SPARTA

D5.2

Demonstrators specifications

Project number	830892
Project acronym	SPARTA
Project title	Strategic programs for advanced research and technology in Europe
Start date of the project	1st February, 2019
Duration	36 months
Programme	H2020-SU-ICT-2018-2020

Deliverable type	Report
Deliverable reference number	SU-ICT-03-830892 / D5.2/ V1.0
Work package contributing to the deliverable	WP5
Due date	January 2021 – M24
Actual submission date	29th January 2021

Responsible organisation	TEC
Editor	Cristina Martinez
Dissemination level	PU
Revision	V1.0

Abstract	This deliverable provides the technical specifications of the CAPE demonstrators (Connected Car and e-Government) and includes contributions on integration mechanisms coming out of the CAPE tasks 5.1, 5.2, and 5.3.
Keywords	assessment, certification, safety, security, connected cars, platooning, e-government, requirements

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830892.

Editor

Martinez Cristina (TEC)

Contributors (ordered according to beneficiary numbers)

Maroneze André (CEA)

Massonnet Philippe, Dupont Sébastien, Grandclaudon Jeremy (CETIC)

Nigam Vivek, Dantas Yuri Gil (FTS)

Plate Henrik (SAP)

Sykosch Arnold, Ohm Marc (UBO)

Cakmak Eren (UKON)

Athanasios Sfetsos (NCSR)

Jiménez Víctor (EUT)

Amparan Estibaliz, López Angel (TEC)

Apvrille Ludovic, Blanc Gregory, Debar Hervé (IMT)

Andrea Bisegna, Carbone Roberto, Verderame Luca, Ranise Silvio (CINI)

Bernardinetti Giorgio, Palamà Ivan, Pellegrini Alessandro, Restuccia Gabriele, Sirbu Gheorghe, Spaziani Brunella Marco (CNIT)

Yautsiukhin Artsiom (CNR)

Porretti Claudio (LEO)

Klein Jacques, Samhi Jordan (UNILU)

Reviewers (ordered according to beneficiary numbers)

Jensen Thomas (INRIA)

Bruce Evaldas (MRU)

Disclaimer

Executive Summary

Deliverable 5.2, CAPE Demonstrators Specifications, is the second deliverable of the CAPE program. CAPE stands for Continuous Assessment in Polymorphous Environments. This scientific activity of the SPARTA project addresses the issue of assessing cybersecurity performance of two environments, security and safety co-design on the one hand, and complex software systems of systems on the other hand.

This deliverable is the continuation of D5.1 (Assessment specifications and roadmap), contributing with D5.3 (Demonstrators Prototypes) to the documentation of the first design-implement-integrate cycle of the CAPE program, providing a full picture of the scientific contributions of the CAPE program.

The first contribution of the deliverable is the development of the SPARTA Cybersecurity assessment framework that maps security assessment tools to the security engineering process for continuous assessment. The framework defines the lifecycle process phases for security and safety engineering, and certification evaluation. To understand how SPARTA assessment tools can be used in these processes, each tool has been associated with one or more lifecycle phases where they can be used. The framework and associated tools provide information that can help design specific cybersecurity assessment processes. In the context of the SPARTA project, we have explored the use of continuous integration methods that provide loosely coupled integration for some of the framework tools. Some of the demonstrators rely on DevSecOps approaches that allow security activities to be integrated in the DevOps process phases.

The second contribution of the deliverable is the development of a methodology and framework to jointly reason on safety and security properties in the context of a critical system, in our case the connected vehicle platooning scenario. This deliverable proposes the first formal framework (to our knowledge) to reason jointly about security and safety in the context of a critical cyber-physical system. The analysis develops the safety and security objectives of the use case independently. Several propositions are then developed to mix the two approaches, trade-off analysis between the safety and security objectives, requirements engineering and security/safety by design.

The third contribution of the deliverable is the development of a methodology to address security in complex software systems that are built in an agile fashion with short and frequent release cycles, and which depend to a considerable extent on 3rd party open source components. It focuses on detecting vulnerable or malicious code, introduced either inadvertently by benign software developers, or intentionally by malicious actors. The tools developed in this context are particularly relevant for maintaining security in large software systems and services, thereby analysing and addressing latest trends regarding supply chain attacks through malicious open source. Beside actual tools, the contributions also comprise public datasets to enable further research within and beyond the SPARTA research project.

The fourth contribution of the deliverable is the specification of our two use cases, the “Connected and Cooperative Car Cybersecurity” vertical and the “Complex System Assessment including large software and open-source environments, targeting e-Government services” vertical. These two vertical use cases are particularly representative of the cybersecurity issues that modern digital systems are facing. Both use-cases are thoroughly described and analysed, in order to provide a strong and common vision of the validation and demonstration activities to be developed in deliverable 5.4 (Demonstrators evaluation).

The fifth contribution of the deliverable is the description (or extended description with respect to D5.1) of 18 tools related to assessment of software systems, covering the entire extended software development lifecycle. Several tools address multiple points of the software validation cycle, often related, such as design on the down-side and validation on the up-side. Out of these 18 tools, 2 are stand-alone and independent of our use cases, 9 are applicable to vertical 1 (Connected Car use

case) and 7 are applicable to vertical 2 (e-government use case). The descriptions of the tools have been significantly improved from D5.1, while keeping a similar formalism to facilitate understanding of the tools. Each of the tools has provided a detailed technical specification, describing the internal functions of the tool.

The deliverable also addresses the program planning for implementation and experimentation with the tools. When we identified tools that had the same (or very close) assessment targets, rather than implement two times the same tool (with different techniques), we harmonized the specification of the tools so that they had complementary goals. This implemented a cooperating rather than a competing governance model, focusing on leveraging synergies and competencies between researchers to extend the coverage of our research activities.

Table of Content

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Scope and Purpose

The CAPE program address the issue of assessing cybersecurity performance, through tasks T5.2 about security and safety co-design, and T5.3 about complex software systems of systems, each of these tasks focusing also on an use case.

The outcome of the WP5 tasks takes the form of a generic continuous assessment framework based on the **V-Model software development** process (see Figure 1):

- Task 5.1 (Assessment procedures and tools) focuses on the **framework specification**, describing how the various tools that compose the framework can contribute to the continuous assessment process.
- Task 5.2 (Convergence of security and safety) proposes techniques for integration of security and safety on the **connected car vertical** such as safety-security co-analysis techniques, requirements engineering, modelling and implementation, safety and security co-verification and validation techniques, etc.
- Task 5.3 (Risk discovery, assessment and management for complex systems of systems) proposes a set of tools that can be used by software development organizations for compliance activities, by detecting the presence of known **security vulnerabilities in 3rd party software** and addressing supply chain attacks.
- Task 5.4 (Integration on demonstration cases and validation) demonstrates the continuous assessment framework in the **connected car** and **e-government** verticals by verifying the **evaluability** of the two verticals.

The first CAPE deliverable was D5.1 [1], delivered at M12, which set the scene of the CAPE program activities, defining the first specifications for the development and demonstration of the assessment tools being developed or extended in the CAPE research program.

D5.2 is the second CAPE deliverable and includes contributions for each task and vertical in the context of the CAPE program. It reports the work that has been conducted by the CAPE partners over the last 12 months on defining technical specifications for the development of the assessment tools and the demonstrators.

The third CAPE deliverable is D5.3 [2], delivered at the same time than D5.2. It starts from the specifications defined in D5.2 and describes the implementation of the tool prototypes and their integration in the demonstrators. The fourth CAPE deliverable is D5.4 [3], to be delivered at M36, that will include the validation and demonstration of the two vertical use cases.

