D2.1

Report about trusted life cycle design methodology for OSH

Project number	101070008
Project acronym	ORSHIN
Project title	Open source ReSilient Hardware and software for Internet of thiNgs
Start date of the project	1st October, 2022
Duration	36 months
Call	HORIZON-CL3-2021-CS-01

Deliverable type	Report
Deliverable reference number	CL3-2021-CS-01/ D2.1/ 1.0
Work package contributing to the deliverable	WP2
Due date	JUN 2023 – M09
Actual submission date	28th June 2023

Responsible organisation	SEC
Editor	Stefano Cristalli
Dissemination level	PU
Revision	1.0

Abstract	This document lays the foundation for defining and modelling the concept of Trusted Life Cycle (TLC) for secure, open source hardware components. Specifically, a starting set of requirements for the TLC is provided. It also contains a novel definition of open source hardware, for evaluating it qualitatively and also quantitatively. Another contribution is represented by the evaluation of current methodologies for component and vulnerability tracking, and a proposition for a modern approach aimed at improving the current situation.
Keywords	Trusted Life Cycle, Secure development, Open source, Hardware, Process requirements, Secure processes, Secure procedures, Component tracking, Bill Of Materials, Vulnerability tracking

Funded by the European Union under grant agreement no. 101070008. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

Editor

Stefano Cristalli (SEC)

Contributors (ordered according to beneficiary numbers)

Volodymyr Bezsmertnyi (NXP)

Guido Bertoni, Filippo Melzani, Massimo Ratti, Stefano Cristalli, Maria Chiara Molteni, Marta Fornasier, Lorenzo Nava, Arianna Gringiani (SEC)

Clarisse Ginet, Olivier Thomas (TXP)

Jan Pleskac (TRPC)

Reviewers

Barbara Gaggl, Michael Käfinger (TEC)

Daniele Antonioli (ECM)

Jan Pleskac (TRPC)

Disclaimer

The information in this document is provided “as is”, and no guarantee or warranty is given that the information is fit for any particular purpose. The content of this document reflects only the author`s view – the European Commission is not responsible for any use that may be made of the information it contains. The users use the information at their sole risk and liability.

Executive Summary

This document lays the foundation for defining and modelling the concept of **Trusted Life Cycle (TLC)** for secure, open source hardware components.

The first challenge that we encountered was providing a definition of *open source hardware*. Although there is extensive literature for both the worlds of “open source” and “hardware” (e.g., [Kelty 2016] [Free Software Foundation 2017], [Baker 2011]), their intersection brings new valuable context, with associated initial challenges, specific concepts, and open problems. There is enough separation within these concepts from existing literature to create a new domain of knowledge. One of such challenges, for example, is agreeing on a clear definition of what *open source hardware* means. To the best of our knowledge, this is the first work that tries to define a systematic approach for such a definition.

After a preliminary reasoning on the semantics that a definition of open source hardware should capture, and comparing it with previous attempts ([OSHWA 2023]), we present a new definition that permits the evaluation of open source hardware, both qualitatively and, for the first time, also quantitatively.

Our model distinguishes between different types of hardware *views*, based on the abstraction level of the development. Our idea is that, although grouped under the umbrella term “hardware”, different developments may have different properties, and therefore deserve separate categorizations (e.g., a technology library vs. a PCB).

With a sound definition of open source hardware, we then proceed listing the requirements for the Trusted Life Cycle. These are “process” requirements about the phases which compose the development of secure open source hardware components.

For this definition task, we cannot ignore the vast literature that exists on the topic of Secure Development Life Cycles (SDLCs). Although focused more on the software domain, and usually lacking any reference to open source topics, existing models for SDLCs contain valuable and reusable knowledge also for ORSHIN’s TLC.

Most of this previous knowledge on SDLCs is consolidated in IT and industry standards; therefore, we present the main relevant works, and try to summarise the commonalities.

Afterwards, starting from a baseline work from ENISA regarding good practices for security in IoT, we draft the requirements for the ORSHIN’s TLC. In particular, we adapt the existing requirements to the new context of open source hardware, and we also draft new requirements that are specific for such domain. We align the definition of TLC requirements with the framework previously defined for the definition of open source hardware, in order to keep a coherent approach that has the possibility of adapting to our previously-defined hardware views.

We also consider a novel approach regarding the definition of process requirements, that is the possibility of adapting the development process according to a threat model. This allows the adaptation of a common set of requirements to specific industry use cases, possibly with dedicated personalization and extensions. Our work is harmonised with ORSHIN's Task 2.2. The content of this deliverable has been employed as a reference for developing the AttackDefense Framework (ADF) proposed by Deliverable 2.2. Specifically, the TLC was used as a reference life cycle and evaluated within the ADF case studies. Moreover, our TLC methodology is the concept at the base of the work produced in WP3, WP4, and WP5.

Finally, after having established a methodology for defining the requirements of the ORSHIN TLC, we tackle important themes related to the development of open source hardware. Specifically, we work on the definition of Hardware Bills Of Materials (HBOMs), as the focal point that has to function properly in order to allow efficient categorization of both hardware components, and of associated information that is relevant for security, such as known vulnerabilities that affect hardware components. We review existing approaches for the definition of Bills Of Materials (BOMs) in general, including relevant work done for Software Bills Of Materials (SBOMs). We challenge the Common Platform Enumeration (CPE), the industry standard for categorising hardware and software components, highlighting some problems that don’t make it suitable, in our opinion, to the context of

secure open source hardware. We make a proposition for a more flexible system, starting from an extension of the Open Worldwide Application Security Project (OWASP) CycloneDX format.

We conclude the document by summarising our contribution, highlighting open problems and listing promising directions for future work.

Table of Content

List of Figures

List of Tables

Chapter 1 Introduction

In this document we report part of the research of WP2, particularly focusing on the work related to the ORSHIN Trusted Life Cycle methodology (Task 2.1). In this project we discuss electronic hardware, and then all the reasoning and examples in this document are focused on that area.

The concept of Secure Development Life Cycle (SDLC) is consolidated and extensively applied in the IT world, with recent applications also touching the Internet of Things (IoT) and Industrial Automation Control Systems (IACS) contexts. However, even the most recent embedded-oriented SDLC variations are heavily lacking when considering the topics of hardware development, and open source.

These topics, central to the ORSHIN project, guide the definition of the Trusted Life Cycle (TLC). TLC is a methodology which aims at providing developers and maintainers of the open source community with practical help for exploring and expanding the cybersecurity dimension of their projects. This methodology focuses mainly on the embedded/IoT/IIoT projects which make partial or total use of open source hardware.

This document is divided into the chapters described in the following Sections.

1.1 Definition of Open source Hardware

Chapter 2 is dedicated to the definition of "open source hardware". Despite being extensively used, this terminology does not yet have a universally accepted definition, so we tackle the challenge of formalising a possible one.

We explore the context of hardware, differentiating developments into *views* based on their level of abstraction.

First, we study properties of hardware developments that influence their *effective* open source status.

Second, we provide qualitative definitions for different levels of open source hardware based on such properties, and we also study the application of properties to the different hardware views. Then, we define a score to compare open source hardware products, aiming at capturing relevant detail while maintaining simplicity of use.

We start by defining our scoring system for components based on the evaluation of single properties, and afterwards we provide a way for calculating the composite score of a device, taking into account the score of its subcomponents.

Finally, we evaluate our novel approach on real-world examples of both open source and non-open source hardware, such as Raspberry Pi4, USB Armory and more.

1.2 Definition of the Trusted Life Cycle

Chapter 3 provides a definition of the Trusted Life Cycle phases, and its requirements, which are the central contributions of this deliverable.

We start by reviewing previous work, which is represented by the different frameworks in the literature for the definition and application of Secure Development Life Cycles (SDLCs). In particular, we focused on sets of practices and requirements that allow developers and system integrators to build secure products and systems in a reliable and repeatable fashion. The beginning of the work on such methodologies dates back to the early 2000s, and it has a good level of maturity in its primary field of application, i.e. software development with IT infrastructure; on the other hand, newer context such as Internet of Things and Industrial Automation and Control Systems have only seen recent effort for porting the practices of SDLCs, and have therefore a lower level of maturity. In particular, hardware development is typically faced by SDLC at a high level of abstraction, without

considering the peculiarities of the hardware industry (e.g., differences while building an IT and IoT processors).
Similarly, open source is seldom mentioned by SDLC methodologies, if at all. Concerns in this direction primarily focus on the security of 3rd-party software libraries, and not much more.

Within the ORSHIN project, both hardware development and open source are central topics, and we propose a methodology that adequately addresses the definition of security requirements for them. We review the most relevant international standards and guidance documents which provide process-oriented security requirements, showing different perspectives to the definition of requirements and best practices.

We select a source that in our opinion represents the best starting point for defining a development life cycle oriented to hardware and open source, due to its starting focus on the IoT world. We perform a selection of requirements, filtering out ones that are not suitable for the properties we want for the ORSHIN Trusted Life Cycle, then we adapt their content to fully adhere to the ORSHIN context.
We draft new requirements specifically for the topics of hardware design and open source, then we provide our finalised proposal with the full list of requirements for the ORSHIN Trusted Life Cycle. Finally, we discuss methodology for the application of requirements.

1.3 Component and Vulnerability Tracking

Chapter 4 faces the important topic of component and vulnerability tracking, which is a fundamental part of the *maintenance* phase of the ORSHIN Trusted Life Cycle.

In order to perform an effective monitoring of the security of developments, it is essential that their composition is known in detail.

This necessity is met by compiling the Bill Of Materials (BOM) for a component, be it software (Software Bill Of Materials - SBOM), hardware (Hardware Bill Of Materials - HBOM) or a combination of the two.

The associated requirement for effective vulnerability management is the ability to gain knowledge about recent vulnerabilities that get published in global databases about products that one wishes to monitor, or about any of their subcomponents.

For this reason, there is a rich vulnerability tracking ecosystem, including:

- Identified instances of vulnerabilities (with the Common Vulnerabilities and Exposures framework - CVE);
- Common weakness that affect various aspects of the design and implementation of systems (with the Common Weakness Enumeration framework - CWE);
- Attack patterns that allow attackers to discover vulnerabilities starting from common weaknesses (with the Common Attack Pattern Enumeration and Classification framework - CAPEC).

We review the state-of-the-art for both this vulnerability-tracking ecosystem, and for its component-tracking framework counterpart, that is Common Platform Enumeration - CPE, which is currently predominantly used for referring to components and products that have been associated with some vulnerability.
We explain how the current limitations of the above systems fail at providing a lightweight and open approach for everyone to use to compile efficient BOMs with rich public information about components, and how the situation could significantly improve by leveraging a related framework from OWASP called CycloneDX.

After declaring the properties that we envision for a modern component and vulnerability tracking system, we see with a practical example how CycloneDX allows to meet most of them. We provide an extension to the format allowing modelling additional details that are relevant for the context of ORSHIN as a consequence of our research (for example, the score for open source hardware). Then, we outline the next steps that would be necessary for global adoption of our framework for satisfying the remaining requirements, which are not format-dependent. For instance, we state that a global public database would be necessary for actual adoption of a new component-tracking system, and that participation from the community would be required at multiple levels, from manufacturers to individual researchers and enthusiasts.

1.4 Conclusion and Next Steps

We review the conclusions of our research in Chapter 5, and outline the next steps to continue research in promising directions.

Chapter 2 Definition of Open source Hardware

2.1 Overview

The object of our work is the so-called *open source hardware*. The initial question we need to answer is: which are the guidelines to define when a hardware device is open source. For example, how can we define an open source product using a closed-design microcontroller or an open source HDL distributed with proprietary toolchain? Is it open source or not?

A starting point for giving a complete definition is the description given by the Open Source Hardware Association (OSHWA) website [OSHWA 2023]. In the introduction it is stated the following:

"Open Source Hardware (OSHW) is a term for tangible artifacts — machines, devices, or other physical things — whose design has been released to the public in such a way that anyone can make, modify, distribute, and use those things."

OSHWA considers a hardware device to be open source if it complies with the following criteria:

- The *documentation* must be provided with the device, and it must be in an open format. In particular, the documentation must include design files, and must allow their modification and distribution.
- The *software* necessary for the hardware under investigation has to be released under an open source licence. It is also desirable to have well documented interfaces, such that it will be easy to write an open source software that allows the device to operate properly and fulfil its functions.
- *Modifications* and *derived works* must be allowed, and they have to be distributed under the same term as the licence of the original work.
- *The redistribution* of the hardware needs to be for free, and it should be possible to sell or give away the project documentation.
- *The Licence* must not be specific to a product and it must *not restrict* other hardware or software. If a part of the product is used or distributed, it has to follow the term of the licence granted for the original work.
- *The Licence* must *not discriminate* against persons or groups, and it must *not restrict* anyone from making use of the work in a specific field of endeavour.

In our opinion, this description and other state-of-the-art notions and definitions (see Section 2.2 - State-of-the-art) in this context are not sufficient to provide a thorough vision. We believe that for the aim of the ORSHIN project and for practical applicability in industry, they can be a good starting point, but they also need to be extended. In this direction, we invested our first efforts in trying to provide an exhaustive and deep definition that we present in the following Sections.

To reach the goal of a complete definition, we propose a categorization of hardware components in sets that we call views (Section 2.3 Views). Once placed in a view, each component is evaluated according to different properties (Section 2.4 - Properties) which express its open sourceness under different perspectives. For any hardware, this evaluation produces a vector of scores (Section 2.5 - How to Score Hardware Open sourceness) which is then combined to reveal how much the hardware component is open source. Some examples of these evaluations are in Section 2.6 - How to Apply our Open source Definition: Case Studies, and the evaluation computed considering also the subcomponents is presented in Section 2.7 - How to Score Hardware with Subcomponents.

2.2 State-of-the-art

The definition of open source was born in the software context. The term open source does not simply mean that the source code is freely accessible, but also that it should follow some criteria [Open Source Initiative][Debian Social Contract 2023]. The highlighted criteria are very similar to those reported in the OSHWA website; this is because OSWHA selected the criteria for defining open source hardware by retracing and elaborating on the steps established in previous years for software.

On the Debian Organization website [Available: https://opensource.org/osd/.], the Debian Free Software Guidelines (DFSG) are listed in the following points.

1. *Free Redistribution*. The licence of a Debian component may not restrict any party from selling or giving away the software. The licence may not require a royalty or other fee for such sale.
2. *Source Code*. The program must include the source code, and the distribution of the source code as well as the compiled form must be allowed.
3. *Derived Works*. The licence must allow modifications and derived works, at which are applied the same terms as the original software.
4. *Integrity of The Author's Source Code*. This point is a compromise in the context of modification of the files. Indeed, the licence may require derived works to carry a different name or version number from the original software, in such a way to preserve the integrity of the original source code.
5. *No Discrimination Against Persons or Groups*.
6. *No Discrimination Against Fields of Endeavour*.
7. *Distribution of Licence*. The rights attached to the program must apply to all to whom the program is redistributed.
8. *Licence Must Not Be Specific to Debian*. The rights attached to the program must not depend on the fact that the program is part of a Debian system.
9. *Licence Must Not Contaminate Other Software*. The licence must not place restrictions on other software that is distributed along with the licensed software.

One of the first software open source projects is the GNU Operating System, supported by the Free Software Foundation [GNU 2021]. It was launched by Richard Stallman in 1983, with the goal of offering a Unix-compatible system that would provide completely free software. GNU packages include user-oriented applications, utilities, tools, libraries, as well as games, namely all the programs that an operating system can offer to the users.

After a first spread in the context of software, the definition of open source took hold also for hardware. In this scenario, the Open Source Hardware Association [OSHWA 2023] was born, with the aim of fostering technological knowledge and encouraging research that is accessible, collaborative and respectful of user freedom. OSHWA organises the annual Open Hardware Summit and maintains the Open Source Hardware certification [OSHWA CERT], which allows the community to quickly identify and represent hardware that complies with the community definition of open source hardware.

2.2.1 *Licence for an Open Source Project*

In choosing a licence, one should first decide whether or not he wants to require people to keep the derivatives of his designs open source. If so, he should use a *copyleft licence*; if not, he could choose a *permissive licence* [OSHWA 2023]. *Copyleft* (or viral) *licences* require derivatives to be licensed under the same terms; on the other hand, *permissive licences* allow other people to make modifications without needing to release the derivative product as open source hardware. A designer of open source software/hardware must allow modification and commercial re-use of a design, so he should not use licences with a no-derivatives or non-commercial clause.

Some examples of licences used for open source projects are listed below.

- CERN OSH [CERN OSH 2023]: the current version of this licence is version 2, that comes with three variants, CERN-OHL-S (strongly reciprocal), CERN-OHL-W (weakly copyleft) and CERN-OHL-P (permissive). For a deeper understanding, see the document [CERN OHL 2020].
- MIT [MIT LICENCE].
- Apache2.0 [APACHE LICENCE 2023].
- GNU General Public Licence [GPL 2022].
- Creative Commons Licences [CCLICENCES].

2.3 Views

In this Section, we propose a categorization of hardware based on *views*.

In the ORSHIN project we discuss electronic hardware, and then all the reasoning and examples in this document are focused on that area. When talking about hardware, we can refer to many different layers, from the technology libraries used for the synthesis of circuits, to the final device, which can include multiple chips. In particular, each hardware component has a specific purpose and contributes to the overall functioning of the device.

The first step toward our definition of open source hardware is the description of these different hardware layers, which we call *views* (Figure 1). Views define different types of hardware components and we identified four of them; the most basic one is the lowest view (V0), i.e. the *technology library*, which is necessary for the synthesis of any hardware component. Starting from that, we have identified other three views, until reaching the most complex hardware level, i.e., the complete *device* (V3).

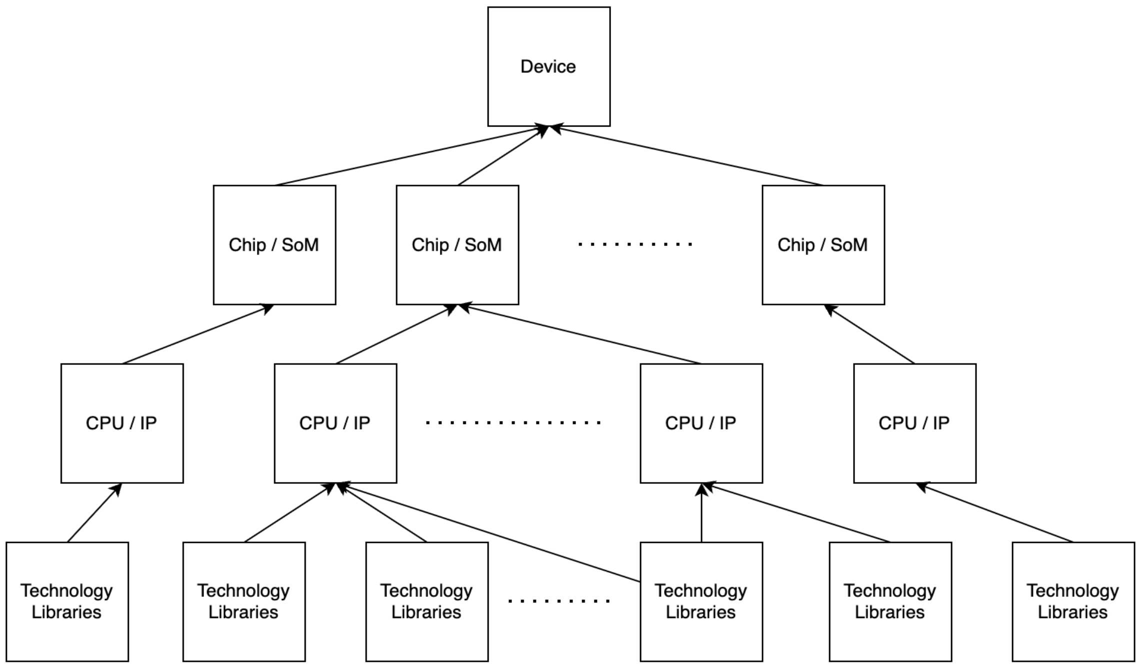


Figure 1:An example of the hierarchical dependency among the hardware views. V0: Technology Libraries, V1: CPU / IP, V2: Chip / SoM, V3: Device.

Each view is described hereafter in detail.

- A **technology library**, also known as a tech library, is a collection of resources, materials, and information related to the building of hardware components. It serves as a repository of knowledge, providing access to a wide range of technological resources. The silicon manufacturers can benefit from technology libraries by accessing resources created by others, keep them updated, and provide insights into best practices and methodologies.

- A **CPU** (Central Processing Unit) is the component of a device responsible for executing instructions and performing calculations. It includes the control unit, arithmetic logic unit, registers, and cache. With the term **IP** (Intellectual Property, i.e., memories, reusable unit of logic, cryptographic accelerator, cell, or integrated circuit layout design) we denote all the other components that are not the CPU and collaborate to the functioning of the final device.

- A **chip** is a physical integrated circuit that is used in electronic devices. It is responsible for the processing, storage, and control of electrical signals within a device. Chip is a hardware component that can be buyed on the market. A **System on Module** (SoM), is a small, self-contained computing module that integrates essential components of a system onto a single board. It is designed to provide a ready-made solution for embedded system development, reducing the time, cost, and complexity of designing a custom hardware solution. A typical System on Module can consist of the following components: processor, memory, I/O interfaces, power management, and connectors.

- A **device** is a physical object that is designed and used to perform specific functions or tasks, and often requires power or energy input to operate. Devices rely on electronic circuits and components to function. For example, they contribute to communication (e.g., smartphones), productivity (e.g., laptops), entertainment (e.g., smart TV), and healthcare (e.g. wearable fitness tracker).

In Table 1 we report the identified views; each view can be linked to one or more referents, i.e. who works and deals with hardware in the correspondent view. This means that, for example, a silicon manufacturer will be interested in components that are grouped in views 0 and 1, while view 2 is related to component integrators. We identify four possible referents:

- **Silicon manufacturer**: company that produces silicon chips. In some cases the chip maker is an Integrated Design Manufacturer (IDM) in other cases is a pure foundry. In the latter case the foundry is producing chips for customers, a typical example is TSMC, while in the case of IDM the company design the chip itself (decide which type of CPUs, interconnection and IPs are integrated in the silicon chip)
- **Chip designer:** It can be a company, an IDM or a fabless (a company without a silicon fab) or an individual that wants to design a chip. The designer decides what should be integrated in the chip and interacts with foundry in order to deliver a set of files for the production.
- **Component integrator**: company or entity that specialises in integrating different electronic components and subsystems into a cohesive and functional system. They play a crucial role in the development and manufacturing of complex electronic products by assembling and integrating various components sourced from different manufacturers. The roles of a component integrator involve: component selection, system design and layout, component procurement, assembly and integration, testing and quality assurance, and documentation and support. We include in the category of component integrator the case of companies or individuals that delivers a final product.
- **Final user**: person or company that is placed at the end of the production-distribution-usage chain. In other words, the final recipient of the object or service that is produced which benefits from its usage.

Table 1: Hardware views.

View	Referent	Description
V0 - Technology Library	Silicon manufacturer	Libraries containing the blocks to build the fundamental bases for a hardware component.
V1 - CPU / IP	Silicon manufacturer or chip designer	The central processing unit and all the subcomponents (Intellectual Property) that are used to build the next view.
V2 - Chip / SoM	Silicon manufacturer, Chip designer or Component integrator	Integrated circuit that combines multiple electronic components and functionalities into a single chip; the subcomponents come from the previous view. It is an item you can buy ready made, directly from the market.
V3 - Device	Component integrator or Final user	The hardware into the hands of the final user, which is designed and used to perform a specific function or task, and which is the final composition of parts from lower views.

In order to model the interdependencies and connections among the components of each of the views, the most straightforward approach is to use a hierarchical topology. Indeed, a device (V3) can be composed of multiple chips and SoMs (V2), and the latter of multiple CPUs and IPs (V1), which are, in turn, based on the technology libraries (V0). A representation of such dependencies is illustrated in Figure 1. Nevertheless, in Chapter 4 - Component and Vulnerability Tracking we will discuss that these dependencies are not always hierarchical, since there are hardware components that are built with subcomponents belonging to the same view. This is, for example, the case of the u-blox cellular module LARA-R6001, which is a view-2 chip containing view-2 subcomponents (see Section 4.5.2 - Practical Example). Anyway, this interdependence of components inside the same view does not affect the considerations made in this Chapter.

2.4 Properties

To understand if a hardware component is open source, we have identified a list of properties to be analysed and scored. Not all of them can be applied to each view, but we tried to make them as homogeneous as possible. Moreover, we tried to be exhaustive, listing all the properties that we considered relevant in a hardware for the definition of how much it is open source.

For each component in the views, we propose the list of properties and descriptions that are in Table 2.

Table 2: View properties.

Property	Description
P0 – Source code and design files	The source code and design files used for building the component. The source code is a collection of instructions or statements written in a programming language that make up the functionalities of the component. Design files serve as a blueprint or reference for implementing the intended design.
P1 - Licences	Which kind of licence is provided. A licence, in the context of software/hardware and IP, is a legal agreement that outlines the terms and conditions under which a person or organisation is permitted to use, distribute, modify, or sell a particular software or intellectual property. Licences help protect the rights of the hardware or intellectual property creators, while providing clear guidelines for users regarding their rights and responsibilities.
P2 - Design tools	Visual design tools (e.g., STM32CubeIDE IOC file), and other tools to support design. Design tools are software applications or platforms that assist designers in creating, editing, and managing the design elements of the hardware.
P3 – Toolchain	Any software tool that processes design files and/or source code and produces artefacts that are necessary for production (e.g., compiler, linker, synthesis tool). A toolchain refers to a set of software tools that are used together in a specific sequence. It consists of various tools that perform different tasks during the development process. Each tool in the toolchain typically takes the output of the previous tool as its input and produces output that can be used by the subsequent tool.
P4 – Software ecosystem	A software ecosystem refers to a collection of software applications, tools, frameworks, libraries, Software Development Kits and platforms that are interconnected and interact with each other to support software development, deployment, and usage. It represents the environment in which software operates and the various components that enable its functioning. A software ecosystem typically includes: operating systems, programming languages, integrated development environments, libraries and frameworks, cloud platforms, etc.
P5 - Firmware	The firmware running on the component and distributed with the product. Firmware refers to a type of software that is embedded within electronic devices and provides low-level control and functionality. It is a specific type of software that is stored in non-volatile memory, such as ROM (Read-Only Memory) or flash memory, and is responsible for controlling the hardware of a device. Since it is strictly related to the hardware component, firmware has to be considered in our scoring method.
P6 - Processes	Any "DevSecOps"-related aspect for which the manufacturer can/has to provide evidence of in order to guarantee environmental security. Processes include all the activities during the hardware development that integrate security practices and considerations into every stage. Generally, the manufacturer emphasises collaboration and shared responsibility among development, security, and operations teams to ensure that security measures are implemented from the beginning of the development process. Examples of

Property	Description
	processes are: security of the private key for signing update images or assurance of having done security testing.
P7 - Replicability	Whether the hardware can be easily/completely replicated starting from the open source information or not. Some key aspects and considerations related to replicability are: methodological transparency, data availability, independent verification, sample size and statistical power, replication studies, methodological rigour and standardisation.
P8 - Documentation	Documentation that describes the design, functionality, specifications, assembly, operation, and maintenance of hardware components or systems. Comprehensive and accurate documentation is crucial to have for various reasons, including facilitating effective communication, ensuring consistency, aiding troubleshooting and repairs, supporting future development, and complying with regulatory requirements.
P9 – Code examples	Examples that can be found in the documentation and/or online.

The presented list of properties is *extensible* according to the hardware component under analysis. More details on this argument can be found in Section 2.8 - Considerations about this Scoring System.

2.4.1 *Applicability of Properties to Views*

Depending on the view that is taken into consideration, it may be not possible to apply a specific property to the hardware component under analysis. For example, taking into consideration the low level view, V0, it is clear that the firmware property cannot be applied. V0 represents the technology libraries, which are used to build the CPUs and IPs (V1); the elements in V0 are not capable of running firmwares. According to the design of the components included in the V0, the “firmware” property is not significant and should not be included into the analysis.

Moreover, for some views, the meaning of the properties may vary with respect to the others. For instance, the software ecosystem is completely different between V0 and V3. In general, the tools and the programming languages required to design or use components from the two views are not easily comparable in terms of functionalities, requirements, and licences. For instance, designing and using components in V0 may involve working with specific EDA designer tools and focusing on hardware-level implementation. In contrast, in V3, the focus shifts to higher-level software development, where more programming languages and frameworks may be available and a different approach is used overall (e.g.: Linux and its utilities).

Furthermore, functionalities and capabilities associated with components can differ significantly across the different views. The components of V0 may be limited to specific and basic operations (such as elementary arithmetic operations), while components of V3 may implement complex functionality (such as structured communication protocols). Thus, applying the properties to these different views is not a trivial task; and differences of a similar nature can be also observed between intermediate views. To overcome this challenge, evaluators must carefully consider the specific context, objectives, and requirements of each view when assessing the properties.

Because of the structure of View 0, our scoring template does not include the following three properties for this view:

- Software ecosystem;
- Firmware;
- Processes.

This choice derives from the previous observations and from the required level of expertise to coherently apply the scoring of such properties in the said context.

2.4.2 *Categorization of Properties in Sets*

The properties listed in the previous Paragraph can be divided in sets, according to their sphere of belonging. Indeed, we identify three sets (Figure 2), that we list below.

- **Component**: in this set are grouped the properties related to the hardware component itself. The properties are:
  - P0 Source code and design files
  - P1 Licences
- **Ecosystem**: all the properties that are related to tools and software running on the hardware. In this set, the grouped properties are:
  - P2 Design tools
  - P3 Toolchain
  - P4 Software ecosystem
  - P5 Firmware
- **Infrastructure**: those properties that are related to a particular aspect of the component, that is not directly linked to the hardware or the running software. These properties are:
  - P6 Processes
  - P7 Replicability
  - P8 Documentation
  - P9 Code examples

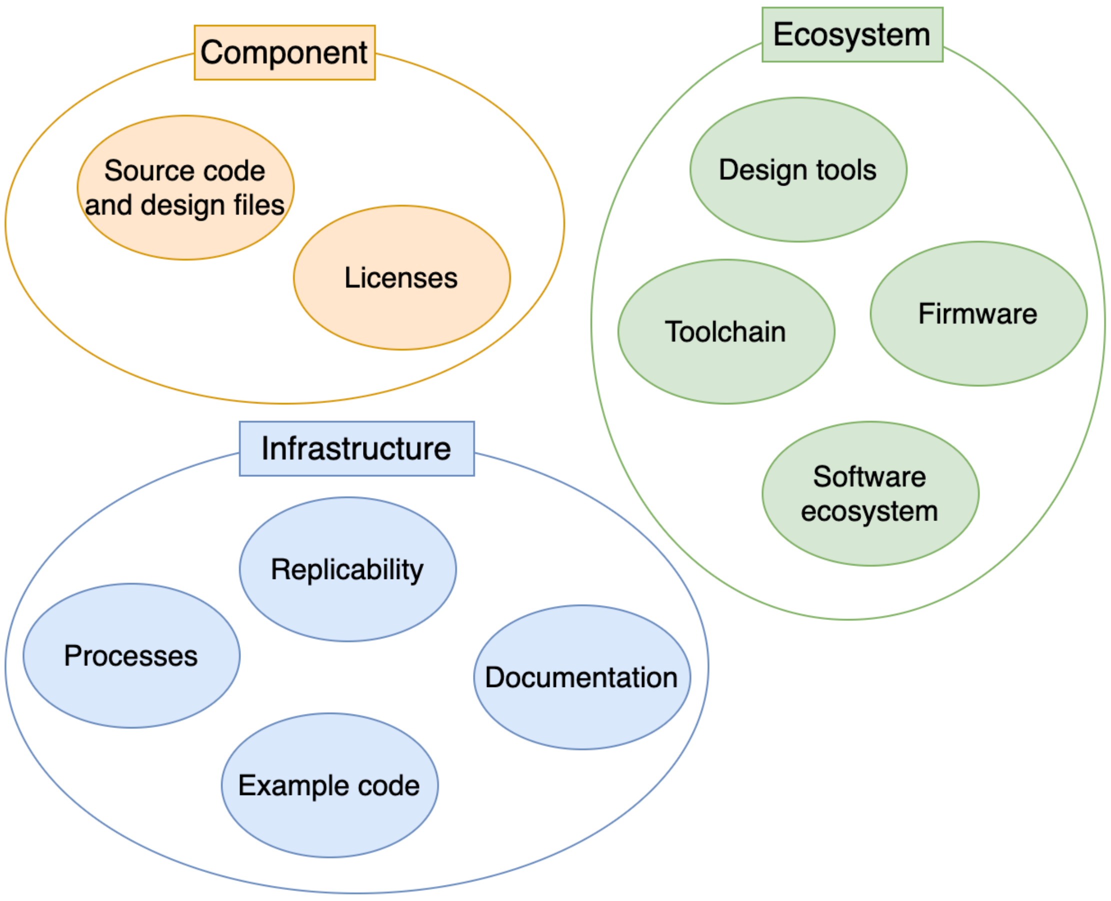


Figure 2:The three sets in which the properties are grouped.

