D5.5
Secure Hardware Platform Specification

Topic	SU-ICT-01-2018
Project Title	Artificial Intelligence-based Cybersecurity for Connected and Automated Vehicles
Project Number	833611
Project Acronym	CARAMEL
Contractual Delivery Date	M21
Actual Delivery Date	M21
Contributing WP	WP5
Project Start Date	01/10/2019
Project Duration	30 Months
Dissemination Level	Public
Editor	DT-Sec
Contributors	DT-Sec

Document History		
Version	Date	Remarks
0.1	11/05/2020	Initial version, filled in all sections with preliminary material
0.1	18/06/2020	All sections filled with planned content, ready for review
0.2	29/06/2020	Review comments integrated, final diagrams added
1.0	30/06/2020	Final editing completed, ready for submission

DISCLAIMER OF WARRANTIES

This document has been prepared by CARAMEL project partners as an account of work carried out within the framework of the contract no 833611.

Neither Project Coordinator, nor any signatory party of CARAMEL Project Consortium Agreement, nor any person acting on behalf of any of them:

- makes any warranty or representation whatsoever, express or implied,
    - with respect to the use of any information, apparatus, method, process, or similar item disclosed in this document, including merchantability and fitness for a particular purpose, or
    - that such use does not infringe on or interfere with privately owned rights, including any party's intellectual property, or
- that this document is suitable to any particular user's circumstance; or
- assumes responsibility for any damages or other liability whatsoever (including any consequential damages, even if Project Coordinator or any representative of a signatory party of the CARAMEL Project Consortium Agreement, has been advised of the possibility of such damages) resulting from your selection or use of this document or any information, apparatus, method, process, or similar item disclosed in this document.

CARAMEL has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 833611. The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed in the deliverable lies entirely with the author(s).

DISCLOSURE STATEMENT

"The following document has been reviewed by the CARAMEL External Security Advisory Board as well as the Ethics and Data Management Committee of the project. Hereby, it is confirmed that it does not contain any sensitive security, ethical, or data privacy issues."

Table of Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial intelligence
API	Application Programming Interface
CA	Certificate authority
CAN	Controller area network
CCAM	Cryptographic Accelerator and Assurance Module
CPU	Central processing unit
D	Deliverable
DC	Direct Current
DEB	Debian Package Format
DCT	Docker Content Trust
DIP	Dual in-line package
DDR	Double Data Rate
eMMC	Embedded Multi-media card
eSE	Embedded Secure Element
EST	Enrolment over secure transport
FS	File System
GPIO	General Purpose Input/Output
GPU	Graphical Processing Unit
HAB	High-assurance boot
HDMI	High-Definition Multimedia Interface
HSM	Hardware security module
HTTP	Hypertext transfer protocol
HW	Hardware
I^2C	Inter-Integrated Circuit
ICT	Information and communication technologies
ID	Identifier
IDS	Intrusion Detection System
IF	Interface
IP	Internet protocol
IPK	Itsy Package
IPS	Intrusion Prevention System
LPDDR	Low power DDR
LTE	Long-term evolution
MIMO	Multiple Input Multiple Output
ML	Machine learning
MMC	Multi-media card
MMU	Memory management unit
NVMe	Non-volatile Memory Express
OEM	Original Equipment Manufacturer
OS	Operating System
OSS	Open source software
OTP	One-time Programmable
PC	Personal computer
QA	Quality Assurance
RAM	Random access memory
ROM	Read-only memeory
RPM	Redhat Package Manager
RSA	Rivest, Shamir, & Adleman (public key encryption technology)
SD	Secure Digital
SHA	Secure Hash Algorithm
SoC	System-on-a-chip
SSL	Secure Socket Layer
TAR	Tape Archive
TCOS	Telekom Card Operating System
ToC	Table of Contents
TOPS	Tera-operations per second
TPU	TensorFlow Processing Unit

UC	Use Case
USB	Universal serial bus
V2X	Vehicle to X
Wi-Fi	Wireless Fidelity
WP	Work Package

Executive Summary

CARAMEL system components will address a wide range of security-related topics and technologies, from cyber threat detection (WP3), cyber-attack prevention (WP4), to in-depth defense mechanisms (WP5). The objective of WP5 is to provide the design, development, and prototype implementation of the CARAMEL anti-hacking device and in-depth defense solution. It will be based on different machine learning-based (ML) algorithms to detect and mitigate cyber-threats, while processing and collecting large volumes of data in future autonomous vehicle scenarios.

In this document we describe in detail the concept of a passive intrusion detection system – the anti-hacking device – implemented as an electronic control unit with integrated machine learning capability in the vehicle to detect threats to the connected vehicle in an agile manner.

Since the anti-hacking device (in-car IDS) shall improve the security of the vehicle and not create a new attack surface on its own the concepts of multi-layered security or defence-in-depth are applied to the anti-hacking device in an innovative way, integrating an embedded HSM, secure boot, secure firmware update, along with machine learning-based threat detection capability in a small form factor, low-cost device.

1 Introduction

1.1 Project Overview

The rapidly growing connectivity of modern vehicles opens numerous opportunities for new functions and attractive business models. At the same time, the potential for cyberattacks on vehicle networks is increasing. These attacks entail risks, especially with regard to functional safety and potential financial damage. CARAMEL's [1] goal is to proactively address modern vehicle cybersecurity challenges by applying advanced Artificial Intelligence (AI) and ML techniques, and also to continuously seek methods to mitigate associated safety risks. By adopting well-established methods from the ICT (Information and communication technologies) sector CARAMEL aims to develop an Anti-hacking IDS/IPS as a commercial product aimed towards the European automotive cyber security market and to demonstrate their value through comprehensive attack scenarios.

1.2 Document Scope

CARAMEL system components will address a wide range of security-related topics and technologies, from cyber threat detection (WP3), cyber-attack prevention (WP4), to in-depth defense mechanisms (WP5).

The objective of WP5 is to provide the design, development, and prototype implementation of the CARAMEL anti-hacking device and in-depth defense solution. It will be based on different machine learning-based algorithms to detect and mitigate cyber-threats, while processing and collecting large volumes of data in future autonomous vehicle scenarios. All these processes will be executed with state-of-art algorithms developed in WPs 3 and 4 of the CARAMEL project – updated in real time depending on the situational awareness about the underlying system at any time.

In this document we describe in detail the concept of a passive intrusion detection system – the anti-hacking device – implemented as an electronic control unit with integrated machine learning capability in the vehicle to detect threats the connected vehicle in an agile manner.

Since the anti-hacking device (in-car IDS) shall improve the security of the vehicle and not create a new attack surface on its own the concepts of multi-layered security or defence-in-depth are applied to the anti-hacking device in an innovative way, integrating an embedded HSM [2][9], secure boot, secure firmware update, along with machine learning-based threat detection capability in a small form factor, low-cost device.

1.3 Document Structure

This document is structured as follows:

Chapter 2 recaps the security features of the anti-hacking device and relates these efforts to other tasks in WP5.

Chapter 3 introduces the three different secure hardware platforms that have been chosen as the base for the anti-hacking device implementations done in the CARAMEL project. The three different hardware platforms address different security, cost, performance, and form factor requirements.

Chapter 4 introduces the concept of layered security or defense-in-depth on a high level. These concepts are the broken down and further detailed in the following chapters.

Chapter 5 looks at how the software load for two of the hardware platforms is created to enhance security and availability.

Chapter 6 gives details of the secure boot mechanism, while chapter 7 focuses on the secure firmware update process.

In chapter 8 a very important aspect of the anti-hacking device software architecture is covered: Docker container technology provides the agility required by an intrusion detection system needed for a swift

reaction to new threats and attack surfaces. This chapter describes how Docker containers can be deployed quickly and securely using Docker Content Trust technology.

Finally, we summarize the findings in this document and give an outlook for the final deliverable D5.6 in WP5.

2 The CARAMEL Anti-hacking Device

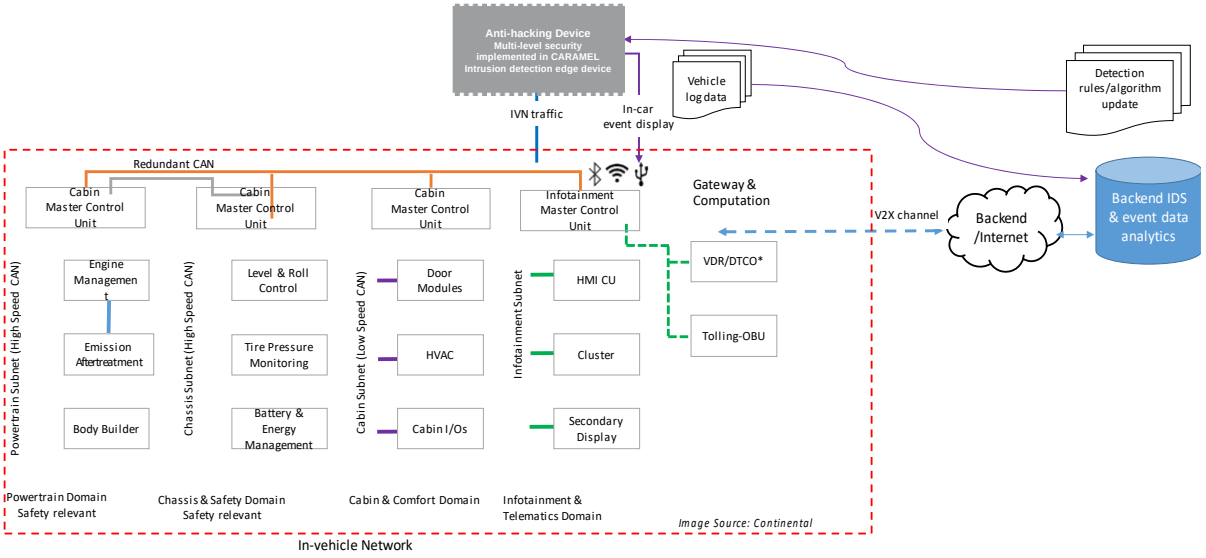

Figure 1: The anti-hacking device in the vehicle

The CARAMEL anti-hacking device is designed as a passive intrusion detection device that is integrated as an additional controller into the vehicle (see Figure 1). The anti-hacking device *passively* listens to the car's internal busses and systems, processes and aggregates raw data from sensors and communication controllers and uses machine learning (ML) and other heuristics to detect possible attacks against the vehicle's systems.

