite model of an application and a formal property, systematically checks whether the property holds for the model. For instance, for an application that controls the daily dosages of medicines for patients, we can check a property “a patient gets a dosage of medicine if and only if it is prescribed for him”. We use model checking technique to verify linear temporal properties (LTL) of applications as required in Deliverable D2.1 [15]: requirements ID[112] and ID[114]. We are using asynchronous communication as applications communicate with the environment in a distributed way.

To do model checking of applications for LTL properties, we plan to use the Spin model checker [33]. As the Spin model checker only accepts application models in the Promela [33] language, we use a new tool to create Promela models from models of applications. The Spin model checker then takes Promela model of an application along with an LTL property to check whether the
property holds for the application. A particular challenge here is back tracing of errors, which will be considered later (requirement ID[113]).

Each application after passing the verification phase has a list of LTL properties that it satisfies and a list that it does not. This information is contained in the application container of an application. In this way, it becomes easy to decide whether an application can be developed in a trustworthy way or not.

3.3.2. Application Container

After a trusted or critical application is developed using the TAPPS toolchain, it can be converted into a binary and wrapped inside a so-called application container. The application container is then ready to be deployed to the App Store. The application container can store trusted or critical applications as requested in Deliverable D2.1 [15] : ID [115]. The application container includes crucial information for deployment, including a description of how the application will operate on the TAPPS device as requested in Deliverable D2.1 [15] : ID [116] and ID[103].

The information stored within this application container includes:

- The version number of the application
- A specification of the CPS that the application has been developed for
- A list of tests that have been performed on the application prior to deployment (e.g., model checking)
- A list of the LTL properties that the application satisfies
- The permissions and hardware resources the application is going to require once installed on the TAPPS device, including the API calls that the application will make to access the hardware
- The WCET of the application, if applicable
- A signature of the application container, including the aforementioned information
- The signature of the vendor that has created the application

The purpose of the application container is to allow the TAPPS device to preemptively determine if the application can be installed successfully on the system without disrupting normal operation or other applications that are already installed. It also enables the TAPPS device to ensure that it has downloaded the correct version of the application from a trusted vendor, as well as to verify the integrity of the application and its properties. The CPS can also verify the hardware resources and API calls that the application is granted access to, prior to installing and executing it.

3.3.3. App Store

Once a trusted application is deployed in form of an application container, the application is ready to be exposed in the App Store. The App Store will provide all the tools necessary for

- A supplier to expose its new developed applications,
- A vendor to create offers, which could be a bundle of multiple CPS devices and applications
- The end-user to be able to subscribe to the offers
To support those functionalities, the App Store is composed of the following components: the back-office, the eShop front end, and the portal application.

The back-office covers the following scope:

- Allows the operator of the marketplace which manages the B2B suppliers and vendors as requested in Deliverable D2.1 [15]: ID[122], ID[118]. Several marketplaces can be implemented by the operator to allow segmented vendor ecosystems and offers as requested in Deliverable D2.1 [15]: ID[122].
- Allows the operator to implement a customized workflow for the application activation process as requested in ID[121] of Deliverable D2.1 [15].
- Allows the supplier to declare new applications, once it has been verified by the toolchain and deployed on the application container.
- Allows the vendor to manage offers, composed of different products as requested in ID[120] of Deliverable D2.1 [15] from different suppliers (see Deliverable D2.1 [15], requirement ID[118]).
- Allows the vendor to manage end-users.

The Store eShop Front End is used by the end-user to subscribe to offers that are containing applications and TAPPS devices. The App Store gives the end-user access to the vendor catalogue and to the subscription processes as requested in Deliverable D2.1 [15]: ID[125] and ID[126].

The Portal application is used by the end user to authorize an application to be downloaded on its related TAPPS device. The portal application covers the following scope:

- Access to an easy dashboard where the user will be able to centrally manage the settings of all applications, where all notifications from applications can be centralized as requested in Deliverable D2.1 [15]: ID[128],
- Access using single sign on (SSO) any application back office portal subscribed on the platform as requested in D2.1 [15]: ID[127],
- Additionally, it can provide ETSI M2M data mediation function able to control the exchange of information between CPS and applications

3.4. Application Services

In the following, we detail several important services for the TAPPS applications, i.e. the inter-execution environment communication and the Safety Integaration Layer.

3.4.1. Inter-Execution-Environment Communication

An important requirement is to ensure secure and authorized communication between the execution environments, as specified in requirement ID[20] of Deliverable 2.1. Below, we describe the approach for such secure communication.

GlobalPlatform [18] is a non-profit association, which develops and publishes specifications for the TEE Standard as a means to cope with the need for uniformity between trusted environments. This association aims to conciliate security and interoperability for embedded systems.
The main contribution of the GlobalPlatform TEE API standard is the concept of Trusted Execution Environment (TEE) [19], which shares the device with the Rich Execution Environment (REE). The specification includes both a hardware and a software architecture, without dictating a particular implementation of either, but offering the security principles and software APIs to build one. As said in Section 3.1.4, ARM TrustZone is a hardware implementation of TEE compliant with the hardware specifications designed by the GlobalPlatform, thus in this section we focus on the software part only.