As discussed for the list of properties, the list of sets can also be *extended*. Indeed, in case a new property is considered in the list in Table 2, but it cannot be included in one of the three sets presented in this Section, then a new set could be defined.

2.5 How to Score Hardware Open sourceness

Following the structure of views and properties described in the previous Sections, it is clear that it is not possible to declare a hardware as entirely open source or closed-source. Therefore, our idea is to associate to a hardware component a view, and then compile for it a vector of scores, one score for each property. The property score ranges from 0 to 3, where 0 means that the property reflects the behaviour and features of closed-source hardware, and 3 of open source hardware. Hence, the higher the score the more open is the hardware component.

2.5.1 *Properties Score*

In Table 3 we report how we score the ten properties that we have defined in Section 2.4 - Properties; for each property, we give a description of what it means scoring that property with levels from 0 to 3.

Table 3: Descriptions of the scores for each property.

Property	Scores
P0 – Source code and design files	0 – The source code and design files are closed-source, and not documented at all. 1 – The source code is poorly documented and the design files are not open source (or not completely). 2 – The source code is well documented, but not sufficiently complete to be considered straightforward to write open source software that allows the device to operate properly and fulfil its essential functions. The design files are not completely open source. 3 – The source code is sufficiently documented such that it could reasonably be considered straightforward to write open source software that allows the device to operate properly and fulfil its essential functions. The design files are completely open source.
P1 - Licences	0 – The licence does not allow any modification or derived work. The licence restricts the parties from selling or giving away the project documentation. 1 – The licence imposes some important restrictions. 2 – The licence imposes few small and irrelevant restrictions. 3 – The licence shall allow modifications and derived works, without commercial restriction. The licence shall not restrict any party from selling or giving away the project documentation.
P2 - Design tools	0 – The most commonly used design tools are proprietary, and are not made available to the public. 1 – The most commonly used design tools are proprietary, and available to the public as paid software. 2 – The most commonly used design tools for the target platform are available under NDA or, in general, with licences that regulate their use. They are not open source. 3 – The most commonly used design tools for the target platform are available online without restriction and are open source.
P3 - Toolchain	0 – The toolchain is proprietary, and it is not made available to the public. 1 – The toolchain is released only under NDA or in general with licences that strictly regulate its use.

Property	Scores
	2 – The toolchain is freely available online, however parts of it are not released under an open source licence. 3 – The toolchain for the target platform is available online without restriction and is completely open source.
P4 – Software ecosystem	0 – No additional libraries / SDK to interact with the product are provided, and it is not possible to recover them. 1 – Additional libraries / SDK to interact with the product can be obtained by paying or signing an NDA. 2 – Additional libraries / SDK to interact with the product are available with minor restrictions. 3 – Additional libraries / SDK to interact with the product are provided together with the product or are available online without restrictions, and they are open source.
P5 – Firmware	0 – The provided firmware is completely closed-source, and not documented at all. 1 – The provided firmware is poorly documented, possibly not completely closed-source. 2 – The interfaces are well documented, but they are not sufficiently complete to be considered straightforward to write open source software that allows the device to operate properly and fulfil its essential functions. 3 – The interfaces are sufficiently documented such that it could reasonably be considered straightforward to write open source software that allows the device to operate properly and fulfil its essential functions. The necessary software is released under an OSI-approved open source licence.
P6 - Processes	0 – The manufacturer is not able to provide any evidence of following industry-standard best practice methodology for security-related aspects in their infrastructure, even on explicit request. 1 – The manufacturer does not provide any formal public evidence of following industry-standard best practice methodology for security-related aspects in their infrastructures; however, they do satisfy some of the standard related requirements and are able to produce evidence on demand (e.g. they perform security testing and are able to declare to do so when asked). 2 – The manufacturer can provide some public evidence of following industry-standard best practice methodology for security-related aspects in their infrastructures; however, they do not follow a Secure Development Life Cycle (SDLC) and do not possess any certification related to this topic. 3 – The manufacturer can provide formal public evidence of following industry-standard best practice methodology for security-related aspects in their infrastructure, such as the implementation of a Secure Development Life Cycle, possibly with related certification (e.g. 62443-4-1).
P7 – Replicability	0 – The component and all of its parts can exclusively be replicated by the manufacturer, with proprietary information. 1 – It is possible to obtain functional replicas of parts of the product with publicly available open source information, however a fully functional replica cannot be achieved. 2 – A fully functional replica of the component may be built by anyone with publicly available open source information; however, some parts of the component may not be open source and may therefore require to be integrated "as-is" (e.g. proprietary Java Card applet for a secure token). 3 – The component can be fully replicated by anyone in its design and functionality, leveraging publicly available open source information.

Property	Scores
P8 - Documentation	0 – The documentation is not furnished with the physical product and cannot be recovered in any way. 1 – The documentation can be obtained by paying for it or signing an NDA. 2 – Documentation is not provided with the physical product. It is possible (but difficult) to find it via the Internet without charge. 3 – The documentation is furnished with the physical product (or it is trivial to find it).
P9 – Code examples	0 – Code examples are not provided with the product, and cannot be found anywhere. 1 – Code examples are not provided with the product, few undocumented ones can be found on the Internet with some effort. 2 – Code examples are not provided with the product, but it is possible to easily find them on the Internet (e.g. from the community). 3 – Code examples are provided with the product (or can be easily found and downloaded directly from the manufacturer's website).

This Table is the result of our best effort, and it is the current state of our work. It is clear that it could be extended, in case that a new property is added in the list of studied properties. Also the range of the scores could be changed in future, becoming stricter or larger, including more possibilities and nuances in the levels of the properties.

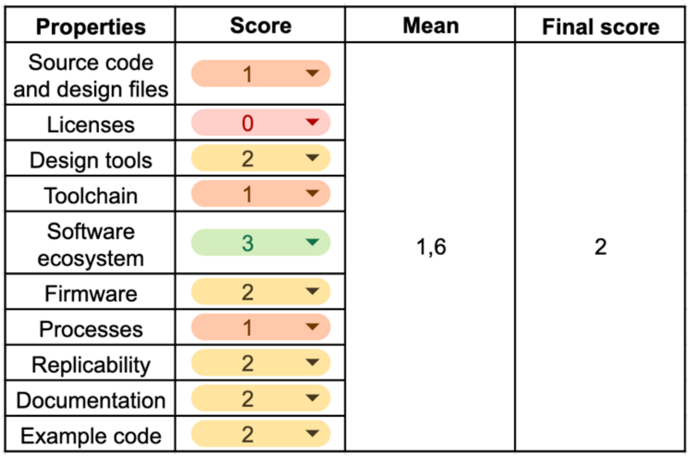
2.5.2 *Final Score*

Using what we described so far, we are able to compute a vector of scores for each hardware component. However, it is useful to compute a final score, which summarises how much an hardware component is open source.

Our first attempt in this direction was to compute the mean of the vector scores, and round the result to the closest integer. For example (Figure 3), a component can have the following scoring vector:

$$c = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 0 & 2 & 1 & 3 & 2 & 1 & 2 & 2 & 2 \\ \hline\end{array}$$

Hence in this case the overall score is $mean(1,0,2,1,3,2,1,2,2,2) = 1.6$ which is rounded as 2.



Properties	Score	Mean	Final score
Source code and design files	1	1,6	2
Licenses	0		
Design tools	2		
Toolchain	1		
Software ecosystem	3		
Firmware	2		
Processes	1		
Replicability	2		
Documentation	2		
Example code	2		

Figure 3: Example: computation of the final score as mean of scores in the vector.

However, this approach cannot be the most realistic one, because the formula gives the same weight to each property. Indeed, we are giving the same weight to all the properties, and consequently we are computing a mean that gives more importance to the properties in the *infrastructure* and *ecosystem* groups, and less to the properties in the *component* group (see Section 2.4.2 - Categorization of Properties in Sets for more details). This is because the number of properties listed under the *component* group are less in number than those in the other two groups.

Thus, our improvement to this approach is to first compute the mean of the scores of each group, and then compute the mean of the means which becomes the final result (*weighted mean*).

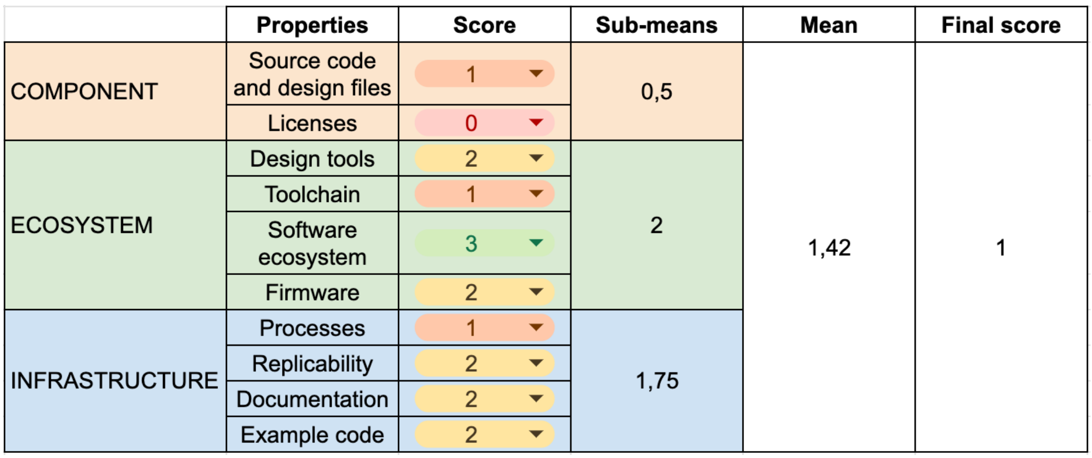
Let c be the scoring vector of a component, as in previous example (Figure 4):

$$c = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 0 & 2 & 1 & 3 & 2 & 1 & 2 & 2 & 2 \\ \hline\end{array}$$

Now we compute:

- The mean of the scores in each group:
  - $m_1 = mean(1{,}0) = 0.5$
  - $m_2 = mean(2{,}1{,}3{,}2) = 2$
  - $m_3 = mean(1{,}2{,}2{,}2) = 1.75$
- The overall score: $mean(m_1, m_2, m_3) = mean(0.5{,}2{,}1.75) = 1{,}42$ which rounded is 1.

Note that by giving more importance to the two properties in the *component* group (that have very low scores), the mean is reduced and then also the overall score.



	Properties	Score	Sub-means	Mean	Final score
COMPONENT	Source code and design files	1	0,5	1,42	1
	Licenses	0			
ECOSYSTEM	Design tools	2	2		
	Toolchain	1			
	Software ecosystem	3			
	Firmware	2			
INFRASTRUCTURE	Processes	1	1,75		
	Replicability	2			
	Documentation	2			
	Example code	2			

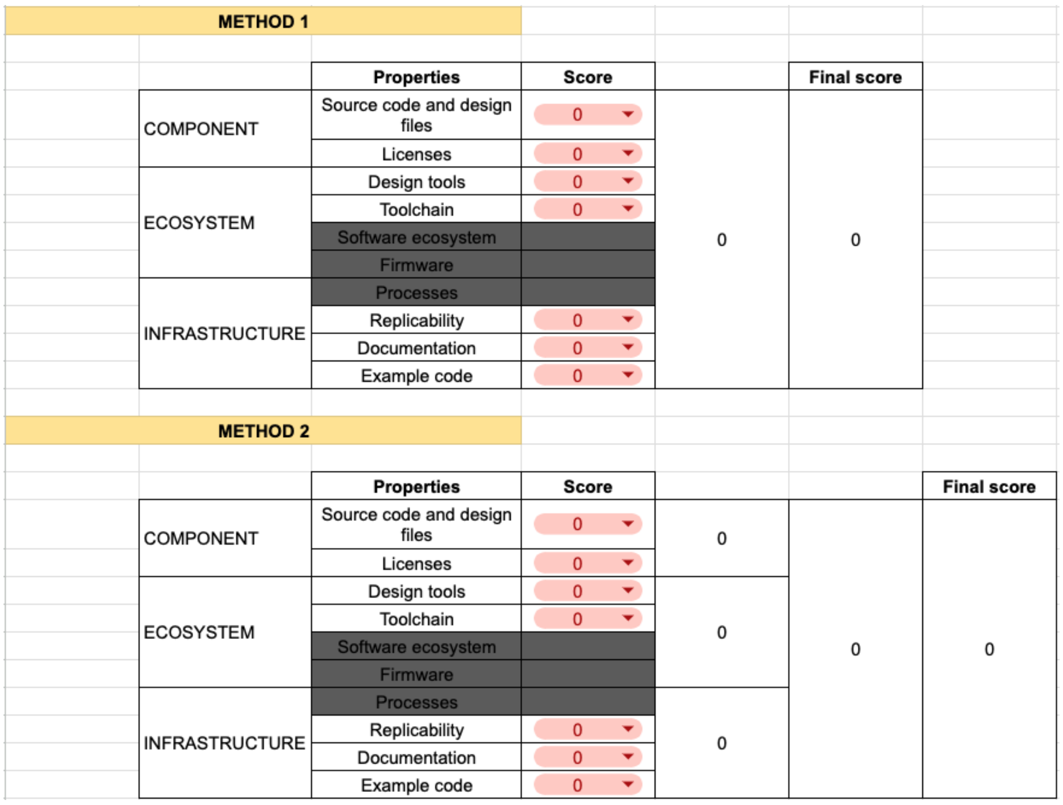
Figure 4: Example: computation of the final score as mean of means of scores in the groups.

We prepared a template that allows the evaluator to easily score a hardware component simply by filling a table with the scores of the properties. In particular, the file has the following sheets:

- *Views*: in this sheet the list of the analysed views and the diagram that we presented in Section 2.3 - Views are reported. This is a useful memo to find the best view in which place the hardware component that is under study.
- *Properties and scores*: this table collects the list of properties, divided into the three sets *Component*, *Ecosystem*, and *Infrastructure* (Section 2.4.2 - Categorization of Properties in Sets). For each property a short description and the meanings of the scores are reported (Section 2.5 - How to Score Hardware Open sourceness).
- *Computation of the final score*: here we present a short explanation on how the final scores are computed. In particular, in this attached file we have reported the two methods of scoring proposed in this Section. We recall that method 1 consists of computing the final score as simply the mean of the scores of all the properties without distinction. On the other hand, in the second method the means of the properties in the sets are computed separately, and the final result is computed as the mean of these means.

- The last four sheets contain the table for evaluation (*Evaluation View 0, Evaluation View 1, Evaluation View 2, Evaluation View 3*). If the hardware component under evaluation is in view 0, then column *Score* in sheet *Evaluation View 0* needs to be filled; similarly, when considering view 1 sheet *Evaluation View 1* needs to be chosen, and so on. Note that the tables for the evaluations are all equal, except for view 0, for which the three properties of *Software ecosystem*, *Firmware*, and *Processes* cannot be evaluated. In each sheet, it is possible to choose between the computation of the score with method 1 or method 2 (or both).

For example, if the hardware component belongs to view 0, then the table that has to be filled (with method 1 or 2) is the one in Figure 5. Note that in this case some properties are dimmed and must not be filled, since these properties can't be defined for a hardware in view 0 (see Section 2.4.1 - Applicability of Properties to Views). By filling the column titled *Score*, the last three columns on the right are automatically compiled. In case of using method 2, the column closest to the *Score* one is filled with the means of the scores grouped in the sets; the next one is the mean of means, not rounded. The last column *Final score* is the last computed mean rounded.


METHOD 1

	Properties	Score		Final score
COMPONENT	Source code and design files	0	0	0
	Licenses	0		
ECOSYSTEM	Design tools	0		
	Toolchain	0		
	Software ecosystem			
	Firmware			
INFRASTRUCTURE	Processes			
	Replicability	0		
	Documentation	0		
	Example code	0		

METHOD 2

	Properties	Score			Final score
COMPONENT	Source code and design files	0	0	0	0
	Licenses	0			
ECOSYSTEM	Design tools	0	0		
	Toolchain	0			
	Software ecosystem				
	Firmware				
INFRASTRUCTURE	Processes		0		
	Replicability	0			
	Documentation	0			
	Example code	0			

Figure 5: Tables for the final score computation in sheet Evaluation view 0.

2.6 How to Apply our Open source Definition: Case Studies

In this Section we present an example of scoring a hardware component for all the views. In particular, the hardware evaluated are listed below:

- V0: UMC
- V1: OpenTitan
- V2: TROPIC01
- V3: Trezor

For each view, the set of properties (defined in Section 2.4 - Properties) are evaluated. The final score achieved by each of the components is a composition of the scores of the singular properties.

In the scoring process of component properties, the evaluator relies on information sourced from official channels: unofficial or unlicensed sources or artefacts that disclose relevant details on a component/development are not taken into consideration when not explicitly authorised by the original author/owner of a component/development (think for example of reverse engineering attempts from the community), as they do not bring any open source-related "merit" to the project. The scores are assigned ranging from 0 (indicating closed-source) to 3 (representing open source). The score is assigned by the evaluator according to its subjective assessment. However, such an assessment shall be influenced by factors such as the difficulty in retrieving information and the physical accessibility of the material. If some material is reserved and some is freely available, the evaluator can pick a score which is between 0 and 3 according to its knowledge and experience. In order to achieve a coherent and unbiased assessment, the results should be reviewed multiple times by differently experienced evaluators.

These examples have been computed using the template presented in Section 2.5.2 - Final Score. We filled the column corresponding to the scores vector, one table per example. Then our templates automatically compute the final scores. For sake of completeness, we report the results with both the presented scoring methods.

2.6.1 *V0 - Technology Library*

As a first example for a component in V0, we present in this Section the **United Microelectronics Corporation (UMC) technology library**.

UMC is a Taiwanese semiconductor company. It is specialised in the manufacturing of integrated circuits (ICs) for a variety of applications.

Among UMC products, we consider the technology library UMC, and in Table 4 we report our scoring table for this component in V0. The evaluation reported in the Table is performed with the second method described in Section 2.5.2 - Final Score, i.e. computing the means for the properties in sets of *component*, *ecosystem*, and *infrastructure*, and obtaining the final result as the mean of these means.

Table 4: Scoring for the Technology Library UMC with the second method.

	Properties	**Score**			**Final score**
COMPONENT	Source code and design files	0	0	0,67	1
	Licences	0			
ECOSYSTEM	Design tools	2	2		
	Toolchain	2			
	Software ecosystem				
	Firmware				
INFRASTRUCTURE	Processes		0		
	Replicability	0			
	Documentation	0			
	Example code	0			

The final rounded score of 1. This first result suggests that this library can be considered more closed-source than open source. By analysing more in depth the individual properties scores, it is possible to note that all the properties belonging to the sets *component* and *infrastructure* are evaluated as the lowest scores possible (all zeros), whereas the properties related to the set *ecosystem* have higher scores (all twos). This means that the closeness of the UMC library is mostly caused by the features related to its components and infrastructure.

2.6.2 *V1 – CPU / IP*

The second example, concerning a hardware component in V1, is **Open Titan**. OpenTitan is an open source silicon root of trust project. It aims to provide a fully open and transparent solution for building hardware security chips. OpenTitan is a collaborative effort led by Google, along with several other organisations.

The main goal of OpenTitan is to address the increasing need for secure hardware in various industries, including data centres, cloud infrastructure, and connected devices. The project focuses on creating a trustworthy, open source reference design and integration guidelines for silicon root of trust chips.

The OpenTitan project incorporates various security features, such as cryptographic accelerators, secure boot, key management, and hardware-based attestation. The project aims to develop a robust and flexible solution that can be customised to meet the specific security requirements of different applications and industries.

In Table 5 we report our scoring table for this component in V1, another time evaluating it with the second method described in Section 2.5.2 - Final Score.

Table 5: Scoring for Open Titan with the second method.

	Properties	**Score**			**Final score**
COMPONENT	Source code and design files	3	3	2,75	3
	Licences	3			
ECOSYSTEM	Design tools	3	3		
	Toolchain	3			
	Software ecosystem	3			
	Firmware	3			
INFRASTRUCTURE	Processes	3	2,25		
	Replicability	0			
	Documentation	3			
	Example code	3			

The final rounded score is 3, suggesting how this IP can be considered actually an open source hardware component. Indeed, by providing an open source reference design, OpenTitan enables transparency, peer review, and collaborative development for hardware security. It allows anyone to access, use, and contribute to the project, fostering innovation, security, and standardisation in the domain of hardware security. Moreover, the project aims to develop a robust and flexible solution that can be customised to meet the specific security requirements of different applications and industries.

As can be noted by the Table, all properties received the maximum score possible, except for Replicability, because it depends on the ASIC manufacturing limitations.

2.6.3 *V2 - Chip / SoM*

The third example, specific for a hardware component in V2, is the **secure element TROPIC01**.

TROPIC01 is the first of Tropic Square's secure element series. It supplies and stores the cryptographic keys of embedded systems. It is built with dedicated secure HW engines to provide cryptographic algorithms, a set of sensors for anti-tampering, and design practices to protect against a wide range of attacks.

In Figure 6 the TROPIC01 schematic is shown, with its parts and functionalities.

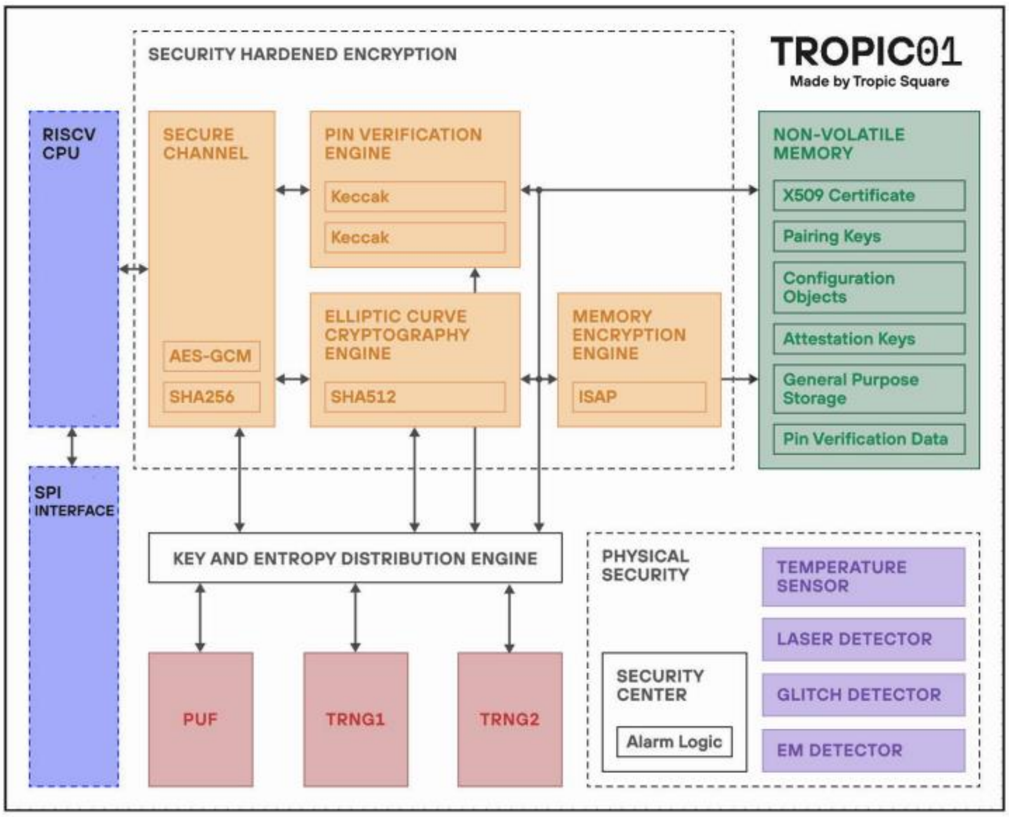


Figure 6: TROPIC01 schematic.

Table 6 shows the scoring table for TROPIC01.

Table 6: Scoring for TROPIC01 with second method.

	Properties	Score			Final score
COMPONENT	Source code and design files	3	3	2,67	3
	Licences	3			
ECOSYSTEM	Design tools	3	3		
	Toolchain	3			
	Software ecosystem	3			
	Firmware	3			
INFRASTRUCTURE	Processes	2	2		
	Replicability	0			
	Documentation	3			
	Example code	3			

The final rounded score results in 3. All but two properties have been evaluated with the higher score (3). The only exceptions that do not have the maximum score are *Processes* and *Replicability*. *Processes* has score 2 because, although the manufacturer (Tropic Square) provides some public evidence of following industry-standard best practice methodology for security-related aspects in their infrastructure, it does not follow a Secure Development Life Cycle. *Replicability* has score 0 because the component and all of its parts have proprietary information and can only be replicated by Tropic Square.

Here, it is possible to note that even though the means are lowered by these not-optimal values, the final score is the highest possible, therefore TROPIC01 can be considered to be open source.

2.6.4 *V3 - Device*

The fourth example, specific for a hardware component in V3, is the **Trezor hardware wallet.**

Trezor is a brand of hardware wallets designed for securely storing and managing cryptocurrencies. Developed by SatoshiLabs, Trezor devices provide an offline, cold storage solution for protecting private keys and conducting cryptocurrency transactions.

In Table 7 we report our scoring tables for this V3 component.

Table 7: Scoring for Trezor with the second method.

	Properties	Score			Final score
COMPONENT	Source code and design files	3	3	2,92	3
	Licences	3			
ECOSYSTEM	Design tools	3	3		
	Toolchain	3			
	Software ecosystem	3			
	Firmware	3			
INFRASTRUCTURE	Processes	2	2,75		
	Replicability	3			
	Documentation	3			
	Example code	3			

Once again, the result is a final rounded score 3. Indeed, Trezor's software and hardware are open source, allowing developers and security experts to review and audit the code for any potential vulnerabilities or issues.

Also, as can be noted by the scores in the Table, the exact score is close to the maximum: all singular values reach 3, except for *Processes* property, which has score 2. As before, the manufacturer follows security best practices, the development is public so there is review by the community and also bounty programs motivate security researchers to contribute. However, a Secure Development Life Cycle is not implemented or certified for the creation of the device.

2.7 How to Score Hardware with Subcomponents

In general, scoring the bare hardware component cannot reflect a very realistic outline of how much a hardware device is open source. For this reason, we decided to push our effort in trying to include, in the component that we are evaluating, also the scores of the subcomponents (or at least the more representative ones).

We define a *subcomponent scoring method* that allows to compute the score of a device or a component belonging to a view different from view 0. Indeed, we can compute the vector of scores for the component considering both its vector of scores and the vector of scores of the subcomponent, merged together with different weights.

More in detail, to apply the subcomponent scoring method, next steps have to be followed.

1. Compute the vector c containing the scores of the properties of the component currently under study. Identify the j-th element of c as c_j.
2. Compute the vectors c^i of the scores of the subcomponents, one score for each property. Identify the j-th element in the i-th vector as c_j^i.
3. Compute the vector of means m of the scores of the subcomponents. The j-th element in m will be $m_j = mean(c_j^0, c_j^1, \dots)$.

4. Give a weight $w_c \geq \frac{1}{2}$ to the scores of the current component and a weight $w_m = 1 - w_c$ to the mean of the scores of the subcomponents.
5. The vector of the weighted scores of the current component is c^*, where each element is given by

$$c_j^* = w_c * c_j + w_m * m_j$$

6. The final score is computed with one of the methods described in Section 2.5.2 - Final Score, applied on the vector of c_j^*.

As specified in point 4., this formula gives more weight to the score of the current component than the scores of the subcomponents, but the final result can be heavily influenced by the latter.

In particular, in our observations and applications of this method to known devices, we decided to assign to w_c the value of 0.75, and to w_m the value of 0.25.

2.7.1 *Numeric Subcomponent Scoring Example*

In this Paragraph we present a numeric example, to understand the application of the method step by step.

1. Vector c containing the scores of the properties of the component that we are hypothetically studying is:

$c =$

3	3	3	3	2	3	3	1	2	3

   The current score of the component, that is the mean of the element in c rounded to the nearest integer, is 3.
2. Suppose that it has 2 subcomponents, c^0 and c^1. The vectors of the scores of these subcomponents are:

$c^0 =$

1	0	3	1	2	0	2	0	2	3

$c^1 =$

0	1	3	2	1	1	3	0	1	1

3. We compute the vector of the means:

$m =$

0.5	0.5	3	1.5	1.5	0.5	2.5	0	1.5	2

4. We assign the weight $w_c = 0.75$ to the score of the current component and a weight of $w_m = 0.25$ to the mean of the scores of the subcomponents.
5. The vector of the weighted scores of the current component is c^* (each element c_j^* is rounded to the closest integer):

$c_j{}^* =$	2	2	3	3	2	2	2	1	2	3

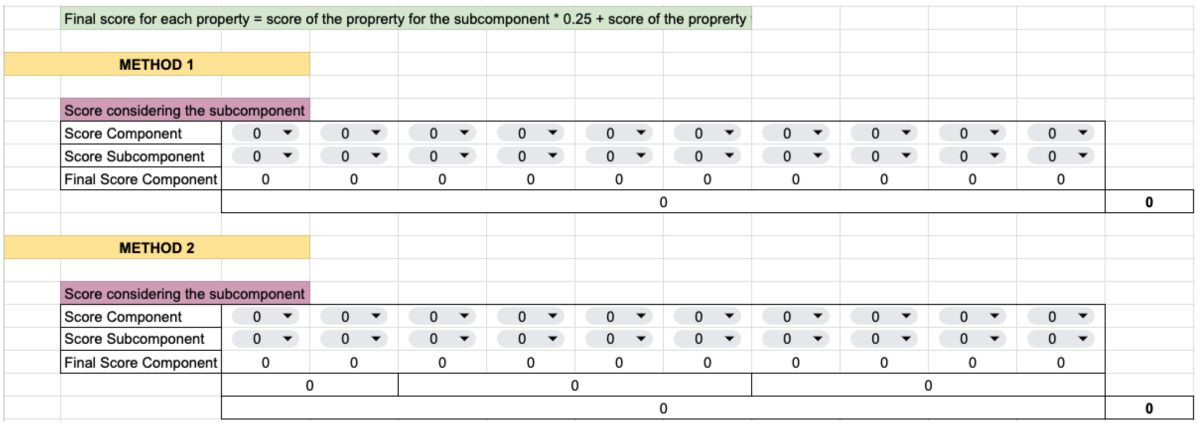
6. To compute the final score of the studied component, we follow the first method presented in Section 2.5.2 - Final Score. Then, the overall score is the mean of c_j^*, which rounded to the closest integer is 2. Note that taking into account the open sourceness of the subcomponents has decreased the score of the studied component.