It then *actively* creates attack reports (events) and sends them to the CARAMEL backend. Details of the integration of the anti-hacking device into the different CARAMEL scenarios are described in the CARAMEL specification [4].

The anti-hacking device needs to be updated very frequently to run updated attack detection algorithms to counter newly discovered attack vectors. This requires frequent updates of the anti-hacking device firmware and application load. From a vehicle safety perspective any corruption of the anti-hacking device by malicious actors must be avoided at all costs.

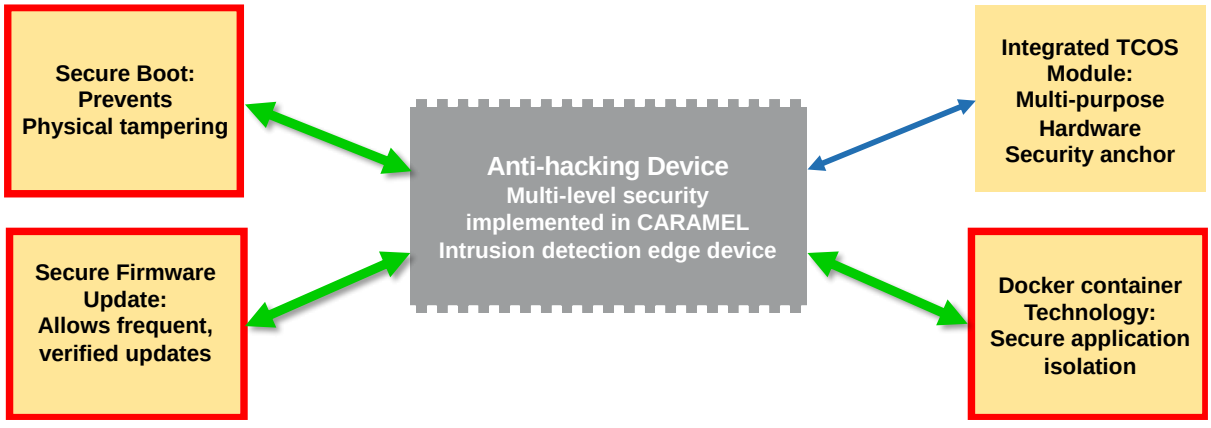

Figure 2: Anti-hacking device security features

To this end multiple security features will be implemented in the project to harden the anti-hacking software against any kind of attacks (see Figure 2):

- **Secure Boot:** The anti-hacking device hardware has fuses (write-once programmable storage locations) that contain the public keys of acceptable boot loader signatures. The anti-hacking device only loads a correctly signed bootloader. The bootloader in turn verifies the signature of

the Linux Kernel and only continues to load a verified kernel. These measures counter any physical tampering attacks on the boot medium.

- **Secure Firmware Update:** The anti-hacking device allows updating the firmware of the Internet (eg. over the vehicle's communication controller via LTE/5G). The anti-hacking device only accepts firmware update files that are properly signed by the anti-hacking device vendor. This protects the device against the installation of manipulated firmware images. In addition to this signature check the anti-hacking device implements also Secure Boot and would reboot to the last known safe state even if the secure firmware signature check were circumvented – effectively implementing multi-level security here.
- **Docker technology:** The anti-hacking device encapsulates the actual detection algorithms and also some system services into Docker containers. This has several advantages: It allows update of detection algorithms without a full firmware update. Additionally, the detection algorithms are separated by the protections offered by the Docker runtime against any mutual interference. As a last security measure, the anti-hacking device only accepts signed Docker images from pre-defined trusted sources, effectively also implementing multi-level security for Docker implementation on the anti-hacking device.
- ***Integrated TCOS (HSM) module:*** *The anti-hacking device contains a hardware secure module (HSM) in the form of a Telekom Card Operating System (TCOS) security chip. Like a smartcard, the TCOS module offers secure storage of private key materials and certificates and the ability to run sensitive cryptographic operations securely on chip. The TCOS module offer these functionalities to Dockerized applications via a high-level security service also implemented as a Docker container.*

This document describes in detail the secure boot and secure firmware updates process as well as the secure Docker technology approach chosen for the anti-hacking device. The integration of the TCOS HSM (Telekom Card Operating System Hardware Secure Module) have been described in the deliverables D5.1 "Hardware Security Module Specifications" [2] and D5.4 "CARAMEL IDS/IPS Security Module" [9].

3 Secure Hardware Platform

Three different anti-hacking device hardware platforms have been chosen for the project. The rationale behind supporting three and not only one platform is that:

- Project partners have expressed the need to support different performance points for the machine learning algorithms developed in WPs 3 and 4. Some of the partners use only heuristic methods to detect threats that require no machine learning capability. Therefore, we needed to introduce different hardware platforms with different kinds of computational resource profiles.
- For the exploitation phase we anticipate that different OEMs have different requirements when it comes to prices and required security level. Therefore, we have selected hardware platforms that are able to meet different price and security targets.
- Finally, we strive to develop a flexible solution that is applicable for a wide range of application and integration scenarios. In order to prove the viability of the anti-hacking solution in such a wider context we have chosen to support three different hardware platforms for the CARAMEL project.

3.1 Anti-hacking Device Overview

The CARAMEL anti-hacking solution is an important part of the project innovation. In this section, we have a deeper look on the general architecture and functionalities of it.

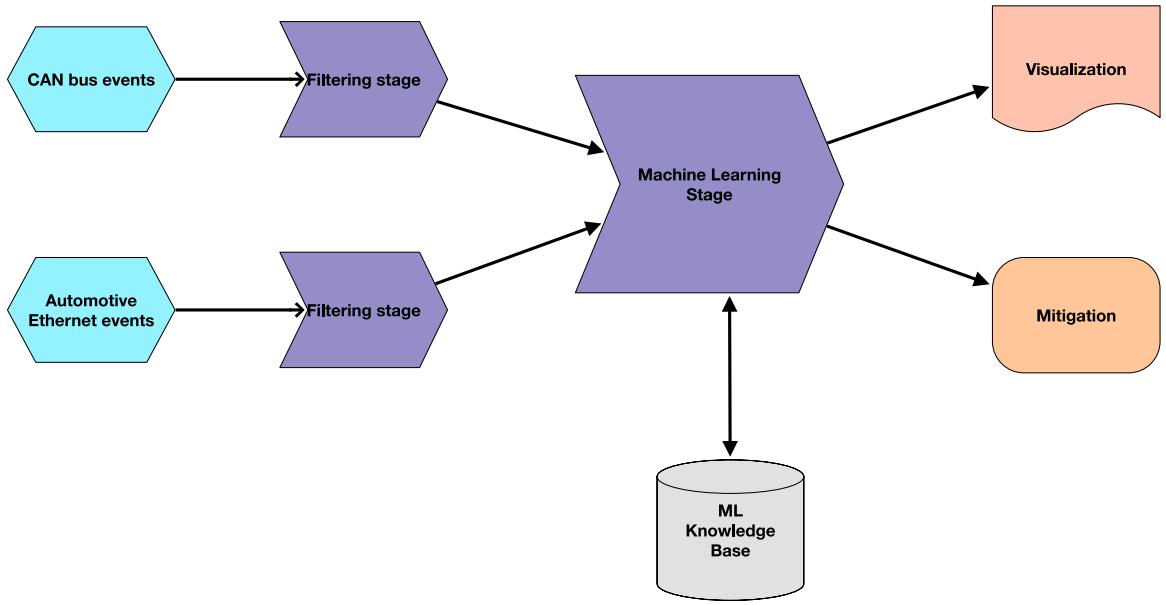

Figure 3: Machine Learning Pipeline

The anti-hacking device is a physical controller that is integrated into the car and acts as an attack detection device. In the Autonomous Mobility scenario its task is to run pre-trained ML models that work on the sensor data to detect anomalies that might point to malicious attacks. Additionally, the anti-hacking solution might be used for different functions in the context of the CARAMEL project, i.e. if needed it can ensure security for an embedded application platform. In this case, the software layer of the solution might be employed only. Further details about this approach will be presented in the rest of this document.

The anti-hacking device is connected to the busses in the car carrying the sensor data. It passively monitors the bus traffic (e.g. CAN bus frames) and extracts the raw sensor data.

Figure 3 shows the ML pipeline where raw data, e.g. from the CAN bus, is pre-filtered and aggregated to make it suitable for the following machine learning stage to detect threats and attacks. Any security-relevant events are then forwarded to the visualization and mitigation components in the car.

The ML knowledge base (model) is pre-loaded into the anti-hacking device. The model will have been created offline on a more powerful system based on simulated and real-world training data.

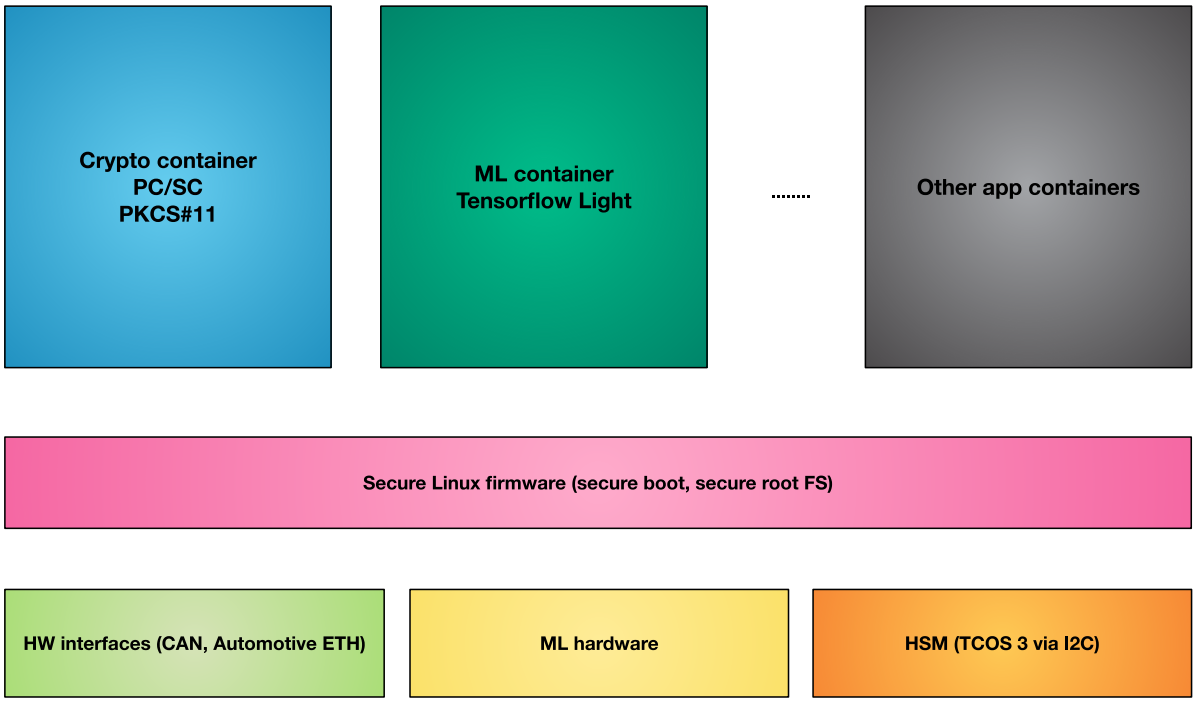

Figure 4: Anti-hacking Device Software Architecture

Figure 4 shows an overview of the software and hardware architecture of the anti-hacking device. From bottom-up the following components make up the anti-hacking devices:

- Hardware (HW) Interfaces: The anti-hacking device will be connected to the in-car systems via appropriate interfaces used in the automotive industry such as the CAN bus or Automotive Ethernet connections. For integration into development and simulation frameworks standard Ethernet will also be supported.
- The anti-hacking device will also support machine learning (ML) hardware. One variant of the anti-hacking device is based on the Coral Dev Board where the TensorFlow Lite Processing Unit (TPU) is the hardware element to support ML. The NVidia-based anti-hacking device has ML support in the SoC. For a development and simulation configuration the Coral USB Accelerator is also supported. This USB accelerator can also be used to add ML capability to the Kontron K-Box.
- HSM (hardware security module): To provide security-related functions of the anti-hacking device the hardware will integrate a Secure Element or HSM in the form of a TCOS (Telekom Card Operating System) embedded smartcard module that supports secure storage of private keys and different cryptographic operations.
- The anti-hacking device itself is based on an NXP Freescale i.MX processor that supports security functions such as hardware-assured boot.
- On this security hardware runs a Yocto-based firmware layer (a Linux embedded meta distribution), or Ubuntu 18.04 in the NVidia case.
- On top of this firmware substrate Docker-based application-specific containers can be loaded. Out-of-the box there will be crypto containers supporting the security functions of the anti-hacking device. ML workloads will be also be implemented as containers that have access to the underlying ML hardware as well as the crypto functions exported by the crypto container.

- The anti-hacking device could also act as a secure run-time environment for other functions as needed by the different use cases.

3.2 Coral Dev Board

3.2.1 Hardware Overview

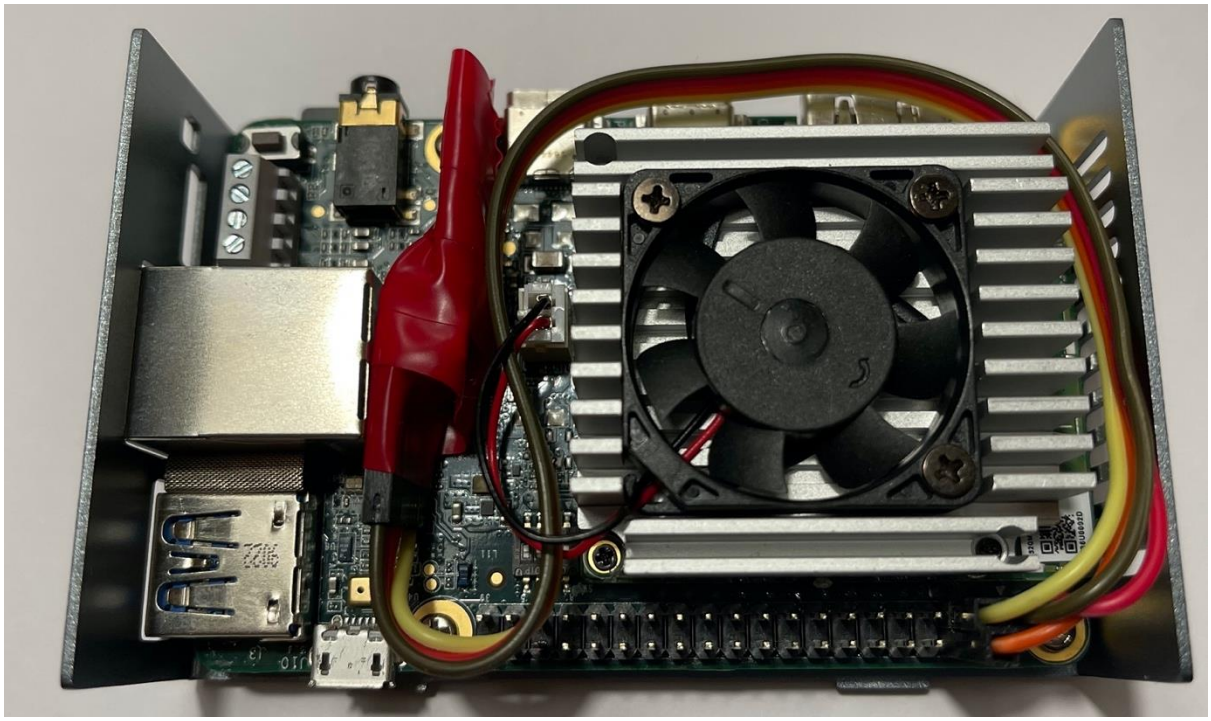

Figure 5: Anti-hacking Device Hardware with TCOS module via I2C

Figure 5 shows a picture of both the final target hardware - the Coral Dev Board[1] and the TCOS security module.

The Coral Dev Board has the following hardware specifications:

- CPU: NXP i.MX 8M SOC (quad Cortex-A53, Cortex-M4F)
- GPU: Integrated GC7000 Lite Graphics.
- Coprocessor: Google Edge TPU.
- RAM: 1GB LPDDR4.
- Flash memory: 8GB eMMC.
- Connectivity: Wi-Fi 2×2 MIMO (802.11b/g/n/ac 2.4/5GHz) Bluetooth 4.1.
- Dimensions: 48 x 40 x 5mm.

The i.MX8 SOC includes advanced security features such as HAB (high-assurance boot) and CCAM (Cryptographic Accelerator and Assurance Module) that will support the security features of the Anti-hacking device. The firmware for the i.MX8 SOC will be created using the Yocto environment which is an industry-standard toolkit to create custom embedded firmware images in a reproducible manner.

[1] https://coral.ai/docs/dev-board/get-started/

Our build process will support signed bootloaders and Linux kernel in order to prevent tampering with the anti-hacking device software and configuration.

The Coral Dev Board also has many connectivity options integrated on the board:

- Ethernet port (can be used for IP-based connections in a simulation and test environment, or to attach Automotive Ethernet adapters if needed)
- GPIO and I2C ports (used for connecting the HSM module, can be used for other purposes as well)
- USB port (used in the project to connect USB-to-CAN-bus converters)
- Wireless connectivity - Wi-Fi and Bluetooth

The Edge TPU processor integrated into the Coral Dev Board supports the execution of TensorFlow Lite models, performing 4 trillion operations (tera-operations) per second (TOPS), using 0.5 watts for each TOPS (2 TOPS per watt). The same Edge TPU is integrated into the USB Accelerator stick, so similar performance can be expected in the Anti-hacking Device simulation environment on when using the Kontron K-Box.

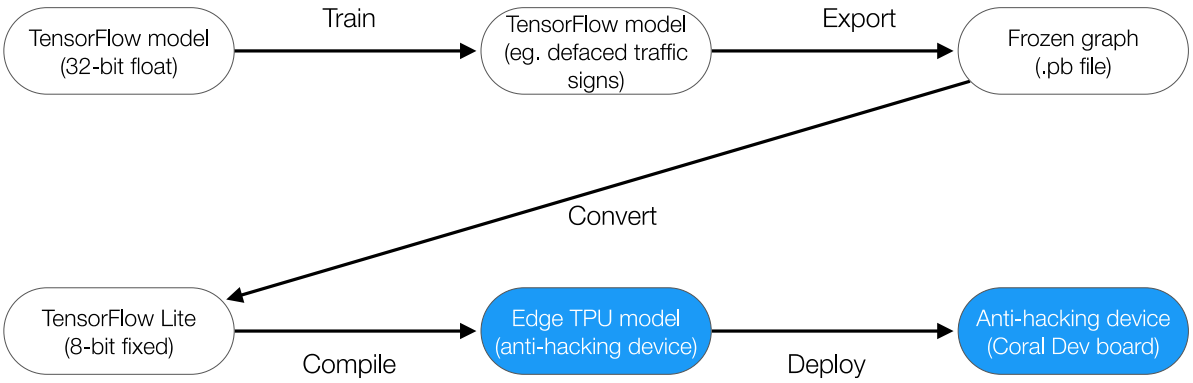

Figure 6: Conversion of Tensorflow model for use with Edge TPU

Figure 6 shows how TensorFlow models created by a machine learning process (e.g. running in the cloud or on project hardware) can be converted for use with either Coral Dev Board or the Coral USB Accelerator.

The I2C ports of the Coral Dev Board will be used to connect an HSM (hardware security module) based on the TCOS (Telekom Card Operating System) specification to act as an embedded Secure Element (eSE) and security anchor for the Anti-hacking device. The HSM is meant to support the following functions:

- Authentication of the Anti-hacking device for remote provisioning and updates
- Provide support for other CARAMEL use cases that need HSM functionality
- Authentication of the anti-hacking device against central systems such as Automotive SOC (Security Operations Centre) for event reporting and alerting

Figure 7: Coral Dev Board in Aluminum Case

Figure 7 shows the final integration of the Coral Dev with I2C module into an aluminum case. This configuration is suitable for deployment into vehicles and test environments.

3.2.2 Initial Software Installation

The firmware for the Coral Dev Board is specifically created for the CARAMEL project using a Yocto-based Linux firmware build process [11]. The details of the build process will be described in D5.6. The firmware is provided for installation on a 32 GB Micro SD card inserted in the Coral Dev Board.

First, install the official Mendel OS on the internal eMMC as described in [5].