Figure 11: The GlobalPlatform TEE architecture (source: TEE System architecture specification 1.0)

The software of the GlobalPlatform TEE is basically a set of APIs that enables communication between the REE and the TEE. But it allows also the Trusted Applications to communicate with one another and with the Trusted Operating System (Trusted OS) as required in Deliverable D2.1 [15]: requirement ID[80]. The following are the main actors of the GlobalPlatform TEE software specification (Figure 11):

- **TEE Client API and Communication agent**: The TEE Client API [20] is an interface that allows Client Applications in the REE to communicate with Trusted Applications in the TEE, while the TEE Communication Agent provides support for messaging between the Client Application and the TEE. The functional API has not yet been fully specified in the GlobalPlatform specification and, as a consequence, this component will not be further discussed in this document.

- **The Trusted OS components**: The Trusted Operating System (OS) Components are the Trusted Core Framework that enables OS like functionality for Trusted Applications, the Trusted Functions that provide facilities for development, the TEE Communication Agent, counterpart of the REE Communication Agent in the REE, and finally the Trusted Kernel.

- **Communication of the Trusted Applications**: Regarding the Trusted Applications (TA), their communication is performed through the TEE Internal API that defines the fundamental software capabilities of a TEE.
The GlobalPlatform REE hosts the TAPPS REE & TEE compartments, while the TAPPS CEE is isolated in the TEE.

In addition, it is also important to mention that the GlobalPlatform TEE specification is built on top of the system “chain of trust”, which provides reliability to the TEE. The root of this chain is called root of trust, and it is based on the hardware features used to protect cryptographic keys, perform authentication and verify software. Software wise, this root is represented by the booting process (more concretely the bootloader), as it is the first step taken during the execution. For this reason, GlobalPlatform defines what it is called secure boot process.

![Diagram of secure boot process](source: TEE System architecture specification 1.0)

As shown in the Figure 12, the TEE bootloader boots from a ROM memory and soon after validates the authenticity of the Trusted OS. If the check passes, the Trusted OS checks and starts the REE initialization firmware, which verifies the authenticity of the Rich OS and eventually boots it. If the root of trust is not breached and these steps are followed, then the system is dependable.

Finally, in addition to security, the TAPPS Secure Boot mechanism will be optimized to provide high performance, as specified by TAPPS requirements (D2.1 [15]: ID[14]).

**Trusted Execution Environment ↔ Critical Execution Environment**

An extended TEE API (see Section 2.2.2) is the interface used to enable the communication and the exchange of data between the Trusted and the Critical Execution Environments (CEE).

As shown in Figure 13, the typical interaction between TEE and CEE starts with a request from a VM application to access a critical service (e.g., the CAN/TTTEthernet bus) through a virtualized instance of the TEE Client vAPI (Point (1) in Figure 13). This request goes through the host TEE Client API, which asks the Rich OS kernel (Linux) driver to forward it to the Trusted OS (FreeRTOS)
When the request has arrived in the CEE through the REE and TEE communication agents (3), it is assessed by the TAPPS Safety Integration Layer (SIL), which enforces security policies based on the identity of the caller, the authenticity of the request, etc. Depending on the SIL decision, the request reaches the target trusted app (4). This application will then interact with the trusted peripherals if needed (5), before providing the result of the request back to the TEE guest’s application (6).

The REE is not involved at all in this interaction.

Rich Execution Environment ⇔ Critical Execution Environment

The interaction between the REE and the CEE targeting the access to trusted peripherals is based on a device mirroring technology with read only permissions for untrusted applications (Figure 14). When these applications want to access CAN/TTTEthernet devices, they use a driver in the guest kernel (Point (1) in Figure 14), which presents the same interface of the real device they are aiming at.

The backend of this driver is a QEMU device (2), which is able to access through a read only mechanism the information provided by a trusted memory device application (3). The read only permissions are enforced by the hardware, while the information shared by the CEE with the REE are filtered by the trusted device mirror application and the SIL (4).
The TEE is not involved at all in this interaction.

### 3.4.2. Safety Integration Layer

To coordinate the different resource accesses from the different apps within the proposed architecture, while still ensuring a sound real-time behavior and a non-interfering execution of apps, the Safety Integration Layer (SIL) is in place. Matching with the TAPPS requirements ID[79], ID[81], and ID[107] described in Deliverable D2.1 [15]: it shall provide an internal, asynchronous, communication API for critical apps running within the CEE. This communication API shall enable predictable accesses to the peripherals of the hardware platform, as well as a predictable communication among different C-Apps.

As requested by requirement ID[84] in Deliverable D2.1 [15], individual apps shall be capable of providing their own API to the system that can be used by other apps. On the one hand, this includes functions provided to less qualified apps (trusted and untrusted) through GlobalPlatform’s TEE API, and on the other hand, functions that can only be used by other Critical apps. The Safety Integration Layer shall provide methodologies to connect the functions provided by different apps with each other and for discovering available functions in the system, with respect to the system’s security and safety policies.