Table 8 resumes all the previous computations.

Table 8: Numerical example of how to score a component taking into account its subcomponents.

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	Final score (rounded mean)
c	3	3	3	3	2	3	3	1	2	3	**3**
c_0	1	0	3	1	2	0	2	0	2	3	
c_1	0	1	3	2	1	1	3	0	1	1	
m	0.5	0.5	3	1.5	1.5	0.5	2.5	0	1.5	2	
c^* (exact)	2.38	2.38	3	2.63	1.88	2.38	2.88	0.75	1.88	2.75	
c^* (rounded)	2	2	3	3	2	2	2	1	2	3	**2**

We prepared a template in which the final score is automatically computed. In this case, the file is composed of one sheet. In this sheet, there are two Tables (Figure 7), one for method 1 and one for method 2, and the user can decide which method to apply to compute the final score (see Section 2.5.2 - Final Score). In these Tables it is possible to insert the vector of the actual scores of the component under study, the vector of the scores of the subcomponents, and automatically the new score for the component is computed, taking into account also the subcomponents. Note that we assign a weight of 0.75 to the component's properties, and a weight of 0.25 to the subcomponents properties. Lower-right cell is the final rounded score for the hardware component under study.


Final score for each property = score of the proprerty for the subcomponent * 0.25 + score of the proprerty

METHOD 1

Score considering the subcomponent

Score Component	0	0	0	0	0	0	0	0	0	0	
Score Subcomponent	0	0	0	0	0	0	0	0	0	0	
Final Score Component	0	0	0	0	0	0	0	0	0	0	
	0										**0**

METHOD 2

Score considering the subcomponent

Score Component	0	0	0	0	0	0	0	0	0	0	
Score Subcomponent	0	0	0	0	0	0	0	0	0	0	
Final Score Component	0	0	0	0	0	0	0	0	0	0	
	0		0				0				
	0										**0**

Figure 7: Tables for the computation of the score of a component considering also the score of subcomponents; one Table is for method 1 and one is for method 2.

2.7.2 *Case Study: Raspberry Pi 4*

Raspberry Pi 4 is a single-board computer launched by the Raspberry Pi Foundation in 2019. It is the latest addition to the Raspberry Pi series, at the time of writing. Raspberry Pi 4 is a machine that can be used for a wide range of applications, including IoT projects and industrial solutions. The Raspberry Pi 4 is built with a Broadcom BCM2711 SoC, based on quad core ARM A-72 processor that runs at 1.5GHz. The board comes with up to 8GB of RAM and supports USB 3.0, dual-band Wi-Fi and Bluetooth 5.0 module. The Raspberry Pi 4 runs on the latest version of Raspberry Pi OS, which is built on Debian Linux (Figure 8).

Figure 8: Raspberry Pi 4 [Raspberry Products].

In this Section, we test our subcomponent scoring method to evaluate its effectiveness and its robustness by analysing the Raspberry Pi 4. Although the Raspberry Pi 4 was designed mainly to spread the use of compute modules by developers [Raspberry About] and the Raspberry Pi Foundation has never claimed its products as open source solutions, it is commonly associated with the open source world among the developer community. This perception can be related to the plethora of resources that are available from and by the community (e.g., project [Home Assistant]). In particular, the aspect we observe in this analysis regarding Raspberry Pi 4 is that it defines a clear division between software and hardware: the software being open source and the hardware being strictly closed.

Table 9: Analysis of the Raspberry PI4.

RASPBERRY PI4	**Properties**	**Score**	
COMPONENT	Source code and design files	2	
	Licences	2	2
ECOSYSTEM	Design tools	3	
	Toolchain	3	
	Software ecosystem	3	
	Firmware	3	3
INFRASTRUCTURE	Processes	1	
	Replicability	2	
	Documentation	3	
	Example code	3	2,25
FINAL SCORE	**2**		

Table 10: Analysis of the Broadcom BCM2711.

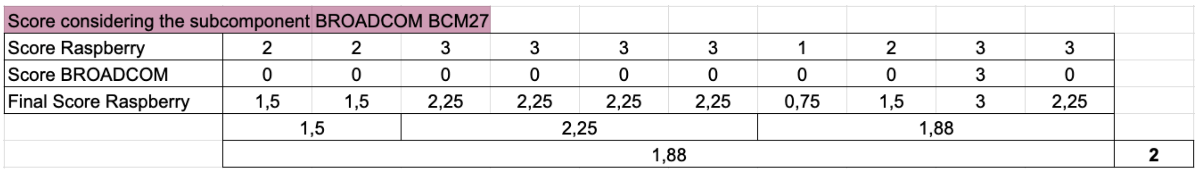
BROADCOM BCM2711	**Properties**	**Score**	
COMPONENT	Source code and design files	0	
	Licences	0	0
ECOSYSTEM	Design tools	0	
	Toolchain	0	
	Software ecosystem	0	
	Firmware	0	0
INFRASTRUCTURE	Processes	0	
	Replicability	0	
	Documentation	3	
	Example code	0	0,75
FINAL SCORE	**0**		

As shown in the previous Tables, the analysed views are two. Table 9 represents the device, which is the Raspberry Pi 4 itself. In such a view (V3), the score is 2 and it is easy to retrieve information about the components and the software that drives them.

The analysis is interrupted at the very next view: Chip/SoM (V2), in Table 10. The BCM2711 SoC is manufactured by Broadcom, which does not provide detailed information about the schematics and the architecture of its subcomponents. Since it is not possible to easily access and evaluate the Chip details, being trade secrets, we assign 0, the lowest score for this level. Without having access to the information of this level, it is not easy to recover the details about the subcomponents, thus the analysis stops.

From the analysis, it is clear the separation between the software and hardware. For example, a specific observation can be made about Linux, which is free and open source and allows many of the licensing restrictions software-related to be avoided. However, the latter observation should be considered true for all devices that allow the installation of open distributions and variants of the operating system. Thus, the evaluation of this aspect needs to be softened by the weight of the evaluations of the other (more specific) properties. Moreover, the fact that the operating system is open source, does not imply that the whole product can be considered as such. The closed aspects of its low level components should be part of the final score. Our scoring method is capable of managing this information and altering the final score according to each subcomponent.

We apply our scoring system and retrieve the final score for Raspberry Pi4 considering the subcomponent BCM2711 (Figure 9). Note that now the not rounded final score of Raspberry Pi4 is lower than before (i.e., final score 2.4 without BCM2711, final score 1.9 with BCM2711).



Score considering the subcomponent BROADCOM BCM27											
Score Raspberry	2	2	3	3	3	3	1	2	3	3	
Score BROADCOM	0	0	0	0	0	0	0	0	3	0	
Final Score Raspberry	1,5	1,5	2,25	2,25	2,25	2,25	0,75	1,5	3	2,25	
	1,5		2,25				1,88				
	1,88										2

Figure 9: Scoring Raspberry Pi4 considering the subcomponent BCM2711.

For comparison purposes, we evaluate the Toradex Apalis IMX6 (Figure 10) and its subcomponents, in Table 11. Apalis is a scalable System on Module (SoM) / Computer on Module (CoM) family that aims to provide high performance in a compact form factor. In particular, the Apalis IMX6 implementation embeds an IMX6 processor by NXP, in Table 11.

Figure 10: Apalis IMX6 [Toradex Apalis].

The Apalis IMX6 is not meant to be an open source alternative to the Raspberry PI 4. However, the score reached by the Apalis IMX6 is higher.

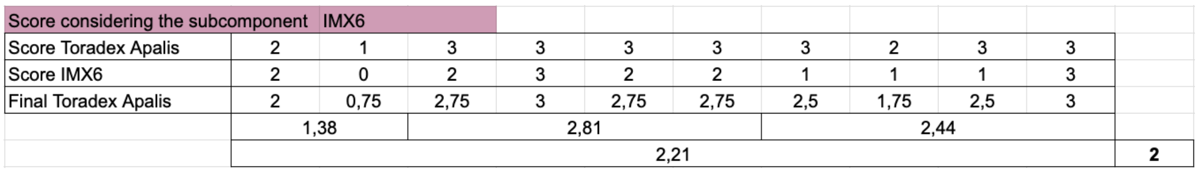
Table 11: Analysis of the Toradex Apalis IMX6.

TORADEX APALIS IMX6	**Properties**	**Score**	
	Source code and design files	2	
COMPONENT	Licences	1	1,5
	Design tools	3	
	Toolchain	3	
	Software ecosystem	3	
ECOSYSTEM	Firmware	3	3
	Processes	3	
	Replicability	2	
	Documentation	3	
INFRASTRUCTURE	Example code	3	2,75
FINAL SCORE	**2**		

Table 12: Analysis of the NXP IMX6.

IMX6	**Properties**	**Score**	
COMPONENT	Source code and design files	2	1
	Licences	0	
ECOSYSTEM	Design tools	2	2,25
	Toolchain	3	
	Software ecosystem	2	
	Firmware	2	
INFRASTRUCTURE	Processes	1	1,5
	Replicability	1	
	Documentation	1	
	Example code	3	
FINAL SCORE	**2**		

In Figure 11 the score for Toradex Apalis considering the subcomponent IMX6 is shown. The higher score w.r.t. the one in Figure 9 is justified by taking into account the subcomponents. The Raspberry Pi 4 embeds the BCM2711 processor, which is a custom and close implementation of the ARM architecture from Broadcom. On the other hand, the Apalis module embeds the IMX6 from NXP, which provides some information freely or just by creating an account on the official website. In conclusion, although the IMX6 is not an open source solution, the amount of available information is greater.



Score considering the subcomponent	IMX6									
Score Toradex Apalis	2	1	3	3	3	3	3	2	3	3
Score IMX6	2	0	2	3	2	2	1	1	1	3
Final Toradex Apalis	2	0,75	2,75	3	2,75	2,75	2,5	1,75	2,5	3
	1,38		2,81				2,44			
	2,21									**2**

Figure 11: Scoring Toradex Apalis considering the subcomponent IMX6.

A fully open-hardware solution that can be compared with the previous two products is the USB Armory Mk II from F-Secure now named WithSecure [USB Armory]. The USB Armory is an open source hardware design, implementing a flash drive sized computer. The main focus of this product is to provide a system that takes advantage of the shelf components and can be completely customizable.

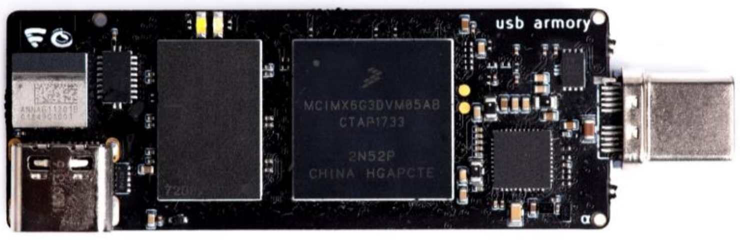


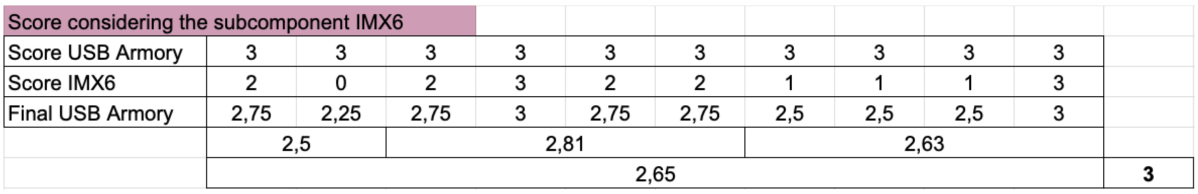
Figure 12: USB Armory Mk II [USB Armory Mk II].

The analysis in Table 13 highlights a higher score with respect to the previous cases. As in the previous cases, the final score is altered by the subcomponents. Since the processor on the USB Armory Mk II is the NXP IMX6, for this subcomponent we consider the scores reported in Table 12.

Table 13: Analysis of the USB Armory Mk II.

USB Armory	**Properties**	**Score**	
COMPONENT	Source code and design files	3	
	Licences	3	3
ECOSYSTEM	Design tools	3	
	Toolchain	3	
	Software ecosystem	3	
	Firmware	3	3
INFRASTRUCTURE	Processes	3	
	Replicability	3	
	Documentation	3	
	Example code	3	3
FINAL SCORE	**3**		

In Figure 13 the score for USB Armory considering the subcomponent IMX6 is shown. The mean of the scores in the vector is slightly lower than the previous one, because we took into account the subcomponent IMX6, which is not an open source solution. However, the final rounded score doesn't suffer from that, and remains the highest possible (3).



Score considering the subcomponent IMX6											
Score USB Armory	3	3	3	3	3	3	3	3	3	3	
Score IMX6	2	0	2	3	2	2	1	1	1	3	
Final USB Armory	2,75	2,25	2,75	3	2,75	2,75	2,5	2,5	2,5	3	
	2,5		2,81				2,63				
	2,65										**3**

Figure 13: Scoring USB Armory considering the subcomponent IMX6.

We conclude that the model that we propose in this work is valid to analyse most common cases. The scoring inheritance allows us to evaluate the final product according to the score of all the elements of the system. However, some improvements can be introduced to enhance the evaluation and to support different or unusual scenarios.

2.8 Considerations about this Scoring System

Our scoring system offers valuable insights in the context of open source hardware, such as the capability to define a hardware component as part of a specific view and the possibility to compare two components in the same view on their open sourceness. However, it has the potential for improvement and may not encompass all possible scenarios.

One possible improvement involves the enhancement of the scoring criteria and the properties defined. Currently, the properties proposed in this document can cover most of the main aspects of an hardware design; however, according to further research, more or different properties may be identified, in order to provide support for novel scenarios. The same reasoning applies to the property categorizations.

Another limitation of the scoring system is related to the granularity of the scores. In an attempt to provide an overview of the main properties of the entire system, rounded scores are often utilised. This approach allows evaluators to visualise an overview of the entire system, without requiring to focus on small (or irrelevant) details to balance scores in search of perfect consistency across different analyses. Moreover, such an approach permits an easier interpretation for the non-experts, who might not have instruments or interest in understanding nuances of the analysis that are too technical.

On the other hand, having a rounded scoring system implies the loss of some information. By rounding, certain distinctions among different properties are inevitably omitted. This approach potentially limits the depth of analysis and understanding. This rounding can result in the merging of scores that may have slight variations, blurring the distinctions between them. As a consequence, the final scores of the analysis tend to be flattened towards mean values (e.g.: scores equal to 1 or 2). For example, according to the documentation available at the time of writing, it is not clear whether the Raspberry Pi4 provides hardware support for secure boot, as opposed to the IMX6, which clearly supports the Secure Boot. Nevertheless, the Raspberry Pi4 obtains good scores for all the properties, since specific topics are not individually classified.

However, when thinking about the actual distribution that our scoring system should model, it is natural that most evaluated hardware should score either 1 or 2; indeed, the extremes of 0 and 3 have been designed with the goal of modelling, respectively:

- completely closed-source projects, which are very frequent, but not the target of our evaluation;

- completely open source projects on most properties, which are rare.

In practical terms, if we do not consider completely custom solutions and we focus on system designed to be publicly used, by performing the evaluation over a large number of parameters it is difficult to encounter a product that obtains a maximum or a minimum score over all the three categories defined by our model: component, ecosystem, infrastructure.

Another aspect that might be considered is the *verifiability* property. This property provides the possibility of an evaluation about the components adopted in a system, with different levels of granularity up to formal verification. Open source systems promote a transparent, collaborative and inclusive approach. In order for a system to be considered transparent, it is necessary that anyone with the necessary skills and expertise can review the source code, design files and up to hardware components to assess the system's integrity, reliability, and adherence to established standards. The verifiability property provides important information about how open a system can be considered. However we think that the verifiability property is not fitting properly in the current model, since it requires the evaluator to deeply analyse and understand the material provided by the manufacturer in order to assign a coherent and fair score. Some aspects of verifiability will be considered in the other work packages of the project.

In conclusion, our scoring system provides an improvement with respect to the current situation (no universal definition/scoring system available) and it is robust to many different typical scenarios, as shown in the previous Sections. As any other scoring system, it is susceptible to abuse aimed at making it ineffective, as for the so-called Cobra Effect. The original concept derives from a historical anecdote involving a failed attempt to control a cobra population. The Cobra Effect in scoring systems highlights how well-intentioned solutions can lead to unexpected and undesirable outcomes. In practical terms, a manufacturer may be interested in maximising the scores without providing support for the core values of the open source design concept. These fallacies can occur in the real world. For example, a recent case involves a well-known graphic card manufacturer. The latter was recently at the centre of a debate for claiming to provide open source drivers. However, prior to release to the community, proprietary technologies were moved from the driver to the device's closed firmware. From our point of view, mitigating such cases requires foresight, transparency, and ongoing evaluation to ensure fairness and meaningful results.

Chapter 3 Definition of the Trusted Life Cycle

3.1 Overview

After having discussed a definition and scoring method for open source hardware, we now proceed with analysing the core contribution of our research reported in this document, that is the definition of the Trusted Life Cycle (TLC) for ORSHIN including its phases and process requirements.

We start our journey with introductory considerations on requirements, then move onto analysing process-oriented international standards, and afterwards we make our proposal for the definition of the TLC phases and requirements.

The development of a secure component relies on two pillars:

- A set of requirements and best practices for the technologies used to provide the component with a certain level of security and privacy. We call these **product requirements**.
- A set of requirements and best practices for structuring the development process itself, the security of the development environment, and any process/procedure not part of, but related to, the development of the secure component. We call these **process requirements**.

Some cybersecurity-focused standards are focused on the process (for example, ISO 27001), while others are targeting the product (for example, ISA/IEC 62443-4-2). Many standards include a mixture of these types of requirements, and do not make a clear distinction (this is the case, for example, of NIST SP 800-53). Our goal is to explore the definition of a Trusted Life Cycle for secure open source hardware, specifically focusing on process-related aspects for Task 2.1.

Given the literature on process requirements, we start from existing knowledge, and try to shift towards the relatively new and unexplored context of open source hardware. In fact, most existing cybersecurity standards either come from the world of IT systems, and therefore focus almost exclusively on software requirements, or recently from the world of Industrial Automation Control Systems (IACS) and Internet of Things (IoT) systems, which have some hardware-related aspects. However, even when standards are related to IoT systems, they lack the depth and specificity to accurately model low-level hardware development, and any reference to open source.

Back to our initial goal and given the above considerations, we make the remarks that follow.

1. We wish to preserve the valuable knowledge from public documents and repositories, international standards and regulations. Therefore, we start from a review of the state-of-the-art, gathering:
    a. Requirements that apply to a generic secure life cycle, regardless of its technological nature. This is the case, for example, of governance requirements and cryptography best practices.
    b. Requirements that do not specifically apply to the context of open source hardware, but which can provide valuable content if adapted or partially rewritten.
    c. Knowledge that is specific to the worlds of hardware and open source, but does not yet possess the status of "requirement" for any secure life cycle.

    We build on this information to produce content that is coherent with existing knowledge on secure life cycles.
2. We provide definitions for the phases of our Trusted Life Cycle for open source hardware, specifically:
    a. Explaining what is a life cycle and how it becomes *trusted* as a result of the composrition of various, adequately chosen security requirements.
    b. What peculiarities that are specific to hardware design and development must be considered, and how they impact the definition of the life cycle and its phases.

Our work on this topic is articulated in the next Sections. We first explore the state-of-the-art (Section 3.2 - State-of-the-art) and review some of the main cybersecurity standards that provide valuable input with respect to secure life cycles and process requirements (Section 3.3 - Review of Main Cybersecurity Standards for Process Requirements). Afterwards, we choose a promising source as a starting point for drafting the requirement of our Trusted Life Cycle (Section 3.4 - Definition of the Trusted Life Cycle Phases), which is then adapted in order to draft a first version of the ORSHIN Trusted Life Cycle process requirements (Section 3.5 - Process Requirements for the ORSHIN Trusted Life Cycle).

3.2 State-of-the-art

The life cycle of a system refers to the complete sequence of stages and activities that the system undergoes, from its creation to its eventual retirement. It encompasses design, development, implementation, operation, and disposal, providing a structured framework for managing and maintaining the system throughout its entire lifespan.

The literature is rich of definitions and examples for the concept of Secure Development Life Cycle (SDLC). Under this term are grouped a series of procedures and best practices for trying to guarantee that a development process has the higher chances of producing, as output, a product that has a good starting security posture, and for which any security defect discovered after release will be handled appropriately.

The fundamental idea behind the adoption of a SDLC is that of *security-by-design*, namely the idea that security should be integrated in development practices from the beginning of a product's life cycle (i.e., the definition of the functional context and of the requirements) to its end (i.e. when the product is decommissioned and retired from the field).
This concept is opposed to the wide-spread bad habit of *security-after-the-fact*, i.e. considering it as a step of the product life cycle, typically late in development and close to release.

To understand why the *security-after-the-fact* approach is dangerous, consider the topic of testing: if security-related tests begin only after development, it is easy to see how issues can have unforeseen impact, requiring changes in design and implementation choices that were supposed to be consolidated (we refer to them as *milestones*).

An example of such a milestone is the final choice of hardware that will compose the product, constituting its Hardware Bill Of Materials (HBOM, see Section 4.3 State-of-the-art: Component Inventory for more details). Typically, changing the HBOM after it is approved will range from costly (if the wrong components have been already ordered) all the way to impossible (if production has already happened), with related consequences on the business and of the security of the product.

A *threat model* is a conceptual representation or framework that identifies and evaluates potential threats, vulnerabilities, and attack vectors within a system or application. It helps understand the security risks, prioritise mitigation efforts, and guide the development of effective security measures to protect against potential threats.
If the security requirements are clear from the beginning, a threat model is made accordingly, and the product design conforms to such a threat model, the selection of components for the HBOM will be easy, whereas if these activities are performed erratically, it is more probable that components with the wrong features will be selected.
To summarise, *security-by-design* is an approach that allows distributing the risk of such a mistake over time, with multiple checkpoints that allow to catch human error, whereas *security-after-the-fact* concentrates all the risk in single points in time, making it possible for single high-impact decisions to be taken without justification.

A typical composition of a Secure Development Life Cycle for a hardware product, therefore, will typically touch the following definitions:

- Product's operating context.
- Security requirements.
- Product design best practices.
- Implementation best practices.
- Testing types and related best practices.
- Vulnerability management process.

- Maintenance operations (e.g., definition of procedures for the release of security updates).
- Cybersecurity product information
- Procedure for the retirement of the product from the field.

Most of these aspects are not recent, and can benefit from consolidated knowledge coming from the domain of IT cybersecurity. Nevertheless, the definition of comprehensive approaches for the secure development life cycle is more recent, especially in the relatively newer contexts of Industrial IACS and Internet of Things (IoT).

One of the first works to extensively cover the concept of Secure Development Life Cycle is [Howard 2006]. The book describes Microsoft's history and choices with respect to their implementation of an internal Secure Development Life Cycle. Microsoft SDL has been first developed internally, and made available publicly since 2004, and remains one of the standard approaches for implementing a Secure Development Life Cycle.

Since then, alternative models and frameworks for implementing secure life cycles have been proposed. Notable examples include the Software Assurance Maturity Model (SAMM) [OWASP SAMM]; the Building Security in Maturity Model (BSIMM); and the Comprehensive, Lightweight Application Security Process (CLASP), but also the Secure Development Life Cycle described by IACS cybersecurity standard ISA/IEC 62443, particularly by its section ISA/IEC 62443-4-1.

An article from 2010 covers the topic of actual adoption of SDLCs in companies [Geer 2010], and provides insight of real-world difficulties in conforming to these relatively new methodologies.

3.3 Review of Main Cybersecurity Standards for Process Requirements

Nowadays there are several cybersecurity standards to certify the security of connected devices, and the number is growing. All of them require that some processes are in place to guarantee the security properties of the final product.

We made a selection among the most relevant generic cybersecurity standards, prioritising the ones with a context closer to the one of ORSHIN; we analysed their characteristics to find their commonalities and differences, and also their gaps in terms of coverage of the domains of hardware and open source.

The standards that we have considered are the following:

- ISA/IEC 62443
- ISO 27001
- NIST SP 800-53
- ENISA Good Practices for Security of IoT
- CSA IoT Security Controls Framework
- ETSI EN 303 645

Among these, ENISA Good Practices for Security of IoT, CSA IoT Security Controls Framework and ETSI EN 303 645 provide requirements and recommendations specific for the security of IoT devices and systems, therefore we mostly consider their requirements as a starting point for drafting the requirements of the ORSHIN Trusted Life Cycle.

In the following Section, we describe the standards listed earlier and present our considerations regarding their applicability in the ORSHIN context.

3.3.1 *ENISA Good Practices for Security of IoT*

The “ENISA Good Practices for Security of IoT” is a document authored by ENISA in 2019, the European Union Agency for Cybersecurity, collecting good practices for IoT security. The document primarily focuses on software development guidelines for ensuring the security of IoT products and services throughout their entire life cycle.

Although these guidelines are specific to IoT devices, which include software and hardware components, the ENISA document addresses mainly the security of the software components (including firmware, communication protocols, operating systems, and device drivers). Indeed, the purpose of the document is to define security requirements for the Software Development Life Cycle (SDLC). This includes the definition of security measures that apply to the entire IoT ecosystem (e.g., communications, networks, and cloud) to strengthen the security of the development process.

The phases of the SDLC identified by ENISA are the following.

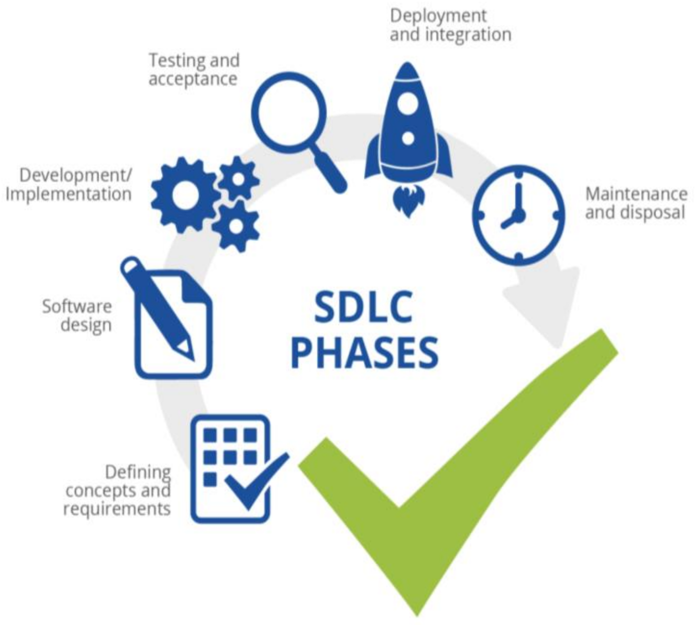

Deployment and integration
Testing and acceptance
Development/ Implementation
Maintenance and disposal
Software design
SDLC PHASES
Defining concepts and requirements

Figure 14: SDLC phases defined in ENISA Good Practices for Security of IoT.

1. **Requirements.** This phase involves the definition of business and functional requirements, which will be the starting point for the definition of technical specifications that will guide the later stages. This allows to implement security by design principles, having in mind that security does not come as an afterthought.

2. **Software Design.** During this phase the architecture and the design of the IoT device are outlined. This includes the definition of system specifications (how the IoT solution will work), starting from the business and functional requirements.

3. **Development/Implementation.** This phase includes the part of coding (and, in particular, secure coding), starting from the specifications defined in the previous phase.

4. **Testing and acceptance.** This phase encompasses all necessary steps to ensure that the developed software meets the identified requirements and design principles from earlier phases. It includes both automated and manual testing of the source code and running software.

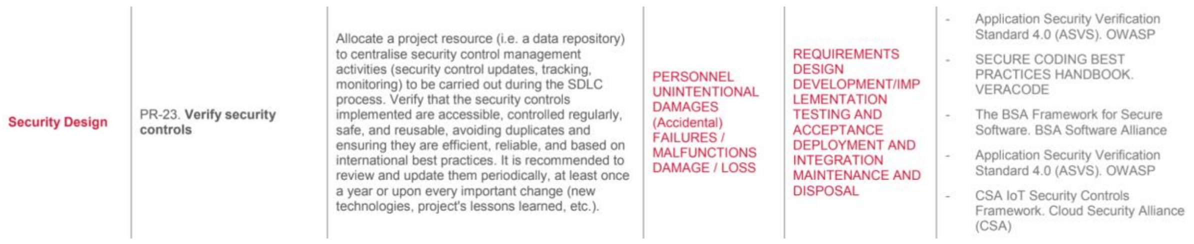
5. **Deployment and integration.** This phase involves integrating all essential software components into the production environment and deploying them.

6. **Maintenance and disposal.** In this phase, continuous maintenance is performed to ensure the availability and integrity of the IoT device’s functionalities. This includes, for example, over-the-air update procedures and security maintenance (e.g., vulnerability continuous monitoring, penetration tests, threat detection and response, etc.). In addition, secure disposal is defined, to be applied when the IoT software becomes obsolete to preserve privacy management.

The phases outlined for the SDLC are similar from those identified in the ORSHIN project for the definition of the Trusted Life Cycle (TLC), which will be detailed in Section 3.4 - Definition of the Trusted Life Cycle Phases. However, ORSHIN's TLC phases are designed to include the steps specific to the implementation of open source hardware, which this SDLC does not take into account.

Overall, the ENISA framework is quite comprehensive and not overly detailed. It defines high-level requirements and associates them with the SDLC phases, standard references (e.g., ISA/IEC 62443) and threats for which the requirements are a countermeasure.

The security requirements are divided into three categories, namely: people, process, and technology. In particular, process-level requirements are defined, which are particularly useful as a reference for defining process requirements for ORSHIN's TLC. An example of a SDLC process requirement is in the following Figure 15.



Security Design	PR-23. **Verify security controls**	Allocate a project resource (i.e. a data repository) to centralise security control management activities (security control updates, tracking, monitoring) to be carried out during the SDLC process. Verify that the security controls implemented are accessible, controlled regularly, safe, and reusable, avoiding duplicates and ensuring they are efficient, reliable, and based on international best practices. It is recommended to review and update them periodically, at least once a year or upon every important change (new technologies, project's lessons learned, etc.).	PERSONNEL UNINTENTIONAL DAMAGES (Accidental) FAILURES / MALFUNCTIONS DAMAGE / LOSS	REQUIREMENTS DESIGN DEVELOPMENT/IMPLEMENTATION TESTING AND ACCEPTANCE DEPLOYMENT AND INTEGRATION MAINTENANCE AND DISPOSAL	- Application Security Verification Standard 4.0 (ASVS). OWASP - SECURE CODING BEST PRACTICES HANDBOOK. VERACODE - The BSA Framework for Secure Software. BSA Software Alliance - Application Security Verification Standard 4.0 (ASVS). OWASP - CSA IoT Security Controls Framework. Cloud Security Alliance (CSA)

Figure 15: An example of process security requirement defined in ENISA Good Practices for Security of IoT.