The latest version of the Yocto-based firmware for the SD card built for the CARAMEL project is available via this link:

https://caramelx.mine.bz/caramelx-h2020-prv_dcs/coral/core-image-base-coral-dev.wic.gz

The following instructions assume that you use the “Dual-boot configuration for development purposes” described in section 3.2.3 and execute all the commands when booted into Mendel OS. If you use any other operating system (such as Linux or MacOS) you will have to adapt the commands appropriately.

First boot into Mendel OS and download the current Yocto-based firmware from the link above. Then execute the following commands to write the firmware to the Micro SD card (32 GB recommended):

```
zcat core-image-base-coral-dev.wic.gz | sudo dd of=/dev/mmcblk1 bs=4M
```

Then resize the root partition on the SD card to fill the rest of the available space:

```
resize2fs /dev/mmcblk1p2
```

This is needed to provide enough space for the Docker images on the card.

At the moment docker-compose is not properly set up in the provided firmware image. Please execute the following commands while connected to the Internet to install the missing dependencies:

```
wget "https://bootstrap.pypa.io/get-pip.py"
python3 get-pip.py
pip3 install "jsonschema<3,>=2.5.1"
cd /usr/lib/python3.7/site-packages/
rm -rf PyYAML-5.1.2-py3.7.egg-info
pip3 install "PyYAML<4,>=3.10"
pip3 install "requests!=2.11.0,!=2.12.2,!=2.18.0,<2.20,>=2.6.1"
```

This problem will be corrected in a forthcoming version of the Yocto firmware build.

3.2.3 Dual-boot Configuration for Development Purposes

For development purposes it is recommended to leave the boot switches as is into order to boot into Mendel OS first. To test the Yocto build you have to boot from the SD card, however. This can be achieved by interrupting the automatic uboot by pressing a key and issuing the following commands on the uboot command line (it make take some time until the kernel is loaded and booting, so don't power cycle but wait):

```
setenv bootdev 1
```

```
setenv bootcmd "ext2load mmc 1:1 ${loadaddr} boot.scr; source; boota mmc0 boot_a;"
```

```
saveenv
```

```
boot
```

Note that you must use the serial console (ie. the Micro USB port on the Coral Dev Board) connected to a PC to interrupt the boot process and enter the abovementioned commands.

Of course, after the Yocto-based firmware has proven stable, it is possible to permanently switch to SD card boot by changing the DIP switch positions as follows:

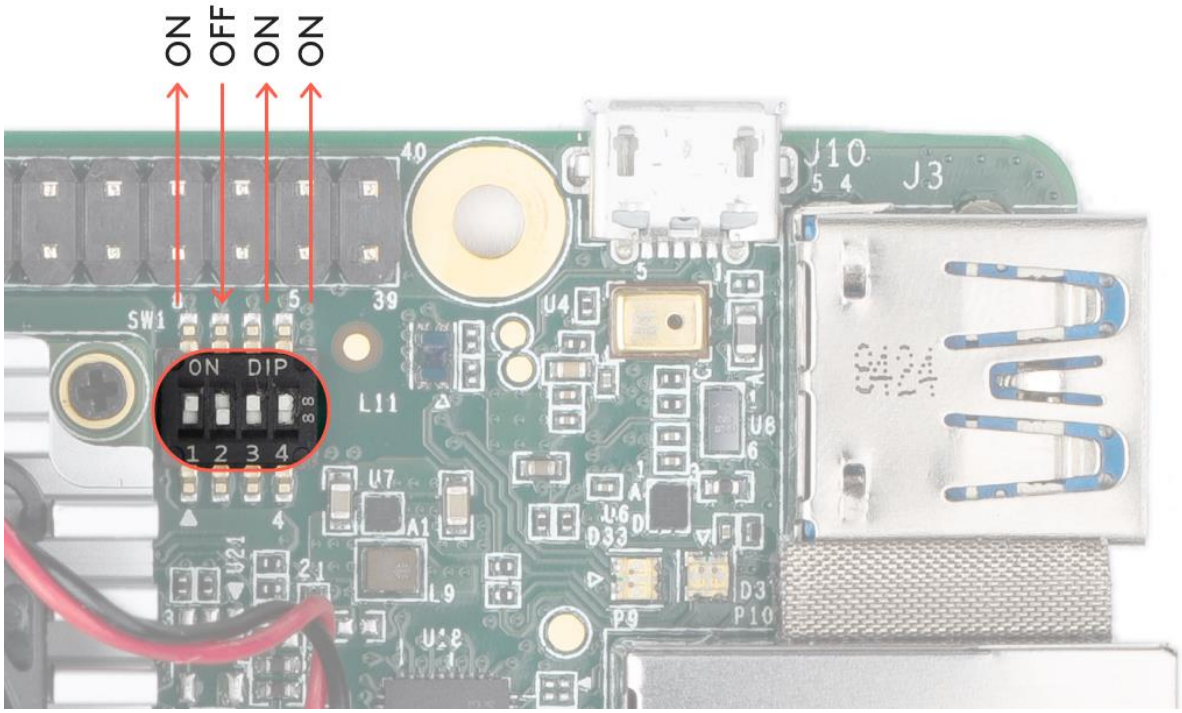

Figure 8: Coral Dev Board DIP switch positions for SD card boot

In order to boot from Mendel OS you then need to interrupt the uboot process again and issue these commands:

```
setenv bootdev 0
setenv bootcmd "ext2load mmc 0:1 ${loadaddr} boot.scr; source; boota mmc0 boot_a;"
saveenv
boot
```

3.3 NVidia Jetson AGX

3.3.1 Hardware Description

Figure 9: NVidia Jetson AGX Embedded Controller

Figure 9 shows the NVidia Jetson AGX embedded controller. The Jetson AGX is the most performant member of NVidia's Jetson range of devices. It features:

- GPU: 512-core Volta GPU with Tensor Cores
- CPU: 8-core ARM v8.2 64-bit CPU, 8MB L2 + 4MB L3
- Memory: 32GB 256-Bit LPDDR4x | 137GB/s
- Storage: 32GB eMMC 5.1
- DL Accelerator: (2x) NVDLA Engines
- Vision Accelerator: 7-way VLIW Vision Processor
- Encoder/Decoder: (2x) 4Kp60 | HEVC/(2x) 4Kp60 | 12-Bit Support
- Multiple USB connectors
- GPIO header with I2C

We have also integrated a 1 TB NVMe SSD to store large datasets locally on the device.

3.3.2 Software Installation

As of the time of this writing the NVidia provides a Jetson-specific firmware based on Ubuntu 18.04 that provides a uniform runtime and development platform for all Jetson devices.[6] The NVidia developer documentation [6] describes the installation processes of the firmware in detail.

Since the anti-hacking device functionality is based on Docker images and the Docker runtime is pre-installed in the Jetson firmware no further software installation is necessary to run the CARAMEL software stack on the device.

3.4 Kontron K-Box K-BOX A-330 MX6

3.4.1 Hardware Description

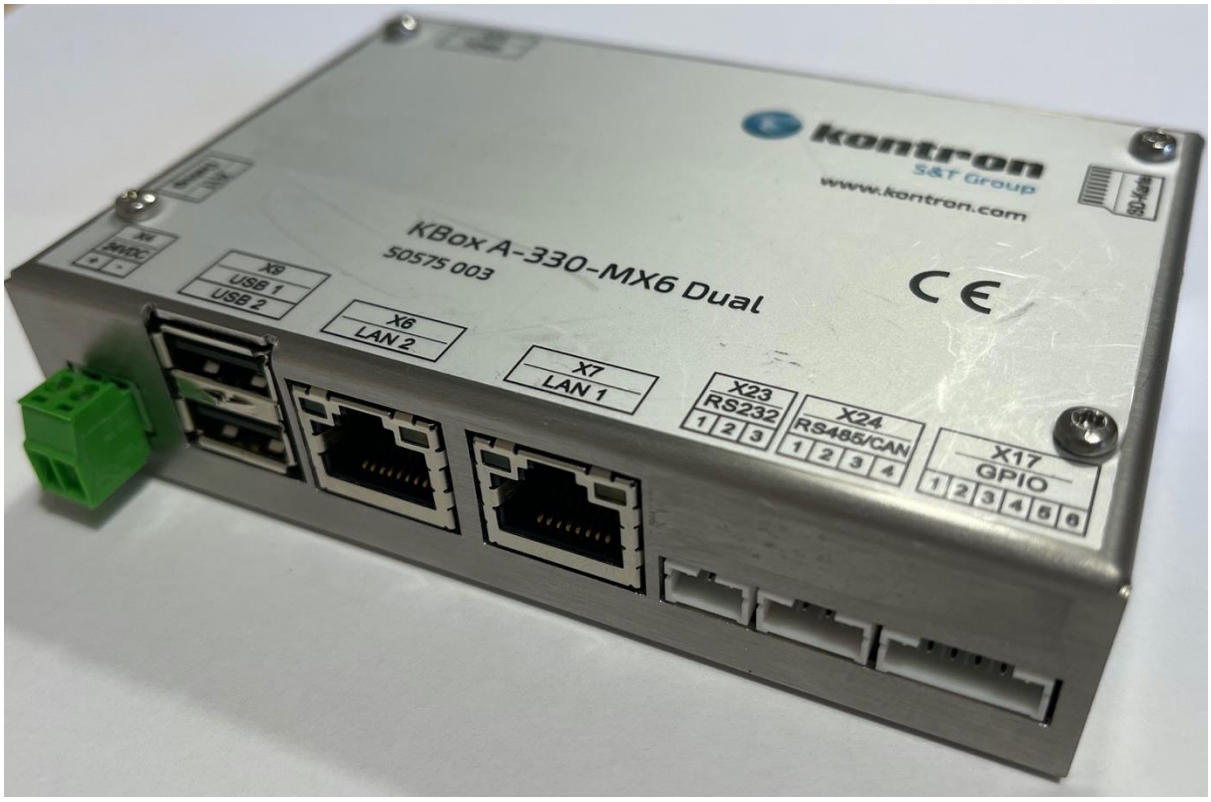

Figure 10: Kontron K-Box K-BOX A-330 MX6 Embedded Controller

CPU	NXP i.MX6 Dual Core/ULL Arm Processor
RAM technology Capacity	DDR3L 512 MByte (MX6-ULL)
Graphics format	1080p – 1920x1080

USB ports Ethernet ports Serial and other ports	2x USB 2.0 2x Fast Ethernet 1x RS232, 1x RS485 (switchable CAN) 4x Digital Out
Graphics port	HDMI
Internal storage External storage	4GB eMMC Micro SD slot
OS	Yocto
Construction Cooling Dimensions (HxWxD) Weight Mounting	Stainless steel Fanless 111 x 25 x76 mm Approx. 0,26 kg for mounting on 35mm mounting rail acc. to EN 60715
Operating temperature Storage temperature Relative humidity	0 °C to +55 °C (32 °F to 131 °F) -20 °C to +80 °C (-4 °F to 176 °F) 93% @ 40 °C, non-condensing
Power supply range	9..32 V DC

Table 1: Kontron K-Box K-BOX A-330 MX6 hardware specifications

Table 1 shows the hardware specifications of the K-Box A-330. The system has a dual-core ARM SoC with 512 MB of main memory (RAM) [3]. Since the 4GB internal eMMC might not be sufficient to host a large number of Docker images we have decided to use 32GB Micro SD cards to install the CARAMEL firmware load. This also allows us to install two images in parallel for the secure firmware update process we plan to implement for the anti-hacking device.