Furthermore, the SIL shall provide fault isolation, which ensures that a faulty app does not affect other apps. This means that resources that are shared between apps like memory or CPU time are divided in a safe way between the apps and with respect to the overall application prioritization, as required by requirement ID[82] of D2.1 [15]. This also includes the access to these resources and thus the control over implicitly shared resources like buses and caches, as well as the handling of unauthorized read and write accesses to the main memory and attacks like denial of service or buffer overflows.

If an installed app is vulnerable to attacks like buffer overflows it could be exploited by hackers and make the whole system less secure and safe if the app would be executed without the Safety
Integration Layer. If the number of app contributors increases it is also possible that faulty apps are created intentionally. To limit the influence of such faulty apps the different components of the execution environment and all apps shall be isolated into their own fault domain. The behavior of an app exploited with a buffer overflow can be changed completely for example by using Return Oriented Programming (ROP). Still it is not possible for the hacker to change the behavior of a non-exploitable app in an uncontrolled way. Beside this safety aspect, it is also not be possible for an exploitable app to read confidential data from another non exploitable app installed in the same system. Additionally no app is authorized to directly write uncontrolled to another app. Such faulty behavior can be hidden deep in the app by evil app contributors for example using obfuscation.

With this concept, we see the strict application encapsulation demanded by ID[89] of D2.1 [15] fulfilled.

As special part of the Safety Integration Layer, the Integrity Manager shall ensure that all critical apps can access there declared resources at any time without interference by other apps and in accordance to their timing requirements. This includes schedulability tests during the deployment of new applications, as well as enforcing the schedule during normal execution with the means of the Safety Integration Layer, which matches the requirements of requirement ID[88] of D2.1 [15]. In combination with the predictable communication API a compositional API for Worst Case Execution Time (WCET) analysis is provided that can be utilized to separately qualify the WCET of an individual app. This fulfills requirement ID[83] of D2.1 [15].

Beside the Integrity Manager, the SIL integrates the components defined in the GlobalPlatform Internal API, and can partially be executed in kernel space of the Real Time Operating System.

Figure 15: Installation process for trusted and critical Apps. The blue path illustrates the process flow for a critical application, and the black one that of a trusted application. The read path between the Crypto Service and the Public Key represents a secured interface.
The process of installing a new application either in the trusted or in the critical execution environment, as requested by Deliverable D2.1 [15]: ID[16] and ID[41] is illustrated in Figure 15. It is assumed that the installation process within the untrusted world is handled by a regular Android app store, like e.g., Google’s Play Store.

The TAPPS trusted installation process is a five steps approach that slightly differs between T-APPs and C-APPs. However, the concepts stays the same. First the application container is downloaded from the app store with the help of a download service (1). The download service adapts to the interface provided by the app store and establishes a secure connection to it over an otherwise untrusted network. The app is then delegated to the responsible installation process (T2/C2). There is one for the TEE, and one for the CEE. These installation services will handle the actual registration of the new apps with the execution environment. The installation service itself first needs to ask the crypto service for an validation of the downloaded app container (T3/C3). This is done by checking the digital signature of the app container with a public key that is securely stored and accessed within the CEE (4). After an app container is successfully validated it is finally installed, i.e., registered, with the respective execution environment (T5/C5). In case of C-Apps the installation service also checks whether the new application would threaten the overall system schedule. Thus, it needs to be checked whether all timing constraints can still be held with the new application in place and whether all hardware resources would be still able to satisfy the demands off all already installed applications. In case the system cannot afford to run the new application, due to resource constraints, it will be rejected. With this process we help to enforce the priorization of applications, and avoid the risk of un-authorized apps interacting with the critical interfaces of the system, which in turn fulfills the requirements ID[86] and ID[118], and partially the requirement ID[82] and ID[75] of Deliverable D2.1 [15].
4. Architecture Interfaces

The TAPPS project aims to build an ecosystem of companies and stakeholders in different market segments (i.e., automotive and healthcare) as well as a community of third party developers. In order to accomplish this goal, open standards and a clear definition of the interfaces between the TAPPS platform components is of pivotal importance.

In this section, the interfaces between the most important functional components of the TAPPS architecture will be detailed.

4.1. App Store Interfaces

The App Store exposes REST/XML API to allow the different roles in the TAPPS architecture to achieve their main functions.

- A supplier deploys a new application in form of an App Container. The application supplier must provide a list of information that are described in Section 3.3.2, and requested in Deliverable D2.1 [15]: ID [116]).
- A supplier registers a new application on the App Store.
- The end-user subscribes to an offer containing CPS devices and the application
- The end-user provision the devices that will be used on the platform
- The App Store notifies the application supplier that a new subscription has been done
- According to the permission granted, the CPS device can contact the app store to download the application as requested in Deliverable D2.1 [15]: ID [123]
- The app store activates the application activation process to allow the CPS device to download the application
- Once the application is deployed, the CPS can check if the application deployed is conformed. The CPS must contact the app store to verify the conformity of the applications (see Deliverable D2.1 [15]: ID [117]). It will use the manifest described in the app container. See Deliverable D2.1 [15]: ID [124])

The App Store APIs and the triggering of download service will be developed for the TAPPS project.