CAPE tasks have addressed several ambitious technological challenges over the last 12 months which are described in detail in this document.

The main technical challenges in **T5.1** (see Chapter 2) include the adaptation of the various tools to provide incremental and continuous operation modes, and the lack of common ground for integration between the tools. Those challenges led to the development of connectors to continuous integration and deployment orchestration platforms (ex. Gitlab-CI and GitHub Actions) for several of the tools that can automate previously manual assessment steps, and the use of standard protocols and exchange formats (SARIF, SCAP, ...) for the communication between tools when transitioning between the various certification and assessment steps.

The main technical challenges in **T5.2** (see Chapter 3) include the development of common semantics for safety and security analysis as well as the clarification of possible interactions between safety and security analysis. These challenges need to be tackled successfully to provide, e.g., safety and security co-analysis techniques. T5.2 also provides validation techniques for safety and security by using formal verification. Technical challenges to achieve this include the formalization of a platoon model, the formalization of parametric intruder models to subvert communication channels to carry out attacks, as well as the implementation of such models in a formal verification tool to automatically verify platoon specifications.

The main technical challenges in **T5.3** (see Chapter 4) relate to the unavailability of public datasets with detailed, code-level information about known vulnerabilities in open-source components and open-source supply chain attacks. Such data is essential for developing detective and preventive countermeasures, especially when it comes to AI/ML-based techniques. Several works performed in T5.3 address this lack of public datasets, namely the two open-source projects Project KB and Backstabbers Knife Collection, as well as SafeCommit, which aims at identifying security-introducing commits, and whose results will be published as part of the before-mentioned datasets.

D5.2 also contains the specification of the CAPE use cases, the “Connected and Cooperative Car Cybersecurity” (a.k.a. Connected Car) vertical and the “Complex System Assessment including large software and open-source environments, targeting e-Government services” (a.k.a. e-Government) vertical. These two vertical use cases are particularly representative of the cybersecurity issues that modern digital systems are facing. Both use-cases are thoroughly described and analysed, in order to provide a strong and common vision of the validation and demonstration activities to be developed in deliverable 5.4 (Demonstrators evaluation).

The main technical challenges in the **Connected car** use case (see Chapter 5) are the implementation of countermeasures to mitigate the injection of false messages into the CACC communication channels (Basic Scenario). Dynamic orchestration of security services in cloud/edge infrastructures raises the challenges of how to guarantee the continuity of the assessment. Inputs and output of several assessment steps such as vulnerability scan reports and risk assessment need to be identified and made available as early as possible (Scenario 2). The Verification Tool Scenario (Scenario 3) takes into account the inputs provided by previous steps performed by safety and security co-analysis techniques. This leads to a technical challenge consisting of a mix between currently standards, such as CC standard, and design of a HW and SW setup for testing the rovers. One of the goals of using the OpenCert tool for Safety and Security compliance assessment and certification (Scenario 4) is the digitization of both safety and security standards. The main challenge is the use of both standards in parallel but keeping in mind that one standard is not in conflict with the other. If there is a conflict, there shall be another goal to assess what the conflict is and how to obtain the most adequate scenario. The challenge of the Sabotage tool in the Fault-injection and analysis of faulty scenarios (Scenario 5) is to perform an early analysis of the algorithm used by the plausibility checks developed in the Basic Scenario. The goal is to observe through simulations the behaviour of the algorithm under the different effects that may be produced by carrying out attacks on the vehicles. Based on the simulation results, modifications may be made to the algorithm in the early stages of its development.

The main technical challenges in the **e-Government** use case (see Chapter 6) are posed by the complexity of the real-world innovative authentication solutions based on the usage of the Italian national electronic identity card. The challenges include the identification of the relevant components of the complex system in the scope of the demonstration and their security requirements. Then, it is envisaged the selection of the CAPE tools capable to increase the security of the components and the specification of DevSecOps pipelines to properly integrate the CAPE tools in the complex environment already in place. The final challenge is posed by the assessment of the adoptability of the proposed framework, by showing how the deployed DevSecOps scenarios can be used by end-users willing to include the CAPE assessment tools in their pipeline and perform a security assessment of their complex systems.

Finally, please note that the vertical related to financial services that was identified in the SPARTA DoA has not been further pursued. As it was explained in D5.1 [1], this vertical was originally meant to demonstrate assessment tools developed in the context of CAPE task 5.3, however further investigation revealed that those tools are largely independent of a given industry or vertical and their specific security and certification requirements. Moreover, it turned out that many tools developed by CAPE partners target specific technologies that are not present in the software application part of the financial services use case, thus, cannot be demoed in this context. For those reasons, it was decided to demonstrate tools developed as part of CAPE task 5.3 at the example of the other use-cases, which also allows to focus CAPE partners’ efforts.

1.2 Structure of the Document

The structure of the document is organized as follows:

- **Chapter 1** is the current section presenting the objectives, scope and structure of the document.
- **Chapter 2** presents the technical specifications of the Assessment Procedures and Tools resulting from the work in T5.1.
- **Chapter 3** details the technical specifications of the techniques for integration of security and safety that have been developed by the partners in T5.2.
- **Chapter 4** details the technical specifications of the techniques for detecting security vulnerabilities in 3rd party software and addressing supply chain attacks that have been developed by the partners in T5.3.
- **Chapter 5** presents the technical specifications for the implementation of the Connected Car vertical use case.
- **Chapter 6** presents the technical specifications for the implementation of the e-Government vertical use case.
- **Chapter 7** describes the technical specifications of the CAPE Assessment tools.
- **Chapter 8** presents the conclusions of the report.

Chapter 2 Assessment Procedures and Tools (T5.1)

2.1 Context and Background

Task 5.1 addresses the aspects related to assessment automation, augmenting the assessment toolbox to support pre-assessment by users, as well as incremental assessment and continuous monitoring.

The assessment tools being developed or extended in the CAPE research program are presented in the form of a cybersecurity assessment framework. The role of the framework is to describe in which phase of the security engineering process each of the assessment tools can be used. The framework also takes into account safety engineering and cybersecurity certification evaluation processes in order to explain how each of the assessment tools could also be useful in these processes.

Figure 1 shows the **SPARTA Cybersecurity assessment framework** that has been created in the context of the SPARTA CAPE program. Using the V-Model is a good compromise to compare security engineering and safety engineering processes. The security engineering process covers both software and hardware development; however, the focus is on software development. The safety certification process is not considered in the SPARTA assessment framework because it is beyond the scope of the SPARTA project. For the cybersecurity certification process the Common Criteria standard (ISO/IEC 15408 and ISO/IEC 18045) is used.

The framework covers the following phases of the software lifecycle:

- the **design phase** is assumed to be iterative and covers requirements, architecture, design, development, unit testing, integration testing, acceptance testing and deployment;
- the **operation phase** when a system is running in its target environment;
- the **end of life** phase when the system is taken out of operation.

The assessment tools of the SPARTA assessment framework can be used during different phases of the software lifecycle (see D5.1 [1], section 3.1.1.1).