ENISA Good Practices for Security of IoT has been chosen as the starting point for defining the process requirements for ORSHIN's TLC. It constitutes a baseline for the definition of the TLC phases in the ORSHIN context, but it lacks specific requirements for open source hardware.

3.3.2 *ISA/IEC 62443*

ISA/IEC 62443 is a series of standards developed by ISA99, i.e., the ISA (International Society of Automation) committee for Industrial Automation and Control Systems Security, and produced by IEC (International Electrotechnical Commission). It is a structured and extended standard, composed of 14 work documents, and divided in four tiers, i.e., General, Policies & Procedure, System, and Component. These tiers, shown in the image below, characterise different aspects of an organisation's security.

ISA/IEC 62443 introduces an approach for building secure systems based on the composition of secure components, and provides both procedural and technical requirements for implementing such a model.

Specifically, Tier 3 focuses on the system level, and the work document ISA/IEC 62443-3-3 contains system-scoped requirements.

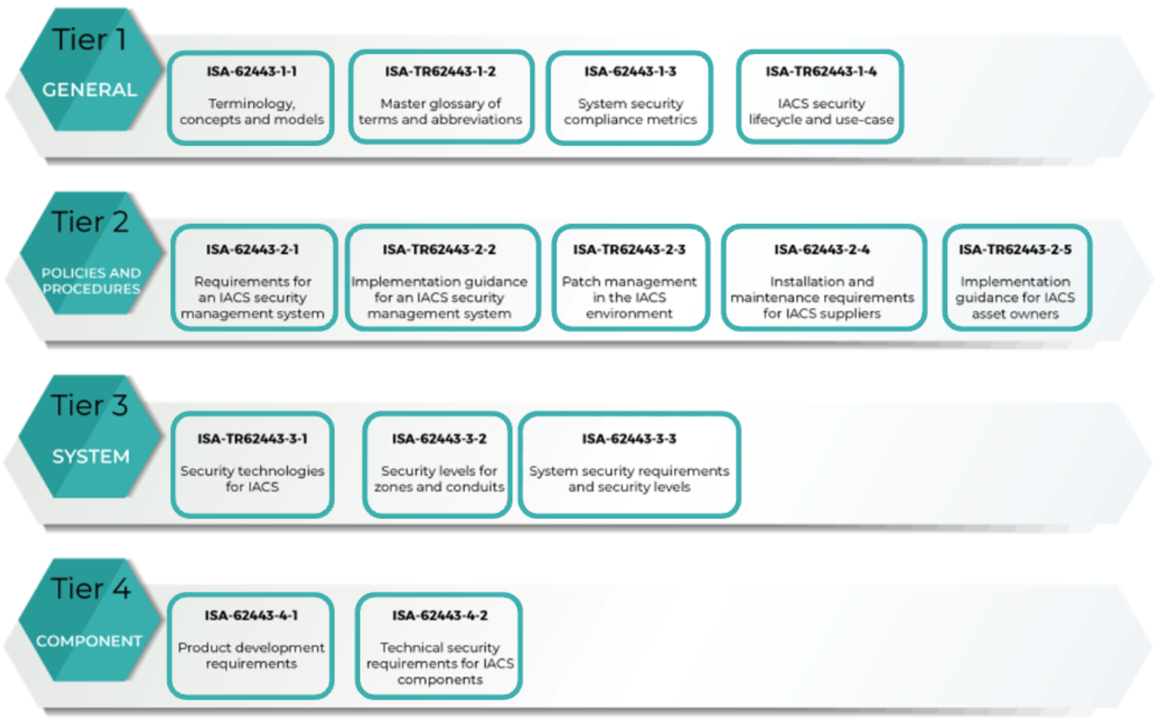

Tier 1
GENERAL
ISA-62443-1-1
Terminology, concepts and models
ISA-TR62443-1-2
Master glossary of terms and abbreviations
ISA-62443-1-3
System security compliance metrics
ISA-TR62443-1-4
IACS security lifecycle and use-case
Tier 2
POLICIES AND PROCEDURES
ISA-62443-2-1
Requirements for an IACS security management system
ISA-TR62443-2-2
Implementation guidance for an IACS security management system
ISA-TR62443-2-3
Patch management in the IACS environment
ISA-62443-2-4
Installation and maintenance requirements for IACS suppliers
ISA-TR62443-2-5
Implementation guidance for IACS asset owners
Tier 3
SYSTEM
ISA-TR62443-3-1
Security technologies for IACS
ISA-62443-3-2
Security levels for zones and conduits
ISA-62443-3-3
System security requirements and security levels
Tier 4
COMPONENT
ISA-62443-4-1
Product development requirements
ISA-62443-4-2
Technical security requirements for IACS components

Figure 16: The document structure of ISA/IEC 62443.

Tier 4 focuses on the component level, and is divided into two work documents:

1. ISA/IEC 62443-4-1 (Product Security Development Life Cycle Requirements) deals with how a product's development life cycle shall be managed to guarantee that the product's security level can be ensured throughout its lifetime. Specifically, it contains the requirements for implementing a Secure Development Life Cycle; it also defines a maturity model, against which it is possible to evaluate a specific implementation of a SDLC compliant with ISA/IEC 62443-4-1.
   The Tier 4 SDLC requirements are divided into 8 categories, called "Practices":
   a. Security management
   b. Specification of security requirements
   c. Secure by design
   d. Secure implementation
   e. Security verification and validation testing
   f. Management of security-related issues
   g. Security update management
   h. Security Guidelines

   Practise *a* is a category containing mostly high-level, general requirements for proper set up of the SDLC framework; Practices *b* to *h* instead follow the product's life cycle from the beginning to the end.

2. ISA/IEC 62443-4-2 (Technical Security Requirements for IACS components) deals with what technical features the product should contain to meet the user's expectations and needs in terms of the product's capability to respond to threats. Specifically, it contains the technical requirements that a secure component must have in order to reach a certain security level, representative of the resistance of the component against adverse cybersecurity events.

   These levels are formally defined in ISA/IEC 62443, as follows:

Table 14: ISA/IEC 62443-4-2 Security Levels.

Security Level (SL)	Definition
SL 0	No specific requirements or security protection necessary
SL 1	Protection against casual or coincidental violation
SL 2	Protection against intentional violation using simple means with low resources, generic skills and low motivation
SL 3	Protection against intentional violation using sophisticated means with moderate resources, IACS specific skills and moderate motivation
SL 4	Protection against intentional violation using sophisticated means with extended resources, IACS specific skills and high motivation

Requirements in ISA/IEC 62443-4-2 are already mapped to the above Security Levels; for reaching a certain level, a component must implement all the corresponding requirements.

As of today, it is possible to use ISA/IEC 62443-4-2 to evaluate the security of four distinct types of components:

- Embedded devices
- Host devices
- Software applications
- Mobile devices

ISA/IEC 62443-4-2 contains both component requirements that apply to all four categories and requirements that are specific for a particular type of component.

For ORSHIN, ISA/IEC 62443-4-1 is a valuable reference for the definition of Trusted Life Cycle process requirements, while ISA/IEC 62443-4-2 constitutes a good reference for outlining more technical requirements.

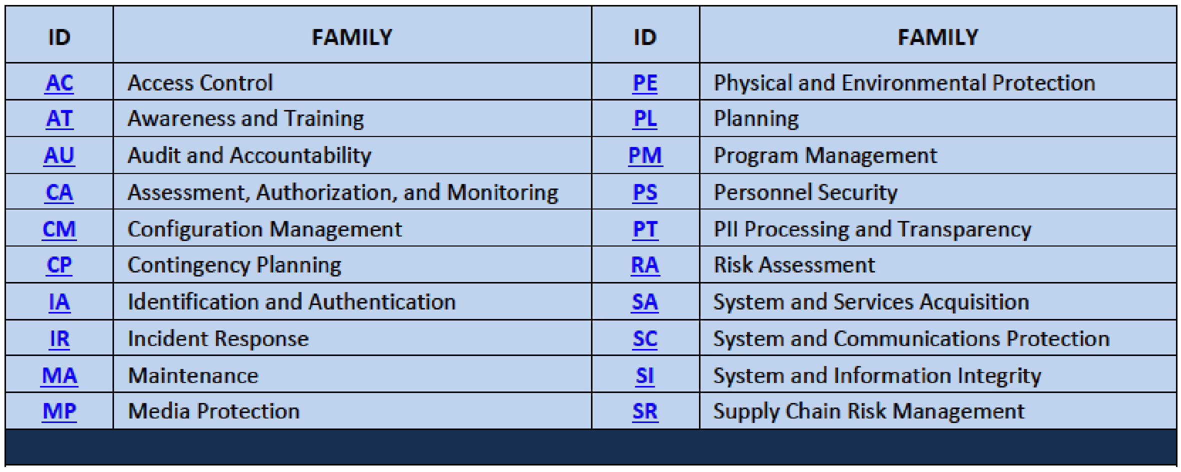
3.3.3 *NIST SP 800-53*

NIST Special Publication 800-53, “Security and Privacy Controls for Information Systems and Organizations”, is a widely recognized standard defined by the National Institute of Standards and Technology (NIST), that provides a comprehensive set of security and privacy controls for federal agencies (and other organisations) in the United States.

NIST SP 800-53 was initially published in 2005 as “Recommended Security Controls for Federal Information Systems” and revised periodically thereafter. With the last revision (Revision 5), published in 2017 and updated in 2020, the word “federal” was removed to indicate that the regulations may be applied to all organisations, not just federal ones.

A *control* is defined as a measure that modifies or maintains risk, including processes, policies and practices. NIST SP 800-53 defines a risk-based framework that guides the management and implementation of security and privacy controls, to protect information systems and organisations from a diverse set of threats. Security and privacy controls described in this standard have a well-defined organisation and structure and address many different areas.

The controls are organised into 20 families, each distinguished by a two-character identifier (e.g., AC for Access Control). The following Table provides a schematic representation of the control families.



ID	FAMILY	ID	FAMILY
AC	Access Control	PE	Physical and Environmental Protection
AT	Awareness and Training	PL	Planning
AU	Audit and Accountability	PM	Program Management
CA	Assessment, Authorization, and Monitoring	PS	Personnel Security
CM	Configuration Management	PT	PII Processing and Transparency
CP	Contingency Planning	RA	Risk Assessment
IA	Identification and Authentication	SA	System and Services Acquisition
IR	Incident Response	SC	System and Communications Protection
MA	Maintenance	SI	System and Information Integrity
MP	Media Protection	SR	Supply Chain Risk Management

Figure 17: NIST SP 800-53 security and privacy control families.

Each family comprises base controls, that serve as foundational measures, and control enhancements that can be implemented to augment the functionality and specifications of the base controls, strengthening their effectiveness.

This framework is comprehensive and granular, and includes multiple controls. In fact, it can be considered to be a baseline for the definition of more specific frameworks tailored to the particular requirements of the context in which it is applied, rather than being strictly enforced for every control. Many organisations use it as a starting point for developing their own security and privacy programs.

The catalogue of requirements offered by this framework is also flexible and offers various levels of detail for each family of controls. This adaptability allows organisations to establish both high-level and highly-detailed specifications (e.g., using control enhancements). In terms of high-level controls, NIST SP 800-53 includes requirements that are comparable to those found in standards such as ISO 27001, which can be classified as organisational controls. On the other hand, it includes technical controls that delve into specific aspects of information security. For instance, within the family of SA (System and Services Acquisition), are defined enhancements for Developer Testing And Evaluation control concerning Static Code Analysis, Threat Modeling And Vulnerability Analyses, Independent Verification Of Assessment Plans And Evidence, Manual Code Reviews, Penetration Testing, or within the family of SI (System and Information Integrity) are defined enhancements for Software, Firmware, and Information Integrity control for Verify Boot Process and Protection Of Boot Firmware.
For ORSHIN, NIST SP 800-53 provides value as a possible source and reference for requirements, showing a comprehensive approach that embraces many different aspects of IT security.

3.3.4 *ISO 27001*

ISO/IEC 27001 is an international standard for information security management systems (ISMS) developed by ISO, the International Organization for Standardization. It provides a framework for organisations to establish, implement, maintain, and continually improve their information security management.
ISO 27001 was first published in 2005, then revised in 2013 and 2022 to better accommodate the changing information security challenges. The current version is called ISO 27001:2022 [ISO 27001].

The high-level requirements of the standard are addressed in 7 clauses, schematised in the following Table.

Table 15: The 7 clauses from ISO 27001.

Clause	Explanation
Context of the organisation	The first step is to identify the organisation's issues related to information security and to determine interested parties' expectations and requirements, with the aim of assessing the scope of the ISMS and establishing it.
Leadership	Top management should support the importance of the ISMS by ensuring its integration and effectiveness. Management should establish an information security policy and assign roles and responsibilities to manage the ISMS.
Planning	This clause is about planning of actions to address risks and opportunities. Risk assessment and treatment processes should be defined by determining the necessary controls. In order to protect the information asset, information security objectives should be established and planned.
Support	It is important to provide an adequate level of resources, competences and employee awareness for the implementation and maintenance of the ISMS. Everything related to the ISMS should be documented and kept updated.
Operation	This clause is about implementing controls to ensure the outcomes of the ISMS are achieved. Risk assessment and treatment should be adequately implemented.
Performance evaluation	The organisation should determine which and how data, processes and controls need to be monitored and measured to evaluate the effectiveness of the ISMS. Internal audits should be conducted and planned, and management reviews should be performed.
Improvement	The effectiveness of the ISMS should be continually tested and improved, and the system should be corrected whenever an unconformity occurs.

ISO 27001 is related to ISO 27002, or Annex A, which defines a checklist of generic information security controls designed to be used by organisations within the context of an ISMS. After a risk assessment process, the organisation should determine which of the reference ISO 27002 controls are relevant based on the identified risks. The chosen measures are then described in a central document for the ISMS called the Statement of Applicability, where inclusion or exclusion of reference controls need to be justified, together with presence or lack of their implementation.
The current version of the document, called ISO 27002:2022 [ISO 27002], presents significant changes with respect to the 2013 version: the list of controls was reconstructed and compacted from 114 to 93 controls, structured in the following four categories:

- *Organisational controls* (37 items) define rules and behaviours regarding users, devices and systems, including organisational information policies, cloud service use and asset use.
- *People controls* (8 items) provide security knowledge and awareness to employees and relevant third parties. This type of controls concern remote work, screening, confidentiality and non-disclosure measures.
- *Physical controls* (14 items) include maintenance, facilities security, media storage and security monitoring.

- *Technology controls* (34 items) are those implemented in informative systems utilising software and hardware components, and concern authentication, encryption and data leak prevention.

In addition to the category, for each control are specified attributes which have the aim to gain different perspectives on the controls and allow filtering. ISO 27002 suggests the following 5 attribute classes with respective attribute values:

- Control type: Preventive, Detective or Corrective.
- Information security properties: Confidentiality, Integrity and Availability.
- Cybersecurity concepts: Identify, Protect, Detect, Respond and Recover.
- Operational capabilities such as Governance, Asset Management, Information Protection, and others.
- Security domains: Governance and Ecosystem, Protection, Defence and Resilience.

Ultimately, ISO 27001 is a comprehensive standard that not only equips companies with essential knowledge to safeguard their valuable information but also allows them to obtain certification, demonstrating their commitment to data protection to customers and partners. Similarly, individuals can enhance their professional credentials by becoming ISO 27001 certified through training and examination, showcasing their expertise in implementing or auditing Information Security Management Systems.
The applicability context of this standard is quite distant from the ORSHIN context, and for this reason ISO 27001 does not represent a good starting base for the process requirements of the ORSHIN TLC. However, it provides a solid reference for governance-oriented requirements, and in general for the integration of risk management topics.

3.3.5 *CSA Security IoT Controls Framework*

The Internet of Things (IoT) Security Controls Framework, developed by the Cloud Security Alliance (CSA), provides a set of security controls to mitigate risks associated with an IoT system, that incorporates multiple types of connected devices, cloud services, and networking technologies. It is applicable to many IoT domains, ranging from systems processing only “low-value” data with limited impact potential to highly sensitive systems that support critical services.

The most recent version of the framework is version 2. The version 3 is planned for the future with some improvements such as the definition of the ENISA Guidelines for Securing the Internet of Things.

The framework is provided in an Excel document that includes the following pages:

- Domain Definitions: the list of domains covered by the framework's controls, along with a short name, a list of sub-domains and a definition.
- IoT Controls Matrix: the table containing all controls of the framework. For each control are specified:
  - An identifier (Control ID).
  - The control domain and sub-domain associated.
  - The related identifiers from the CSA Cloud Controls Matrix (CCM), a cybersecurity control framework for cloud computing.
  - The impact levels on Confidentiality, Integrity, Availability.
  - Some additional directions (supplementary information detailing special requirements, explanations of terms, etc.).
  - Some references.
  - An implementation guidance (e.g., how organisations can implement the controls and the frequency with which each control measure should be enacted).
  - The architectural allocations (at which level the control can be applied, that can be one or more among Device, Network, Gateway and Cloud Services).

Alongside the Excel document, the CSA Guide to the IoT Security Controls Framework is provided to help users and organisations in understanding and applying the framework effectively.

The IoT Security Controls Framework serves as an invaluable asset for designers and developers responsible for crafting secure IoT systems, as well as individuals evaluating the effectiveness of such systems. This tool enables designers and developers to consistently assess the security measures implemented in their IoT projects, allowing them to ensure the robustness of their implementation at every stage of the development life cycle. By providing a comprehensive evaluation, the framework guarantees compliance with industry-approved standards and recommended practices for IoT systems.

For ORSHIN, this source offers a good reference as it applies to the IoT context, which includes both hardware and software elements.

3.3.6 *ETSI EN 303 645*

ETSI EN 303 645 is a standard for cybersecurity of IoT devices. It provides a set of requirements and recommendations for the security of IoT devices and systems. It constitutes a very high-level guide, lacking specific technical content.

The standard is a continuation of the UK initiative of 2018 about the definition of a code of practice for IoT cybersecurity [Code of Practice for Consumer IoT Security]. The UK initiative started with the definition of the involved stakeholders: device manufacturers, IoT service providers, mobile app developers and retailers. The initiative also identifies what are the types of IoT devices that are included in this activity, like connected children's toys, connected safety products, smart camera, smart TVs, smart speakers, home automation, fitness trackers and similar devices. The initiative collected open comments from whoever had an interest in commenting and presenting information. The activity is summarised in the final report. This initiative has been recognized as well structured and as a tradeoff between the technical standards that are too focused on the details of a specific sector and the needs of the end users that are not represented in the technical standards.

ETSI defines some aspect that we summarise in the following points:

- *No universal default password.* If passwords are used as a method of authentication they should be defined by the user, otherwise securely implemented pre-installed password unique per device should be used. Whenever other forms of authentications are in force, they shall use best practice cryptography and implement countermeasures against brute-force attacks via network interfaces.
- *Implement a means to manage reports of vulnerabilities*. The manufacturer shall make a vulnerability disclosure policy publicly available to specify the process for reporting issues. It is crucial to build the so-called Software Bill of Materials (SBOM), a list of third party components and the version used, that allows manufacturers to continually monitor for product security vulnerabilities.
- *Keep software updated.* The device should check periodically for updates and verify their authenticity and integrity by using best practice cryptography. Configurable automatic update mechanisms should be used. For constrained devices that cannot have their software updated, the product should be isolable and the hardware replaceable.
- *Securely store sensitive security parameters.* Secure storage mechanisms should be used. The implementation of hard-coded security parameters should resist physical and software tampering. Lastly, any critical security parameter used for integrity and authenticity checks of software updates and for communication with associated services shall be unique per device and securely produced.
- *Communicate securely.* The device shall use reviewed implementations of best practice cryptography, and the cryptographic algorithms should be updatable. Authentication is required for security-relevant device functionality and for communication of security parameters that happens via a network interface.
- *Minimise exposed attack surfaces.* The network interfaces of the device shall minimise the disclosure of security-relevant information, and they should be disabled when unused. Code, hardware physics interfaces and software privileges should be minimised.

- *Ensure software integrity.* The IoT device should verify its software using secure boot mechanisms, e.g., with a hardware root of trust. If an unauthorised change is detected to the software, the device should alert the right stakeholder.
- *Ensure that personal data is secure.* The confidentiality of personal data transiting between the device and a service, especially associated ones such as cloud services, should be protected with best practice cryptography. All external sensing capabilities of the device shall be clearly documented.
- *Make systems resilient to outages.* The IoT device should stably connect to networks, it should remain operating in the case of a loss of network access and cleanly recover in the case of a loss of power.
- *Examine system telemetry data.* If data such as usage and measurement data is collected from the device or associated services, it should be examined for security anomalies.
- *Make it easy for users to delete user data.* There should be a simple functionality to erase data from the device and associated services, which should provide clear confirmation of the deletion.
- *Make installation and maintenance of devices easy.* The manufacturer should provide users with guidance on how to securely set up their device and how to check whether the set up is correct.
- *Validate input data.* The device software shall validate data input via user interfaces, or transferred via Application Programming Interfaces (APIs), or between networks in services and devices.

Along with the ETSI 303 645 there is also a very important document, the ETSI TS 103 701. This document puts in perspective how a device manufacturer can perform a series of tests to demonstrate that the device is in line with the requirements of the 303 645. In the case of the baseline requirement it is possible to perform a self assessment of the cybersecurity posture of the device.

It is possible to see that the standard is a mix of technical requirements (like no default password) and procedures (like “keep software updated”).

The very positive aspect of the standard is the pragmatic approach in defining tangible results that the IoT device manufacturer should reach and how they can be demonstrated.

This standard has been also adopted by Singapore [CLS CSA Singapore], indeed the Cybersecurity Singapore Agency defined a labelling scheme for helping consumers in identifying the level of security of IoT devices.

For ORSHIN, we make sure that the definition of the TLC requirements does not contradict the best practice guidelines of ETSI EN 303 645.

3.4 Definition of the Trusted Life Cycle Phases

We now define the Trusted Life Cycle for ORSHIN secure open source hardware. The concept of Trusted Life Cycle (TLC) is comparable to the one of Secure Development Life Cycle, in that it provides a methodology for secure development. In particular, the goal of the TLC is to describe a *systematic* and *generic* approach for designing and developing *secure* and *trustable* hardware devices with open source components.

One key point for the Trusted Life Cycle is the concept of "trust", which refers to the possibility for manufacturers to build secure hardware without necessarily dealing with the cybersecurity architecture and specification of every single component used in the design, but rather relying on a chain of trust that leads to the adoption of reusable secure hardware through the multiple-step and multiple-stakeholder process of building hardware. Even though requirements of verifiability about 3rd-party developments are already present in most SDLCs, the TLC takes this concept one step further, exploring the generic "1st-party"/"3rd-party" distinction and decomposing hardware designs into their layers, down to the lowest abstraction level, and providing a way to evaluate security for hardware developments of any kind. Openness of designs and source code plays a crucial role in this evaluation, allowing for more efficient collection and distribution of the details of a hardware component that impact its security, thus facilitating the validation of trust. Although the ORSHIN TLC requirements potentially fit any type of hardware and embedded development, the performance of the TLC is maximised when applied to open source components.

As a result of our review of existing standards for process requirements, our opinion is that none describes a development life cycle that fully matches our expectations in this regard. However, valuable common themes of consolidated SDLC knowledge can be reused, and we make sure to build on such information.

We define the ORSHIN Trusted Life Cycle as a set of process requirements, which can be used to derive product requirements and practically define an instantiation of the Trusted Life Cycle for a particular single piece of hardware, or for more complex products.

The first building block of the TLC are its phases. For the development of ORSHIN hardware, we identify the following *seven* phases (Figure 18).

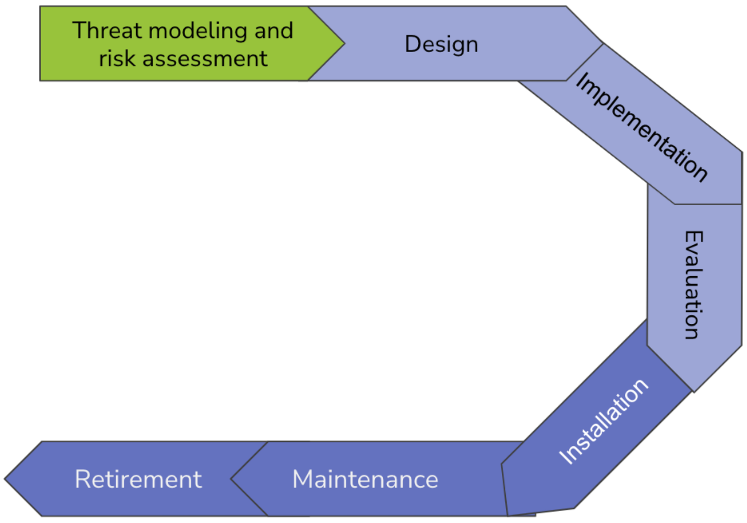


Figure 18: Seven phases of the Trusted Life Cycle.

3.4.1 ***Threat Modelling and Risk Assessment***

Being built on a security-by-design approach, the ORSHIN Trusted Life Cycle must have its roots in threat modelling and risk assessment.

These activities allow for the definition of *security and privacy requirements*, a set of product requirements for guaranteeing certain cybersecurity properties.

The ways in which these requirements can be gathered are various:

1) They may come from non-controllable sources: for example, the commissioner may impose certain requirements on the entire system, thus including the product under development.
2) They may come from adequately selected sources: this is the case of requirements and control lists provided by international standards and other guidance documents.
3) They may be extracted as a result of ad hoc cybersecurity analyses: for instance, threat enumeration strategies such as STRIDE [STRIDE] may yield interesting threats when run on an accurate model of the system; mitigation of such threats may result in the introduction of new security requirements.

Regardless of their source, the common denominator of all security requirements is that their selection should always be validated by threat modelling and risk assessment activity. The revised list of security requirements is the output of this first phase, and constitutes the foundation of the entire Trusted Life Cycle.

3.4.2 *Design*

In this phase, the product hardware is designed, starting from the specifications and functional requirements, and taking into consideration the output of the previous phase, i.e. the security requirements that are to be implemented.

The design activities are not strictly limited to the hardware only, but involve any other relevant product component (e.g., if the product being developed is an IoT device, the firmware will go through its own design phase as well).

The output of this phase is the complete product design in the form of artefacts that can be used for implementation. These include, as an example, hardware schematics, hardware and firmware design files, system diagrams, sequence diagrams for the design of protocols.

From a cybersecurity perspective, it is crucial that this phase is based on a security-by-design approach, meaning that the design activities follow a series of process requirements and general design best practices to:

- Integrate with the output of the threat modelling and requirement definition activities, making sure that the design does not contradict any of the defined security requirements;
- Output designs that have minimal chance of introducing security issues due to design errors;
- Output designs that facilitate a secure implementation;
- Output designs that facilitate testing and validation activities.

3.4.3 *Implementation*

In the implementation phase, starting from the design, the development of hardware and software/firmware components are carried out. Single parts can be in-house designs, but it is also possible to integrate external custom developments, or pre-developed 3rd party components, to be used off-the-shelf. Eventually, all parts are connected to implement the complete design plan. In this phase, the topic of open source becomes important. While open source components are found everywhere in the software world, from small developments to enterprise applications, in the hardware world the trend is different, with the vast majority of low-level hardware still being composed of proprietary IP designs, and with only slight improvements in the stance of high-level hardware CPU, SoCs, and devices.

As for the design phase, the implementation phase should be based on solid principles and best practices for security, with goals that include, but are not limited to, the following:

- Avoid or minimise the chance of implementation errors such as coding errors in software/firmware;
- Guarantee that the implementation will be coherent with the input from the design phase;
- Include secure components in the development, and keep track of them (see Chapter 4 - Component and Vulnerability Tracking);
- Test individual parts of the implementation as early as possible, developing evaluation strategies and tools together with the product.

3.4.4 *Evaluation*

Starting from the implementation, in the evaluation phase the developed components are tested, both from the functional and the cybersecurity points of view.
For example, one test may assess if power or energy consumption is within a specific budget. For connected devices, it may be checked if communication throughput and latency are acceptable. Security-specific tests aim at validating that:

- The security requirements have been adequately implemented;
- The product is free from known vulnerabilities, which may be introduced by 3rd party components;

- The product resists more thorough security-specific testing, such as fuzz testing, penetration testing, or fault-injection testing (described below), in which the product is tested for hidden security weaknesses.

While in general the evaluation is performed on the outcome of the implementation, it is also possible to evaluate models. In this case, it is possible to call this *presilicon evaluation* for models that simulate the power consumption or that allow the evaluation of fault injection attacks. Some of these types of testing are very context-specific, and some of them are not part of generic Secure Development Life Cycles. For example, fault-injection is crucial for ensuring that the hardware and firmware of an embedded device will have a sufficient level of resistance to adversarial signal injection.

A *fault* is defined as a physical defect, imperfection, or flaw that occurs within some hardware or software component. *Fault injection* can be defined as the validation technique of the dependability of fault tolerant systems, which consists in the accomplishment of controlled experiments where the observation of the system's behaviour in presence of faults is induced explicitly by the introduction (injection) of faults in the system. There have been many efforts to develop techniques for injecting faults into a system, including prototypes or models. These techniques can generally be grouped into five main categories:

- Hardware-based fault injection: involves introducing errors into the system by physically altering the hardware of the system. This can include techniques such as disturbing the hardware with environmental parameters such as heavy ion radiation or electromagnetic interferences, injecting voltage sags or power supply disturbances on the power rails of the hardware or using laser fault injection to modify the values of the pins of the circuit.
- Software-based fault injection (software-implemented fault injection): force errors that would occur in hardware at the software level. This technique aims to reproduce hardware faults in the system through software means to understand how the system behaves under these conditions.
- Simulation-based fault injection: involves introducing errors or faults into high-level models, such as Hardware Description Language (HDL) models, to evaluate the dependability of the system when only a model is available. This method utilises different description languages to target different levels of abstraction, and a cohesive environment is needed to promote compatibility between abstraction levels and integrate validation into the design process.
- Emulation-based fault injection: uses Field Programmable Gate Arrays (FPGAs) to simulate errors or faults in a system and study the circuit's actual behaviour under these conditions. This method is considered an alternative solution to simulation-based fault injection campaigns, as it can speed up the process and provide more accurate results. The technique involves emulating the circuit in the application environment, taking into account real-time interactions. However, it is important to note that the HDL description used for the emulator must be synthesizable.
- Hybrid fault injection: This method combines software-based fault injection with hardware monitoring.

Another way to categorise fault injection methods is by distinguishing between invasive and non-invasive techniques. Invasive techniques are those that leave a noticeable impact on the system, while non-invasive techniques are able to introduce faults without affecting the system's normal behaviour. The challenge with complex and time-sensitive systems is that it may be difficult to eliminate the impact of the testing mechanism on the system, regardless of the type of fault injected. Invasive techniques may cause a permanent effect on the system, while non-invasive techniques aim to minimise their impact and leave the system's behaviour unchanged.

In the context of ORSHIN secure hardware, significant attention will be given to security audits of hardware and software. The evaluation phase provides meaningful feedback information to the design and implementation phases before deployment.

Of particular importance for hardware is the process of *vulnerability analysis* (VA), which for a future device is a multi-step process, split between pre-production and post production/prototyping.