4.2. Critical Apps ➔ CAN

CAN communication infrastructure is configured via the CEE. The driver can be socket-based following the transition from legacy CAN driver to the SocketCAN [26]. Even though a character device interface to send and receive raw CAN frames, directly to/from the controller hardware is easy to program, the SocketCAN concept allows the exposure of many CAN interfaces to the CEE (TEE processes communicate through the CEE counterparts that actually accesses CAN interface). However, in view of predictability and robustness advantages of RTOS executing in the CEE we envision the integration of plain CANbus driver functionality.

CAN-frames can be transmitted one at a time to avoid priority inversion issues, while interrupt or polling-based transmission modes can be selected on a case basis, in relation to the real-time behavior of each ECU. CAN promises the delivery of a message unless the bus is overloaded or the node is in Bus Off mode. In particular, it is required to assess if using the CAN transmit
interrupt is the fastest way to be notified that a message has indeed been sent successfully and therefore the quickest way to inform the main application to proceed with e.g. an application state machine [13].

As CAN is a serial bus protocol that supports priority-based message arbitration and non-preemptive message transmission, it is important that any authentication add-on to the driver or firmware layer will respect/abide with system real-time constraints.

In this scope the CAN API comprises a set of core functions that allow using the CAN network without having to commit attention to all the details of setting up and communicate with the CAN peripherals. API functions can be categorized to initializing the CAN peripherals, a set of Transmit and Receive functions for communicating a message and Error functions.

### 4.3. Critical Apps ↔ TT/Ethernet

When TAPPS applications are running in a distributed fashion, i.e. they are deployed on multiple nodes and require communication between them, TAPPS-Device to TAPPS-Device communication is required. This communication can be handled by various means, one of them being the TT-Ethernet network.

The communication to the TT-Ethernet network is handled by a driver and the respective TT-Ethernet controller hardware. The driver resides inside the operating system and provides the ports for time-triggered, rate-constrained and/or best-effort traffic to the applications. It furthermore provides access for each application to its configured memory partitions in the TT-Ethernet controller.

The dataflow between two TAPPS devices is depicted in Figure 16. Note that for simplicity reasons the figure does not depict the whole TAPPS architecture, but only the Critical Execution Environment with FreeRTOS.
4.3.1. Deterministic Ethernet Configuration

In order to use deterministic Ethernet for real-time communication, two key requirements are:

1) The availability of a global timebase that is shared between all the nodes

2) The availability of a communication schedule that defines when each node is allowed to send its data using TT links.

For the global timebase, a synchronisation algorithm is running in deterministic Ethernet nodes (in switches and end-systems, for details, see the SAE AS6802 TTEthernet standard). Properties of the synchronisation can be configured during design time, e.g. the precision, redundancy, etc.

For the communication schedule, the communication behaviour of the time-critical applications must be known and fixed at design-time. This behaviour must be known in order to guarantee the correct and deterministic transmission of data. Furthermore, the physical and logical architecture must be defined in order to calculate the respective routes and schedule of data at design time.

The Deterministic Ethernet configuration toolchain provides the necessary tools to calculate all the required configuration data for the network components and to load these configuration into the respective switches and nodes. The overall TTEthernet configuration toolchain is depicted in Figure16. It consists of the following main parts:

- **TTEPlan**: TTEPlan is the TTEthernet network planning tool. Based on input provided to the tool, TTEPlan creates the whole network configuration databases.
• **TTEBuild**: TTEBuild allows converting XML-based device configuration database files into binary configuration images required by the TTE Switches and the TTE End Systems.

• **TTELoad**: TTELoad is an application suitable to configure a TTE Switch based on TTEthernet switch IP that also supports bootstrap configurations of TTE Switches.

• **TTEView**: This TTEthernet frame dissector for Wireshark¹ 1.x is a plug-in to Wireshark which supports the recording and analysis of over 300 Ethernet and internet protocols including TTEthernet.

An overview of this toolchain showing input and output files is presented in Figure 17. TTEPlan can be used to configure a network from scratch, or to migrate an existing configuration to a network description file. The configuration output of the toolchain is a schedule that can be downloaded or otherwise communicated to the TTEthernet network components. It defines the time-slots during which communication on the network will occur including a separation along the different communication types co-existing in the network.

![TTEthernet configuration toolchain](image-url)

Figure 17: TTEthernet configuration toolchain
In TAPPS, this toolchain must be used at design-time. When defining the architecture of a system in which distributed applications will be used, the toolchain ensures the correct configuration of the switches and nodes in the network.

The toolchain covers requirements ID99 and ID100 for the deployment of distributed TAPPS systems.
5. Architecture Utilization in Use Cases

The following two sections describe the use cases considered in the TAPPS project, as well as specific requirements and instantiation of the TAPPS architecture for the use cases.