3.4.2 Software Installation

The default software installation is described in detail in [10].

The software build process is based on the Yocto firmware build system [11]. In order to implement pre-installed Docker containers and Docker container autostart as well as secure boot and secure firmware update, we have implemented additional recipes for the CARAMEL project that augment the Kontron base Yocto firmware build files. These will be described in a later deliverable in WP5.

The CARAMEL firmware image has to be installed to a Micro SD card to support the pre-installed Docker images (crypto and other containers) as well as the dual-boot setup for the secure and reliable firmware update process we plan to implement.

Our custom Yocto Linux firmware also supports all the features in sections 3.2.2 and 3.2.3 for the Coral Dev Board, ie. secure boot with signed boot loader and Linux Kernel, dual boot, signed and secure firmware updates, as well as signed Docker containers.

4 Layered approach to security

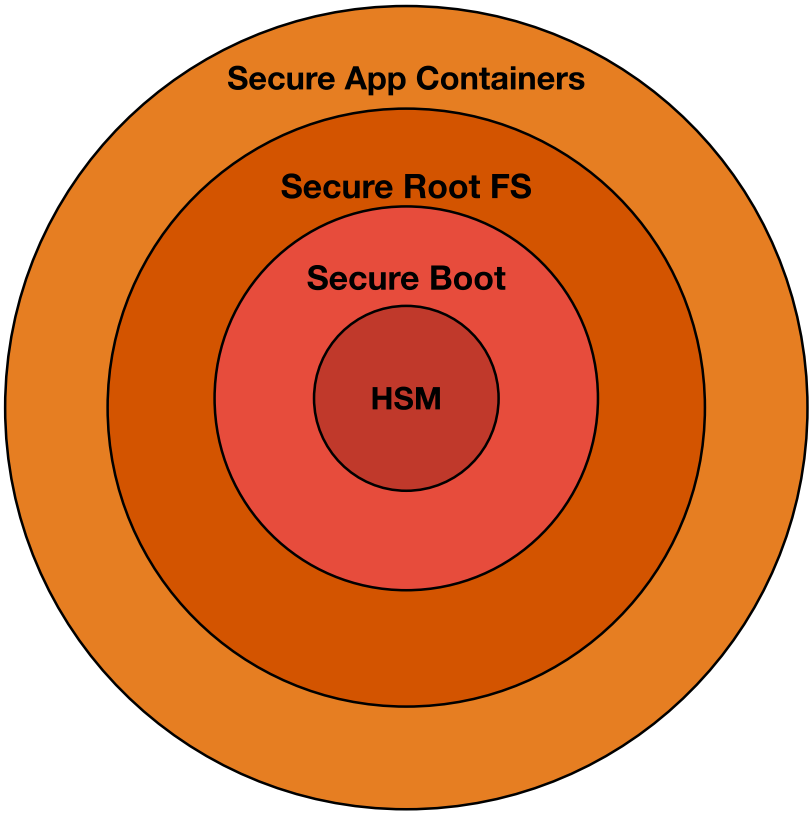

Figure 11: Layered approach to security

Since the anti-hacking is supposed to be a passive intrusion detection system for the vehicle that should improve the security of the overall system architecture, it should pose a security risk on its own. To this end, we have implemented a layered security approach, also known as defense-in-depth, that enforces secure operation of the anti-hacking device at several levels (see Figure 11).

- Integrated HSM (secure element) – described in deliverables D5.1 [2] and D5.4 [9]
- Secure boot – described in chapter 6
- Secure userland (root FS) – refer to chapter 5 for details
- Secure application containers – see chapter 8
- Secure firmware update – described in chapter 7

This comprehensive security-first approach also poses challenges that will be addressed by the design and implementation of the solution:

- Performance/resources of embedded anti-hacking device – embedded devices have less computing power and memory sizes than servers or desktop computers. Security solutions must take these resource constraints into account.
- Manage different hardware platforms for different purposes – there is no “one size fits all” solution for scenarios where the anti-hacking device could be deployed in an automotive context. To this end we have introduced different kinds of anti-hacking devices that target different performance, price, and capability points.
- Providing the abovementioned layers for all devices – this probably cannot be achieved actually for all devices, but in fact this is no problem: There will also be different security targets depending on the OEM’s security requirements and risk model. We will strive to showcase all the security features described in this document across the range of anti-hacking device platforms listed in chapter 3.

5 Deterministic Firmware Build Using Yocto

Yocto Linux [11] is not a specific Linux distribution, but a build system to create tailored Linux distributions for specific hardware platforms and specific usage scenarios.

We have chosen Yocto as a platform for the anti-hacking device firmware load due to the following considerations:

- Low overhead – only packages that are needed for the task at hand are included in the firmware image
- High security due to deterministic build process – no reliance on binary packages, all packages are fetched from the original sources, their integrity is checked, and then they are compiled in our controlled build environment. Additionally: since only a minimal number of packages is installed, the attack surface for hackers is minimized.
- Good hardware support from vendors for the Coral Dev board and Kontron K-Box versions of the anti-hacking device (BSP configurations).
- Docker technology support can be added easily.

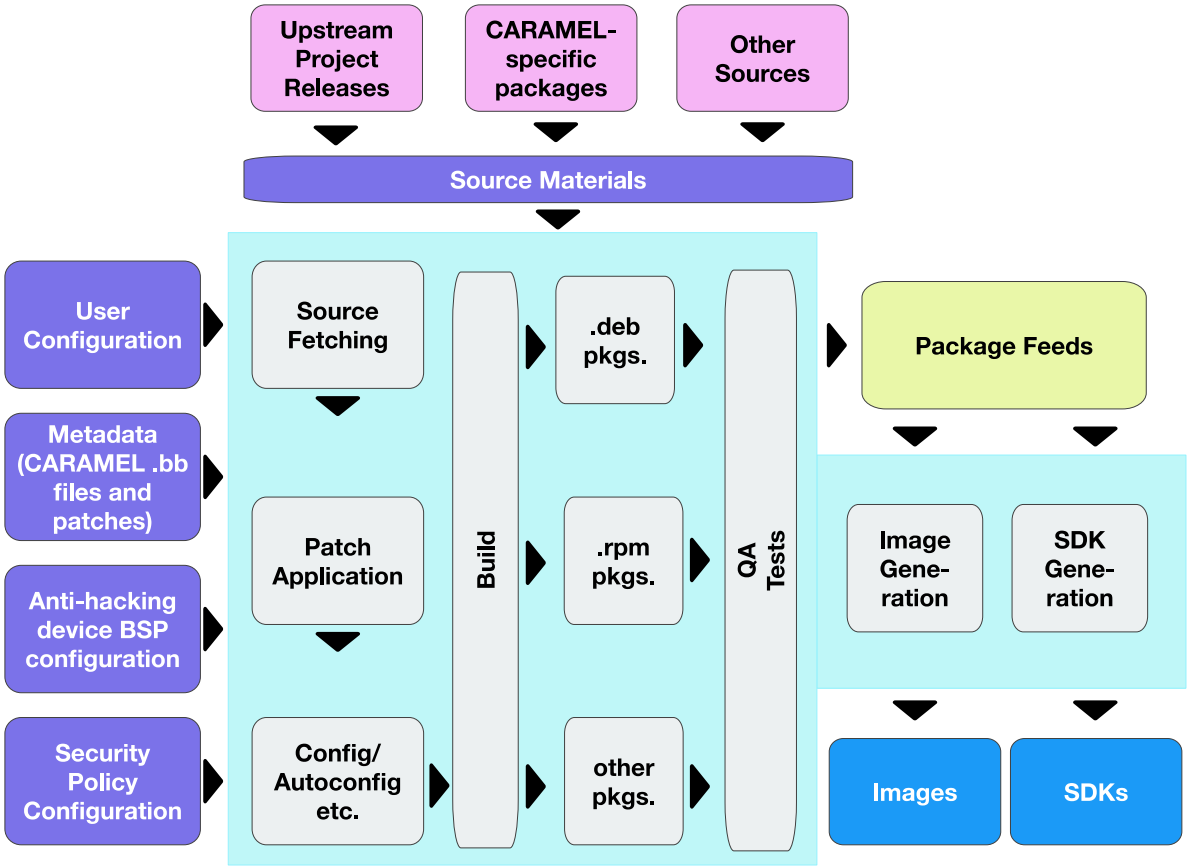

Figure 12: Overview of the Yocto build system

Figure 12 shows a high-level overview of the Yocto build system. The workflow for the creation of a new image is as follows:

- Developers specify architecture, policies, patches and configuration details.
- The build system fetches and downloads the source code from the specified location. The build system supports standard methods such as TAR files or source code repositories systems such as Git.
- Once source code is downloaded, the build system extracts the sources into a local work area where patches are applied and common steps for configuring and compiling the software are run.

- The build system then installs the software into a temporary staging area where the binary package format you select (DEB, RPM, or IPK) is used to roll up the software.
- Different QA and sanity checks run throughout entire build process.
- After the binaries are created, the build system generates a binary package feed that is used to create the final root file image.
- The build system generates the file system image and a customized Extensible SDK (eSDSK) for application development in parallel.

For the CARAMEL project we have added a layer to the Yocto build configuration that adds the following project-specific features:

- Support for signed bootloader and kernel
- Docker support
- Docker image for TCOS HSM integration
- Docker image for secure firmware upload

The details of these configurations will be described in the forthcoming deliverable D5.6 “CARAMEL Secure Hardware Platform”.