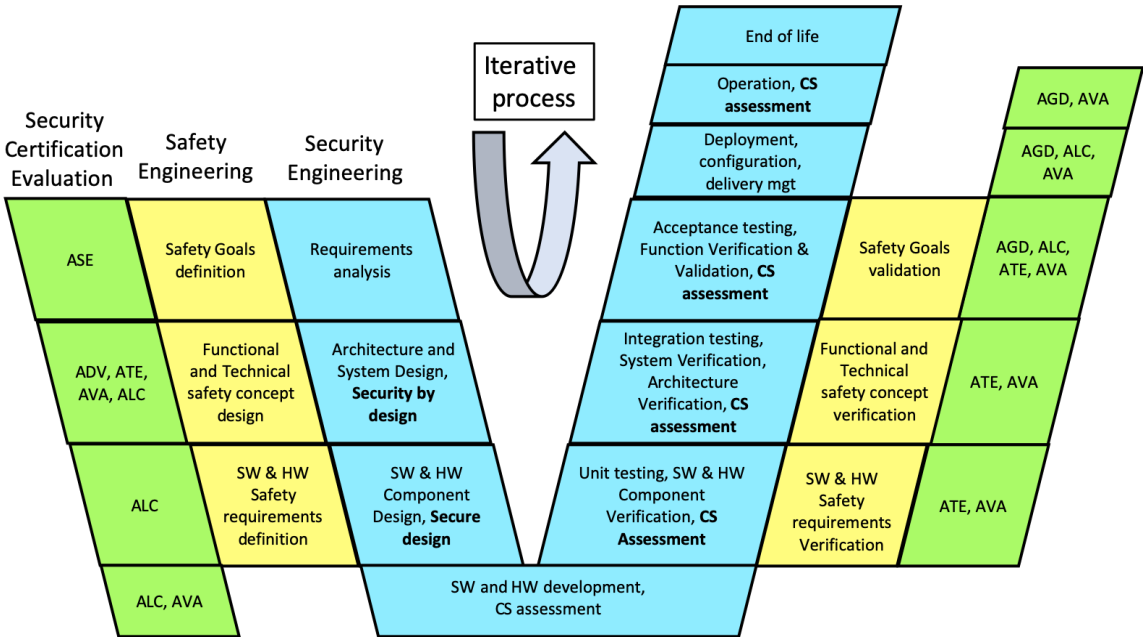

Figure 1: V-Model - Certification for safety and security

Figure 2 shows the **Roadmap** that was defined in D5.1 [1] for the development of the tools:

- M12-M15 / M25-M26: **detailed design** of the changes and additions to the various tools based on the use cases.
- M14-M18 / M27-M29: **implementation of a first/second prototype** version of the use cases.
- M14-M23 / M29-M36: **verification and validation** that the framework tools software requirements are satisfied by the implementation.
- M16-M23 / M27-M36: **integration** of the various tools to obtain a first/second prototype version of the use cases.

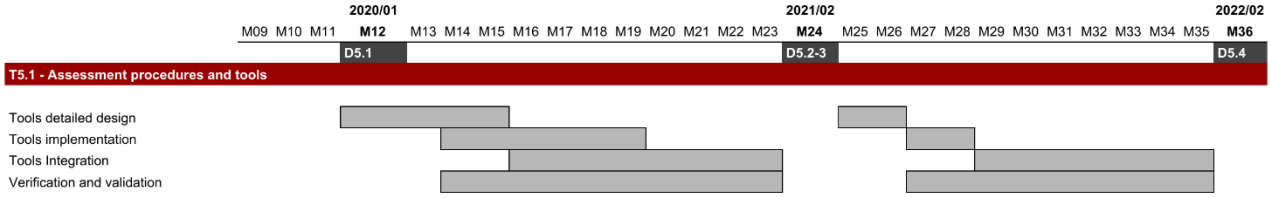

Figure 2: CAPE T5.1 Roadmap

2.2 CAPE Assessment Tools

Table 1 and Figure 3 provide a summary of the CAPE Assessment tools. For each tool, we indicate its name and acronym, the partner in charge of its development, the V-model phases supported by the tool in the SPARTA Cybersecurity assessment framework (see Figure 1), the related task in the CAPE program and the vertical use case in which the tool will be validated. Those tools that cannot be demonstrated in the context of the verticals will be demonstrated independently.

Note that, in order to facilitate the reading of the document, the technical specifications of the CAPE tools prototypes have been included at the end of this document (see Chapter 7).

Tool	Partner	V-model Phase	Task	Scenario	Tech. Specif.
Approver (RAA)	CINI	Development process	T5.3	e-Government (Vertical 2)	Section 7.1
AutoFOCUS3 (AF3)	FTS	Development process; All phases	T5.2	Connected Car (Vertical 1)	Section 7.2
Buildwatch (BW)	UBO	Application development	T5.3	e-Government (Vertical 2)	Section 7.3
Frama-C (FC)	CEA	Development, Unit testing	T5.3	Connected Car (Vertical 1)	Section 7.4
Legitimate Traffic Generation System (LTGen)	IMT	Operations	T5.1	Stand-alone	Section 7.5
Logic Bomb Detection (TSOpen)	UNILU	Design (from unit testing to acceptance testing)	T5.3	e-Government (Vertical 2)	Section 7.6
Maude (MAU)	FTS	Verification and Validation	T5.2	Connected Car (Vertical 1)	Section 7.7
NeSSoS Risk assessment tool (RA)	CNR	Risk Management process at the global level	T5.1	e-Government (Vertical 2)	Section 7.8

Tool	Partner	V-model Phase	Task	Scenario	Tech. Specif.
OpenCert (OC)	TEC	Safety Goals definition; Safety Goals validation, Safety Analysis, Trade-Off Analysis, Assessment	T5.2	Connected Car (Vertical 1)	Section 7.9
Project KB (KB)	SAP	All phases	T5.3	e-Government (Vertical 2)	Section 7.10
Risk assessment for cyber-physical interconnected infrastructures (MRA)	NCSR	Requirements analysis	T5.1	Connected Car (Vertical 1) - security profile	Section 7.11
Sabotage (SB)	TEC	Functional and technical Safety concept design; Functional and technical Safety concept verification	T5.2	Connected Car (Vertical 1)	Section 7.12
SafeCommit (SF)	UNILU	Software development (of the libraries used by the application)	T5.3	Connected Car (Vertical 1)	Section 7.13
SideChannelDefuse (FS)	CNIT	Deployment	T5.1	Stand-alone	Section 7.14
Steady (VA)	SAP	Design (from component design to deployment) and Operations	T5.3	e-Government (Vertical 2)	Section 7.15
SysML- Sec (TTool)	IMT	All phases	T5.2	Connected Car (Vertical 1)	Section 7.16
VaCSInE (VCS)	CETIC	Operations	T5.1	Connected Car (Vertical 1)	Section 7.17
Visual investigation of security information (VI)	UKON	Security Analysis, Verification and Validation	T5.3	e-Government (Vertical 2)	Section 7.18