5.1. Energica Superbike

The current existing Energica dashboard has a proprietary operating system. It implements traditional software for automotive ECUs and it is connected to the external gateway unit via the CAN off-chip network. Protection on Energica Superbike has been achieved by partitioning components across distributed modules, which communicate over a CAN bus network (Deliverable D2.1[15]: ID[1], ID[5], ID[6]). Energica Superbike use case need to integrate applications with different safety levels. In order to enhance the current version of the motorbike by supporting applications directly inside the motorbike using TAPPS technologies Energica plans to design an evolution of the current architecture by enhancing the dashboard as well as the Gateway unit. We could define this demonstrator as “Trusted Dashboard” (T-DASH) unit network (D2.1 [15]: ID[3], ID[4]). T-DASH includes a dashboard unit and a Gateway unit may be merged in a single unit. Trusted dashboard will implement TAPPS architecture (D2.1 [15]: Section 2.1 and ID[10], ID[11]). Thanks to the TAPPS Technology the current Energica dashboard will be improved. About the Motorbike Policies and Regulations that have been illustrated in Deliverable D2.1 [15] (ID[7], ID[8], ID[9]) they will guide the design and the development of the TAPPS project but it is not planned to carry out a complete certification process during the project lifetime.

The T-DASH unit shall provide three classes of apps called untrusted apps, trusted apps and critical apps. TAPPS application examples for the Energica Superbike are listed below. The classification has been carried out taking into account the three execution environments (TAPPS REE, TAPPS TEE and TAPPS CEE).

<table>
<thead>
<tr>
<th>Application Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TAPPS REE</strong></td>
</tr>
<tr>
<td>- Request Assistance for Vehicle</td>
</tr>
<tr>
<td>- Upload Lap Times/ Track/ Statistics</td>
</tr>
<tr>
<td>- Download Waypoint with attributes (Charging Points and Point Of Interest)</td>
</tr>
<tr>
<td>- Reserve Charge Station for next charge</td>
</tr>
<tr>
<td>- Live Data Stream of a limited set of information: Position, Speed, Battery Status, RPM, Temperatures and other useful data</td>
</tr>
<tr>
<td>- Request Vehicle Actual Position</td>
</tr>
<tr>
<td>- Navigation System</td>
</tr>
<tr>
<td>- Social Apps</td>
</tr>
<tr>
<td><strong>TAPPS TEE</strong></td>
</tr>
<tr>
<td>- Configure User Parameters</td>
</tr>
</tbody>
</table>
- Control of Charging Parameters
- Deactivate (lock) Stolen Vehicle
- Monitor Actual and Past Error frames
- Upload Diagnostic Log
- Perform End of Line Procedures
- Lock/Unlockable features (e.g. traction control and sport motor map)
- Trip Capability Check
- Dynamically Adjust of Braking Settings
- Dynamically Adjust of the Range
- Motorbike Management

| **TAPPS CEE** | - Remote Firmware Upgrade: Secure platform for firmware download and recovery (Premium FOTA)
 | - Configure advanced parameters |

Applications are described in more detail in D2.1 [15] – Requirements, Use Cases Report (Section 2.1, Pages 38, 39, 40 and ID[15], ID[16], ID[17], ID[18], ID[19]).

Assuming that the TAPPS display will have a 6.5” dimension and a resolution of 1024X600 (D2.1 [15]: ID[12]) :

- Data to be evaluated by TAPPS REE could be displayed for example in the following blue box (512x300 pixels);
- Data to be evaluated by TAPPS CEE and TEE could be displayed outside of the blue box;
- The blue rectangle is painted by TAPPS TEE;

![Figure 18: Display sharing organization](image)

If more space will be required in order to properly display all the TAPPS information scroll bars and/or a paging system may be implemented. Possibly more than one area could be reserved for TAPPS information.
For the TAPPS demonstrator suitable Use Cases (D2.1 [15]: Section 2.2) will be selected between:

- Use Case 1: Premium FOTA
- Use Case 2: Integration of 3rd-party services
- Use Case 3: Motorbike Management
- Use Case 4: Adjustment of Braking Settings
- Use Case 5: Trip Capability Check

In order to provide a good example of intended interaction and cooperation behavior of T-Apps and C-Apps the focus will be on the UseCase 5 “Trip Capability Check” (or Range Estimator): Whenever the driver sets the trip destination, the T-App monitors the state of charge of the vehicle (SOC), the state of the health (SOH) of the vehicle, residual WattHour of the battery and ambient temperature. Based on this information T-App can calculate the estimated range. The C-App check the following two conditions:
  1) Energica is able to reach the destination without performance limitations;
  2) Energica is able to reach the destination with performance limitations e.g. speed, torque and motor map.