During testing of a prototype, in case of detected vulnerabilities one would go back to a previous step in the design & implementation process. Generally earlier steps of VA are less precise, but are also much less expensive and thus can be repeated with several different designs, while later steps would engage more costly re-design if issues are found. Thus, device manufacturers are interested in detecting as many security issues as possible at early stages of device development.

Hardware and software vulnerabilities vary in nature and may be introduced and detected at different stages of the Trusted Life Cycle. The attack scopes are also different, but hybrid testing is required to efficiently cover the attack surface.

3.4.5 *Installation*

After evaluation embedded devices get deployed and installed, possibly in restricted or constrained environments. For example, it may not be possible to access or recall them physically.

Great care must be taken to ensure that the installation of devices conforms to the threat modelling and the security context intended for them; even the most secure designs and implementations may fail to meet security requirements due, for example, to:

- Improper understanding of the security context - this is the case in which, for example, an IoT device which does not implement any network controls, but rather needs them implemented in the environment to guarantee a certain level of security, is deployed within an insecure network.
- Improper configuration - this is the case in which robust security mechanisms provided by a device (for example, secure boot) fail or are affected by weaknesses due to an incorrect configuration of the mechanisms themselves; in the case of secure boot, for example, this could translate in not correctly isolating production keys from development ones, using keys that are cryptographically weak, and other similar problems related to key generation and management.

To counter these problems, clear guidelines for the installation of secure devices should be made available to the installer, which should ideally be complete with actionable, checklist-style tests. For IoT devices, a section in the product manual dedicated to cybersecurity is advisable.

3.4.6 *Maintenance*

We define the "maintenance" phase as the period between the installation and the retirement of the product, so all the time that the product spends in the field counts as maintenance. After installation, embedded devices may be remotely monitored and maintained in the field, for example monitoring their correct functioning and providing firmware updates. Also, it may be possible for 3rd parties to continuously run specific integrated tests and confirm that the device behaves as specified.
Another crucial topic for the maintenance phase is the continuous monitoring of the product's security; even with an implementation of a secure and trusted development life cycle, it is only possible to guarantee a certain level of security up to the product's release, but the risk of future vulnerabilities discovered in the product and in its subcomponents cannot be eliminated. The continuous monitoring of the product security could be addressed by implementing a vulnerability management process, which is composed of several key items:

- A process for receiving vulnerabilities that are reported on the products. These may come from various sources, for example
    - Users
    - Developers
    - Security researchers
- Processes for reviewing reported vulnerabilities, assessing their impact, and addressing them accordingly
- Active monitoring of public vulnerability databases

A fundamental piece of information that is required for an efficient implementation of the above processes is the complete and accurate knowledge of the product's composition, from a software and hardware perspective. The list of components is referred to as the Bill Of Materials (BOM), which can be divided into its software-only version (Software Bill Of Materials, SBOM) or hardware-only version (Hardware Bill Of Materials, HBOM). Currently, there exist different ways to compile this information, but there is no consensus on a universally preferred BOM standard. Along with the BOM, the information regarding vulnerabilities need also to be tracked.

We discuss these topics in detail in Chapter 4 - Component and Vulnerability Tracking

3.4.7 *Retirement*

When a device reaches the end of its life cycle, it is time for its retirement from active deployment. This phase holds significant importance from a security standpoint and should be considered as the final stage of the Trusted Life Cycle. It is crucial not to overlook the security implications associated with device retirement. Instead, this phase should be approached with utmost care and attention.

During the retirement process, one must take diligent measures to ensure that sensitive data stored on the device does not persist beyond its expected lifespan. This entails implementing both procedural and technical requirements to guarantee the secure erasure of sensitive data from the device. Techniques like secure memory erasure can be employed to accomplish this task effectively. It is essential to prevent any unwanted remnants of sensitive information from remaining on the retired device.

Furthermore, it is imperative to address the dissemination of any data that the device might have shared within the system. This data should be thoroughly erased along with the retirement of the device to prevent any unauthorised access or unintended exposure.

Additionally, any access or privileges granted to the device for system and infrastructure resources, such as cloud services, should be permanently revoked. This ensures that the retired device no longer retains any privileged access that could potentially compromise the security of the system.

By adhering to these requirements, the secure retirement of a device can be achieved, mitigating the risks associated with data exposure and ensuring the protection of sensitive information throughout the last phase of the life cycle of the device.

3.5 Process Requirements for the ORSHIN Trusted Life Cycle

We now present a proposition for the process requirements composing the ORSHIN Trusted Life Cycle.

We select the *ENISA Good Practices for Security of IoT* (Section 3.3.1 - ENISA Good Practices for Security of IoT) as a starting source for drafting the requirements. In our opinion, among the process-oriented standards and reference documents that we analysed, it represents the best compromise between completeness, which is a property the ORSHIN TLC should strive for, hardware and IoT applicability, that are crucial for the context of ORSHIN, and simplicity, which makes it possible to focus the task of adaptation primarily on the core content of best practices, rather than on related systems for definitions, scoring, related informative content, etc.

3.5.1 *Selecting the Requirements*

Even though the Good Practices for Security of IoT is a good starting point, we perform an accurate review of the best practices for selecting the ones that pertain to the ORSHIN context, and filter out best practices that do not fully apply.

3.5.1.1 Process Requirements vs. Product Requirements

The first important point to clarify is that the Trusted Life Cycle is a development life cycle, described by process requirements, and not product requirements.

Unfortunately, although such a distinction is easy to operate on a conceptual level (see Section 2.1 - Overview), not all cybersecurity reference sources operate it, and thus the definition of purely process-based life cycles is not immediate.

For example, both NIST SP 800-53 and ISO 27001 standards contain a mixture of process requirements and technical product requirements.

Our chosen reference source for drafting ORSHIN TLC requirements, ENISA Good Practices for Security of IoT, contains mostly process-oriented best practices, but with occasional product requirements, that we wish to filter out.

An example of such a case is represented by requirement ***TC-11, Implement secure session management***. TC-11 states that for all sessions that take place in IoT, it is essential to ensure that active sessions are unique and cannot be shared or guessed, and that they are timed out and invalidated when no longer necessary. Session tokens should be unique for each session, guaranteeing a minimum level of entropy. They must never be disclosed in URLs or error messages. Cookie-based sessions must have the 'Secure', 'SameSite', and 'HttpOnly' attributes enabled.

Such a good practice references specific technical properties of sessions, and it clearly applies to the single product being developed rather than being an indication for the development methodology.

3.5.1.2 Context Differences

Although the Good Practices for Security of IoT describes best practices that are suited for a generic IoT development, we wish to leverage the context of ORSHIN secure hardware to make the requirements of the TLC more specific.

Given the specific context we chose to adopt, some requirements need to be adapted, while we think that others cease to apply.

For example, consider the following items:

- ***PE-09 Designate a physical security officer.***

  Designate a resource responsible for fulfilling the plan or procedure defined to take actions when risks have to be mitigated and to contain them and prevent them from resulting in additional risks if information regarding the SDLC or spaces where it is stored are compromised due to a fire, flood, electric show, etc.
- ***PE-12 Designate a Security Champion figure.***
  Designate a role to centralise all issues related to software development security. This figure should not be responsible for the implementation of security functions, but for coordination, follow-up, planning, and monitoring efforts and activities related to security. This position should be understood as a bridge, a security catalyst among organisation statements (developers, team leaders and decision- makers).
- ***PR-07 Contractually require controlling and monitoring the external services through KPI's.***
  By means of contractual clauses, ensure that both internal and external service providers implement security controls to measure the quality of the service (e.g. service incident response time, unavailability terms, etc.) and detect potential flaws, stipulating a reporting period (e.g. on a weekly basis) for the KPIs to assess the service, along with measures to be taken to prevent impacts on the SDLC phases, such as, for instance, the maintenance phase.
- ***PR-13 Automate the SDLC process.***
  Processes supported by tested tools should be automated in order to reduce costs and human efforts and errors. The main objective is to improve the monitoring and measurement of development progress, as well as the implementation of security measures for the process. The result of automated testing must be analysed, since automated tools are based on patterns that can suffer modifications, which may not be detected and produce false positives. In cases where this is not possible, manual tools should be used. It is recommended to execute this process in every iteration (sprint).

The non-exhaustive list above contains best practices that we wish to exclude from the ORSHIN TLC due to significant context differences.

Governance/"people" items PE-09 and PE-12 are too strict on the definition of roles, and would be difficult to apply as-is, but also to translate for small development teams peculiar to open source projects, which constitute a different reality from the enterprise organisations that could more easily benefit from such requirements.

Similarly, process items PR-07 and PR-13 contain indications that make sense for larger organisation, but not so much for the context of all hardware/embedded/IoT developments (including small ones and open source ones), where contractual obligations may not exist, and automation of the SDLC may not be an immediate necessity, considering that implementing such a best practice would require a nontrivial amount of resources.

3.5.2 *Adapting the Requirements to ORSHIN*

On the remaining best practices that were not excluded for the reasons described in the previous Section, work still needs to be done in order to fully adapt them to the context of ORSHIN.

Thus, we performed a review of all the best practices, and adapted their content based on the following goals:

- *Terminology:* trivially, we want to use ORSHIN-specific terminology; for example, requirements will not reference a generic "SDLC" anymore, but rather they will reference the "TLC"
- *Inclusiveness*: rather than targeting a particular reality (e.g., enterprise IoT development), ORSHIN requirements aim at covering the broadest possible range of hardware/embedded/IoT developments; we adapt the terminology accordingly, for example referring to the "team" rather than referring to the "organisation"
- *Generality:* even though the best practices already reference the context of IoT, some of them primarily apply to software. We make sure to use appropriate terminology to also reference hardware, where applicable.

3.5.3 *Hardware-specific topics*

Besides having many commonalities with the software development, the development life cycle of hardware components differs on various aspects. In particular, the production of hardware artefacts poses constraints in terms of time and cost. Fixing even a tiny bug in a hardware component late in the cycle may range to being very hard, up to requiring to re-spin a new production cycle. Furthermore hardware components belong to different classes, as detailed in Section 2.3 - Views, each one with its own peculiarities in terms of involved entities, production dynamics and costs. For instance, the manufacturing of a silicon IP requires access to highly specialised semiconductor fabrication plants, also called foundries, and this forces the need to plan milestones in advance and to strictly adhere to the timeline. Due to the highly specialised tools and processes required, a production cycle is also very expensive.

This means that extended effort is put in the verification of the design before the tape-out, due to the high penalty in terms of money and time in case of any malfunction of the design. It is common in the hardware design of ASICs to apply extensive sets of testing and verification methodologies at different levels of abstraction, from behavioural RTL simulation to gate-level simulation, formal verification of protocols, prototyping on FPGA, physical simulation of power consumption and electromagnetic emissions, etc. The state-of-the-art in this field involves the “**design for testability**” (DFT) principle, which defines specific design constructions devoted explicitly for testing, such as scan chains for synchronous elements in the design. These elements do not have any functional role in the design but are necessary to validate that the product hardware contains no manufacturing defects that could adversely affect the product's correct functioning.

In addition, the manufacturing process itself is subject to variability and it imposes constraints. Achieving high-yielding hardware designs is an extremely challenging task due to the miniaturisation of the silicon technology as well as the complexity of leading-edge design. In order to keep under control the manufacturing, it is common practice for the hardware designs to apply principles of “**design for manufacturing**” (DFM), for which the easiness and reproducibility of manufacturing of a specific cell element is taken into consideration in the design choices. For instance, cells with high yield are preferred over others with the same functionality but worse in performances.

Different principles apply to different classes of hardware products. In general, compared with software development, the development life cycle of hardware elements has the following peculiarities:

- rigid milestones are defined to split phases for which the feedback loop is extremely expensive or slow, and the responsibility is assigned to different entities (e.g., the design vs the manufacturing);
- it is slower because of the timing and complexity of the manufacturing process;
- a big effort is put on testing and verification as early as possible in the development stages in order to minimise the chance of defects in the final product.

The points highlighted above also impact the Trusted Life Cycle, because on one side they add new constraints to the TLC, and on the other hand they extend the properties that must be covered by trust among the providers of hardware components.

For the reasons exposed before, there are process requirements for the TLC that are peculiar for the design implementation and operation of hardware components. Due to the different dynamics in the development of hardware components, some requirements apply to the process itself. For instance, a defined schedule with the main milestones becomes fundamental in a context where the development and the manufacturing rely on highly specialised and expensive resources, that are possibly offered by external parties. Closely tied to the schedule for the milestones there is the need for well defined roles and responsibilities. When heterogeneous hardware components with different life cycles are integrated in a product, a proper development procedure requires identifying, for instance, the trusted party in charge of the sign-off, who in turn must rely on guarantees from each component provider. The penalty in case of any failure in such a process imposes a stricter management of interfaces between consecutive phases of the development, that is usually not enforced in a more agile cycle applied to purely software components.

In the design phase of hardware components, evaluations and simulations play a relevant role, together with prototypes when possible. For software products, it is common to start with a proof of concept, which then evolves in a featured component over multiple iterations. Performances and other metrics are usually evaluated on artefacts that are close to the current state of the component under development, for instance through live profiling. This approach is usually not possible for hardware components. This means for instance that design choices must be defined in advance relying on experience from similar platforms and on simulations. It is common practice for hardware designs to prototype several variants of the final components and to simulate/evaluate the key metrics before starting with the actual development of the product. For instance, in the development of hardware IPs as defined in the Section 2.3 - Views it is common to explore multiple solutions for area occupancy versus throughput and maximum frequency. This is done by simulating and synthesising different designs in order to get data points in support of the choice for the best architecture. In a similar way, when a required throughput must be reached it is necessary to simulate the system in order to decide how many processing engines to instantiate in parallel, the size of the data buffers, the width of the data paths and so on. All these design choices get harder to modify when going forward in the development process.

In a similar way, physical properties of the hardware components are accurately simulated well before any physical instance of the component will be available. Things such as the power consumption, the electromagnetic emission, the thermal performances must be kept under control since the beginning. Starting from silicon technology cells, to hardware IPs, to SoCs, and electronic boards, every stage must strictly profile and guarantee the physical properties, because this aspect alone can make the difference between a good product and a design that simply cannot work in reality. The benefit and importance of such early evaluations are well recognised even if the evaluations usually are heavy and long to execute. The relative weight of pre-design evaluations among the whole development process is much higher than for software development.

These approaches that are nowadays consolidated in the hardware development are pushed a step forward in terms of challenge when security properties for a TLC must be taken into account. Protections against physical attacks such as side-channel attacks or fault injections require to perform accurate evaluations about the physical behaviour of the resulting object a lot before any physical realisation could be available for actual evaluation. Therefore, it is necessary to have models and tools that allow the early simulation of those physical properties. For instance, in the case of side-channel attacks, both the power consumption and the electromagnetic emission should be simulated of models of the device. These models and tools can leverage the consolidated know-how built over the years for purposes other than side-channel attacks (e.g., power consumption to size the power supplies and characterise the thermal management). However, they must often be adjusted for the specific purpose, for instance isolating data-dependent consumption in relative terms, rather than accurate average absolute values of consumption.

In terms of “trust”, a common approach for software open source is to make the source code publicly accessible and to let the final entity inspect it and build that from scratch up to the final application to be used. This approach is simply not viable in the context of hardware components, with multiple layers (i.e., hardware views), multiple parties that must be involved (e.g., silicon manufacturers) and the associated timings and costs. For this reason, a TLC process must include requirements that aim at providing evidence and at building trust in the design itself. For instance, some design choices may be more trustworthy than others, or exposing some interfaces for inspection may contribute to the overall confidence on the final product. These aspects are sometimes in conflict with the security properties required by the final product. It is usually not possible to make a clear distinction between debugging (and then inspectable) devices and production devices, and therefore the access to some resource must be allowed only under some conditions, commonly related with the life cycle of the device.

Similarly, in a TLC for hardware components that must fulfil some security properties the test must be extended in order to provide evidence to the user of the product about the goodness of the protections. The “design for testability” concept described above extends in order to explicitly cover testing of security properties. Therefore, it is common to have self-tests of basic cryptographic or security properties, which are run automatically at boot or can be requested by the final user in order to check at any time the proper functioning of the device.

3.5.4 *Open source-Specific Topics*

Open source projects can be selected for different reasons, sometimes even just for economics aspects, because they are publicly available, and "good enough" for meeting implementation requirements.

Specifically regarding the context of security, for an electronic product, the property of being open source has always been associated with the benefit of being trustworthy. The reasoning behind this is very simple and involves multiple arguments:

- *Transparency:* a software product for which the source code is publicly available can be inspected in search of any possible security bug, bad practice, malicious piece of code or instruction;
- *Independence:* a software product for which the source code and the building tools are publicly available can be modified by the end user in order to fix any misbehaviour, and rebuilt without the need of relying on external parties.

The two aspects above are solid, but we have to remember that being open source does not always lead to trust by itself, and in particular we believe that the link between open source and trustworthiness should be better elaborated in the case of hardware components and in the specific context of trust of security properties.

When the motivations above are applied to the whole product in order to also include hardware components, it becomes clear that the scenario is more complex.

First of all, it is worth noting that every software product requires a hardware platform to run. Stating that a software component is trusted without having any form of guarantee from the underlying hardware, makes the statement empty and meaningless. This relationship of trustworthiness is true at any level, especially on modern platforms which are composed of multiple layers of components stacked onto one another. In order to be trusted by the final user, a software application must be itself trusted, but must also be built using trusted libraries and frameworks. It must also run on a trusted operating system, which in turn must run on a trusted hardware platform, composed of trusted components. It is clear that the level of trustworthiness in a system ultimately hinges on its weakest component within the stack. This inherent vulnerability undermines the overall security of the entire process, necessitating the pursuit of reliable solutions. To overcome the shortcomings of trust-deficient alternatives, truly secure implementations demand a co-design and evaluation approach that encompasses both hardware and software. This collaborative effort is precisely what ORSHIN focuses on.

In the case of hardware components, even having access to the codebase is usually only a first step towards trusting the final object, but it is far from being sufficient, for two main reasons. First, the independence property is really hard to satisfy because of the highly specialised tools and competencies necessary to process the source code, and the high costs associated with building the final component. This means that in most of the cases one or more third parties must be trusted along the manufacturing chain. Second, by having to rely on a final product manufactured by a third party it is hard (harder than with the software) to guarantee that the physical object has been originated by the exact same codebase that has been inspected. This requires additional trust on third parties that might undermine in practice the benefits in terms of trust provided by a hardware design with an open source codebase. These aspects have generated a research thread related to hardware trojan and how a hardware design might include countermeasures or techniques to detect malicious manipulation during the fabrication. A very recent and very interesting aspect has been discussed in the open source project Precursor [PRECURSOR] where the designer of this messaging device put at the core the possibility for the user to know if the device is malware-free at every level.

In addition, when discussing trust not in a general sense, but related to security properties and guarantees of an electronic component, the access to the source code can help but it cannot be considered enough to trust the final artefact. As this document about TLC aims at demonstrating, the most effective way to build a secure product that can be trusted requires to apply a security-oriented methodology across all the phases. And this involved producing evidence that good practices have been applied during the whole life cycle of the device. An open source component with poor specifications and documentation, without a testing strategy coming from an unknown public repository, can hardly be more trusted than a close-source component from a respected provider with clear documentation and reports about the testing strategy applied. In this context, the foundation of the trust starts with the lower levels of providers, who produce evidence of good practices, which in turn are received and integrated by upper layers up to the final user. Having access to the source code is considered a big plus, which opens the possibility to produce more relevant evidence, but that cannot generate and sustain alone the trustworthiness that this project aims at.

Similarly for the case of hardware development, open source components lead to peculiar requirements that are relevant for the TLC. One set of aspects are related to the need for ensuring reproducibility of artefacts as much as possible, therefore aiming at using open source tools for all the stages of the development, including simulations, evaluations, testing, synthesis, layout design. Results of all these stages should be publicly available, together with instructions to replicate the results. Also the licence should enable the propagation of as much information as possible along the integration chain.

3.5.5 *A Proposal for the ORSHIN TLC Requirement List*

We now present a proposal for the process requirements of the ORSHIN Trusted Life Cycle.

These requirements are derived from the ENISA Good Practices for Security of IoT as a starting base, selected using the criteria explained in Section 3.5.1 - Selecting the Requirements, adapted to the ORSHIN context as discussed in Section 3.5.2 - Adapting the Requirements to ORSHIN, and extended considering Hardware-specific topics (Section 3.5.3) and Open source-Specific Topics (Section 3.5.4).

This list of requirements is a solid set of broadly applicable best practices for hardware development, context-aware with respect to open source aspects. Others can reuse it and even extend it.

As the study of open source hardware development progresses and knowledge is expanded, requirements may be added in the future, and it is also possible that specific sets of requirements will be adapted for specific types of development, at different levels of abstraction in the hardware chain (e.g., low-level IP vs high-level device), or depending on the hardware properties (e.g., highly-detailed and specific cryptography requirements for hardware implementing cryptographic functions).

Given the possibility of these future extensions, we think that, as of today, our proposed set of requirements is suitable for starting an open source hardware project that follows the ORSHIN Trusted Life Cycle.

We provide further detail on applying our set of requirements in the related Section 3.5.6 - Applying the TLC requirements.

We keep the division in three categories made by the starting source, but we rename the "People" category with the broader "Governance" term, for including high-level requirements about the organisation/company/development team that is implementing the TLC. We also keep the "Process" category name, in order to align with the ENISA terminology, even though all the requirements listed in this document belong to the general class of process requirements, in the distinction between process VS product requirements.
We also keep the same sub-categories defined in the document from ENISA, adapting their title if necessary, and we add new sub-categories for the hardware-specific and open source-specific extensions.

The following is the resulting category tree for the requirements.

- Governance
  - Training and awareness
  - Roles and privileges
  - Security culture
  - Hardware design
  - Open source
- Process
  - Third-party management
  - Operations management
  - TLC methodology
  - Secure deployment
  - Security design
  - Internal policies
  - Hardware design
  - Open source
- Technology
  - Access control
  - Third-party software
  - Secure communication
  - Secure code
  - Security reviews
  - Security of TLC infrastructure
  - Secure implementation
  - Hardware design

The complete requirements list is provided in Appendix A - List of process requirements for the TLC.

3.5.6 *Applying the TLC Requirements*

From the previous Sections it follows that both process requirements and product requirements are necessary in order to achieve a good security posture. A secure product cannot be built simply implementing a list of technical requirements, such as encrypt the communication channel or enable the secure boot mechanism. Proper management of the cryptographic keys associated with those mechanisms is as fundamental as the mechanisms themselves. Similarly, without proper testing of the security mechanisms it is hard to get confidence on the actual security of the product.

In the context of a **Trusted** development lifecycle the role of the process requirements become even more important. The only way to build trust on the security of the resulting product is to provide evidence about the good practices applied as part of the underlying development process. In the examples above, for instance, trust can be increased by statements about the realisation of a proper key ceremony for the generation of keys used in production, together with the report of the execution of such a ceremony. Or again, the presence of a security testing suite, together with the logs of the execution of the tests on the current version of the product.

For a hypothetical hardware open source project that starts today and wants to provide trust by adhering to the TLC proposed in this document, the challenge is to concretely instantiate the best practices listed in this document.

On one hand it is impossible to detail concrete guidelines about process requirements that can be blindly applied to any development reality. Differently from product requirements, process requirements impact the daily workflow and could impose arbitrary constraints that make the development flow more complex. This is the very reason why none of the evaluated cybersecurity standards, including the “ENISA Good Practices for Security of IoT” used as main reference, are able to detail a practical procedure to be applied. We believe that the TLC should adapt to and extend the development lifecycle that a team or an organisation has defined over the years and that has been proven effective in realising good products. We think that requiring to revolutionise such an established process to strictly adhere to practices defined once for all, would result in worse products overall.

On the other hand, simply leaving a list of generic process requirements without any indication about its application, would result in a theoretical proposal that only few entities will try to translate into practice. One of the main goals of this project is rather to provide indications that can be applied in a wide range of hardware open source projects of different sizes with benefits from the security point of view.

We advise to use the list of requirements provided by this document as a methodology practical starting set of procedures that the hardware open source project should follow as a reference of good practices, but with two observations.

- First, *this set should be considered as a reference*, rather than a compliance list, in the meaning of adherence to a standard. This means that it is possible for the owner of a new project to decide upfront which requirements better fit in the specific case of the project and which provide the most benefits in terms of supporting the generation of evidence towards trust of the project. As elaborated in the previous Sections of the document, some requirements from the reviewed references hardly apply to projects that are fully open source, or realised by a team of few people. But the check on the applicability of each single requirement to a specific project is delegated to the project owner. Reducing too many of the requirements will result in little trust on the project, and vice versa enforcing too many requirements could potentially slow down the project up to the point that it does not produce any valuable result. The selection itself of the requirements, and their actual implementation in practice should be continuously reviewed and adjusted towards reaching the good tradeoff.
- Second, being the subject of the requirements a live process, *it is difficult to declare once for all that a specific requirement has been fulfilled*. This means that in practice, the checklist should be used to define a procedure, and with this regards once the procedure is defined the check mark can be set, but the application of the procedure should be continuously monitored, considering the possibility that at early stages only part of the procedure could be effectively in place. We believe that in order to reach the ultimate goal of the TLC, that is build trust, the project owner should aim at transparency about the processes. Openly declaring which requirements will be enforced in the projects, possibly with the plan to extend them, and showing with evidence (e.g. log of the performed tests) the level of coverage currently reached by the application of the procedures is an attitude towards trust well-perceived by the stakeholders. This approach can build stronger trust in the long-term compared to the simplistic approach of stating that all the requirements are enforced and are fully in place, but providing no evidence.

Our practical advice for a hypothetical hardware open source project is to use the proposed list of requirements for the TLC and to proceed in two stages:

1. First select the relevant requirements for the project and define the associate procedure to fulfil each of those requirements;
2. Then progressively apply in practice the procedures, annotating the evidence of such an execution.

As described, these two stages will be iteratively repeated over time, knowing that the list of selected requirements and the procedures may be adjusted more frequently at the beginning, and then become more stable. Similarly in the initial iterations the evidence produced may be partial and it will be integrated over time. In this process it is more important to set up the proper security posture in the mid-term, rather than rushing towards filling all the checkboxes in a way that is not replicable.

About the selection of the relevant requirements, our advice is to start with a minimal set of requirements that are considered bringing the most value in terms of trust on the project. Once in place and established such requirements in the actual development flow, the list can be extended over time in order to improve and adapt based also on requirements coming from other stakeholders interested in the outcomes of the project. Depending on the size and on the openness of the project, the realisation of the list itself could be subject to discussions and improvements.

The high level classes of requirements, Governance, Process, Technology, can help in splitting the effort of the definition of the actual instantiation of the TLC for a specific project. **Governance requirements** should be defined first, because they impact roles and activities throughout all the development phases. The project owner or a restricted set of people are in charge of these requirements. Due to the width of these requirements, once established can be reused across different projects from the same organisation or different projects in a similar context. In this set, for instance, some of the most relevant and widely applicable are GO-05 and GO-06. They are about proper definition of roles and proper separation of duties among roles. It is clear that such requirements are meaningful for any kind of project and that should be enforced from the beginning, because they are hard to apply late in the development process. Small teams may define only a few roles, while for larger organisations the associated procedures may be definitely more complex.

Process requirements come next, and could be assigned to roles in the project more related to the operations and executions. Also in this class there are requirements that are always relevant, such as PR-05 and PR-07, that are about Incident management plan and Vulnerability management. It is beneficial to set up from the beginning procedures for these aspects, not waiting for an incident to occur. For small projects the associated procedures could be as simple as defining a responsible person and to indicate the procedure to follow in case any security issue is found on the project, with a clear communication channel established for this purpose (e.g. a dedicated email address). Producing the associated evidence, that is having a publicly accessible and clearly defined page with instructions and contacts, even if simple, shows the security-oriented mindset of the project and therefore contributes in building trust. As much as this requirement is easy to set up, the vast majority of open source projects currently do not have such a procedure in place.

Similarly PR-14, about testing strategy, and PR-20 and PR-22, about risk management and threat modelling, in spite of their fundamental relevance in a TLC that aims at security, are in most of the cases absent, and in the few best cases replaced by a specific set of uncommented tests and some general indications about the risks.

Technology requirements could be assigned to technical roles and apply to technology choices for the project, therefore must be set when the actual development phase starts. Here again it is possible to identify some requirements that are commonly applicable to any kind of project, such as TC-08, about use of secure communication protocols, and TC-09 about proven encryption techniques. Nowadays there is an abundance of secure stacks for communication protocols and implementations of state-of-the-art cryptographic algorithms. Especially for widespread protocols, such as TCP/IP, thanks to the existence of established security mechanisms, such as TLS, it would be impossible for a security-oriented project not to use them instead of the unprotected counterpart or implementing ad-hoc custom solutions. Even if many public projects now embeds these state-of-the-art solutions, the vast majority of them do not explicitly provide statements about the selection of the specific solutions and indications about how to exploit at its best the implemented solution from the security viewpoint. This often results in the integrator not understanding the importance and the reason behind the provided solution, and in its misconfiguration (or even intentional disablement) by the integrator.

From our viewpoint, an ideal instantiation of the TLC in an open source project would require to publicly present the requirements, the defined procedures, and the artefacts produced as evidence together with the source code of the design. Even in cases where this cannot be done for a specific project, stating the adherence to the TLC, together with a clearly visible annotation of the artefacts resulting from the application of the TLC, is a fundamental action for a project that aims at building trust towards the stakeholders.

This concretely translates into publishing this material on the repository of the project or in other publicly accessible locations related to the project. In the same way as nowadays it is a good practice to accompany the source code with a suite of tests to validate it by the integrator, the next step we strive for if to also complement with a description of the reasons behind the choices for that specific testing strategy, and with the logs of the executions of the test for the latest version of the design. Security mechanisms implemented in the products alone, without such an evidence of TLC best practices, may generate a solid product but it will hardly build trust on it.

Chapter 4 Component and Vulnerability Tracking

4.1 Overview

In the previous chapter we have provided an overview of the different requirements and recommendations of multiple standards to shape the TLC proposal. The considered standards and the phases of our TLC proposal include the maintenance of the device and its security. This is an essential aspect, particularly for devices that stay in the field for many years. We thus dedicate this Section to the tracking of components and vulnerabilities during device maintenance.

Component and vulnerability tracking is the process of identifying, monitoring, and managing software and hardware components within a device, along with the associated existing security vulnerabilities. Starting from component and vulnerability tracking it's possible to implement an effective and proactive vulnerability management procedure, aimed at evaluating the identified vulnerabilities, prioritising the efforts for finding mitigations or remediations.

The combination of these processes is a crucial aspect to ensure the security of the device, reducing the risk of potential attacks and protecting sensitive data and assets.

Nowadays, hardware and software systems are becoming increasingly complex. As a result, their supply chain components, functionalities, and relationships are difficult to represent, especially in a standardised format. Various methodologies have been recently proposed to represent components used in hardware and software products, and to link them to known vulnerabilities. However, there is often confusion around these standards and they may not always meet all necessary requirements.

Defining an appropriate and common standard for the representation of a product is crucial to have control over the product, its dependencies, licensing (and more) and to facilitate vulnerability identification, monitoring, and management starting from a well-defined input.

In this Section, we examine the state-of-the-art for component and vulnerability tracking and vulnerability management, exploring limitations as well as possible improvements.