Table 1: Summary of CAPE Assessment Tools

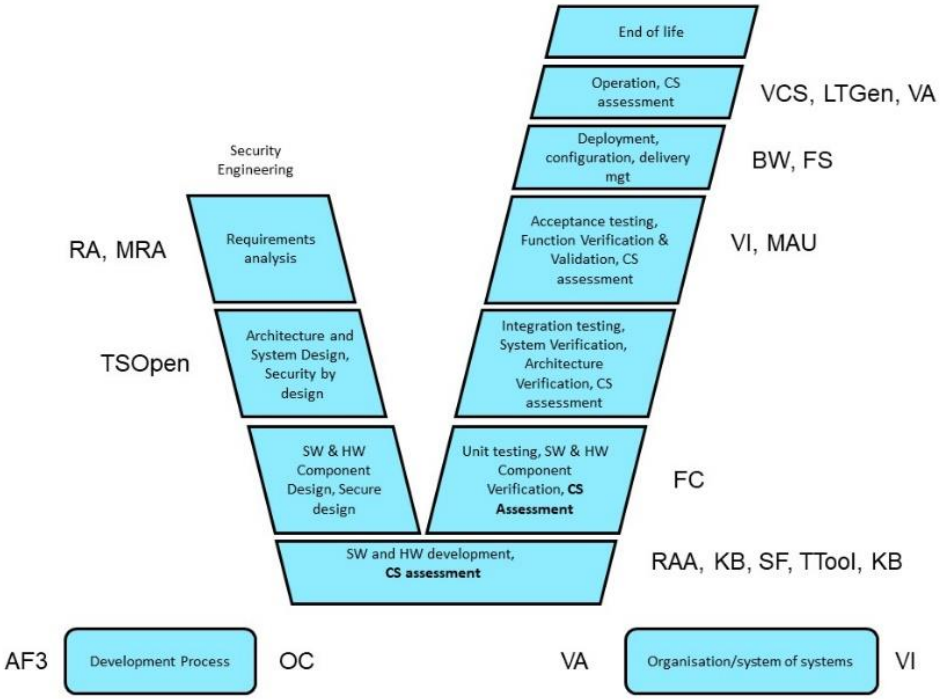

Figure 3: CAPE assessment tools in the Security Engineering V-Model

To provide a clear view of the progress of the tools maturity in the context of CAPE, Table 2 summarises the technology readiness level (TRL) of each tool at the start of the SPARTA project compared to now and what is the target at the end of the SPARTA project. When no development has been planned in the context of SPARTA, the target TRL is marked as "-".

Technology Readiness Levels (TRLs) are indicators of the maturity level of particular technologies. This measurement system provides a common understanding of technology status and addresses the entire innovation chain. There are nine technology readiness levels; TRL 1 being the lowest and TRL 9 the highest [4] [5].

Tool	Partner	Start TRL	Current TRL	Target TRL
Approver (RAA)	CINI	9	9	-
AutoFOCUS3 (AF3)	FTS	7	7	-
Buildwatch (BW)	UBO	1	4	6
Frama-C (FC)	CEA	2	3	4-5
Legitimate Traffic Generation System (LTGen)	IMT	3	3	4-5
Logic Bomb Detection (TSOpen)	UNILU	2	4	5
NeSSoS Risk assessment tool (RA)	CNR	5	6	7
OpenCert (OC)	TEC	5	5	-
Project KB (KB)	SAP	2	4	5
Risk assessment for cyber-physical interconnected infrastructures (MRA)	NCSR	3	3	5

Tool	Partner	Start TRL	Current TRL	Target TRL
Sabotage (SB)	TEC	3	3	4
SafeCommit (SF)	UNILU	2	3	5
SideChannelDefuse (FS)	CNIT	2	4	5
Eclipse Steady (VA)	SAP	9	9	-
SysML- Sec (TTool)	IMT	4	4	5
VaCSInE (VCS)	CETIC	2	4	5
Visual investigation of security information (VI)	UKON	1	4	4

Table 2: SPARTA Assessment Framework tools Technology Readiness Level progress

2.3 Continuous Integration

In recent years, the need to improve software delivery in terms of speed and quality has given rise to a set of practices that combine continuous build, testing, integration, delivery, ... The DevOps approach, closely related to Agile software development method, combines software development (”Dev”) and operations (”Ops”) processes to ensure that new features are added to a software solution in the shortest time possible, and with a high level of quality. This approach emphasizes the importance of communication between the involved parties, including the whole production chain (developers, sys-admins, network team, …), to break the classic “silos” of specialists. DevOps relies on the “CAMS” (Culture, Automation, Measurement, Sharing) characteristics and on a “shift to the left” where aspects such as resilience or security are taken into account sooner in the software development lifecycle (architecture design, coding, pre-production, ...).

DevOps is focused on producing quality code, quickly and reliably. The security problematic is not directly addressed in this approach and DevSecOps is aiming to correct this by complementing DevOps with security procedures to ensure continuous security assessment.

Security is now a shared responsibility between all the actors of a project, at every stage of the software development lifecycle (SDLC). To reach that goal, the reflection has to start from the very beginning, several tools and methodologies will be needed, along with a good deal of automation. DevSecOps, as DevOps, is not only about tooling but also about changing mentality and bad habits.

The benefits of DevSecOps can impact the SDLC in various ways. The left-shift in security integration provides a better approach to security by intervening earlier in the deployment cycle and thus detecting security issues sooner, similarly the automation of security enables a continuous monitoring of the system where vulnerabilities are detected with minimal human intervention. DevSecOps also provides value by reducing the cost of making mistakes, detecting them, investigating their cause and fixing the problems. Finally, security concerns are among the major concerns that limit the adoption of DevOps processes, DevSecOps proposes tools and methodologies to ease this friction.

In the context of CAPE, we leverage DevSecOps to integrate the incremental certification process with the continuous integration. When possible, CAPE tooling provides an interface to use in mainstream continuous integration systems (see for example Frama-C, VaCSIne, TSOpen, Approver and Steady GitLab-CI/GitHub Actions integration), existing security tooling (see VaCSIne integration with OpenSCAP or Frama-C SARIF adoption in Section 5.4), etc. The demonstrations of the verticals illustrate various continuous assessment processes where CAPE tooling is included as steps in the continuous integration.

Various additional technical challenges have been identified during the course of the CAPE activities. For example, static code analysis run time or security services deployment duration need to be reduced in order to provide a more reactive incremental assessment, as they can currently take a long time to complete. Solving those issues would help widespread adoption of SPARTA tools, by improving the reaction time and lowering the entry cost.

Chapter 3 Convergence of Security and Safety (T5.2)

3.1 Context and Background

This section describes the specifications efforts carried out by the CAPE partners in task T5.2 Convergence of Security and Safety. The goal of this task is to advance the techniques and tools for the integration of safety and security, which is particularly important given the increased interconnectivity of safety-critical systems, such as autonomous cars. Since attackers could exploit the increased attack surface to cause harm and accidents by disabling, for example, safety features, countermeasures are needed.

Figure 4 shows the T5.2 roadmap activities that were defined in D5.1 [1]. This roadmap started with the description of the Connected Car vertical (Vertical 1), that was detailed in D5.1 [1]. Then, from the identified scenarios, this deliverable goes on describing the work performed in the following activities: Safety Analysis, Security Analysis, Trade-off Analysis, Requirements Engineering and Safety-Security by design.

The activities related to the last four phases of the Roadmap (Modelling and Implementation, Verification and Validation, Update and Assessment) will be described in the deliverable 5.3 [2].