It is possible to provide a good example of intended interaction and cooperation behavior between U-Apps, T-Apps and C-Apps. In this case it’s possible to talk about an Advanced Range Estimator. T-Apps can pass to the U-Apps the state of charge of the vehicle (SOC), the state of the health (SOH) of the vehicle, residual WattHour of the battery. U-Apps based on this vehicle information and route parameters such as distance, slope, the number of curves and traffic can calculate the estimated range. T-App can evaluate this estimated range. The C-App check the following two conditions:
  1) Energica is able to reach the destination without performance limitations;
  2) Energica is able to reach the destination with performance limitations e.g. speed, torque and motor map.

Energica Superbike specific requirements per compartment are listed in the following table. Only TAPPS CEE may have direct access to CAN bus and direct write permission. Vehicle CAN bus and some TAPPS CEE internal variables may populate special shadow devices to be used also by the TAPPS REE. These shadow devices are managed by TAPPS CEE (see Section 2.2 and Section 3.4).
TAPPS REE and TAPPS TEE do not have direct CAN access. The CAN access is indirect and occurs through the TEE Client API (see Section 2.1.4) for the TAPPS TEE and through the aforementioned shadow devices for the TAPPS REE. With the help of the safety integration layer the TAPPS CEE puts a permission system in place that controls the read and write accesses from the TEE and REE. In a first step the TAPPS REE shall have no write access to the CAN at all.

<table>
<thead>
<tr>
<th>Specific Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAPPS REE</td>
</tr>
<tr>
<td>- Indirect CAN access through the shadow device</td>
</tr>
<tr>
<td>- Read and write permission are managed by the TAPPS CEE</td>
</tr>
<tr>
<td>TAPPS TEE</td>
</tr>
<tr>
<td>- Indirect CAN access through the TEE Client API</td>
</tr>
<tr>
<td>- Read and write permission are managed by the TAPPS CEE</td>
</tr>
<tr>
<td>TAPPS CEE</td>
</tr>
<tr>
<td>- Direct CAN access</td>
</tr>
<tr>
<td>- Direct read and write permission</td>
</tr>
<tr>
<td>- Complete control and enforces security policies for TAPPS TEE and REE</td>
</tr>
</tbody>
</table>

The long range connectivity and the short range connectivity are described in greater detail in D2.1 [15]– Requirements, Use Cases Report (Subchapter 2.3, Page 54: ID[2] ID[13]).

5.2. Health Trolley

The selected use case for the implementation of trusted CPS is represented by the Smart Trolley, an innovative medical device designed to improve the current procedures in the hospital wards. By integrating different technologic resources and devices, this trolley is thought to assist nurses and healthcare professionals during the execution of daily services, such as drug preparation and administration, medications, bedside assistance, vital signs monitoring and so on. Of course, whenever the trolley prototype and its hardware-software architecture will be verified within a reliable validation campaign, it is necessary to demonstrate the feasibility of further functionalities even if not contemplated in the preliminary version (Deliverable D2.1 [15]: ID[26] and [27]).

As TAPPS project use case, a reliable and trusted architecture is needed for three main issues:

- **Security and Safety**: being the trolley designed to lower the drug administration errors, the drawers management requires the higher priority in terms of control reliability (e.g. D2.1 [15]:ID[95], ID[97] and ID[98]);
- **Privacy**: access to personal data both of patient and user must be prevented to unauthorized third parties (D2.1 [15]: ID[25] and ID[94]);
• **Flexibility**: permit to healthcare staff to access to useful app in open marketplace to better conduct their daily work’s activity.

For these reasons, the proposed TAPPS architecture can correctly fit within this use case, by implementing three parallel environments which different apps/functionalities belong to.

Within the **Critical Execution Environment** a firmware layer regulating the essential functions of the Smart Trolley is implemented, mainly managing the communication with the Drawers Unit (D2.1 [15]: ID[30]). A C-based algorithm, automatically generated by a Finite State Machine (D2.1 [15]: ID[96]), act as intermediate layer between commands and signals coming from UI and single drawers unit and the drawers subsystem, guaranteeing the opening of a specific compartment only under well-defined conditions. Moreover, the firmware may include also diagnostic algorithms (D2.1 [15]: ID[40]), to promptly detect unexpected deviations from the nominal operating mode (such as manual opening of the drawers), or power management tools (D2.1 [15]: ID[28]). These algorithms run on a Linux-based operating system, installed on an ARM multicore processor which communicates through the NoC with the other non-critical environments. A serial connection (i.e. USB) is sufficient for the management of the Drawers Unit, which represents the main feature of the device. However, even sCAN communication could be integrated within this architecture, enabling the transmission of real-time, safety-related data: for instance, cameras, bumpers or proximity sensors\(^2\) could be interesting features to detect and avoid obstacles during the motion (D2.1 [15]: ID[31] and [35]).