6 Secure Boot

ARM systems require the TrustZone IP extension for the ARM SoC in order to implement secure boot. Since an NXP-based SoC is used for our demonstrator, we use the NXP variant HAB (high-assurance boot) for the secure boot implementation.

One of the critical points during the lifetime of a secure system is at boot time. Many attackers attempt to break the software while the device is powered down, performing an attack that, for example, replaces the software image on the SD card with one that has been tampered with. If a system boots an image from flash, without first checking that is it authentic, the system is vulnerable.

The principle to apply here is the generation of a chain of trust for all software running on the device, established from a root of trust that cannot easily be tampered with. This is known as a secure boot sequence.

A TrustZone-enabled processor starts in the Secure world when it is powered on. This enables any sensitive security checks to run before the Normal world software has an opportunity to modify any aspect of the system.

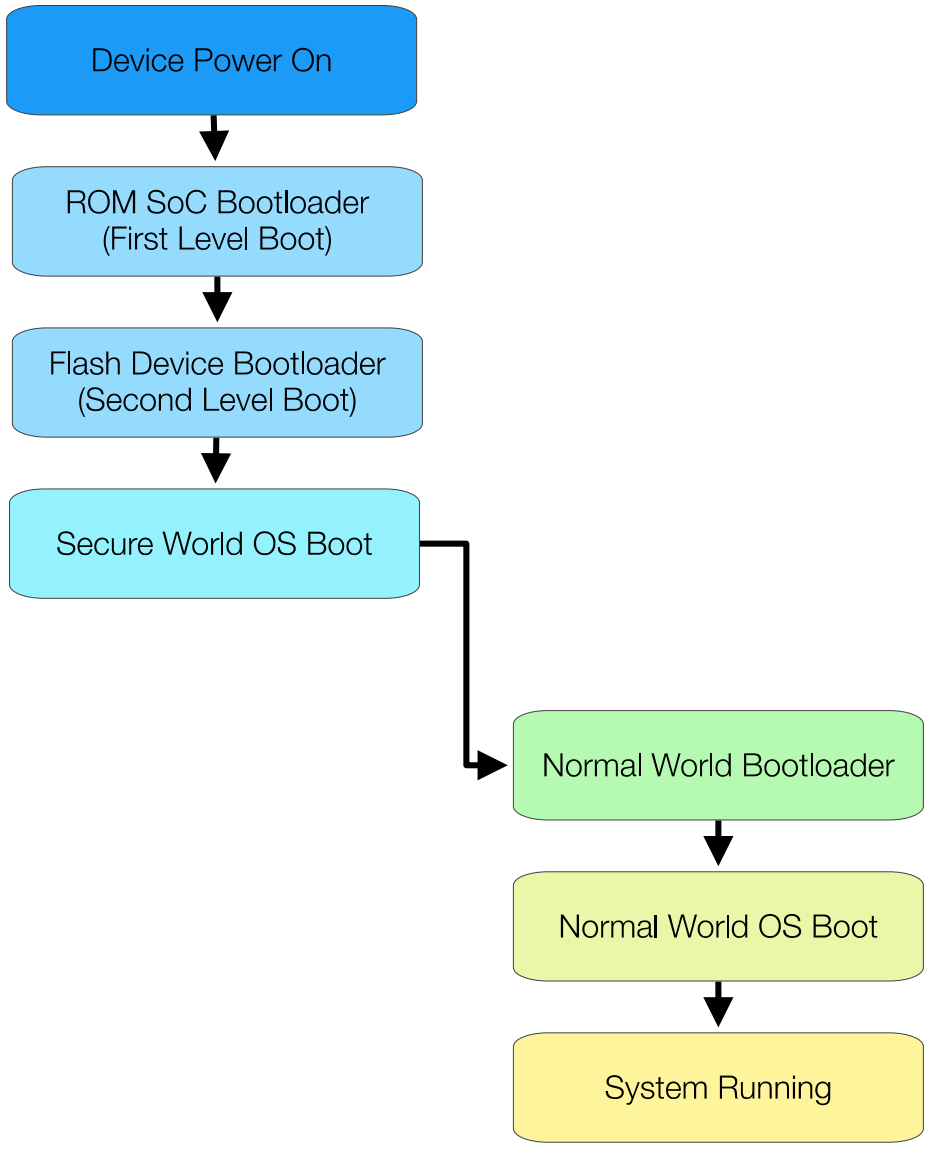

Figure 13. A typical boot sequence of a TrustZone-enabled processor

As shown in Figure 13, after power-on most SoC designs will start executing a ROM-based bootloader which is responsible for initializing critical peripherals such as memory controllers, before switching to the uBoot bootloader located in external non-volatile storage such as an SD card. The boot sequence will then progress through the Secure world operating environment initialization stages, before passing control to the Normal world bootloader. This will progress to starting the Normal world operating system based on Yocto Linux, at which point the system can be considered running.

A secure boot scheme adds cryptographic checks to each stage of the Secure world boot process. This process aims to assert the integrity of all of the Secure world software images that are executed, preventing any unauthorized or maliciously modified software from running.

The most logical cryptographic protocol to apply is one based on a public-key signature algorithm, such as RSA (Rivest, Shamir and Adleman public key system) in the case of NXP HAB. In these protocols a trusted vendor uses their Private Key (PrivKey) to generate a signature of the code that they want to deploy and pushes this to the device alongside the software binary. The device contains the Public Key (PubKey) of the vendor, which can be used to verify that the binary has not been modified and that it was provided by the trusted vendor in question.

The PubKey does not need to be kept confidential, but it does need to be stored within the device in a manner which means it cannot be replaced by a PubKey that belongs to an attacker. The NXP SoCs provide programmable fuses for storage of the PubKey.

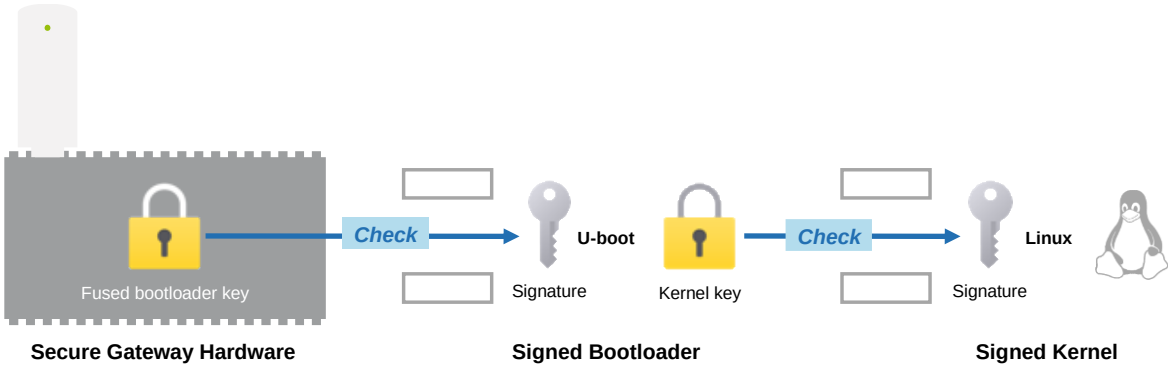

Figure 14: Secure boot for the anti-hacking device

The secure boot process for the anti-hacking device implements a chain of trust (see Figure 14). Starting with an implicitly trusted component, every other component can be authenticated before being executed. The ownership of the chain can change at each stage - a PubKey belonging to the device OEM might be used to authenticate the first bootloader, but the Secure world OS binary might include a secondary PubKey that is used to authenticate the applications that it loads, ie. the Linux kernel.

Unless a design can discount hardware stack attacks the foundations of the secure boot process, known as the root of trust, must be in the on-SoC ROM. The SoC ROM is the only component in the system that cannot be trivially modified or replaced by simple reprogramming attacks.

Storage of the PubKey for the root of trust can be problematic; embedding it in the on-SoC ROM implies that all devices use the same PubKey. This makes them vulnerable to class-break attacks if the PrivKey is stolen or successfully reverse-engineered. On-SoC One-Time-Programmable (OTP) hardware, such as poly-silicon fuses, can be used to store unique values in each SoC during device manufacture. This enables a number of different PubKey values to be stored in a single class of devices, reducing risk of class break attacks.

The simplest defense against stack attacks is to keep any Secure world resource execution located in on-SoC memory locations. If the code and data is never exposed outside of the SoC package it becomes significantly more difficult to snoop or modify data values; a physical attack on the SoC package is much harder than connecting a logic probe to a logic board track or an IC pin.

The secure boot code is generally responsible for loading code into the on-SoC memory, and it is critical to correctly order the authentication to avoid introducing a window of opportunity for an attacker. Assuming the running code and required cryptographic hashes are already in safe on-SoC memory, the binary or PubKey being verified should be copied to a secure location before being authenticated using cryptographic methods.

In case of the NXP i.MX SoCs (eg. as used on the Kontron K-Box or Coral Dev Board) the PubKey is stored in fuses on the board during configuration in our lab. Secure boot is not finally enforced, however, in order to allow re-use of the boards after the CARAMEL project.

7 Secure firmware update

Even though we provide with our Secure Docker container approach a flexible way to update the anti-hacking to newer threat detection mechanisms it might be necessary to do a remote update of the whole firmware at some time, eg. to patch flaws in the base Yocto Linux OS layer. It is important to consider that firmware update security is a process – not only a technology. This is case because cryptographic technology must be properly embedded into a process to be effective. The following topics should be considered when implementing remote update capabilities into an embedded design:

- Hardware Support: Many designs today already contain some type of flash. It is important to consider the intricate details when working with reprogrammable flash. Without proper planning, design, and coordinated hardware and software implementation, performing a remote update can unfortunately render a product non-functional.
- Operating System Support: When implementing remote updates, from an embedded OS perspective, what becomes part of an update? Does the new image contain device drivers, just a new or modified task, or must it include a whole new OS, boot code etc. Since we are using Yocto Linux for the K-Box and the Coral Dev Board, we will opt to update the whole Linux system including applications and Docker support.
- Operational considerations. Embedded systems are found in many diverse environments. If enabling remote updates, operational considerations can often be a high priority. Typically, reprogramming in-circuit flash requires more power than standard operation. A wireless sensor network that harvests energy from the environment will likely have different operational considerations or operation power budget than an enterprise network switch or router or an embedded device in a vehicle.