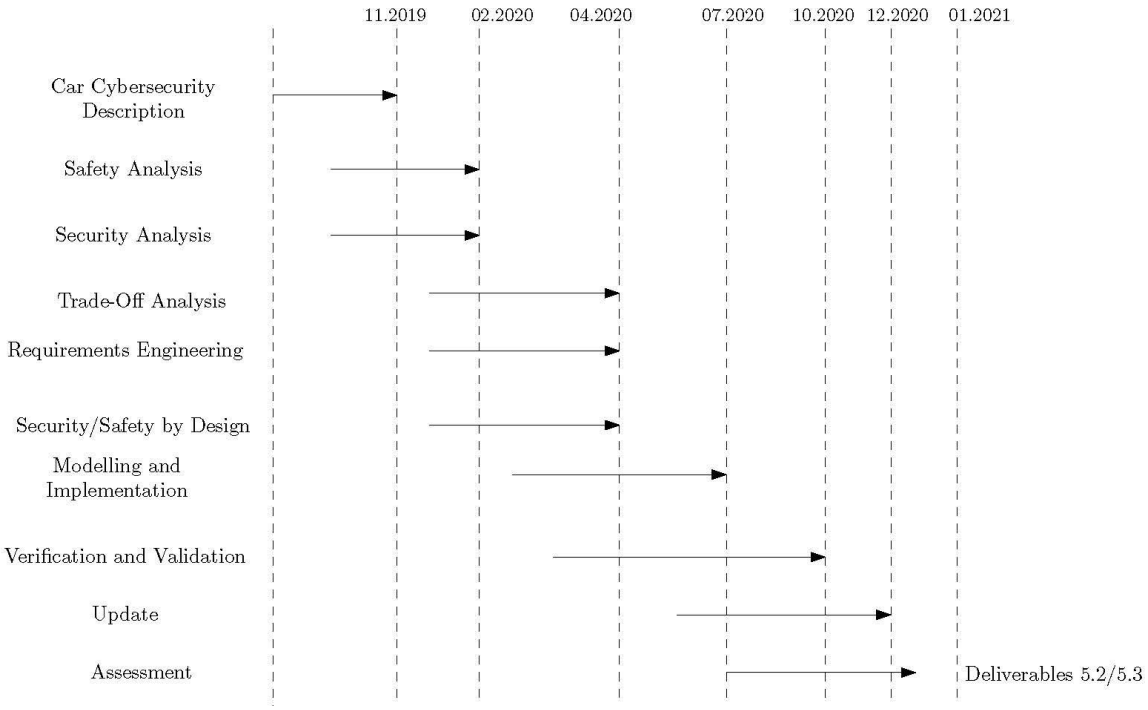

Figure 4: Roadmap for Task 5.2 activities (Source: D5.1 [1])

3.2 Technical Specifications for the Convergence of Safety and Security

3.2.1 Overview

This section builds on the contents of D5.1 [1] that took the first steps in the T5.2 Roadmap shown in Figure 4. For example, D5.1 already describes the Platooning scenario, including basic requirements as well as machinery available among the partners, such as the FTS and TEC Rovers. Moreover, D5.1 also contains Safety and Security analysis of the connected car scenario. Notice as well that this deliverable also breaks down the Platooning scenario into more specific scenarios as described in Chapter 5.

In the following sections, we describe the technical specifications developed since D5.1:

- Section 3.2.2 describes further safety analysis for the Platooning scenario, such as Failure Mode and Effects Analysis (FMEA).
- Section 3.2.3 expands the security analysis in D5.1 that included Attack Defence Trees with other modelling methodologies such as those based in KAOS.
- Section 3.2.4 introduces a new automated methodology for analysing the trade-offs between safety and security based on safety and security architectural patterns.
- Section 3.2.5 describes the efforts in Requirement Engineering for safety and security of the Platooning scenario. This resulted in a protection profile document (in Chapter 12) for a safety and security platoon management module.
- Finally, Section 3.2.6 describes Safety and Security by Design methodologies developed based on model-based engineering and formal verification techniques.

Those contributions that correspond to actual tools are listed in Table 3, and are comprehensively described in the respective subsections of Chapter 7. Contributions of other types, e.g., models, as well as background information are described in the following subsections.

Tool	Partner	Tech. Spec.	Demonstrator use case
AutoFOCUS3	FTS	Section 7.2	Connected car: basic scenario (Section 5.2.1)
Maude	FTS	Section 7.7	Connected car: basic scenario (Section 5.2.1)
OpenCert	TEC	Section 7.9	Connected car: scenario 4 (Section 5.2.4)
Sabotage	TEC	Section 7.12	Connected car: scenario 5 (Section 5.2.5)
SysML-Sec	IMT	Section 7.16	Connected car: basic scenario (Section 5.2.1)
VaCSInE	CETIC	Section 7.17	Connected car: scenario 2 (Section 5.2.2)

Table 3: Overview about tools involved in the context of task 5.2

3.2.2 Safety Analysis

3.2.2.1 FMEA methodology

In the deliverable D5.1[1] a short introduction was made about FMEA (Failure Mode and Effect Analysis) and how this technique is being used in the Platooning scenario. This section builds on it providing a more detailed description of the methodology.

The FMEA methodology is one of the risk analysis techniques recommended by most of international standards as ISO 26262 “Road Vehicles Functional Safety”. This methodology points out potential failures to identify possible failure causes with the aim of removing and locating the failure impacts in order to reduce them. The FMEA process has three main focuses:

- The recognition and evaluation of potential failures and their effects.
- The identification and prioritization of actions that could eliminate the potential failures, reduce their chances of occurring or reduce their risks.
- The documentation of these identification, evaluation and corrective activities so that product quality improves over time.

Several derivatives of FMEAs have been developed, with two basic types: “Design FMEA” (DFMEA) and “Process FMEA” (PFMEA). Design FMEA identifies potential risks introduced in a new or changed design of a product/service, whereas Process FMEA deals with the manufacturing and assembly processes. Nevertheless, both use a common approach, by identifying:

- potential product or process failure to achieve the correct performance;
- potential consequences;
- potential causes of the failure mode;
- application of current controls;
- level of risk, and
- risk reduction.

FMEA is usually performed by filling in a table on a worksheet (see Figure 6). There is not a single or unique process for FMEA development, however we can always find some common elements or terminologies:

- **Item Function:** Item function specifies the function of the part or item under review.
- **Potential Failure Mode:** A potential failure mode is the way in which a failure can occur. The potential failure mode could be also the cause of another potential failure mode in a higher-level subsystem or system or be the effect of one in a lower-level component.
- **Potential Failure Effects:** Potential failure effects refer to the outcome of the failure on the system, design, process or service. The local and global impacts must be analysed. For example, if a local effect is an outcome with only an isolated impact that does not affect other function or if a global effect affects other functions/components affecting completely to the system.
- **Potential Failure Causes:** They identify the root cause of the potential failure mode and provide an indication of a design weakness that leads to the failure mode. The identification of the root cause is very important for the implementation of preventive or corrective measures.
- **Severity (S), Occurrence (O) and Detection (D):** Severity is the seriousness of the effects of the failure. It is an assessment of the failure effects on the user, surrounding people, and environment. Occurrence is the frequency of the failure, in other words, how often the failure can be expected to take place. Detection is the ability to identify the failure before it reaches the user. Figure 5 shows some values for these parameters.

S	O	D	Score
Insignificant	Improbable	Automatic detection with warning	1–2
Marginal	Remote	Automatic detection without warning	3–4
Critical	Occasional	Detected by the operator	5–6
Very critical	Probable	Impossible to detect by the operator	7–8
Catastrophic	Frequent	Impossible to detect	9–10

Figure 5: Score of Security, Occurrence and Detection

- **Risk Priority Number (RPN):** An RPN is a measurement of relative risk. Its value is obtained by multiplying the Severity, Occurrence and Detection values. The RPN is determined before implementing recommended corrective actions, and it is used to prioritize those actions. It is recalculated after the implementation of the corrective actions to assure that the risk priority has decreased.

$$RPN = S * O * D$$

- **Recommended Actions:** The recommended corrective actions are intended to reduce the RPN by reducing the Severity, Occurrence or Detection ranking, or all three together.

After applying the necessary recommended actions, a brief description of the current actions, the responsible and date, and the new security occurrence and detections are recalculated.

FAILURE MODE AND EFFECTS ANALYSIS

System name: _____

													After applying the recommended actions					
Item / Function	Requierement	Potential Failure Mode	Potential Effect(s) of Failure	Severity	Potential Cause of Failure	Current Design				RPN	Recommended Action	Responsability & Target Completion Date	Action Results					
						Controls Prevention	Ocurrence	Controls Detection	Detection				Actions Taken	Severity	Occurrence	Detection	RPN

Figure 6: Example of FMEA datasheet

As mentioned at the beginning of this Section, the FMEA methodology has been applied in the Connected Car vertical use case (see Section 5.3.1).