For both **Trusted** and **Rich Execution Environments** an Android-based operating system running on a smart device (such as a tablet or notebook acting as Trolley Gateway) is required to run trusted and untrusted applications. The TEE can be designed to host those non-critical apps working with limited-access information, such as sensible data about users or patients, or – broadly speaking – requiring a high level of supervision/control by authorized staff and the integration with the Hospital Information System (D2.1 [15]: ID[47] and ID[48]). Drug preparation and administration, remote prescription and management of the therapy are suitable for this environment (D2.1 [15]: ID[42] and ID[43]), but also the management of any device dedicated to patients' screening (D2.1 [15]:ID[21], ID[22], ID[23], ID[45] and ID[46]), e.g. point-of-care-testing (blood/urine samples...), vital signs monitoring (pulsimeter, oximeter, electrocardiogram,...D2.1 [15]: ID[44]) and so on. Serial and WiFi connections should be used for the communication from/to the peripherals (D2.1 [15]: ID[35], ID[36] and ID [37]), while – depending on the UI device – the signals transmission with the CEE should take place through USB or HDMI connections. Any other untrusted app without access to medical data can be executed in the REE, for instance: browsing medical or pharmacy data, streaming of professional contents (such as trolley tutorials) or instant messaging within the hospital staff.

At any time, the user should have access to all the three environments, thus a shared GUI is necessary (D2.1 [15]: ID[34]). As preliminary concept, on the main interface the user should have direct access both to the trusted and untrusted environment, while the background-running critical applications should be hosted on a smaller section, to be displayed whenever necessary. An example of this GUI is shown in Figure 20, while in Table 4Table 4 different application examples for each execution environment are reported.

\(^2\) Not yet integrated in the preliminary version of the trolley.
Table 4: Per execution environment application examples in the health trolley use case

<table>
<thead>
<tr>
<th>Application Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TAPPS REE</strong></td>
</tr>
<tr>
<td>- Browsing</td>
</tr>
<tr>
<td>- Tutorials</td>
</tr>
<tr>
<td>- Instant Messaging</td>
</tr>
<tr>
<td><strong>TAPPS TEE</strong></td>
</tr>
<tr>
<td>- Therapy preparation and drug administration</td>
</tr>
<tr>
<td>- Remote prescription or drug management</td>
</tr>
<tr>
<td>- Devices management (Point of care testing, barcode readers, vital signs monitoring)</td>
</tr>
<tr>
<td><strong>TAPPS CEE</strong></td>
</tr>
<tr>
<td>- Drawers management</td>
</tr>
<tr>
<td>- Trolley diagnostic</td>
</tr>
<tr>
<td>- Powertrain control (not yet implemented) (D2.1 [15]: ID[29])</td>
</tr>
</tbody>
</table>
6. Conclusions

In this deliverable we presented a first version of the system architecture and methods that will be implemented within the TAPPS project. We provided a high-level overview of all hardware and software components focusing on the role they play in the architecture and providing afterwards a more detailed description of the components.

The TAPPS architecture comprises untrusted and trusted execution environments for running applications and supports trusted application to be downloaded at runtime. In fact, three execution environments are envisaged: a rich execution environment (REE), a trusted execution environment (TEE) and a critical execution environment (CEE). This way the Global Platform TEE API standard is extended by an additional execution environment – the CEE – providing additional security mechanisms.

This architecture ensures trusted inter-app and inter-EE communication as well as secure access to peripheral devices via sCAN and deterministic Ethernet that permits for classical encryption, authentication and integrity mechanisms. One main component is the Safety Integration Layer, which is in charge of the coordination of the non-interfering execution of the apps in the CEE, strict app encapsulation and isolation of faulty apps, among other functionality.

A trusted toolchain secures trusted application development based on model-based approach of apps using state-machines and automatic code generation as well as model checking for the verification of apps properties. A trusted installation process ensures integrity and authenticity by crypto services and public key features. In addition, the TAPPS boot loader will take into account the state-of-the-art of the standardization efforts.

Further discussions regarding the architecture and the methods will arise during the progress of the implementation and will lead to extensions, detailing and possibly adjustments to the current version described in this document. The revised version will be reported in Deliverable D2.3 “Updated Architecture and Methods for 2nd Iteration”.

- 48 -
### 7. References

<table>
<thead>
<tr>
<th>Reference</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>Deliverable D2.1. Requirements, Use Case Report, submitted 17.06.2015</td>
</tr>
<tr>
<td>[21]</td>
<td>ARM, ARM Security Technology Building a Secure System using TrustZone Technology</td>
</tr>
<tr>
<td>Reference</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>[38]</td>
<td>GENIVI, accessed: 2015-12-22 <a href="http://www.genivi.org/">http://www.genivi.org/</a></td>
</tr>
</tbody>
</table>
ANNEX A: Related Projects

For the definition of the TAPPS architecture, we inspected several other architectures developed in other projects.

Within the scope of the DREAMS project an architecture for mixed-critical systems is developed, which only takes into account close systems where the applications are known at design time.

SEPIA Develops a security-architecture for mobile and embedded systems, addressing topics such as isolated execution space, virtualization and secure protection of confidential data. The SEPIA project focuses on generic techniques for trusted execution environments. Its use cases are on mobile banking and payment. TAPPS will uses the isolation and virtualization concepts extending them to off chip.

The INTER-TRUST project designs a dynamic and scalable framework that allows creating and deploying critical services and applications; and assures interoperation among devices and systems with different security policies. INTER-TRUST deals with general policies that are likely more general than what we will use in a restricted media setting. However, TAPPS will monitor the work in INTER-TRUST to see if such policies may applicable.