Figure 15 illustrates a generic high-level architecture for implementing secure remote firmware updates.

Figure 15. High-level remote update architecture

There are three main elements that comprise the remote update architecture:

- Embedded device: this is the unit to receive the update, in our case the anti-hacking device.
- Communication path: The embedded device must have a method for receiving an update. This is dependent on the overall design. In some cases, this is an Ethernet connection, while in others it could be a backplane or a combination of a USB memory stick and sneaker-net to deliver the update.
- Firmware repository: this entity maintains a firmware update library for all devices, versions, configurations, etc.

The following summarizes a set of generic requirements for implementing secure remote firmware updates:

- Standards: the end solution should leverage available communication and security standards.
- Authentication: A downloaded image should come from trusted sources.
- Validation: Validate that a downloaded image is complete.
- Versatile: The end solution must support a wide range of deployed network topologies.
- Scalable: The end solution must be able to support potentially hundreds or thousands of

devices, versions, product lines, etc. all requesting upgrades at one time.
- If the embedded device fails to authenticate or validate a downloaded image, an appropriate error is set, and overall operation is not affected.
- Only original software must be accepted by the embedded system – specifically software must not be successfully downloaded to the embedded system that alters its defined behavior.
- Only authenticated parties may alter data (i.e., parameters) stored in the embedded system.

Figure 16 shows the iterative development model for secure firmware updates.

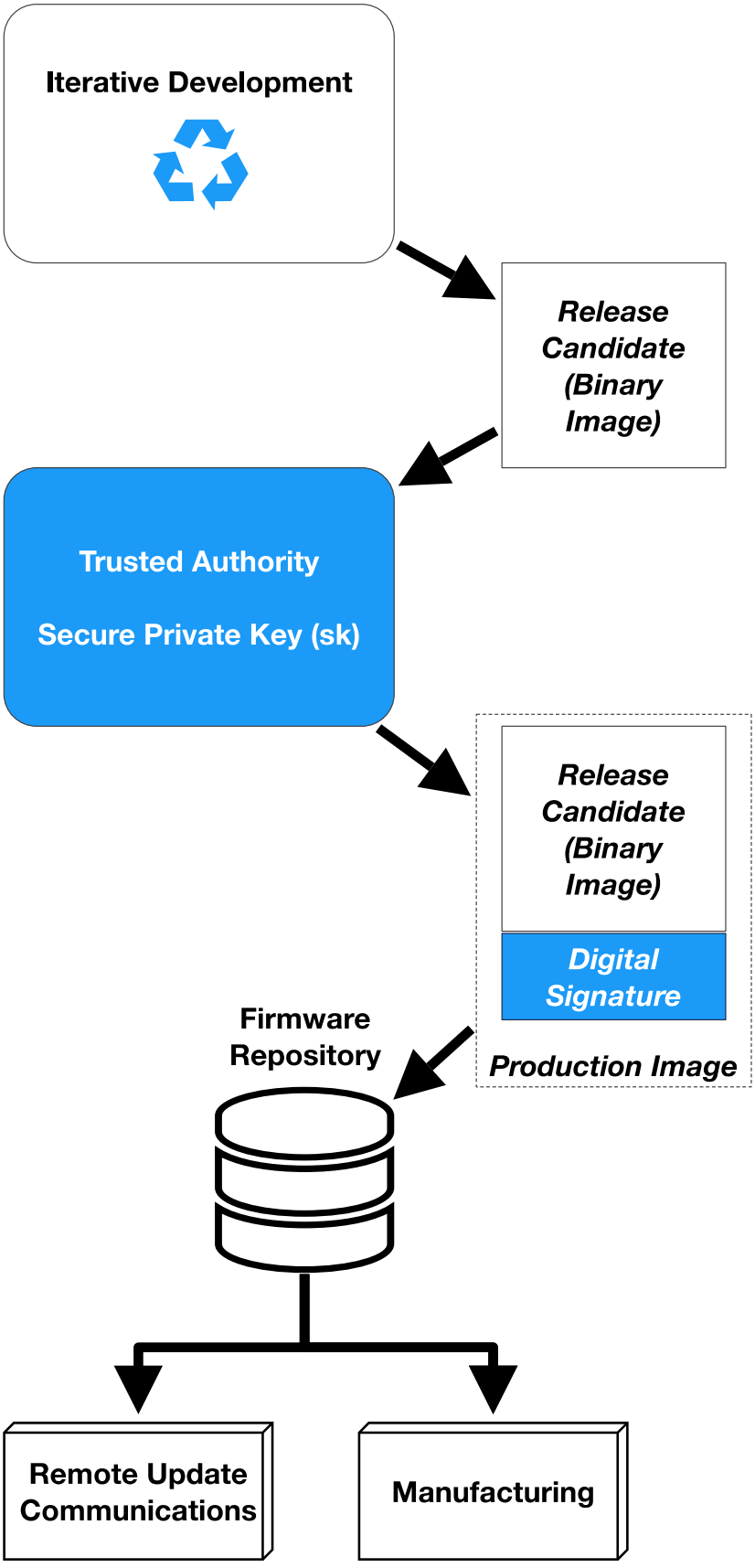

Figure 16. Development model supporting digital signatures

First, a digital signature is calculated for the new firmware load as shown in the Figure 16. The private key sk must be kept secure and is only known to the Trusted Authority. In many cases this is a secure machine that is not connected to a network. Additionally, there are typically well-established procedures for access and use of the secure machine to generate a digital signature.

The iterative process of developing and debugging the Yocto-based software eventually yields a final release candidate that is a binary image. As part of established security processes in place, the binary image is turned over to a Trusted Authority to create a unique digital signature based on the secure private key sk. The digital signature is then appended to the binary image. The firmware repository now stores the production image with the digital signature, typically in some form of database. Likewise, the Trusted authority public key is installed in the embedded device during manufacturing or stored within the program code.

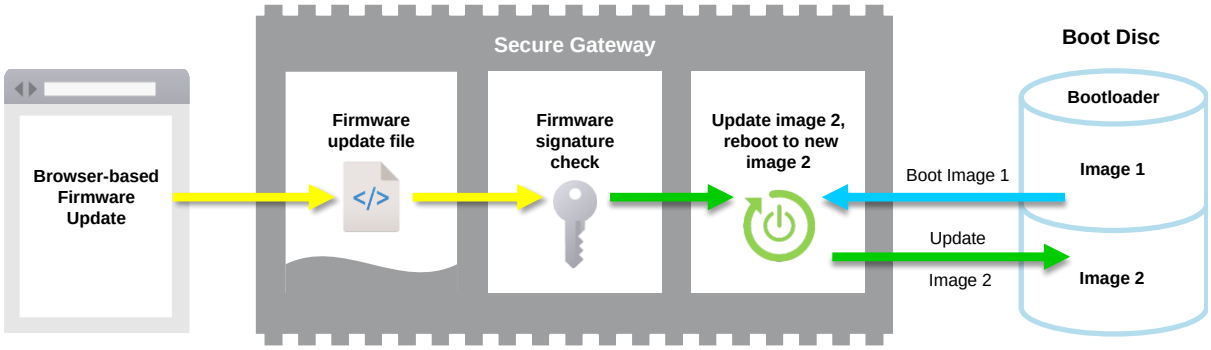

Figure 17: Anti-hacking device firmware update process

Figure 17 shows how the anti-hacking firmware update process works in detail:

1. The anti-hacking device boots from image 1 on the boot medium
2. The firmware update process is initiated via a browser interface
3. The new, signed firmware file is downloaded to the anti-hacking device
4. The firmware signature is checked. Only if the signature is valid the update process proceeds to the next step.
5. The firmware image is unpacked and written to the partition for image 2 on the boot medium.
6. A boot loader flag is set so that on next boot the boot loader script tries to boot from image 2.
7. If all is well, the device boots now into boot image 2.
8. If the boot of image 2 fails, the device reboots.
9. The boot loader checks the flags and boots again to image 1 in this case.

This process ensures that a bad flash cannot negatively impact the operation of the anti-hacking device, increasing its availability.

Figure 18 shows the format of the signed firmware files:

- SHA256 signed digest of the firmware file, followed by a
- ROOT image file containing the root file system of the new firmware that can be directly written to a partition

Figure 18: Signed firmware format

8 Secure Docker containers

One of the requirements for the anti-hacking device is the ability to quickly deploy or update new attack or intrusion detection algorithms in order to mitigate new kind of attacks quickly and in an agile manner. To this end, we specified the use of Docker container technology for the provisioning of the actual threat detection mechanisms to the anti-hacking device.

Such a quick update mechanism can pose a potential security threat, however, since the isolation offered by Docker technology could be circumvented by bad actors (eg. by exploiting implementation flaws or design problems). Therefore, it is of utmost importance that only secure Docker images from trusted sources are accepted by the Docker installation on the anti-hacking device.

The following sections detail our implementation choices to offer a Docker provisioning platform that is both secure and scalable.

8.1 Introduction to Docker Content Trust (DCT)

Docker Content Trust (DCT) is an industry standard technology pioneered by Docker Inc. (see [12] for an in-depth description). The basic idea behind DCT is that specific tags added to Docker images are signed using a local key on the development machine. The images together with their signed tags can then be pushed to a DCT-enabled Docker registry service (eg. the Harbor registry service described in section 8.2). DCT allows publishers of images to use digital signatures, effectively allowing users pulling their images to verify:

- That the content of the image has not been tampered with.
- The identity of the publisher.

Because DCT is based on public-key cryptography, anyone can create a free public/private keypair and start pushing signed images. Publishers need to sign every single individual image before being uploaded and signatures refer to specific tags of their images

On the consumer side (user of signed Docker images) the images with their tags are downloaded and can be verified using appropriate Docker commands or, alternatively, the Docker installation on the consumer side (in our case the anti-hacking device) can be configured to only accept signed images.