4.2 Vulnerability Management Methodology

Component and vulnerability tracking are essential aspects of the vulnerability management process. Vulnerability management is based on a combination of automated tools and manual efforts. The process can be implemented as follows.

1. *Component Inventory Management:* in this step the target is to create and maintain an inventory of all software (applications, libraries, operating systems, etc.) and hardware components present in the system, gathering information about the version numbers, vendor names, and other relevant details. Every identified component is associated with a unique identifier that is used for tracking and referencing other components throughout the whole process.
2. *Vulnerability Tracking:* in this step all the components in the inventory are scanned for known vulnerabilities, comparing the version in use against vulnerability databases. The scan is run periodically, tracked and documented, to maintain a record of identified vulnerabilities (represented via unique identifiers), their descriptions, severity levels, and available mitigations or remediations.
3. *Vulnerability Assessment:* in this step all the identified vulnerabilities are evaluated to define their risk level and understand their potential impact. The assessment takes into account factors such as the vulnerability's potential for exploitation, the effect on the system functionality/availability, the impact on the data confidentiality and integrity. According to the output of this evaluation, it is possible to define a priority in the remediation efforts.
4. *Vulnerability Addressing:* in this step the assessed vulnerabilities are mitigated or remediated according to the defined priority. This step includes monitoring and tracking the availability of patches or updates provided by component vendors or open source communities, applying

the necessary patches or updates, and keeping track of the actions taken to address each vulnerability.

5. *Reporting and Metrics:* in this step are generated reports and metrics to track the progress of component and vulnerability tracking efforts. This helps in assessing the overall security posture, identifying trends, and communicating the status to relevant stakeholders.

4.3 State-of-the-art: Component Inventory

During the Component Inventory Management step, an inventory of all software and hardware components of the system is defined. This is often referred to as Bill Of Materials (BOM).

There are different types of BOMs, including:

- *Software Bill of Materials (SBOM)*: an inventory that lists the software components used in a particular software system.
- *Software-as-a-Service Bill of Materials (SaaSBOM)*: an inventory similar to SBOM, but for software delivered as a service, and providing a logical representation of complex systems.
- *Hardware Bill of Materials (HBOM)*: an inventory that lists the hardware components that make up a system (e.g. processors, memory modules, mechanical housing, etc.).

To facilitate this step and the subsequent phases of the vulnerability management process, it is beneficial to adopt a standard BOM format. Examples of these standard formats include *Software Package Data Exchange (SPDX)* and *CycloneDX*.

Software Package Data Exchange [SPDX] is an open standard for SBOM, which allows the representation of components, licences, copyrights and other data. It is designed for the representation of software components and is not suitable to be reworked for the definition of an HBOM.

On the other hand, CycloneDX [CycloneDX v1.4] is an open standard from OWASP for defining a generic BOM, including SBOM and HBOM.

A generic BOM contains an inventory of the various individual components of a product, which in turn can be represented using different standards, such as:

- *Common Platform Enumeration (CPE),*
- *Software Identification (SWID) tags,*
- *Package URL (PURL).*

Among these, the use of SWID or PURL is limited to software. On the other hand, CPEs can be used to identify both hardware and software components, but there are some criticalities also in this case that will be analysed later in this document.

In the ORSHIN context, the target is to create a security-focused HBOM model to represent an open source hardware product, that can be used to define the inventory of the used components and to track the status of their vulnerabilities.

The only format that gives the possibility to represent an HBOM is the CycloneDX format. However, it lacks some properties for a more accurate representation and there are very few examples available. In addition, there is no centralised tracking of HBOMs, in any format. For these reasons and some others, which we will explain in detail in later Sections, much work still needs to be done.

4.3.1 *Common Platform Enumeration (CPE)*

Common Platform Enumeration (CPE) [CPE Dictionary] is a standardised scheme that can be used to identify applications, software packages, operating systems, and hardware devices. It provides a structured naming scheme that allows for the unique identification and categorization of these components. CPE is defined by the National Institute of Standards and Technology (NIST) and is part of the Common Vulnerabilities and Exposures (CVE) initiative.

4.3.1.1 Naming Scheme

The CPE naming scheme is based upon the generic syntax for Uniform Resource Identifiers (URI), and consists of a formatted string that includes a set of fields representing different attributes of a component. These fields are organised hierarchically in order of decreasing significance from left to right, and separated by a colon as delimiter (":").

The current version, maintained by NIST, follows this format:

`cpe:<cpe_version>:<type>:<vendor>:<name>:<version>:<update>:<edition>:<language>:<sw_edition>:<target_sw>:<target_hw>:<other>`

- `cpe_version`, represents the version of the CPE definition. The latest CPE definition version is 2.3.
- `type`, represents the category of the component identified by the CPE, and can have one of the following values:
  - *a* for Applications,
  - *h* for Hardware,
  - *o* for Operating Systems.
- `vendor`, represents the person or the organisation that manufactured or created the product.
- `product`, represents the most common name of the component identified by the CPE.
- `version`, represents the specific version number of the component identified by the CPE.
- `update`, represents any updates, service packs, or patches applied to a specific version of the component.
- `edition`, represents a specific flavour (or variations) of a component, often used to represent the target OS/software, architecture, and/or feature set of a product.
- `language`, represents the language used in the specific release of the component (any valid language tag defined by IETF RFC 4646).
- `target_sw`, is used to specify the target software or target environment in which the component is used.
- `target_hw`, is used to specify the target hardware in which the component is used.

For example, for the microcontroller ESP32, manufactured by Espressif, the CPE associated is `cpe:2.3:h:espressif:esp32:-:*:*:*:*:*:*:*`. Here, `*` is used as a wildcard character to denote fields that are not specified. The fields used in this CPE are:

- `cpe_version` 2.3
- `type` hardware
- `vendor` espressif
- `name` esp32

Another example of CPE is the one representing the firmware of the ESP32, defined by `cpe:2.3:o:espressif:esp32_firmware:-:*:*:*:*:*:*:*.` In this case, the field `type` is set to Operating System, which is clearly not an appropriate categorization for firmware. Moreover, note that many fields, as demonstrated in both examples, often go unused.

4.3.1.2 Applications and Usage

By providing a standardised framework for describing and identifying components, Common Platform Enumeration (CPE) is primarily used in the field of cybersecurity and vulnerability management to create a comprehensive inventory of software packages, operating systems, and hardware devices within a system.

A CPE is usually associated to metadata to identify:

- the function of a product (e.g., web server, DNS server),
- the existence of product vulnerabilities,
- product configuration compliance,
- product licence usage.

This is useful in scenarios such as:

- *Asset Inventory:* it aids in managing software versions, ensuring compliance with security policies by keeping track of patches and updates.
- *Vulnerability Monitoring and Management:* it helps in identifying and tracking known vulnerabilities by mapping them to specific components, prioritising remediation efforts.
- *Security Incident Response:* during security incidents or breaches, it helps to quickly identify the affected components and assess their potential impact.
- *Interoperability and Integration:* CPE enables different security tools, databases, and platforms to communicate and exchange vulnerability information using a common naming scheme. It facilitates interoperability and integration between various vulnerability management solutions, making it easier to share and analyse vulnerability data.

4.3.1.3 Limitations

Although CPE is a standardised method for describing and identifying software applications, operating systems, and hardware devices, it does have several limitations.

1. **Limited Coverage**: The type field of a CPE can assume as value only *a*, *o* and *h*, since it primarily focuses on software applications, operating systems, and hardware devices. These three categories are too generic and cannot be used to represent other systems components, such as communication interfaces, protocols and firmware. Moreover, often these categories are misused. Indeed, take as an example `cpe:2.3:o:espressif:esp32_firmware:-:*:*:*:*:*:*:*`, representing the firmware of the ESP32 [ESP32 Firmware CPE]. In this case, the firmware is tagged as Operating System, which is not very appropriate.
2. **Incomplete and Inconsistent Data**: The accuracy and completeness of CPE data rely on the information provided by vendors and maintainers. Sometimes the data may be incomplete, inconsistent, or outdated, leading to challenges in accurately identifying and categorising components.
3. **Lack of Granularity**: CPE uses a hierarchical naming scheme that may not provide sufficient granularity. Most of the fields available in the CPE naming scheme are often not used and substituted with an `*`. Typically, a component is represented using only the fields `cpe_version`, `type`, `vendor`, `name` and `version`. This may not capture nuanced differences within software versions or hardware variants, which can impact vulnerability management and tracking accuracy.
4. **Maintenance and Updates**: CPE requires regular updates to reflect changes in software versions, product names, and other relevant information. However, keeping the CPE data up-to-date can be a challenging task, and outdated or inaccurate information may impact its effectiveness.
5. **Lack of Standardised Taxonomy**: While CPE provides a structured naming scheme, the categorization and classification of software and hardware components are subjective and may vary between vendors or organisations. This lack of standardised taxonomy could create challenges in consistent interpretation and usage of CPE data.
6. **Components relationships**: The CPE naming scheme does not allow to represent links and relationships among components. For example, it is not possible to define if a component is derived from another one (e.g., a project forked from another one), if a set of components is deployed together, or if a component implies the presence of other components (e.g., a device containing specific hardware components). In the context of cybersecurity, knowing these relationships is useful to perform an in-depth analysis of existing vulnerabilities and understand whether one component is the direct source of a weakness or if it inherits it from a linked component. Understanding if a set of components is deployed together also helps to understand whether or not an update should involve all of them.
7. **CPE database**: CPE databases are strictly related to their associated vulnerabilities. This means that if a vulnerability is discovered for a component, a new CPE (if not already defined) is created to keep track of the vulnerability. But on the other hand, if a component has no

vulnerability found yet, it is highly improbable that a CPE would be created to represent it. As a result, it is impossible to use CPEs to proactively identify and monitor components for potential vulnerabilities.

4.4 State-of-the-art: Vulnerability Tracking

A vulnerability is a flaw in a software or hardware component resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, availability, or other aspects (e.g., authenticity, privacy) of a system.

In the context of device security, it is necessary to identify and enumerate not only the hardware and software components, but also the known vulnerabilities and the weaknesses that may affect them.

Vulnerability tracking involves monitoring known vulnerabilities associated with the software and hardware components of a device, to ensure their timely identification and to enable the possibility to implement an efficient vulnerability management process.

The most common method for identifying and tracking vulnerabilities is through the **Common Vulnerabilities and Exposures (CVE)** system.

The CVE database is maintained by MITRE, a non-profit organisation that runs federally funded research and development centres. MITRE collaborates with government agencies to provide technical expertise in areas such as defence, cybersecurity, healthcare, and aviation.

The primary goal of the CVE system is to define a standardised naming and identification scheme for publicly known vulnerabilities. It serves as a central repository of vulnerability information and is publicly accessible to support the efforts of the cybersecurity community in securing their systems and networks. Over time, the CVE database has grown significantly, with thousands of vulnerabilities being assigned CVE IDs each year.

CVE may not cover all security issues of a device, since the system is collecting only known vulnerabilities and exposures. To ensure comprehensive coverage of potential risks and vulnerabilities in their systems, organisations should complement CVE tracking with other practices, such as penetration testing and monitoring of vendor security advisories.

4.4.1 *Common Vulnerabilities and Exposures (CVE)*

The Common Vulnerabilities and Exposures (CVE) [CVE] was established in 1999 by the U.S. Department of Homeland Security's Cybersecurity and Infrastructure Security Agency (CISA) in partnership with the Mitre Corporation.

The CVE system provides a unique identifier, known as a CVE ID, for each reported vulnerability. The format of a CVE ID is "CVE-YYYY-NNNNN," where "YYYY" represents the year of the ID creation, and "NNNNN" is a sequential number assigned to the vulnerability. By assigning unique identifiers (CVE IDs) to vulnerabilities, the CVE system enables consistent and efficient tracking, communication, and remediation of vulnerabilities across the cybersecurity community. In fact, within the information provided for each CVE, it is possible to identify references for mitigations or patches for the vulnerability.

CVE IDs are assigned by CVE Numbering Authorities (CNAs), which include major software and hardware vendors, security research organisations and open source projects. When CNAs assign a CVE ID to a vulnerability, it becomes a part of the CVE database and is made publicly accessible.

Security researchers, vendors, and organisations can report vulnerabilities to CNAs, who ensure that reported vulnerabilities are valid, unique, and meet the criteria for inclusion in the CVE database. CNAs are then responsible for assigning CVE IDs, and publishing the details of the vulnerabilities (such as a description, severity level, affected software versions, and any available references or patches) in the CVE database.

The CVE system has become an essential part of the cybersecurity ecosystem, widely used by organisations and security tools to track, prioritise, and address vulnerabilities in software and systems. By monitoring the CVE database, organisations can stay informed about newly discovered vulnerabilities, can assess the severity and the potential impact on their systems and software, can retrieve patches and mitigations provided by vendors or the community.

4.4.2 *Common Weaknesses Enumeration (CWE)*

The Common Weaknesses Enumeration (CWE) is a framework developed by MITRE which provides a comprehensive taxonomy of the known and recurring weaknesses of software and hardware systems. The primary goal of the CWE framework is to define a common language and a baseline for identification, mitigation, and prevention efforts for each of the most common types of weaknesses [CWE].

In practical terms, CWE is a list of common software and hardware weakness categories, maintained and developed by the community. The CWE list covers a range of different scenarios, such as flaws, faults, bugs in the code, the design, the architecture of the systems that are affected by the listed weaknesses. Moreover, the CWE list contains suggestions and techniques that should be implemented to avoid or mitigate the issues.

4.4.2.1 Limitations

The CWE is an enumeration system, thus it inherits some of the common limitations of this kind of frameworks.

Although the CWE list is designed with the goal of enumerating all the common weaknesses that arise in software and hardware system development, some weaknesses may not be included in the database. This issue relies on the nature of frameworks like CWE, since new weaknesses are discovered continuously and the management of the database requires heavy manual work. Such manual work includes manual reviews and evaluations that may be impossible to automate.

The delay introduced by the manual management of the database may also indirectly affect the frameworks that focus on the vulnerabilities, since the latter directly lean on the weaknesses. In other words, temporary or continued absence of CWE entries easily causes *inconsistencies* in the other evaluation ecosystems (e.g.: CVE).

Moreover, since defining an evaluation method for manual activities is not a trivial task and strictly depends on the sensitivity of the evaluators, the final subjective interpretations of the weaknesses may lead to *inconsistent evaluations*. The inconsistency may be introduced as erroneous, ambiguous or incomplete classifications. For example, it is possible that two similar vulnerabilities are linked to completely different weaknesses, depending on the evaluator(s) decision. According to CVE-2019-15894, an attacker who uses fault injection to physically disrupt the ESP32 CPU can bypass the Secure Boot digest verification at startup. This vulnerability is associated with CWE-755, “Improper Handling of Exceptional Conditions”. CWE-755 is quite ambiguous, and would have been better to select a CWE from category 1388, “Physical Access Issues and Concerns”. For example CWE-1332 “Improper Handling of Faults that Lead to Instruction Skips”, seems more appropriate. Other vulnerabilities associated with fault injection attacks such as CVE-2022-47549 and CVE-2022-42961, are classified as CWE-347, a generic “Improper Verification of Cryptographic Signature”, or not classified at all.

An additional limitation arises considering the experience required to use the system effectively. The CWE framework requires a *high level of expertise* in order to be effective in practical situations; this is mainly due to the fact that the CWE system is hierarchically complex (i.e., it is not a flat data model), and that its content is fragmented (i.e., it has been provided by various contributors). Understanding the content and the meaning of the CWE entries according to the specific application scenarios is a non-trivial task. Security experts are typically involved in the evaluation process, the system lacking a reliable tool designed to apply the conceptual contents of the database to the practical field. In addition, the CWE entries may hardly adapt to the specific peculiarities or implementations of the final products of each organisation.

4.4.2.2 CWE-1194 – Hardware Design

In recent years, a set of hardware weaknesses has been embedded in the CWE list. The latter weaknesses have been grouped in a dedicated category, released in 2021: CWE-1194 [CWE-1194]. The CWE-1194 is a specific view that contains *all the most common weaknesses related to the hardware*, such as side channel leakages, unmanaged faults, security flaw issues, etc.

The CWE-1194 subcategories are listed below:

1194 - Hardware Design

- Manufacturing and Life Cycle Management Concerns - (1195)
- Security Flow Issues - (1196)
- Integration Issues - (1197)
- Privilege Separation and Access Control Issues - (1198)
- General Circuit and Logic Design Concerns - (1199)
- Core and Compute Issues - (1201)
- Memory and Storage Issues - (1202)
- Peripherals, On-chip Fabric, and Interface/IO Problems - (1203)
- Security Primitives and Cryptography Issues - (1205)
- Power, Clock, Thermal, and Reset Concerns - (1206)
- Debug and Test Problems - (1207)
- Cross-Cutting Problems - (1208)
- Physical Access Issues and Concerns - (1388)

Each sub-category contains the CWE related to hardware weaknesses, which are mainly submitted and maintained by hardware experts.

4.4.2.3 CWE-1000 – Research Concepts

CWE-1000 is a specific entry within the Common Weakness Enumeration (CWE) catalogue that contains *the whole CWEs that appear in the database*. As part of the comprehensive taxonomy of vulnerabilities, CWE-1000 serves as a reference view intended to facilitate research into weaknesses, considering their specific inter-dependencies. CWE-1000 organises the CWE entries according to their abstractions of behaviours instead of their detection strategy.

1000 - Research Concepts (pillars)

- Improper Access Control - (284)
- Improper Interaction Between Multiple Correctly-Behaving Entities - (435)
- Improper Control of a Resource Through its Lifetime - (664)
- Incorrect Calculation - (682)
- Insufficient Control Flow Management - (691)
- Protection Mechanism Failure - (693)
- Incorrect Comparison - (697)
- Improper Check or Handling of Exceptional Conditions - (703)
- Improper Neutralization - (707)
- Improper Adherence to Coding Standards - (710)

Each element of the first level can either include base elements or groups (such as classes or composite).

4.4.3 Common Attack Pattern Enumeration and Classification (CAPEC)

We have seen CVE items, which describe single vulnerabilities, and CWE items, which describe generic weaknesses.
An effort to link these entities, explaining how attackers could exploit weaknesses to obtain instances of vulnerabilities is done by MITRE initiative CAPEC [CAPEC].

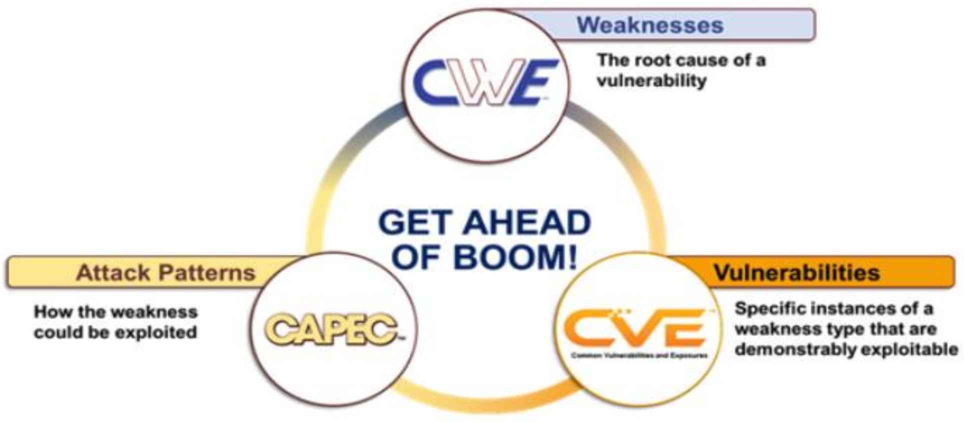

Weaknesses
The root cause of a vulnerability
CWE
GET AHEAD OF BOOM!
Attack Patterns
How the weakness could be exploited
CAPEC
Vulnerabilities
Specific instances of a weakness type that are demonstrably exploitable
CVE
Common Vulnerabilities and Exposures

Figure 19: Comparison of CVE, CWE and CAPEC. Copyright © The MITRE Corporation.

An attack pattern is the common approach to the exploitation of a weakness in a software or a hardware component.

The *Common Attack Pattern Enumeration and Classification* (CAPEC) provides a publicly available catalogue of common attack patterns that helps users understand how adversaries exploit weaknesses in products, components or applications. Each attack pattern captures knowledge about how specific parts of an attack are designed and executed, and gives guidance on ways to mitigate the attack's effectiveness. Initially released in 2007 by the U.S. Department of Homeland Security, the CAPEC List continues to evolve with public participation and contributions to form a standard mechanism for identifying, collecting, refining, and sharing attack patterns among the cybersecurity community.

4.5 Modern Approach for Component and Vulnerability Tracking

In this Section we present a modern and efficient approach to component and vulnerability tracking. The goal is to establish a methodology for identifying and representing the components of a product in an unambiguous and accessible format. Through this methodology it should be possible to define the Bill Of Materials (BOM) of a device in a simple way, facilitating the aggregation and scaling of information at a global level. Being able to track components in this format would then also facilitate and automate the monitoring of their vulnerabilities.

Desirable features for a component tracking system include the following:

1. *There should be the possibility to track components freely, even when there are no known vulnerabilities, and to allow contributions from the community.*

   As seen in Section 4.3.1.3- Limitations, the current situation is not ideal. We think that a global database (centralised or distributed) should allow contribution from various sources, including at least:

   - Manufacturers/vendors wanting to track their own components, and providing authoritative information;
   - Community members wanting to add new components or add/correct information on existing ones.

2. *There should be a way to indicate whether a (hardware) component is open source.*

   Specifically for the context of ORSHIN secure hardware, but also generally due to the rising importance of open source hardware designs, we think that there should be a way to indicate whether a particular product declares to be open source, or uses open source subcomponents. In Chapter 1 - Definition of Open source Hardware we faced the problem of agreeing on what it means to be open source, and also we tried to answer the question of whether the property of being open source can be defined not only qualitatively, but also quantitatively.

We make a proposition for a component-tracking format that will allow to specify such information, using our own definition as an example; regardless of the actual system being used in the future, we think that it is important to capture this open source-related information inside BOMs.

3. *There should be the possibility to specify product licences.*

   Licensing information is important when cataloguing components, providing information about their allowed use. The relevance is not limited to software licences, but also applies to hardware designs, and gets even more relevant in the context of open source, as different open source licences greatly vary in terms of the actual permissions they concede.

4. *It should be improved the specification of vendor, author, and contributor information, and this could include a "community" vendor option, for example.*

   For standard products, a single "manufacturer" or "vendor" is enough, but the reality of open source projects is more variegated. For instance, the maintainer of an open source project may change over time, or there might not be a single entity behind the development, but rather a "community" or a "team" composed of individuals.

   Currently, this information is hardly tracked outside of the repository or web page of the individual project, and sometimes it's not even clear from such sources; as an example, consider an open source project on GitHub with a single owner and contributor, identified only by its GitHub handle.

   We expect that in the next few years, the catalogues for components will have to give the possibility of specifying all this information pertaining to authorship and identification.

5. *Lastly, more useful metadata, such as links to the main project page and security advisories, could be added.*

   Links to security-relevant documents are always useful, such as security advisories published by the vendor, external independent blogs describing features of the product, and so on. Due to irregularities, it may not be possible to devise a model that perfectly fits for every possible case, so a desirable feature for a tracking model would be to have enough flexibility to specify both common and unexpected metadata.

To provide these features, the first step is to agree on unique and shareable identifiers for components. A cloud infrastructure should therefore be implemented to keep track of components in a database, observing the above properties.

It should be possible to apply for registration to the system with different types of users and roles, which include the possibility of submitting proposals for components, submitting revisions and updates, and approving them.

The system should have a form of consensus whereby multiple users can review and approve the proposals of components. For example, there could be defined roles for *reviewers*, who can propose changes to components or propose new ones to be added, and roles of *administrators*, who can vote to approve new components or revisions (with an agreed threshold for approval).

There should be no ambiguity between components in the database: there should be no duplicates, or at least it should be possible to identify duplicate components and perform deduplication.

After careful evaluation of the state-of-the-art (Section 4.3- State-of-the-art: component inventory, Section 4.4- State-of-the-art: Vulnerability Tracking), we consider CycloneDX a promising format for efficient component tracking. It already possesses most of the features we desire to model a generic BOM, and HBOMs in particular.

Moreover, through its Vulnerability Exploitability eXchange (VEX) feature, it is possible to provide information about vulnerabilities inside BOMs, thus linking component tracking to vulnerability tracking. Single vulnerabilities can be represented by providing their CVE identifier, along with related metadata (if present/relevant).

With a few adaptations to facilitate context-specific information for hardware and for open source, it is the ideal format for representing BOMs of ORSHIN secure hardware in the scope of the ORSHIN Trusted Life Cycle.

Given these considerations, the data format we propose to be used in a hypothetical global database for component tracking is compliant with the CycloneDX format. It can be used to represent BOMs for single components (e.g., a library, a specific hardware module), or for more complex devices.

4.5.1 *Model for Component Tracking Definition*

To define our model for component tracking, we start by providing a conceptual example of the Bill Of Materials (BOM) for a device, which we call ORSHIN Device. The proposed format follows the CycloneDX schema [CycloneDX v1.4].

Our main focus is to model the hardware part, without also including in our BOM software components that run on top of it. The BOM of the example ORSHIN Device will have the following structure, in JSON format:

```
{
  "$schema": "http://cyclonedx.org/schema/bom-1.4.schema.json",
  "bomFormat": "CycloneDX",
  "specVersion": "1.4",
  "serialNumber": "urn:uuid:3e671687-395b-41f5-a30f-a58921a69b79",
  "version": 1,
  "components": [
    {
      "bom-ref": "device-1",
      "type": "device",
      "name": "ORSHIN Device",
      "version": "1.0.0",
      "supplier": {
        "name": "ORSHIN",
        "url": [ "https://www.orshin.com" ],
        "contact": [ { "email": "orshinmember@gmail.com", ... }, ... ]
      },
      "author": "ORSHIN",
      "publisher": "ORSHIN",
      "licenses": [ ... ],
      "description": "Example of device for the ORSHIN project",
      "externalReferences": [
        {
          "type": "other",
          "url": "https://www.orshin-device.com"
        }
      ],
      "properties": [
        {
          "name": "orshin:view",
          "value": "1"
        },
        {
          "name": "osrhin:opensource:score",
          "value": "TBD"
        }
      ],
      "components": [ ... ]
    }
  ],
  "dependencies": [
    {
      "ref": "device-1",
      "dependsOn": [ ... ]
    }
  ]
}
```

The proposed schema consists of the following elements, defined by CycloneDX fo.

- The `schema`, `bomFormat` and `specVersion` properties, containing information about the schema itself.
- The `serialNumber`, the unique identifier of the BOM, which must conform to RFC-4122.
- The `version` of the BOM, which is 1 by default and should be incremented whenever the BOM is modified, either manually or through automated processes.
- The `components`, an array of the software and hardware components of the product represented by the BOM.
- In addition, other properties can be specified. For example the property `metadata`, which can be used to provide additional information about a BOM (e.g. authors, manufacturer, supplier, etc.).

As concerns the `components` array, for each element of this list, some of the properties that can be specified are listed below.

These properties are all derived from the CycloneDX format. In our model, we have only extended the possible values that the `type` of a component can take and added ORSHIN-specific properties in the field where CycloneDX allows custom values to be included. In particular, the fields marked with (*) were not defined by CycloneDX or were readjusted for the purpose of adherence to the ORSHIN context.

1. `bom-ref`: an optional identifier which can be used to reference the component in the BOM. Every `bom-ref` must be unique within the BOM.
2. `type`: the type of component. For software components, CycloneDX allows a specific and appropriate classification, while for hardware it lacks granularity. We propose the following comprehensive list to choose from, which also allows us to model the ORSHIN views defined in Section 2.3 – Views.
    - `application`: software application.
    - `framework`: software framework.
    - `library`: software library. All third-party and open source reusable components will likely be a library. If the library also has key features of a framework, then it should be classified as a framework.
    - `container`: packaging and/or runtime format, not specific to any particular technology, which isolates software inside the container from software outside of a container through virtualization technology.
    - `operating-system`: operating system.

- `firmware`: special type of software that provides low-level control over a device's hardware.
- `file`: computer file.
- `device` (*): hardware device into the hands of the final user, which is the final composition of parts from lower ORSHIN views.
- `chip` (*): integrated circuit that combines multiple electronic components and functionalities into a single chip (CPUs and IPs).
- `cpu` (*): central process unit of a hardware component.
- `ip` (*): stands for Intellectual Property, subcomponent of a system-on-module / chip (e.g. a secure element).
- `technology-library` (*): library containing the blocks to build the fundamental bases for a hardware component.

3. `name`: the name of the component (a shortened, single name of the component).
4. `version`: the component version. The version should ideally comply with semantic versioning but is not enforced.
5. `licences`: a list of licences of the component, that can be represented by a string or a more detailed structure with id, name, text and URL properties.
6. `description`: a brief description for the component.
7. `externalReferences`: a list of sites and other information (which may also include other BOMs) that may be relevant, but which are not included with the BOM.
8. `supplier`: an object describing the organisation that supplied the component, which may often be the manufacturer, but may also be a distributor or repackager. Can include names, URLs and email addresses.
9. `author`: the person(s) or organisation(s) that authored the component.
10. `publisher`: the person(s) or organisation(s) that published the component.
11. `properties`: list of properties which can be customly defined and represented in a name-value format. This provides us the flexibility to include data not officially supported in the CycloneDX. We define the following properties:
    - `Orshin:view` (*): the optional number identifying the ORSHIN view of the component.
    - `Orshin:opensource:score` (*): the optional scoring value for the ORSHIN open source evaluation of the component.
12. `components`: list of software and hardware components included in the parent component. This is not a dependency tree, while it provides a way to specify a hierarchical representation of components. Items are structured as above, i.e. can have all above properties, including the array of components itself.
13. `dependencies`: array of items that define the direct dependency of a component, using bom-ref identifiers.

We now provide some observations on the defined model, the convenience of using CycloneDX, its advantages and disadvantages.

Aspects that we consider important for the representation of a BOM and that CycloneDX fulfils are multiple. It offers the possibility of expressing components and subcomponents, thus representing the inclusion relationship between them. There is the possibility of expressing links between components, using the dependency property. An example of dependency can be the one existing between a library defined in a repository and a derived version of it, defined in a fork of its repository.

In addition, components can be represented without using CPEs. The major improvement brought by this alternative system is that devices, and BOMs in general, become uniquely identifiable objects, which can be structured, searchable, linkable, all from within the CycloneDX format. With CPEs, these features were somewhat granted by the surrounding environment (e.g. the NIST database), but the CPE format itself could not be used for any rich, meaningful representation of these objects (refer to Section 4.3.1.3 – Limitations for further detail).

Even if it is easy to agree on the improvement brought by this change, the CPE system remains widespread nowadays, related databases do contain most of relevant information useful for tracking software and hardware (and link them to vulnerabilities), and so replacing it all at once would not be realistic. CycloneDX makes it possible for each component to also define the property `cpe`, providing a way to preserve this information, and make the transition to a new tracking framework smoother.

On the other hand, CycloneDX does not have a system to enforce the definition of globally unique identifiers; as of today, to the best of our knowledge there are no public global databases that use CycloneDX to track components.

However, using a combination of the serial number and bom-ref fields, components in CycloneDX can be uniquely identified and referenced [BOMLINK]. Therefore, by defining and implementing the appropriate infrastructure, we think that this representation format is suitable for modelling BOMs of arbitrarily complex hardware, and serve a central global reference and inventory for hardware components.