The OVERSEE project provided high-level security services for secure communication, entity authentication, secure storage, secure software management, secure policy decision as well as secure policy management. It employed virtualization technology to support multiple, separated applications, but does not consider hardware assisted system virtualization, while TAPPS also addresses security services on top of on-chip and off-chip network.

The recently completed EVITA project focuses on automotive safety applications based on vehicle-to-vehicle and vehicle-to-infrastructure communication and examines different threats. While EVITA project examined the use of crypto engines, TAPPS focuses on protection and isolation via compartments implemented for on-chip and off chip networks. In TAPPS we plan to address 3rd part apps.

The D-MILS project develops a trusted architecture with multiple, independent layers of security and corresponding management. Results from D-MILS will be taken up and extended in TAPPS to consider open apps platforms and integration with trusted hardware.
# ANNEX B: Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>App</td>
<td>Application</td>
</tr>
<tr>
<td>B2B</td>
<td>Business to Business</td>
</tr>
<tr>
<td>B2C</td>
<td>Business to Consumer</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>C-Apps</td>
<td>Critical Applications</td>
</tr>
<tr>
<td>CEE</td>
<td>Critical Execution Environment</td>
</tr>
<tr>
<td>CPS</td>
<td>Cyber Physical Systems</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>EE</td>
<td>Execution Environment</td>
</tr>
<tr>
<td>FOTA</td>
<td>Firmware-Over-the-Air</td>
</tr>
<tr>
<td>GPL</td>
<td>General Public License</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HDMI</td>
<td>High Definition Multimedia Interface</td>
</tr>
<tr>
<td>HMI</td>
<td>Human–machine interface/interaction</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel Virtual Machine</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>M2M</td>
<td>Machine to Machine</td>
</tr>
<tr>
<td>MISRA</td>
<td>Motor Industry Software Reliability Association</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on Chip</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PI</td>
<td>Package Installer</td>
</tr>
<tr>
<td>PM</td>
<td>Package Manager</td>
</tr>
<tr>
<td>QEMU</td>
<td>Quick Emulator</td>
</tr>
<tr>
<td>Qt</td>
<td>Cross-platform application framework</td>
</tr>
<tr>
<td>REE</td>
<td>Rich Execution Environment</td>
</tr>
<tr>
<td>REST/XML</td>
<td>reStructuredText Extensible Markup Language</td>
</tr>
<tr>
<td>ROM</td>
<td>Read-Only Memory</td>
</tr>
<tr>
<td>RPM</td>
<td>Revolutions per Minute</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real Time Operating System</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>sCAN</td>
<td>Secure Controller Area Network</td>
</tr>
<tr>
<td>SIL</td>
<td>Safety Integration Layer</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Sockets Layer</td>
</tr>
<tr>
<td>SOC</td>
<td>State of Charge – Electric vehicles</td>
</tr>
<tr>
<td>SOH</td>
<td>State of Health - Electric vehicles</td>
</tr>
<tr>
<td>T-Apps</td>
<td>Trusted Applications</td>
</tr>
<tr>
<td>T-DASH</td>
<td>Trusted Dashboard</td>
</tr>
<tr>
<td>TEE</td>
<td>Trusted Execution Environment</td>
</tr>
<tr>
<td>TPM</td>
<td>Trusted Platform Module</td>
</tr>
<tr>
<td>TSN</td>
<td>Time Sensitive Networking</td>
</tr>
<tr>
<td>U-Apps</td>
<td>Untrusted Applications</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VS</td>
<td>Vital Signs</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-case execution time</td>
</tr>
<tr>
<td>WORM</td>
<td>Write Once Read Many</td>
</tr>
</tbody>
</table>
ANNEX C: Relation to Requirements in Deliverable D2.1

The requirements of Deliverable D2.1 are mentioned in the appropriate sections above. The following list contains those requirements of Deliverable D2.1 that have not been refereed within this document, due to the justifications given on the last column of the table.

<table>
<thead>
<tr>
<th>ID Number</th>
<th>ID Name</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Storage Space</td>
<td>Is not a requirement for the architecture, will be considered later.</td>
</tr>
<tr>
<td>38</td>
<td>Execution Environment</td>
<td>This is fulfilled but also obsolete as we support now more that two execution environments.</td>
</tr>
<tr>
<td>77</td>
<td>Restrict System API</td>
<td>Will be considered for the SIL later.</td>
</tr>
<tr>
<td>78</td>
<td>Software based Tracing Support</td>
<td>To be addressed later.</td>
</tr>
<tr>
<td>119</td>
<td>Dual role support for Vendor and Supplier in App store</td>
<td>To be addressed later.</td>
</tr>
<tr>
<td>129</td>
<td>App Store Security</td>
<td>This Requirement is not covered at this stage. Indeed, the ETSI M2M communication was originally planned to be used between the CPS Device and the applications. This is not relevant anymore.</td>
</tr>
</tbody>
</table>