3.2.2.2 GSN modelling

In D5.1 [1], we detailed other safety modelling techniques based on Goal Structure Notation (GSN). The Goal Structuring Notation is a semi-formal language that has been successfully used to express safety arguments, called GSN-arguments. GSN-arguments are trees formed by different types of nodes, such as Goal nodes with safety requirements, whose satisfaction is argued by strategy nodes, and Solution nodes referencing evidence for the satisfaction of safety requirements. We refer a more interested reader to [6]. Several argumentation patterns, such as GSN Hazard Pattern, GSN FTA Pattern and GSN FMEA Pattern have been proposed as templates for expressing safety patterns [7]. We applied these patterns in D5.1 for the safety analysis of the Connected Car use case (Vertical 1). These were modelled in the AutoFOCUS3 tool.

Moreover, as also reported in D5.1, we extended AutoFOCUS3 to enable the quantitative evaluation of GSN following the work in [8]. They have proposed mechanisms for associating GSN-arguments with quantitative values denoting the confidence level. These values are inspired by Dempster-Shafer Theories containing three values for, respectively, the *Belief*, *Disbelief*, and *Uncertainty* on a safety assessment.

3.2.3 Security Analysis

We have continued the work reported in D5.1 [1] on developing models for the security analysis of polymorphic systems, such as the vehicle platooning scenarios. In D5.1, we carried out the following activities:

- Development of the machinery to model attack defence trees in AutoFOCUS3.
- Designed methodologies for the automated extraction of security relevant information from safety cases. These methodologies are described in D5.1 and also in the paper [9].
- Application of these methodologies to the Connected Car use case (Vertical 1).

Following the work reported in D5.1, we have expanded the security analysis carried out for the Connected Car vertical, now focusing on the basic scenario described in Section 5.2.1. For example, we have identified attack scenarios where intruders can cause harm, e.g., vehicle crash, by exploiting the communication channels used by vehicles in a platoon. These analyses have been used for the definition of requirements documented as a protection profile described in Section 5.3.4.

For the analysis of the firewall reconfiguration and update scenarios described later on in Section 5.2.2 the KAOS goal-oriented requirements engineering methodology [10] was used. The KAOS methodology was used to model some of the safety and security goals of the Connected Car vertical related to the firewall reconfiguration and update scenarios (see Section 5.3.2). The aim was to experiment safety-security co-engineering [9] on the Connected Car vertical.

The KAOS methodology provides a specification language to capture why, who, and when aspects in addition to the usual what requirements; a goal-driven elaboration method; and meta-level knowledge used for local guidance during method enactment. Hereafter we introduce some features of the language and the meta level that will be used later.

The KAOS methodology language, a multi-paradigm specification formalism which combines **semantic nets** for the conceptual modelling of goals, constraints, agents, objects and operations in the system; **temporal logic** for the specification of goals, constraints and objects; and **state-based specifications** for the specification of operations. The language has a rich ontology which is explicitly defined and accessible at the meta-level. The KAOS language, the specification language, provides constructs for capturing a rich variety of concepts involved in the requirements engineering lifecycle, namely, goals, constraints, agents, entities, relationships, events, actions, views, and scenarios. There is one construct for each type of concept. The following types of concepts will be used in the sequel.

- **Object**: an object is a thing of interest in the domain whose instances may evolve from state to state. It is in general specified in a more specialized way -as an entity, relationship, or event according as the object is autonomous, subordinate, or instantaneous, respectively. Objects are described formally by invariant assertions.
- **Action**: an action is an input-output relation over objects; action applications define state transitions. Actions may be caused/stopped by events. They are characterized by pre-, post- and trigger conditions.
- **Agent**: an agent is an object acting as a processor for some actions. An agent performs an action if it is effectively allocated to it; the agent knows an object if the states of the object are made observable to it. Agents can be humans, devices, programs, etc.
- **Goal**: a goal is an objective the system should meet. Refinement links relate a goal to a set of subgoals. The goal refinement structure for a given system is in general an AND/OR directed acyclic graph. Goals often conflict with others. Goals concern the objects they refer to.
- **Requirement**: a requirement is an implementable goal, that is, a goal that can be assigned to some individual agent in the system.

Goals must be AND/OR refined into requirements. Requirements in turn are AND/OR operationalized by actions and objects through strengthening of their pre-, post-, trigger conditions and invariants, respectively. Alternative ways of assigning responsibilities for a constraint are captured through AND/OR responsibility links; the actual assignment of agents to the actions that operationalize the constraint is captured in the corresponding performance links.

Meta-level knowledge: Domain-independent knowledge is used for local guidance and validation during goal-driven elaboration. In particular, a rich taxonomy of goals, requirements, objects and actions is defined at the meta level together with rules for specifying concepts of the corresponding sub-type. Here are a few examples.

- Goals are classified by pattern of temporal behaviour they require, where <> is the operator for a formula f eventually becoming true (sometime in the future), and [] is the operator for a formula always remaining true (always in the future):

  Achieve: P => <> Q or Cease: P => <> not Q

  Maintain: P => [] Q or Avoid: P => [] not Q

- Goals are also classified by type of requirements they will drive with respect to the agents concerned (e.g., *SatisfactionGoal*, *InformationGoal*, *ConsistencyGoal*, *SafetyGoal*, *PrivacyGoal*, etc.).
- Requirements are in the *HardRequirement* category if they may never be violated, or in the *SoftRequirement* category if they are likely to be temporarily violated.
- Actions are *Modify* or *Inspect* actions according as they modify some object state or not.

Such taxonomies are constrained by rules, e.g.,

- *SafetyGoals* are *AvoidGoals* to be refined in Hard-Requirements.
- *PrivacyGoals* are *AvoidGoals* on Knows predicates.
- *SoftRequirements* must have associated *ModifyActions* to restore them.

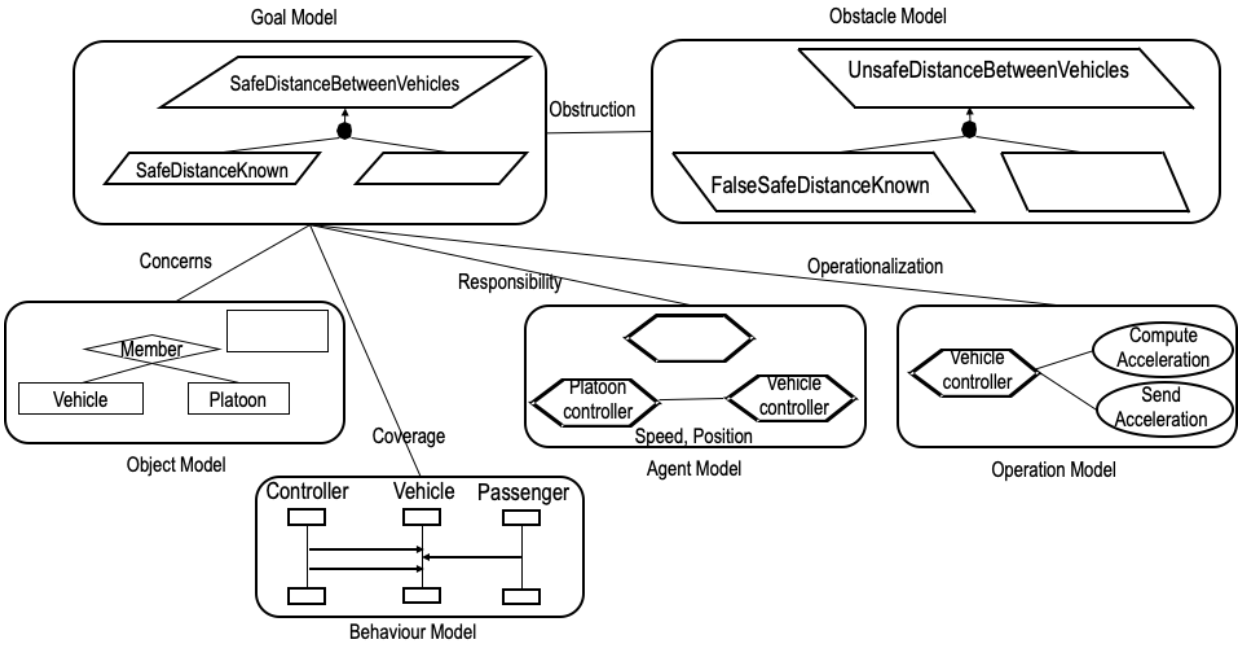

Figure 7: Complementary KAOS system views

Figure 7 shows the complementary system views between KAOS sub-models:

- The **Goal** model presents the intentional view of the system by modelling the system's functional and non-functional goals in terms of attributes describing their specification, type, or priority, and inter-relationships, such as their contributions to each other, their potential conflicts, and their alternative refinements into software requirements and environment assumptions.
- The **Obstacle** model focuses on what could go wrong with goal modelling. Obstacles are conditions that prevent reaching goals and are modelled in a similar way to risk trees. New goals for a more robust system are then added to the goal model as countermeasures to the modelled obstacles.
- The **Object** model provides a structural view of the system, by defining concepts as an entity, attribute, relationship, event, or agent. Objects can be structured by aggregation and specialization with inheritance. The object model is derived from the goal model.
- The **Agent** model provides a view of responsibilities of the system. It identifies the agents forming and their restricted behaviour to meet the goals they are responsible for. Agent capabilities are described in terms of ability to monitor or control the objects involved in goal specifications.

- The **Operation** models provides a functional view of the system by identifying operations performed by system agents under specific conditions.
- **Goals**. Operations are described in terms of pre and post conditions and restricted further to meet requirements. Behaviour is described in terms of scenarios and state machines.

Modelling Safety and Security Goals: When modelling safety-critical or security-critical systems it is important to capture high priority safety and security properties. These properties can be captured in goals of type safety and security. Given the duality between goals and obstacles, safety goals are obstructed by hazard obstacles, and security goals are obstructed by threat obstacles. For the latter disclosure obstacles obstruct confidentiality goals, corruption obstacles obstruct integrity goals, and denial-of-Service obstacles obstruct availability goals.

Threat analysis is carried out by identifying threats as obstacles to security goals and refining the root obstacle that negates it into threat trees. Attacker goals are captured as anti-goals that are intentional obstacles that can be monitored or controlled by an attacker. The last step of threat analysis involves defining countermeasures in the form of new security goals to leaf anti-goals/obstacles.

3.2.4 Trade-off Analysis

Our vision is to provide methods for automating safety and security co-analysis with patterns. These methods shall incorporate safety and security reasoning principles and consider the trade-offs between safety and security. The remainder of this section motivates why such automated methods are needed.

System interconnectivity has been a motivating factor behind the evolution of, e.g., autonomous cars. This interconnectivity, however, leads to new challenges for safety and security. That is, an intruder might cause catastrophic events by remotely targeting safety-critical systems. For example, an intruder might exploit a connection vulnerability in an autonomous car to remotely disable safety features, such as airbags or the braking system, in order to put passengers in danger [9]. Also, an intruder tries to remain undetected so that safety incidents look like hazards. A better integration between safety and security is then appealing. Standards and guidelines for avionics [12] and automotive [13] industries have already taken steps towards this integration. They specify interaction points between the analyses performed by safety and security engineers. That is, when information gathered by safety engineers shall be made available to security engineers and vice versa [14]. The goal is a co-analysis between safety and security engineers to address, respectively, malfunctioning behaviour and intentionally caused harm on safety-critical systems.

Such co-analyses can, however, lead to at least three interrelations:

- Conflicts between safety and security: for example, a security function, e.g., encryption of messages, may increase the latency of safety flows thus reducing the capacity of the system to control hazards.
- Synergies between safety and security: for example, a security function, e.g., the use of machine authenticated messages, plays a similar role as CRC checks used for safety.
- No conflict nor synergy between safety and security: for example, a security function may not interfere with any safety function, e.g., typically, security measures for privacy or IP-related issues do not affect the safety of systems.

The challenge is to understand what the trade-offs between safety and security analyses are, and how to proceed when conflicts or synergies are found.

Our goal is to provide automated methods for safety and security co-analysis that account for trade-offs. Before achieving this goal, we first investigate how much of the safety analysis and security analysis w.r.t pattern selection can be automated.

Safety engineers commonly use hazard analysis and risk assessment (HARA) to identify the main hazards that might potentially cause harm. To control the identified hazards, safety engineers may use safety architectural patterns [15] (e.g., watchdogs or safety monitors). Security engineers focus

on threat detection and mitigation under the presence of an intruder, using, e.g., threat assessment and remediation analysis (TARA). Security engineers may use security patters (e.g., firewall or encryption) to mitigate the identified threats.

Currently, however, safety and security analyses are mostly performed manually by safety engineers. That is, the reasoning of which pattern to use at which part of the target system to control which hazard is documented is mostly in textual form or by means of models, such as GSN-models [16], with limited support for automation. As a result, it is not possible to automatically check whether all hazards have been properly controlled by, e.g., safety patterns.

The methodology that we are applying in SPARTA is summarized as follows:

- First, we propose a domain specific language with safety and security patterns.
- Based on the proposed language, we specify safety and security reasoning principles with patterns during the definition of system architecture for embedded systems. We specify these principles using logic and logic programming as they are suitable frameworks for the specification of reasoning principles as knowledge bases.
- Then, we use logic programming engines to automate the trade-off analysis between safety and security.
- Finally, we validate our current results with an example of safety-critical embedded system taken from the automotive domain.

Figure 8 illustrates our approach. After specifying the domain specific language that includes the types of functions, channels, hazards, threats, and architectural patterns, we specify reasoning rules. In the figure, two reasoning principles are illustrated using logic programming notation. The first one specifies when a safety monitor (*safMon*) can be used to control a hazard (*hz*). The second reasoning principle specifies when a firewall can be used to mitigate a threat.

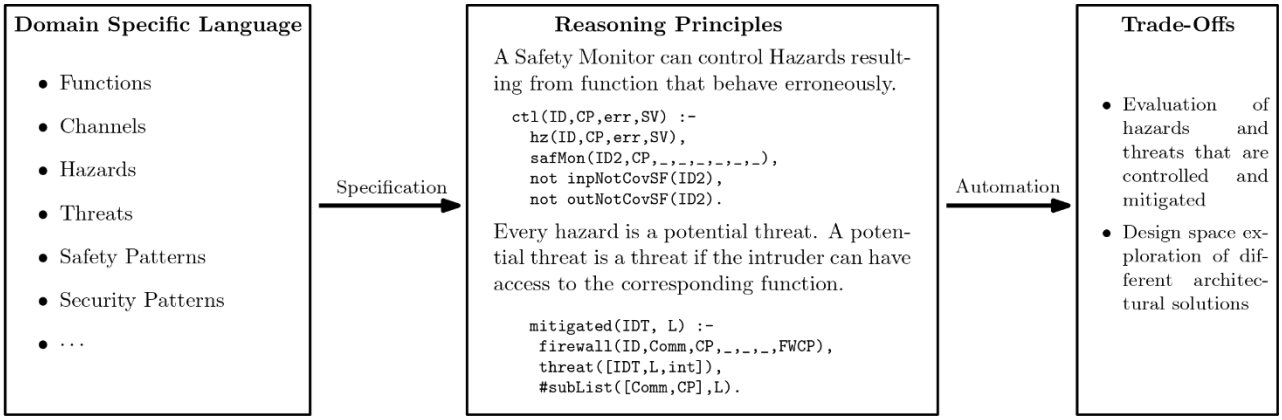

Figure 8: Illustration of the methodology for trade-off analysis

Since these reasoning rules are specified as logic programs, we can use logic programming engines to understand the trade-offs of placing safety and security patterns in a given architecture. Moreover, we can also explore the different architectures obtained by placing safety and security patterns. By using this machinery, engineers can automatically evaluate whether a design has enough control and countermeasures and understand the trade-offs between them. Formal proofs can be applied only on high-level models or sub-systems and the trade-offs should be performed as soon as possible in the development cycle.