With appropriate effort and allocation of resources from key actors, it could be possible to initiate a transition towards this modern tracking system in the near future, and make it fully operational within the next years.

4.5.2 *Practical Example*

Now we show how to use our BOM methodology on a device, which we call ORSHIN Device, with some real components, to provide a more complete and practical example. The device is composed by two chips:

1. NXP MIMXRT685 (see [NXP RT600 Datasheet] for details),
2. U-blox LARA-R6001 (see [U-blox LARA-R6 Datasheet] for details).

The NXP MIMXRT685 is a dual-core microcontroller from the NXP i.MX RT family. It is based on the ARM Cortex-M33 and the Xtensa HiFi4 Audio DSP CPUs. The u-blox LARA-R6001 is a cellular module of the LARA-R6 series. Among this series it is one of the smallest LTE Cat 1 multi-mode solutions with comprehensive support of RAT and bands for global connectivity, including 18 LTE FDD/TDD bands plus 3G/2G fallback in single SKU.

In order to define the BOM, it is necessary to understand which are the subcomponents of the two chips. These can be derived by looking at the NXP MIMXRT685 and u-blox LARA-R6001 schematics and extracting the CPUs and PINs information. The two Figures below illustrate the subcomponents diagrams of the two chips.

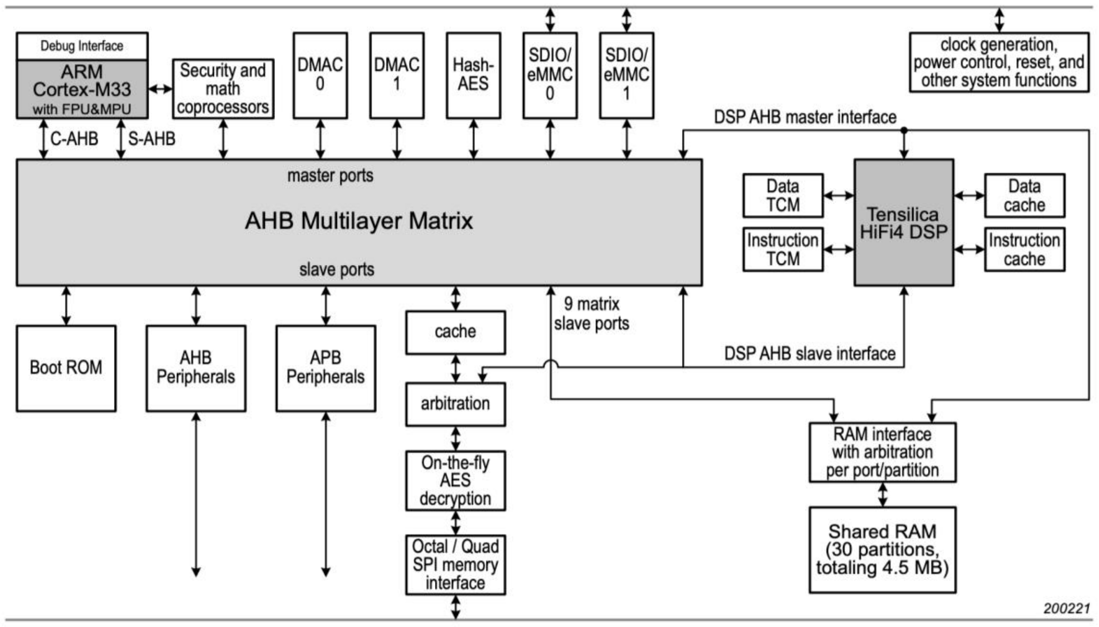


Figure 20: NXP MIMXRT685, source [NXP RT600 Datasheet].

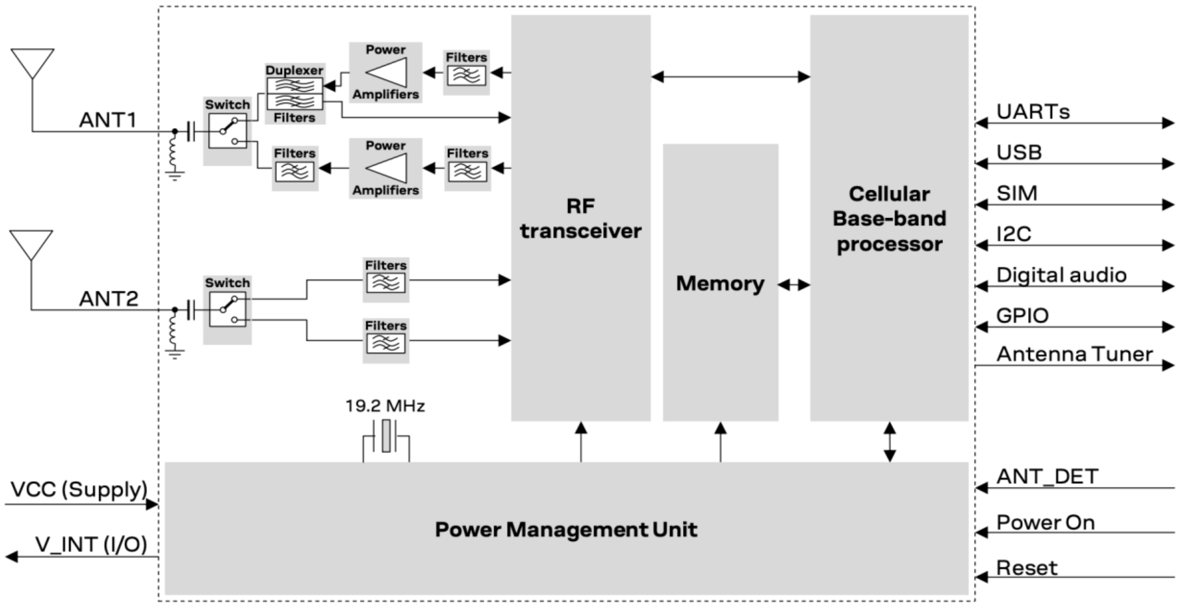


Figure 21: U-blox LARA-R6001 simplified block diagram, source [U-blox LARA-R6 Datasheet].

As can be seen from the first diagram and from the schematic document, NXP MIMXRT685 is composed of two CPUs (Cortex-M33 and Tensilica HiFi4 DSP) and several IPs (e.g. PowerQuad and Casper coprocessors, Boot ROM unit, RAM memory, etc.).

On the other hand, the LARA-R6001 schematic is documented with less detail and it is more complex to extract information regarding its subcomponents. However, it is possible to determine that it consists of a cellular base-band processor, which should be the MDM9207-1 by Qualcomm (composed, in its turn, by a Cortex-A7 CPU), an RF transceiver, a Flash memory and Power Management Unit.

The CPUs and IPs of the two chips of the device are in turn composed of different technology libraries. In the case of our example, the underlying technology libraries are not open source, so the representation of this information is not applicable here. In addition, we decided not to report in detail all IPs and all properties of the device under consideration, in order to keep the example easy to understand (from the pictures above all known components that are part of the considered chips can be found).

The BOM of the ORSHIN Device example will represent the components as outlined in the following picture.

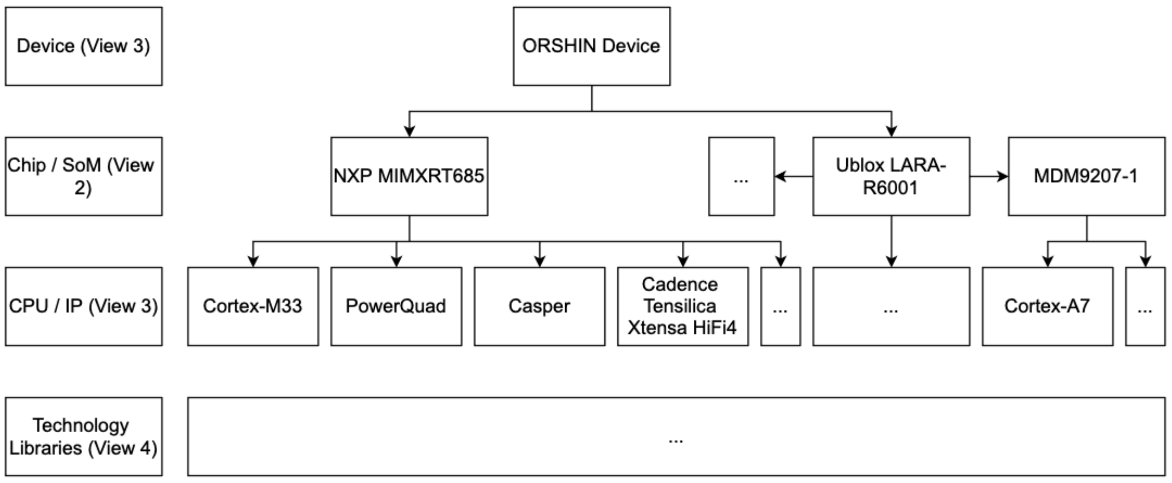


Figure 22: The BOM of the ORSHIN device example.

Therefore, the BOM of the ORSHIN Device will have the following scheme.

```
{
  "$schema": "http://cyclonedx.org/schema/bom-1.4.schema.json",
  "bomFormat": "CycloneDX",
  "specVersion": "1.4",
  "serialNumber": "urn:uuid:3e671687-395b-41f5-a30f-a58921a69b79",
  "version": 1,
  "components": [
    {
      "type": "device",
      "name": "ORSHIN Device",
      ...
      "components": [
        {
          "type": "chip",
          "name": "NXP MIMXRT685",
          "description": "Main application processor",
          ...
          "properties": [
            {
              "name": "orshin:view",
              "value": "2"
            },
            {
              "name": "orshin:opensource:score",
              "value": "TBD"
            }
        ],
        "components": [
          {
            "type": "cpu",
            "name": "Cortex-M33",
            "description": "Arm Cortex-M33 core",
            "properties": [
              {
                "name": "orshin:view",
                "value": "3"
              },
              {
                "name": "orshin:opensource:score",
                "value": "TBD"
              }
            ]
          },
          {
            "type": "cpu",
            "name": "PowerQuad",
            "description": "Hardware accelerator for fixed and floating point
            DSP functions",
            "properties": [
              {
                "name": "orshin:view",
                "value": "3"
              },
              {
                "name": "orshin:opensource:score",
                "value": "TBD"
              }
            ]
          },
          {
            "type": "cpu",
            "name": "Casper",
            "description": "Crypto/FFT engine",
            "properties": [
              {
                "name": "orshin:view",
                "value": "3"
              },
              {
                "name": "orshin:opensource:score",
                "value": "TBD"
              }
            ]
          },
          {
            "type": "cpu",
            "name": "Cadence Tensilica Xtensa HiFi4",
          "description": "DSP processor core",
          "properties": [
            {
              "name": "orshin:view",
              "value": "3"
            },
            {
              "name": "orshin:opensource:score",
              "value": "TBD"
            }
          ]
        },
        ...
      ]
    },
    {
      "type": "chip",
      "name": "U-blox LARA-R6001",
      "description": "LTE interface module",
      "properties": [
        {
          "name": "orshin:view",
          "value": "2"
        },
        {
          "name": "orshin:opensource:score",
          "value": "TBD"
        }
      ],
      "components": [
        {
          "type": "chip",
          "name": "MDM9207-1",
          "description": "Cellular base-band processor",
          "properties": [
            {
              "name": "orshin:view",
              "value": "2"
            },
            {
              "name": "orshin:opensource:score",
              "value": "TBD"
            }
          ],
          ...
          "components": [
            {
              "type": "cpu",
              "name": "Cortex-A7",
              "description": "Arm Cortex-A7 core",
              "properties": [
                    {
                      "name": "orshin:view",
                      "value": "3"
                    },
                    {
                      "name": "orshin:opensource:score",
                      "value": "TBD"
                    }
                  ]
                },
                ...
              ]
            },
            ...
          ]
        }
      ]
    }
  ]
}
```

This representation is intended to be a baseline example of the proposed format for representing an HBOM. It should be clear from this example and from what is described in this Section how to extend and complement it.

Other than the specific properties of the HBOM format, we also think that a relevant role in the evolution of component and vulnerability tracking will be played by the related infrastructure, for which we have outlined the desired properties.

Chapter 5 Conclusion and Next Steps

In this deliverable we have reported part of the research of WP2, particularly focusing on the work related to Task 2.1, Trusted Life Cycle methodology.

This methodology aims at providing developers and maintainers of the open source community with practical help for exploring and expanding the cyber security dimension of their projects, and particularly the embedded/IoT/IIoT projects which employ open source hardware.

We took inspiration from state-of-the-art standards and embraced the open source philosophy, discharging any security-by-obscurity practice. At the beginning of an open source project, the developers can use the TLC as a reference for shaping its cyber security dimension, and for creating the required evidence that the project is adhering to certain security requirements.

The methodology has been detailed with the user/adopter of open source projects in mind; we put ourselves in the shoes of the user that would like to select an open source project, and who would like to have indications on whether the cybersecurity dimension has been considered when initiating a development, and if it is still considered and maintained over time after the product has gone to market.

There are two additional and important outcomes from our research. First, we have worked to clarify the definition and meaning of "open source hardware". Despite being extensively used, this terminology does not yet have a universally accepted definition. Before our research, the closest effort to having such a definition is represented by the Open Source Hardware (OSHW) Definition 1.0 [OSHWA 2023]; however, the property that are used to characterise open source in that document are not easy to measure, do not differentiate between different types of hardware, and are generally oblivious of the technical context.

We tried to make an improvement to the state-of-the-art, by providing a definition for open source hardware covering all these missing properties. After differentiating hardware developments based on their level of abstraction, we study relevant properties that affect their ability to effectively be "open source", initially from a qualitative perspective, then also from a quantitative point of view, providing a practical way to calculate a score for how "open source" a component effectively is. Finally, we provide examples on how such a system can be applied to real-world use cases.

On the other hand, there is a growing research in finding vulnerabilities in software, firmware and hardware. Which has an impact on the projects that are using components coming from other open source projects or from commercial providers, like silicon manufacturers. The adopter of the open source project would like to see from the maintainers an effort for tracking components and the associated vulnerabilities over time. This is a fundamental aspect. Some projects are doing it via release notes, this is a good start but remains hard to follow. We define a modern method adopting the CycloneDX and expanding it to the open source hardware dimension. We also provide guidance on the next steps that would need to be executed by key actors in order to advance the adoption of this model.

Our proposal for the ORSHIN Trusted Life Cycle is flexible and allows adopters to customise it based on their needs. Our list of security requirements should be considered as a base from which a project can immediately benefit from.

The next steps that we envision for our TLC mainly concern the involvement of interested stakeholders; we think that, more than a theoretical contribution to the state-of-the-art, our approach could also practically constitute a first step in consolidating secure life cycles for previously unexplored contexts, namely the ones of hardware and open source.

By adopting our classification and scoring system for open source hardware, manufacturers will be able to both measure their effectiveness when trying to adhere to open source initiatives, and at the same time they will be able to provide a sort of scorecard to interested parties for measuring how "open source" their products are.

By using our definition for the TLC phases and requirements, manufacturers who wish to initiate hardware developments with open source components will be able, for the first time, to adopt a methodology for defining and implementing context-aware process requirements in such areas.

Finally, if key actors in the area of component and vulnerability tracking agree on our proposal for implementing a global database for allowing the community to track information related to Bill Of Materials in an efficient way, overcoming the limitation of the current *de facto* standard system, it will be possible to evaluate the benefits of a framework that is more open, coherent, and hardware-friendly than what is available today.

List of Abbreviations

Abbreviation	Translation
AC	Access Control
ADF	AttackDefense Framework
API	Application Programming Interface
ASIC	Application Specific Integrated circuit
BOM	Bills Of Materials
BSIMM	Building Security in Maturity Model
CAD	Computer Aided Design
CAPEC	Common Attack Pattern Enumeration and Classification
CCM	Cloud Controls Matrix
CCTV	Closed-Circuit Television
CISA	Cybersecurity and Infrastructure Security Agency
CLASP	Comprehensive, Lightweight Application Security Process
CNA	CVE Numbering Authorities
CoM	Computer on Module
CPE	Common Platform Enumeration
CPU	Central Processing Unit
CSA	Cloud Security Alliance
CVE	Common Vulnerability and Exposures
CWE	Common Weakness Enumeration framework
DevSecOps	Development Security Operations
DFM	Design for Manufacturing
DFSG	Debian Free Software Guidelines
DFT	Design for Testability
DNS	Domain Name System
DSP	Digital Signal Processing
EDA	Electronic Design Automation
ENISA	European Union Agency for Cybersecurity
ETSI	European Telecommunications Standard Institute
FDD/TDD	Frequency Division Duplex/Time Division Duplex
FIDO	Fast Identity Online

Abbreviation	Translation
FPGA	Field Programmable Gate Array
GB	GigaByte
HBOM	Hardware Bills Of Materials
HDL	Hardware Description Language
HW	Hardware
IACS	Industrial Automation Control Systems
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IETF	Internet Research Task Force
IIoT	Industrial Internet of Things
IOC	Indicator Of Compromise
IoT	Internet of Things
IP	Intellectual Property
ISA	International Society of Automation
ISMS	Information Security Management System
ISO	International Organization for Standardization
IT	Information Technology
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
LTE	Long Term Evolution
LTS	Long Time Support
NDA	Non Disclosure Agreement
NIST	National Institute of Standards and Technology
OS	Operation System
OSHW	Open Source Hardware
OSHWA	Open Source Hardware Association
OWASP	Open Worldwide Application Security Project
P0, P1, P2, ...	Property 0, Property 1, Property 2, …
PCB	Printed Circuit Board
PIN	Personal Identification Number
PURL	Package URL
RAM	Random Access Memory
RAT	Radio Access Technology

Abbreviation	Translation
RF	Radio-Frequency
RFC	Request For Comments
ROM	Read-Only Memory
RTL	Register Transfer Level
SA	System and Services Acquisition
SaaSBOM	Software-as-a-Service Bill of Materials
SAMM	Software Assurance Maturity Model
SBOM	Software Bills Of Materials
SDK	Software Development Kit
SDLC	Secure Development Life Cycle
SI	System and Information Integrity
SIEM	Security Information and Event Management
SKU	Stock Keeping Unit
SMM	Security Maturity Model
SoC	System on a Chip
SoM	System on Module
SP	Special Publication
SPDX	Software Package Data Exchange
SW	Software
SWID	Software Identification
TCP/IP	Transmission Control Protocol/Internet Protocol
TLC	Trusted Life Cycle
TLS	Transport Layer Security
TS	Technical Specification
UK	United Kingdom
UMC	United Microelectronics Corporation
URI	Uniform Resource Identifiers
URL	Uniform Resource Locator
USB	Universal Serial Bus
V0, V1, V2, V3	View 0, View 1, View 2, View 3
VA	Vulnerability Analysis
VEX	Vulnerability Exploitability eXchange
WP	Work Package

Bibliography

Kelty 2016

C. M. Kelty, "The Cultural Significance of free Software - Two Bits", Duke University Press, 2016.

Free Software Foundation 2017

Free Software Foundation, "Categories of Free and Nonfree Software", 2017. [Online]. Available: https://www.gnu.org/philosophy/categories.html.en.

Baker 2011

R. J. Baker, "CMOS: Circuit Design, Layout, and Simulation", John Wiley & Sons, 2011.

OSHWA 2023

OSHWA, “Open Source Hardware (OSHW) Definition 1.0”, visited 2023. [Online]. Available: https://www.oshwa.org/definition/.

Open Source Initiative

Open Source Initiative, “Open Source Definition”, 2007. [Online].

Available: https://opensource.org/osd/.

Debian Social Contract 2023

Debian org, “The Debian Free Software Guidelines (DFSG)”, 2020. [Online].

Available: https://www.debian.org/social_contract#guidelines.

CERN OSH 2023

CERN, “CERN Open Hardware Licence”, visited 2023. [Online].

Available: https://cern-ohl.web.cern.ch/.

CERN OHL 2020

CERN, “CERN OHL version 2. An Introduction and Explanation”, 2020.

Available:
https://ohwr.org/project/cernohl/wikis/uploads/0be6f561d2b4a686c5765c74be32daf9/CERN_OHL_rationale.pdf

MIT LICENCE

Open Source Initiative, "The MIT License", visited 2023. [Online].

Available: https://opensource.org/license/mit/

APACHE LICENCE 2023

APACHE software foundation, “APACHE LICENSE, VERSION 2.0”, visited 2023. [Online].

Available: https://www.apache.org/licenses/LICENSE-2.0

GPL 2022
GNU Operating System, “Licenses”, 2022. [Online].
Available: https://www.gnu.org/licenses/licenses.en.html

CCLICENCES
Creative Commons, “About CC Licenses”. [Online].
Available: https://creativecommons.org/about/cclicenses/

GNU 2021
Free Software Foundation, "GNU Operating System", visited 2023. [Online].
Available: https://www.gnu.org/

OSHWA CERT
OSHWA, “Open Source Hardware Association”, 2023. [Online].
Available: https://certification.oshwa.org/

CPE Dictionary
Official Common Platform Enumeration (CPE) Dictionary, 2023. [Online].
Available: https://nvd.nist.gov/products/cpe

Howard 2006
M. Howard, S. Lipner, “The Security Development Lifecycle”, Microsoft Press, 2006.

Geer 2010
D. Geer, “Are Companies Actually Using Secure Development Life Cycles?”, 2010.

Raspberry About
Raspberry Pi Foundation, “About”, visited 2023. [Online].
Available: https://www.raspberrypi.com/about/

Raspberry Products
Raspberry Pi Foundation, “Products”, visited 2023. [Online].
https://www.raspberrypi.com/products/

USB Armory
With Secure, “USB Armory”, visited 2023. [Online].
Available: https://www.withsecure.com/en/solutions/innovative-security-hardware/usb-armory

USB Armory Mk II
With Secure, “USB Armory Mk II”, visited 2023. [Online].
Available: https://www.crowdsupply.com/f-secure/usb-armory-mk-ii

Toradex Apalis

Toradex, “Apalis arm family”, visited 2023. [Online].
Available: https://www.toradex.com/computer-on-modules/apalis-arm-family

OWASP SAMM

Open Web Application Security Project, "OWASP SAMM", visited 2023. [Online].
Available: https://owasp.org/www-project-samm/

CycloneDX v1.4

Open Web Application Security Project, "CycloneDX v1.4 JSON Reference", visited 2023. [Online].
Available: https://cyclonedx.org/docs/1.4/json/

BOMLINK

Open Web Application Security Project, "CycloneDX BOM-Link", visited 2023. [Online]
Available: https://cyclonedx.org/capabilities/bomlink/

PRECURSOR

Sutajio Kosagi, Precursor, visited 2023. [Online]
Available: https://www.crowdsupply.com/sutajio-kosagi/precursor

SPDX

Linux Foundation, "SPDX Project", visited 2023. [Online].
Available: https://spdx.dev/

CVE

Common Vulnerabilities and Exposures
Available: https://www.cve.org/About/Overview

NXP RT600 Datasheet

NXP RT600 Product data sheet Rev. 2.0, 1 April 2022.
Available https://www.nxp.com/docs/en/data-sheet/DS-RT600.pdf

U-blox LARA-R6 Datasheet

U-blox LARA-R6 series single or multi-mode LTE Cat 1 modules with Secure Cloud data sheet, 09 Feb 2023.
Available https://content.u-blox.com/sites/default/files/LARA-R6_DataSheet_UBX-21004391.pdf

CWE

The MITRE Corporation, "Common Weakness Enumeration", visited 2023. [Online]
Available: https://cwe.mitre.org/

CWE-1194

"CWE VIEW: Hardware Design", visited 2023. [Online]
Available: https://cwe.mitre.org/data/definitions/1194.html

CAPEC

The MITRE Corporation, "Common Attack Pattern Enumerations and Classifications", 2023. [Online] Available: https://capec.mitre.org/

ISO 27001

ISO, "ISO 27001:2022", 2022.

ISO 27002

ISO, "ISO 27002:2022", 2022.

CLS CSA Singapore

CSA Singapore Government, "Cybersecurity Labelling Scheme". [Online] Available: https://www.csa.gov.sg/our-programmes/certification-and-labelling-schemes/cybersecurity-labelling-scheme

STRIDE

Microsoft Corporation, "The STRIDE Threat Model", 2009. [Online]
Available: https://learn.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)

Code of Practice for Consumer IoT Security

UK Government, "Code of Practice for Consumer IoT Security", 2018. [Online]
Available: https://www.gov.uk/government/publications/code-of-practice-for-consumer-iot-security

ESP32 Firmware CPE

The CPE used by NIST to represent ESP32 firmware. [Online]
Available: https://nvd.nist.gov/products/cpe/detail/143A95B6-D13D-434A-AD4A-3F0C4FCD6122?namingFormat=2.3&orderBy=CPEURI&keyword=cpe%3A2.3%3Ao%3Aespressif%3Aesp32_firmware%3A-%3A*%3A*%3A*%3A*%3A*%3A*%3A*&status=FINAL

Home Assistant

Project Home Assistant, "Raspberry Pi", visited 2023. [Online]
Available: https://www.home-assistant.io/installation/raspberrypi/

Appendix A - List of process requirements for the TLC

GO-01	**Governance | Training and Awareness**
Title	Define a team strategy for specific security training
Description	Ensure that all personnel participate in awareness-raising activities and training, focusing on how to apply security in a TLC process. These activities must be customised depending on roles and responsibilities in the TLC. Security knowledge must be a requirement before starting any TLC project. The training should include information about best practices to ensure a safe work environment, security roles and responsibilities within the project phases, and security tasks, as well as security policies, standards, applicable regulations and legislation.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

GO-02	**Governance | Training and Awareness**
Title	Promote security awareness
Description	Include security activities to raise awareness among the team (courses, simulations, talks, etc.) about how to address security during the development process. If the entire team is sensitised to security, it will be easier to implement the necessary measures to achieve a process as secure as possible.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

GO-03	**Governance | Training and Awareness**
Title	Assess the security skills to be updated
Description	A team must stay up to date with the latest security knowledge and certifications of its members, by means of activities, exams, certifications,

	etc. At least once a year this information must be updated.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

GO-04	**Governance | Training and Awareness**
Title	Allocate resources to stay up to date with security topics
Description	Appoint resources and promote the implementation of monitoring, tracking and update activities by means of threat intelligence in order to be aware of the status of current vulnerabilities and new types of attacks that may affect your projects. Along with security lessons learned, this information must be centralised in a repository. The result of these tasks will help to prevent future security issues.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

GO-05	**Governance | Roles and Privileges**
Title	Establish security roles and privileges within the development project
Description	Ensure that development teams work alongside security teams by means of the definition, identification and allocation of functions, responsibilities and tasks in relation to security in all phases of development. This measure ensures that security is addressed when required.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

GO-06	**Governance | Roles and Privileges**

Title	Implement a separation of duties in the work team
Description	It is essential to ensure a proper separation of duties during the development process, implementing security controls in order to prevent security impacts. Without a separation of duties, people could carry out fraudulent activities in any phase by leveraging their privileges. The goal is to avoid the possibility of users having admin rights or inadequate profiles for critical tasks.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

GO-07	**Governance | Roles and Privileges**
Title	Protect the process against privilege abuse
Description	The integrity of the development process must be guaranteed. Implement security measures to access project resources so as to prevent any team member (insider, third-party) with privileges from disabling security controls, establishing or modifying policies and guides, collecting sensitive data, etc. Perform audits periodically to ensure the integrity of information.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

GO-08	**Governance | Roles and Privileges**
Title	Allocate resources for process monitoring
Description	Designate a person to perform, review and put forth improvement actions for the business continuity plan: safeguarding critical points that may slow down or compromise the development process (TLC), like the unavailability of third-party services, the uncontrolled access to sensitive locations where information is stored, the lack or expiry of licences involved in the TLC, etc.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation

	Installation Maintenance Retirement

GO-09	**Governance | Security Culture**
Title	Consult with security experts to improve the process
Description	Engage internal or external security support to complement, support, or cover security aspects and to contribute during specific activities, such as: - Use of external penetration testers during the testing phase to provide different perspectives, adding robustness to the process. - Use of specific expert in security tools to control access to the process resources, increasing the confidentiality and integrity throughout all phases - Use of a coach to bring security into the TLC phases, etc.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

GO-10	**Governance | Security Culture**
Title	Monitor and respond to the supporting security incidents
Description	Allocate resources to monitor, operate and respond to alarms generated by events resulting from the loss or poor performance of the infrastructures that support the TLC, which are essential for correct functioning. This would be the case of communications slowing down or being lost, as well as the loss or unavailability of data repositories, be they owned or through a cloud service, etc.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

PR-01	**Process | Third-Party Management**
Title	Implement a supply chain management plan
Description	During a TLC process, components or services are outsourced to a third-

	party (external supplier). A supply chain management plan should be implemented and integrated into this process to ensure the integrity of the TLC. This plan should include information related to security frameworks to be used, risk management, third-party acquisition management, purchasing contract definition, etc.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

PR-02	**Process | Third-Party Management**
Title	Assess the software and hardware dependency process
Description	Ensure a proper management of third parties and dependencies of the software and hardware development using risk management and integrating security requirements in contracts, ensuring the visibility and traceability of components, documenting all components and subcomponents acquired, managing incidents, scanning dependencies, etc. Public vulnerability databases must be consulted when choosing a third-party software/hardware and dependencies must be checked periodically or every time they are updated. An open source update plan for IoT must be considered and followed, monitoring and managing third-party vulnerabilities. Contract with third parties involved in the TLC should include a clear liability distribution.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

PR-03	**Process | Third-Party Management**
Title	Verify third-party software, hardware and services
Description	Verify that the component provided by third parties meets the TLC security requirements or can guarantee having followed an equivalent process for secure development. It is advisable to check that the requirements have been met at least every time a delivery occurs.