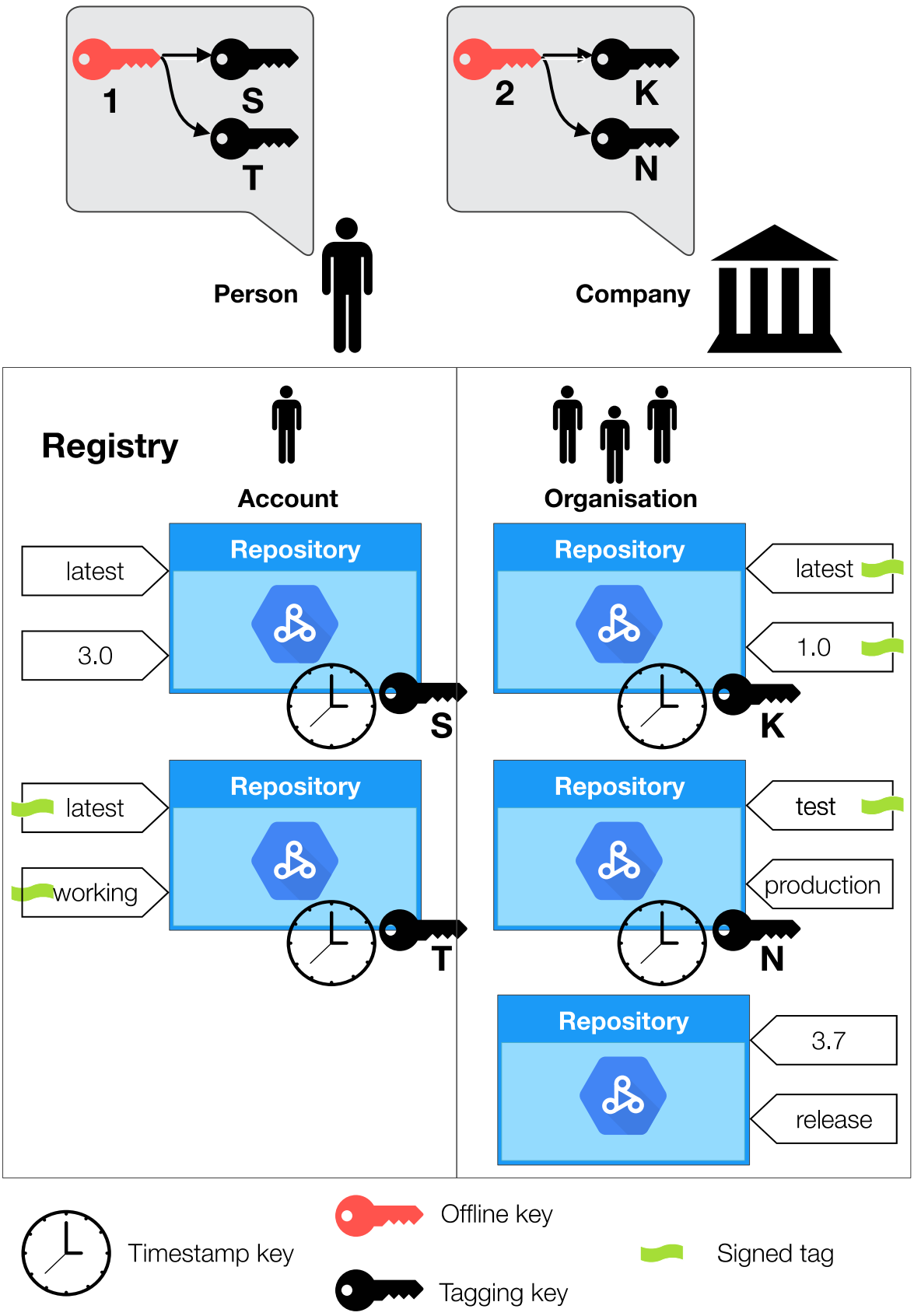

Figure 19: Docker content trust

Figure 19 shows a sample configuration of a DCT deployment suitable for use in the CARAMEL project: Persons or organizations (partners in the project) hold their local signing keys (shown as red lock symbols). Some of the tags that are meant for deployment to the anti-hacking device are signed (with the green wave symbol attached).

It is important to note that for development purposes the whole system can be used without any restrictions or changes to the familiar Docker workflow: Only when security needs to be enabled the developer has to generate a signing key und use specific commands to sign tags to be deployed securely on the anti-hacking device.

The anti-hacking device itself can be configured to only accept signed Docker images from well-known sources. Since this configuration can be done via an environment variable, it can easily be switched off for development and test purposes by the CARAMEL partners.

The actual details of the necessary commands to enable DCT on MacOS, Linux, and Windows clients will be published on the project web site and also available in the forthcoming deliverable D5.6 “CARAMEL Secure Hardware Platform”.

8.2 Harbor Registry Service

The Registry and Notary software offered by Docker Inc. are the reference implementation of the Docker Content Trust technology. These implementations do not allow fine-grained access control and are only configurable via Docker environment variables and configuration files.

Another, fully compliant implementation of the DCT and registry specifications is the Harbor registry service developed by the project with the same name. [13] The Harbor registry offers the following advantages over the simple reference implementations by Docker Inc.:

- User and group management
- Different repositories with adjustable security policies
- Fine-grained role-based access control to these repositories
- Full support of Docker content trust
- Full implementation of Docker Registry specification
- Kubernetes support for scalable agile Docker deployments

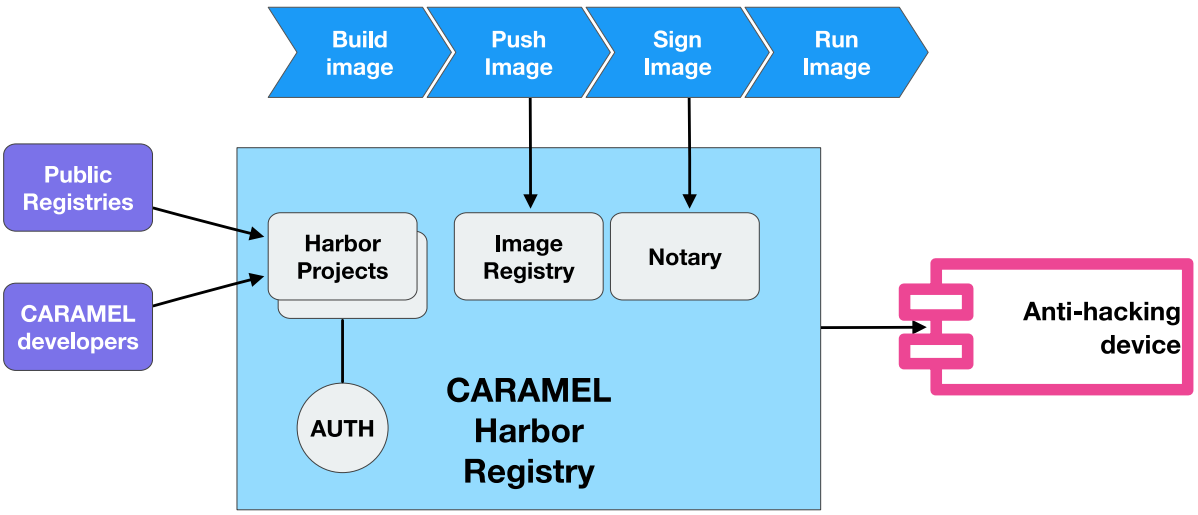

Figure 20: The Harbor registry service

Figure 20 shows the planned Harbor deployment in our lab:

- In the user authentication and authorization section (UAA) we plan to configure all partners that need access to the repository.

- There will be a main CARAMEL harbor project as well as some test projects to which the partner's development teams will have read/write access.
- The Notary server implements the DCT processes on the registry side.
- On the right of the diagram, you can see the clients (anti-hacking devices) that pull the signed images from the registry server.

9 Conclusions and Next Steps

In this document we have described in detail the concept of a passive intrusion detection system – the anti-hacking device – implemented as an electronic control unit with integrated machine learning capability in the vehicle to detect threats the connected vehicle in an agile manner.

Since the anti-hacking device (in-car IDS) shall improve the security of the vehicle and not create a new attack surface on its own the concepts of multi-layered security or defence-in-depth are applied to the anti-hacking device in an innovative way, integrating an embedded HSM, secure boot, secure firmware update, along with machine learning-based threat detection capability in a small form factor, low-cost device.

To this end, we have introduced the security features of the anti-hacking device and relates these efforts to other tasks in WP5.

We have also specified the three different secure hardware platforms that have been chosen as the base for the anti-hacking device implementations done in the CARAMEL project. The three different hardware platforms address different security, cost, performance, and form factor requirements.

Taking the high-level concept of layered security or defense-in-depth as a starting point we have specified how the software load for two of the hardware platforms is created in order to enhance security and availability. Then we have given details of the secure boot mechanism and the secure firmware update process.

Finally we have covered an important aspect of the anti-hacking device software architecture: Docker container technology provides the agility required by an intrusion detection system needed for a swift reaction to new threats and attack surfaces. We have described how Docker containers can be deployed quickly and securely using Docker Content Trust technology.

As a next step we will refine the already existing implementations of these concepts on all three secure hardware platforms and make them available for CARAMEL project partners as part of the work in WP6. Additionally, demonstrations will be prepared in order showcase the different security technologies mentioned in this document. This effort will be documented in the upcoming deliverable D5.6 “CARAMEL Secure Hardware Platform” of WP5.

References

[1] European Commission – Grant Agreement Number 833611 - CARAMEL. 2019.

[2] CARAMEL – D5.1: Hardware Security Module Specifications. 2020.

[3] Kontron: KBox A-330-MX6 datasheet. Kontron, 2020.

[4] CARAMEL – D2.4: System Specification and Architecture. 2020.

[5] Coral: Update or flash the board. https://coral.ai/docs/dev-board/reflash/

[6] NVidia Jetson Developer Guide. https://docs.nvidia.com/jetson/l4t/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide/introduction.html#.

[7] Kontron: KBox A-330-MX6. https://www.kontron.com/products/systems/embedded-box-pc/kbox-a-series/kbox-a-330-mx6.html.

[8] Kontron: Description of the iMXceet Solo / Dual S Demoboard. https://docs.kontron-electronics.de/yocto-ktn/build-ktn-rocko/board-imxceet-solo-dual-s/.

[9] CARAMEL – D5.4: CARAMEL IDS/IPS Security Module. 2021.

[10] Kontron Electronics: Quickstart - Kontron Electronics Docs - NXP i.MX6 (Rocko). https://docs.kontron-electronics.de/yocto-ktn/build-ktn-rocko/quickstart/.

[11] Yocto Project: The Yocto Project. https://www.yoctoproject.org.

[12] Docker: Content trust in Docker. https://docs.docker.com/engine/security/trust/.

[13] Harbor: Harbor. https://goharbor.io.