Phases	Implementation Evaluation

PR-04	**Process | Third-Party Management**
Title	Disseminate a communication procedure to request external support
Description	Establish a procedure for the team to know the steps to be taken in the event of requiring support from external providers to face events or incidents concerning cloud services, testing services, etc., indicating at least the person of contact in charge of the service, the request model and communication channel, incident follow-up and management, the remediation, documentation updates, version, etc.
Phases	Installation Maintenance

PR-05	**Process | Operations Management**
Title	Define an Incident Management plan
Description	Provide guidance for the definition and allocation of roles, responsibilities and activities to be implemented by the teams in the event of security incidents. Security incidents pose a higher impact as the TLC process reaches the last stages, so it is crucial to manage it following an established resolution process. This process should contain at least: - Incident detection and registration. - Classification and initial support. - Research and diagnosis. - Solution and service restoration. - Extract security guidance from incident for next generation - Incident closure. - Monitoring, follow-up and communication of the incident. Maintain, to the greatest extent feasible, a full inventory of third party components and dependencies, and track vulnerabilities, patches, and updates to those components to preserve security. An Incident Management Plan should be defined and periodically updated.
Phases	Installation Maintenance

PR-06	**Process | Operations Management**
Title	Define a Change Management plan

Description	A Change Management Plan should be defined to manage any changes that may take place during the TLC process. This entails ensuring control over the budget, schedule, scope, communication, and resources. The main focus is to minimise the impact a change throughout the process could have on the different assets: business, team, users, and other important stakeholders. Change management is a highly important activity both in the development and integration phases (changes may affect the requirements) as well as in the maintenance and retirement, during updates, patches or functionalities changes. The plan should detail a procedure containing at least: - Identification and formal request. - Impact analysis and assessment. - Validation. - Planning and testing. - Implementation. - Monitoring, follow-up and communication of the change.
Phases	Design Implementation Evaluation Installation Maintenance Retirement

PR-07	**Process | Operations Management**
Title	Implement Vulnerability and Patch Management
Description	Develop a process for vulnerability and update management as well as for vulnerability disclosure from external and internal parties to reduce the risk of system failures, especially in operation. This process must encompass identification and patching processes and the communication process with the relevant stakeholders when a vulnerability is discovered. This guide should document the process and controls to be carried out by the project team, such as: - Vulnerability discovery/disclosure - Identification of the affected asset - Development of the solution or patch - Testing, solution compliance - Patch implementation, update - Update follow-up
Phases	Design Implementation Evaluation Installation Maintenance

PR-08	**Process | Operations Management**
Title	Implement Configuration Management
Description	Configuration management focuses on maintaining the integrity of the system, ensuring that uncontrolled changes are implemented during the deployment and maintenance phases of the TLC process. It must be configured in a restrictive way to guarantee maximum resistance against malicious or unintentional attacks.
Phases	Installation Maintenance

PR-09	**Process | TLC Methodology**
Title	Establish a Control Access and Authorisation policy
Description	The access to resources and processes should be protected to prevent users without authorisation from accessing restricted resources (e.g. data repository, password storage, test reports, etc.) at any stage of the TLC process. By establishing user access privileges, it is possible to ensure the confidentiality, integrity and availability of data and process: - Only authorised persons (based on their privileges) will be able to access restricted resources. - The control access will make it possible to identify and audit the accesses that have taken place, establishing internal security controls.
Phases	Implementation Evaluation Installation Maintenance

PR-10	**Process | TLC Methodology**
Title	Define security metrics
Description	Implement security metrics, which should be defined and tracked in order to verify that the specified security requirements have been fulfilled during the development process. Checking the security metrics should be a necessary requirement to: - Evaluate the security maturity and identify actions to improve the process (SMM). - Reassure quality for all TLC phases. - Assess the status of an ongoing process. - Track potential risks.

	- Discover process issues before they become critical. - Evaluate the ability of the project team to control the quality of products. - Update the security metrics during the whole TLC process
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

PR-11	**Process | TLC Methodology**
Title	Define and document the TLC process
Description	Define security guides establishing the performance of security tests during the different phases of development, defining best practices such as the generation of use cases, the performance of penetration tests during development, the use of tools, the performance of security tests at the end of the process, etc. It is recommended to execute this process in every iteration (sprint) or when a modification is implemented.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

PR-12	**Process | Secure Deployment**
Title	Define a disposal strategy
Description	A plan for the withdrawal of the solution at the end of the life cycle must be considered. The plan must include measures to formally retire stored data according to the needs (organisational, data privacy, regulatory compliance) including third-party components and the communication to the stakeholders. To ensure the disposal process, an audit log must be maintained.
Phases	Maintenance Retirement

PR-13	**Process | Secure Deployment**

Title	Establish process for TLC vulnerabilities follow-up, monitoring and updates
Description	Establish a procedure to inform of new published vulnerabilities (e.g. establishing mechanisms to receive feedback from the security research community) that may affect the development life cycle (including those that affect third party components), so that they can be taken into account in all phases. This information measure can help the organisation not to incur into known errors, and to take them into account as security requirements in the requirements phase of the TLC for future developments.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance

PR-14	**Process | Secure Deployment**
Title	Implement a testing strategy
Description	Define a testing strategy and ensure accuracy of testing processes for the development. This strategy should contain considerations such as test scope definition, criteria to be used, quality control points, procedures to solve errors, etc., Testing must be initiated as soon as possible in the development process, with standard development-oriented testing activities, such as security requirements testing, vulnerability assessment, penetration testing. Testing may continue even after production, by regularly repeating tests and performing other activities such as active piracy monitoring, and red teaming
Phases	Implementation Evaluation Installation Maintenance Retirement

PR-15	**Process | Secure Deployment**
Title	Define a secure deployment strategy
Description	Define effective and secure deployment strategy, weighing the options in terms of the impact of change on the targeted systems, and the end-users. It must be considered that only qualified personnel must have access to the deployment environment, audit systems for all deployments establishing versions control, acceptance threshold, person who conducted it, etc.
Phases	Installation

	Maintenance

PR-16	**Process | Security Design**
Title	Provide a secure framework
Description	Adopt a security framework encompassing the necessary requirements in order to define and provide guides and policies to be implemented throughout the Trusted Life Cycle process. Known frameworks minimise risks and threats that could affect the process. Define a secure framework to ensure in-depth defence and observe security by design considering the entire life cycle of the solution and comprising the design, maintenance, and retirement phases.
Phases	Threat Modelling and Risk Assessment Design Maintenance Retirement

PR-17	**Process | Security Design**
Title	Apply least privilege principle
Description	Ensure that user and software privileges are strictly limited to features required to carry out the operations. Limiting permissions and rights in the tasks to be performed is an important activity during the TLC process, gaining greater relevance in the Design and Testing phases. Privileges must have a resilient configuration against unauthorised changes, and must be in line with authorisation and access control policies.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

PR-18	**Process | Security Design**
Title	Verify security controls
Description	Allocate a project resource (i.e. a data repository) to centralise security control management activities (security control updates, tracking, monitoring) to be carried out during the TLC process. Verify that the security controls implemented are accessible, controlled regularly, safe, and reusable, avoiding duplicates and ensuring they are efficient, reliable, and

	based on international best practices. It is recommended to review and update them periodically, at least once a year or upon every important change (new technologies, project's lessons learned, etc.).
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance

PR-19	**Process | Security Design**
Title	Perform a design review
Description	During the Design phase of the TLC process, solutions must be reviewed, by a team which is independent from the designers, from the point of view of security, ensuring that security requirements, which have been previously defined, have been met, identifying the attack surface, carrying out a threat modelling, providing security mechanisms, and scheduling periodic reviews throughout the development process based on milestones. It is recommended to execute this process in every iteration (sprint).
Phases	Design

PR-20	**Process | Security Design**
Title	Specify security requirements
Description	Establishing security requirements prior to development makes it possible to implement security functionalities that ensure compliance with standards and laws and avoid known vulnerabilities. The definition of these security requirements makes it possible to industrialise the security standards that apply to different developments, complying with a series of standard security controls, making it possible to fix past problems, and helping to prevent future flaws. Some best practices would be the performance of security and requirement compliance assessments, the specification of requirements based on known risks, the definition of requirements in agreement with providers, the implementation of security user stories, and the performance of security audits. They must be reviewed periodically, at least every time known best practices and regulations are updated, or each time a security issue is discovered
Phases	Threat Modelling and Risk Assessment

PR-21	**Process | Security Design**
Title	Perform a risk assessment
Description	Identify risks throughout the development process, at every level (system, hardware, software, network, etc., analysing the sources, data storage, applications or third parties. As part of the analysis, make sure that the data to be protected are reliable, and that there are measures in place to prevent the unauthorised access, loss, destruction or manipulation thereof. A security risk assessment should include: - The analysis of the potential risk if the security of each of the following components were compromised: sources, storage, sensitive data, applications, data stores, cloud services. - The analysis of data classification mechanisms and data security capabilities in order to protect sensitive data from unauthorised use, access, loss, destruction or sabotages. - The analysis of the potential for trusted insiders to misuse their privileged access to data. Based on these analyses, implement best practices for the mitigation of each potential security threat. This process must be periodically reviewed, at least once a year.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

PR-22	**Process | Security Design**
Title	Implement Threat Modelling
Description	In the design phase, it is necessary to study the architecture and the design of the system by means of threat modelling techniques. Threat modelling thoroughly identifies key assets thus far hidden, as well as their associated risks. Through this technique, developers can focus their efforts on subsequent phases, applying tools oriented to the uncovered risks. Developers should regard the following aspects as best practices: - Building and maintaining threat models for each application, defining the profile of potential attackers by means of the architecture. - Building and maintaining abuse case models per project, establishing threat assessment systems. Explicitly evaluate the risk of third-party components and generate threat models with security controls.
Phases	Design

PR-23	**Process | Security Design**
Title	Implement data classification
Description	Data are a critical asset from the point of view of security. Based on the classification of information (status, use, owner, risk, etc.), assign a level of sensitivity to the data in the risk assessment phase to establish the corresponding protection measures throughout the TLC process (ensuring the privacy of data at rest by means of encryption, preventing unauthorised access by means of control access, etc.).
Phases	Threat Modelling and Risk Assessment Design

PR-24	**Process | Security Design**
Title	Ensure that the hardware requirements derived from software requirements are considered
Description	Bear in mind that, as part of the functional requirements, it is essential to take into account the implications for hardware derived from software security requirements. Implement controls during the Requirements phase in order to associate/map software security requirements and hardware requirements and ultimately fulfil them. For instance, associate secure boot mechanisms with the use of chips/modules supporting this technology (Root-of-Trust), identifying hardware needs based on the communication protocol chosen in order to determine the power source depending on consumption, etc.
Phases	Threat Modelling and Risk Assessment

PR-25	**Process | Internal Policies**
Title	Establish a communication plan for security measures
Description	Develop a communication plan targeted at all persons involved in the development process (specially third-parties) in order to report on the security measures that must be observed for a proper development, such as applicable regulations, security frameworks and methodologies to be used, security best practices, etc. This plan must be reviewed, validated and disseminated in the team at least once a year.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation

	Installation Maintenance

PR-26	**Process | Internal Policies**
Title	Control the process against information disclosure
Description	Ensure that process information is not disclosed or tampered with by any stakeholder throughout the life cycle without prior authorisation, as it could result in a compromise of intellectual property, a breach of regulatory compliance, reputational losses, etc. Security measures should be considered such as role-based access control, authorisation, permission assignment, non-disclosure clauses in the contracts, etc.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance

PR-27	**Process | Internal Policies**
Title	Verify and ensure the availability of updated security documents
Description	Ensure the availability of security policies, procedures, guides, applicable regulations and requirements for developers. Throughout the process, a centralised repository must be accessible. Organisations have to implement change management to guarantee the integrity of data and avoid introducing errors in the process.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance

PR-28	**Process | Internal Policies**
Title	Plan an alternative for unavailability cases
Description	Distribute your resources so as to not centralise security knowledge in a single resource, be it internal or through a third party, with a view to avoiding cases of unavailability that may bring the TLC process to a standstill in any

	of the phases. This would be the case, for instance, when there is only one security pentesting specialist during the Testing phase. This measure focuses on providing an alternative for TLC critical points (resources redundancy).
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance

TC-01	**Technology | Access Control**
Title	Implement authorisation
Description	Implement access control in the infrastructure to ensure that the system verifies that users and applications have the right permissions allocated to their roles to access system resources. This can be done by means of the least privilege principle and a strategy regarding authorisation policies, controls, and design principles for different categories of data. If a password is being used for authentication, the asset should force the user to change the password at first use. Furthermore, typed characters should be masked.
Phases	Threat Modelling and Risk Assessment Design Implementation Installation Retirement

TC-02	**Technology | Access Control**
Title	Secure storage of users' credentials
Description	Ensure that user credentials of infrastructures are secured. Passwords must always be hashed with a salt. Password bolts are often used to hardcode credentials for system communications, so that the system has to request the credentials before accessing a resource. This measure prevents access to sensitive functionalities and data (e.g. source code).
Phases	Installation Maintenance

TC-03	**Technology | Access Control**

Title	Deploy physical protection for systems
Description	Systems and their corresponding hardware must be protected against unauthorised modification attempts and direct access, as well as other dangers (fire, water, cooling issues, etc.). Physical access must be controlled and unused physical interfaces must be disabled or inaccessible. Removing unnecessary items helps to reduce the attack surface.
Phases	Threat Modelling and Risk Assessment Design Implementation Installation Maintenance

TC-04	**Technology | Access Control**
Title	Implement Key management and authentication mechanisms (e.g. FIDO)
Description	Ensure the secure management of service credentials for your TLC systems. They must be temporary and single-use, and the right communication privileges have to be allocated for the different service credentials (e.g. user credentials vs. System credentials).
Phases	Design Installation Maintenance

TC-05	**Technology | Access Control**
Title	Control the physical access to the critical facilities
Description	Implement a physical access control system with authorization mechanisms to identify users and their privileges. This system should be monitored and provide event logs for all accesses, including unauthorised access attempts. The access to physical facilities storing information concerning the TLC or systems that support the process (repositories, network equipment, documentation files, etc.) must be adequately protected. This measure can be stipulated in contracts with external providers concerning the control of facilities containing information about the service hired. Additionally, a CCTV surveillance system could be configured to communicate with an alarm system (e.g. SIEM) and send signals alerting to unauthorised access attempts.
Phases	Threat Modelling and Risk Assessment

TC-06	**Technology | Third-Party Software**
Title	Use third-party components that are patched for latest known vulnerabilities
Description	Ensure that your TLC model enforces the use of the latest versions of third-party components to safeguard their integrity. The most costly and extensive attacks have been caused by this issue. Check the versions of your dependencies at least quarterly once the software or hardware under construction is in production.
Phases	Design Implementation Evaluation Maintenance

TC-07	**Technology | Third-Party Software**
Title	Use known secure frameworks with long-term support
Description	For the foundation technologies of the software under development, use and verify known software security frameworks from third party providers supplying LTS (Long Time Support) or similar. Some software have associated security flaws, so it is essential to make sure that these components can be trusted in the long term. These components should be chosen considering if they are maintained by a private organisation or an active group, if security patches are available in a short time when a vulnerability is disclosed and if developers can be contacted if a vulnerability is identified.
Phases	Design Implementation

TC-08	**Technology | Secure Communication**
Title	Use secure communication protocols
Description	Ensure that security-relevant communications are always encrypted. Additionally, it is also recommended to implement mechanisms to authenticate communications.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance

TC-09	**Technology | Secure Communication**
Title	Use proven encryption techniques
Description	Security-relevant data must be encrypted, both at rest and in transit, using a recognised encryption algorithm. However, even resilient algorithms are not efficient if they are not properly used (e.g. sufficient key length). It is necessary to use an initialisation vector and to guarantee a minimum level of entropy. It is highly recommended to apply hashes to protect electronic signatures. These measures apply both to original data and to any existing backups.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance

TC-10	**Technology | Secure Code**
Title	Implement secure coding practices
Description	In the TLC process, secure coding practices must be implemented during different phases, including at least: - Proven strong authentication mechanism to access the software (e.g. two-factor authentication, minimum password length, secure transfer, secure connection, secure credential management, etc.). - Handling all errors and anomalous conditions that can compromise of sensitive information about the application - Parameterisation of queries by binding the variables in the corresponding languages to prevent code injections in the query language, and - Validation of input and output for forms' submissions such as with respect to language, characters, etc. (e.g. whitelisting mechanisms). These should be addressed in the TLC to ensure the design, implementation and testing take this into account.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation

TC-11	**Technology | Secure Code**
Title	Provide audit capability

Description	Your TLC model must ensure that the software under development (and IoT systems) include non-repudiation features (design, implementation, testing, etc.). High-value functionalities must be tracked to control critical aspects of the software. This could be mandatory, or highly advisable for regulatory compliance.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance

TC-12	**Technology | Secure Code**
Title	Follow the principles of security by design and by default
Description	Many decisions are made during the design phase, when the final functionality of the solution is devised, including access verifications. These decisions apply to the entire scope of the TLC, implemented in the implementation/development phase, and tested before and after the production environment. The fail-safe principle must be taken into account to prepare the device for errors, anticipate potential disruptions of the service, and respond appropriately to ensure recovery. The principle of least privilege must also be observed to prevent unnecessary or unauthorised accesses. This set of measures is aimed at safeguarding data from being compromised. Implement strong user authentication by enforcing the change of passwords upon first use, and the periodic renewal of passwords (e.g. at least once in 90 days to every 6 months) and session / time lockout upon multiple failed authentication attempts (password, or other).
Phases	Design Implementation Evaluation

TC-13	**Technology | Secure Code**
Title	Implement software development techniques
Description	Use development techniques that make application architecture more flexible. Modular architectures provide great benefits, not only during the operation to speed up updates or identify and troubleshoot, but during development. Developing large and indivisible blocks implies having a large team and making it difficult to define the scope. However, using techniques such as microservices, a large block can be broken down into several to make the development agile, increase flexibility and scalability, facilitate the

	definition of scopes and functionalities, and decrease errors.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance

TC-14	**Technology | Secure Code**
Title	Verify production code
Description	Ensure that production code comes with secure compiler options and does not contain forgotten debug code or debug symbols. In the production environment, security is crucial and it must be carefully controlled by ensuring not only the integrity of the tools but the person's competence conducting these activities.
Phases	Implementation Evaluation

TC-15	**Technology | Security Code**
Title	Ensure security for patches and updates
Description	Patches must be carefully managed and deployed to prevent additional issues with update capabilities. It is necessary to ensure that all IoT elements can be updated and patched, and developers enable notifications of updates and security patches so that users can receive them for having information if, when and how patch software. The installation of security patches and updates should be user-friendly (e.g. automatic or in a few clicks). Update mechanisms include secure/encrypted delivery of updates, validation of signatures on the device before installing the patch (secure boot), etc. Secure over-the-air updates should be considered through a secure mechanism that is cryptographically signed. This must be considered for all IoT systems, as well as for the software under construction already in production (patching as soon as possible for critical vulnerabilities). This measure prevents CVEs exploited by threat agents, and potential legal consequences may arise if due diligence is not in place to keep the systems in a well-fit state.
Phases	Maintenance

TC-16	**Technology | Secure Code**
Title	Implement measures against rogue code and fraud detection
Description	Ensure malicious code is adequately managed (perform manual reviews, protect the code repository against tampering, etc.) in your TLC model. Validate the application source code and third-party libraries (e.g. lack of backdoors, time bombs), and that the application does not grant unnecessary permissions. This measure includes the review of all changes before the deployment of the change.
Phases	Implementation Evaluation

TC-17	**Technology | Secure Code**
Title	Implement anti-tampering features
Description	There must be logical tamperproof measures in IoT systems, that is, measures to monitor and ensure that the most critical assets (e.g. code) have not been tampered with (e.g. code-signing). Tampering could ease the access to sensitive functionalities or data for threat agents, and allow the insertion of rogue code in the software under construction.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance

TC-18	**Technology | Security Reviews**
Title	Apply secure code review
Description	Ensure that your TLC model includes source code reviews. Code reviews can be manual or automated. Good practices recommend performing it manually for each candidate release (i.e. a member of the development team reviews what another team member has developed to ensure quality and share knowledge about the development with the team). This is the only tool available to detect malicious code. Automated code reviews are commonplace and more cost- effective compared to manual ones.
Phases	Implementation Evaluation

TC-19	**Technology | Security Reviews**
Title	Perform an attack surface analysis
Description	Carry out this activity during the design phase to detect any potential threats resulting from weaknesses. Ensure that your TLC model includes this activity to provide value in other phases. It ensures the control of what is susceptible to be misused in the software under development, as well as of potential entry points. It helps to avoid unauthorised activities and data leakages.
Phases	Design

TC-20	**Technology | Security Reviews**
Title	Perform IoT SDLC tests
Description	Ensure that your TLC model makes software, firmware, and hardware undergo testing prior to production to ensure it has no vulnerabilities before deployment. This can be done by means of an audit, and it should be performed at least, annually or for each candidate release.
Phases	Evaluation Installation Maintenance

TC-21	**Technology | Security Reviews**
Title	Design a contingency plan
Description	Take into consideration contingency plans designed to be integrated into the TLC. Some activities of the contingency plan, such as the development of contingency planning policy and completion of the business impact analysis, must be executed in the initial phase of the TLC. However, all the activities of the contingency plan are involved in all the phases but the last one, since once the system is operational, the contingency planning becomes a core part of continuous supervision and other ongoing security management tasks.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance

TC-22	**Technology | Security Reviews**
Title	Monitor requirements to ensure the SDLC success
Description	Implement a system to monitor the requirements agreed by contracts. During the TLC, a partial or full breach of compliance with a requirement is a critical aspect. It would entail an increase in the project vulnerabilities and might even lead the project to fail. It is essential to perform a correct follow-up of the level of compliance reached by the requirements. To this end, key compliance indicators can be used (regarding quality, result required, scope, etc.) by means of a requirement matrix.
Phases	Threat Modelling and Risk Assessment

TC-23	**Technology | Security of SDLC Infrastructure**
Title	Ensure secure Logging and Monitoring Implementation
Description	The components and systems within the development and production infrastructure have to generate high-quality logs, containing information related to security events, and preventing the inclusion of sensitive information. Logs have to be monitored (if possible, in real time using automatic systems), reviewed and analysed by security staff.
Phases	Threat Modelling and Risk Assessment Design Implementation Installation Maintenance

TC-24	**Technology | Security of SDLC Infrastructure**
Title	Implement physical detection systems
Description	Deploy detection systems to control the critical physical environment (workplace, server rooms, etc.) where the TLC infrastructure supports as temperature control, fire/smoke detection, alimentation loss, etc.) in order to avoid the loss of essential support for the TLC such as organisation network, external communication, external services as cloud, internet, surveillance, etc. Deploy backup systems for critical points.
Phases	Threat Modelling and Risk Assessment

TC-25	**Technology | Security of SDLC Infrastructure**
Title	Define a mitigation plan for physical damages

Description	Implement a procedure describing the steps to be taken in order to mitigate the damages that could be caused to the systems where data are stored during the TLC process (communication systems, network equipment, servers, disks, data repositories, computers, etc.), as well as the spaces where they are hosted, to prevent them from being compromised due to a fire, flood, electric shock, etc. It is also important to have a redundant system in place to provide support and prevent alterations in the TLC process.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance

TC-26	**Technology | Security of SDLC Infrastructure**
Title	Use whitelists for allowed applications
Description	Whitelist-based monitoring makes it possible to strengthen the security of connections and servers by controlling the applications. Only authorised applications can be run, thus preventing the execution of unauthorised software or malware. Whitelists must be periodically updated in order to include the latest applications, software has to be patched and tested to verify their functionality, etc.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance

TC-27	**Technology | Security of SDLC Infrastructure**
Title	Audit the access to the TLC infrastructure
Description	Collect security logs to audit access to the TLC resources, such as access to information in servers, files, data stored in physical rooms, etc. Regardless of whether the accesses are physical or logical, they have to be analysed with security tools (e.g. SIEM) to register the events (access to information, downloads, modifications, erasure attempts, etc.), identify users, and monitor the correct functioning of the process in order to generate alarms if security is compromised. These logs must be stored in a safe location and erased once the period of time stipulated by the industry

	elapses (e.g. erasure of financial data after 5 years).
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

TC-28	**Technology | Security of SDLC Infrastructure**
Title	Implement an identification protocol in your facilities
Description	Disseminate among internal and external employees of the organisation a policy on how to adequately identify themselves in the facilities, and on how to act and where to go if they detect unauthorised individuals attempting to access the facilities of the organisation for malicious purposes such as sabotage, industrial espionage, or the theft of confidential information.
Phases	Threat Modelling and Risk Assessment

TC-29	**Technology | Secure Implementation**
Title	Enforce the change of default settings
Description	Security does not end once the hardware is produced. During the operation it is necessary to enforce the end users to safely utilise the device. Therefore, mechanisms must be established during the TLC process to ensure it, namely: not allowing operation with password and user by default, ensuring that passwords have a minimum level of security (length, characters, etc.), including functions to manage user passwords (e.g. enforcing change cycles every 90 days, etc.), closing the user session after an inactivity time, locking the access out after multiple authentication fails, enable user notifications of updates, etc.
Phases	Installation Maintenance

TC-30	**Technology | Secure Implementation**
Title	Use substantiated underlying components
Description	Choose well-supported underlying components that do not require customizations that may lead to losing security oversight and use proven tools to apply security hardening practices.

Phases	Implementation Evaluation

TC-31	**Technology | Secure Implementation**
Title	Provide secure configuration options for end users
Description	Ensure that the TLC process addresses the provision of adequate measures in order to include different setting options for end-users upon first usage of an IoT solution to enable a continuous improvement of security, such as, for instance, the ability to disable features or functionalities that are not going to be used or to add automatic security check mechanism.
Phases	Installation Maintenance

TC-32	**Technology | Secure Implementation**
Title	Implement interoperability open standards
Description	Whenever possible, implement technologies based on open standards to ensure that communication and integration between different devices is secure and reliable.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

TC-33	**Technology | Secure Implementation**
Title	Enable devices to advertise their access and network functionality
Description	By enabling devices to advertise their intended and supported functionality, the threat surface can be significantly reduced. An indicative practical example involves the use of IETF RFC 8520 on Manufacturer Usage Description Specification.
Phases	Design Implementation

The following requirements are introduced by us as context-specific for ORSHIN:

ORSHIN-HD-01	**Process | Hardware design**
Title	Create a design milestone schedule
Description	Define a schedule of relevant milestones for various steps of hardware design, according to the specific technological needs that emerge from the hardware production process. For instance, the manufacturing of a silicon IP requires access to highly specialised semiconductor fabrication plants, also called foundries, and this forces the need to plan milestones in advance and to strictly adhere to the timeline. Failure to adequately meet the milestone schedule can result in inefficient management of very expensive resources, leading to unexpected costs and delays.
Phases	Design Implementation Evaluation Installation Maintenance

ORSHIN-HD-02	**Governance | Hardware design**
Title	Implement role management for milestone schedule
Description	Identify appropriate roles for milestone management and sign-off.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

ORSHIN-HD-03	**Process | Hardware design**
Title	Create a resource/performance evaluation strategy
Description	Define a strategy for evaluating the resource/performance ratio, according to industry-standard criteria and market consensus.
Phases	Design Implementation Evaluation

ORSHIN-HD-04	**Process | Hardware design**

Title	Implement a testing-oriented design approach
Description	Insert dedicated testing logic in the product that allows efficient testing and debugging of root causes for issues when prototyping. Testing logic can be separated from the product's functionalities (i.e. can be non-necessary for implementing the product essential functions).
Phases	Design Implementation Evaluation

ORSHIN-HD-05	**Process | Hardware design**
Title	Monitor and measure the production yield
Description	Define metrics that can be monitored and measured during the whole production cycle, in order to identify weaknesses and critical points, and to plan remediations
Phases	Installation Maintenance

ORSHIN-HD-06	**Technology | Hardware design**
Title	Apply hierarchical and modular design approach
Description	Apply a hierarchical modular approach to design, by recursively dividing systems into modules, reuse regular modules when possible, and define well-formed interfaces between modules and sub-systems.
Phases	Design

ORSHIN-HD-07	**Process | Hardware design**
Title	Design/implementation transparency
Description	Design the product so that it is possible to map the implementation onto the design through reverse engineering, in a simple way that maximises transparency and minimises friction. The goal is twofold: ensuring that the design has not been altered in the implementation (e.g. for inserting backdoors), and that it is easy to evaluate the product security, in particular certifying that it does not rely on "security-through-obscurity".
Phases	Design Implementation

	Evaluation

ORSHIN-HD-08	Process | Hardware design
Title	Hardware/software co-design
Description	Ensure that communication between the hardware and software project teams is facilitated and encouraged, in order to have a coherent product development without the creation of "silos".
Phases	Design Implementation

ORSHIN-HD-09	Process | Hardware design
Title	Implement a design to facilitate fuzz testing
Description	Design the product to permit and facilitate its own testing through fuzzing techniques; also, provide information and tools to support the fuzzing of the interfaces of the design. A prerequisite for this is having clear specification for software/hardware protocols that are to be tested with techniques on the various interfaces of the product.
Phases	Design Implementation Evaluation

ORSHIN-HD-10	Technology | Hardware design
Title	Side-channel protection
Description	Ensure that the design of the product and its implementation take into account the threat of side-channel attacks, and ensure appropriate resistance against them, with a level that is compliant with the product's threat modelling.
Phases	Design Implementation Evaluation

ORSHIN-HD-11	Technology | Hardware design
Title	Fault-injection protection

Description	Ensure that the design of the product and its implementation take into account the threat of fault-injection attacks, and ensure appropriate resistance against them, with a level that is compliant with the product's threat modelling.
Phases	Design Implementation Evaluation

ORSHIN-HD-12	**Technology | Hardware design**
Title	Evaluate design tools
Description	Investigate design tools to make sure synthesis or other stages of the processing do not insert weaknesses inside the design.
Phases	Design Evaluation

ORSHIN-HD-13	**Process | Hardware design**
Title	Make sure that board layout does not exposes weakness
Description	Evaluate layout in terms of attacker physical access. (Critical signal routing / removal of test pads on final hardware / etc)
Phases	Implementation Evaluation

ORSHIN-HD-14	**Process | Hardware design**
Title	Create a cycle taking into account evaluation and design
Description	Plan security testing at each step of the process (design, layout, physical implementation and go back in the process according to fix the issues). Define testing strategies for different stages of the production of a hardware design (design, layout, netlist).
Phases	Design Implementation Evaluation

ORSHIN-HD-15	**Technology | Hardware design**

Title	Instruction Modification Protection
Description	For CPU developments, instructions coherency should be checked. Use of an instruction trap for "undefined" opcodes should be put in place.
Phases	Design Implementation Evaluation

ORSHIN-HD-16	**Technology | Hardware design**
Title	Instruction Flow Modification Protection
Description	For CPU developments, instruction flow modification being the basis of extraction attacks, checking its coherency seems a legitimate feature
Phases	Design Implementation Evaluation

ORSHIN-OS-01	**Process | Open source**
Title	Use online repositories to share open source hardware project
Description	Ensure a clear way of sharing open source hardware projects files, through the use of an online repository (like GitHub or GitLab). All files (design, bill-of-materials, assembly instructions, code, etc) should be version controlled where possible. Most online repositories also include issue trackers, which are a good way to keep track of the bugs in and future enhancements, in a way that others can view and comment on. As an alternative to an online repository, an online CAD tool (like Upverter) or a site like Thingiverse can be used. Reference: https://www.oshwa.org/sharing-best-practices/
Phases	Design Implementation Evaluation Installation Maintenance

ORSHIN-OS-02	**Process | Open source**
Title	Licence open source hardware project designs and derivative works
Description	Apply an open source licence to the hardware design files and other

	documentation. In this way, ensure to make clear the ways in which third-parties should use the project designs. In particular, the licence shall allow modifications and derived works, and shall allow them to be distributed under the same terms as the licence of the original work. Reference: https://www.oshwa.org/sharing-best-practices/
Phases	Installation Maintenance

ORSHIN-OS-03	**Governance | Open source**
Title	Establish official communication channels for open source hardware projects
Description	Establish official communication channels, such as mailing lists, web forums, blogs and public meetings, for discussions, announcements and other relevant communications about open source hardware projects.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

ORSHIN-OS-04	**Governance | Open source**
Title	Implement role management for open source-related aspects
Description	Identify appropriate roles for management of open source-related aspects, such as the definition and updating of open source licenses, the monitor of the product's use with respect to said licenses, etc.
Phases	Threat Modelling and Risk Assessment Design Implementation Evaluation Installation Maintenance Retirement

ORSHIN-OS-05	**Process | Open Source**
Title	Selection of tools for hardware design
Description	Use free and open source software design (CAD) tools where possible. If

	that’s not feasible, try to use low-cost and/or widely-used software packages. Ref: https://www.oshwa.org/sharing-best-practices/
Phases	Design Evaluation

ORSHIN-OS-06	**Process | Open Source**
Title	Selection of third-party components
Description	To make it easier for others to replicate and modify the hardware, when possible it is better to prefer the use of free and open source third-party components, as opposed to proprietary technology.
Phases	Design Evaluation