Another approach to support the trade-off analysis in SPARTA is provided by the OpenCert tool (see Section 7.9). The tool facilitates the trade-off analysis between safety and security by providing a collaborative editor that allows all actors (safety and security engineers) to work simultaneously on the same compositional assurance case of security and safety aspects of the platooning in real time, showing automatically the last contents provided by any of the OpenCert clients.

Collaborative work on the same argumentation will make it easier to see the dependencies between safety and security goals. To enable this collaboration, each actor should have an OpenCert client that is connected to the same OpenCert central server. This server will store all the data of the platooning safety project, such as the assurance case information.

Since Tecnalia is the only partner using OpenCert for creating an assurance case in the Connected Car scenario, these collaborative features will not be necessary. For this reason, to simplify the OpenCert infrastructure needed to work in the Scenario 4 of the Vertical 1 (see Section 5.2.4), both the client and the server will be installed locally on the same computer.

3.2.5 Requirements Engineering

In order to have a convergence of Security and Safety it is necessary to foresee a safety/security co-engineering process where safety and security are analysed together. Even though there are many different ways for safety and security co-engineering, certification processes also need to be integrated.

The effort made in the SPARTA CAPE program was to define safety and security requirements in the same Common Criteria protection profile (see Section 5.3.4 and Chapter 12).

As described in Section 2.1, the V-model of the SPARTA Cybersecurity Assessment Framework includes a Common Criteria Assurance Class mapping. This was obtained considering the following mapping scheme (see Figure 9) where the various assurance classes, that characterize the activities carried out during a process of evaluation/certification of a TOE, go perfectly to map the needs defined in the various phases of the Cyber security process.

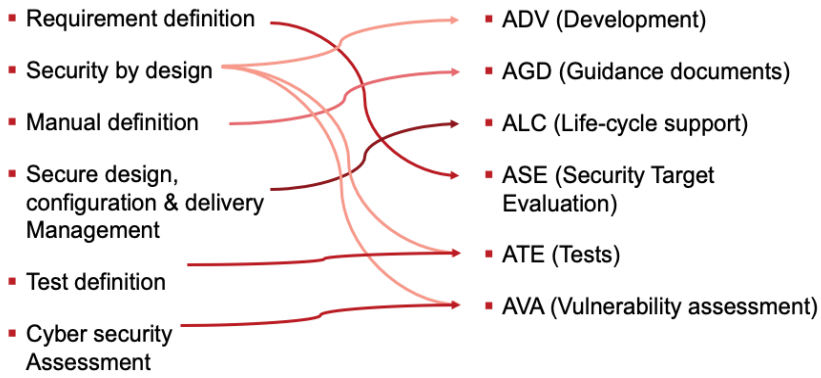

Figure 9: Cyber Security process and Common Criteria Assurance Classes mapping

There are many ways that safety and security requirements can be co-engineered. Figure 10 shows the planned approach for the SPARTA project, where safety and security analysis are re-reconciled during trade-off analysis to produce safety and security requirements that are linked together. The above Protection Profile approach has been experimented on the Connected Car vertical, using ISO 26262 “Functional Safety Road Vehicles” for the safety certification, and ISO/IEC 15408 “Common Criteria” and SAE J3061 “Cyber Security guidelines” for security certification.

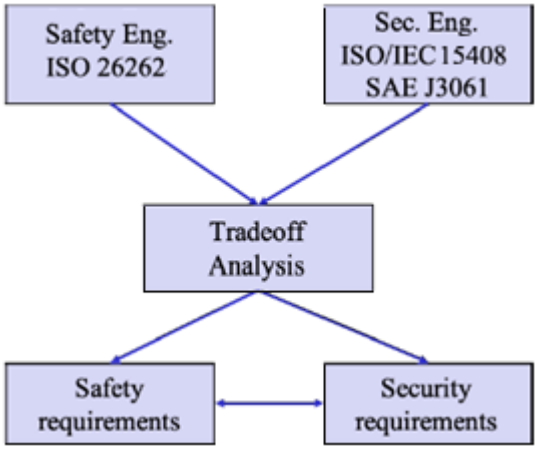

Figure 10: Safety and security trade-off analysis

Safety analysis and cybersecurity analysis are being performed in parallel. A trade-off analysis then needs to be carried out to determine which trade-offs between safety and security need to be taken.

The results of the trade-off analysis are then documented in the common safety/security protection profile, as it is described in Section 5.3.4.

3.2.6 Security/Safety by Design

This section describes some of the methodologies that we have developed for ensuring the safety and security by design of complex systems, such as cyber-physical systems as the Connected Car system considered for Vertical 1. Notice that some of the methodologies described in the previous sections support the development of safety and security by design. For example, the use of models for safety and security analysis and the use of logic programming engines for trade-off analysis provide analysis that can be used by engineers early on to design safe and secure systems.

In the following section, we build on these techniques and elaborate further methodologies based on precise mathematical models for the (automated) analysis of systems. We propose a formal assessment framework for specification and verification of cyber-physical systems, such as vehicle platooning.

3.2.6.1 Formal Verification of Cyber-Physical Systems

Designing safe and secure systems is challenging as intruders may carry out attacks by exploiting corner-cases or implicit requirements overseen by developers. For example, several communication protocols have been shown to be vulnerable to attacks, some of which have been discovered decades after they have been developed [20]. The safety and security of vehicle platooning have the additional complexities of cyber-physical systems, including speed, time to react, and position. Engineers must ensure that intruders cannot exploit these aspects, as in the injection attacks described by [21].

The use of formal verification provides further evidence about the security of platoons using CACC. An advantage of formal verification over, e.g., simulation analysis, lies on the fact that its methods are based on precise mathematical models that specify the behaviour of the analysed system. By using formal verification, implicit requirements are made explicit thus exposing existing vulnerabilities. Moreover, from such models, automated tools can determine whether undesired events are possible by traversing all behaviours including corner-cases.

Existing formal frameworks for platooning [22], [23] and other agent-based cyber-physical systems [24], [25] have successfully been used to verify the safety of agent-based cyber-physical systems, such as platoon joining manoeuvres and strategies used by Unmanned Aerial Vehicles [26]. These frameworks, however, do not consider security aspects. They do not include intruders and therefore,

it is not possible to verify in such frameworks whether an intruder may attack a system and cause harm, e.g., a vehicle crash.

To the best of our knowledge, this deliverable proposes the first formal framework to consider platooning, CACC and security. Our main contributions are three-fold:

- **Vehicle Platoon Behaviour Specification:** Our first contribution is a platoon model that includes specifications of both cyber aspects, e.g., specifications for the communication protocols, and physical aspects, e.g., speed, acceleration, positions of vehicles. Our model enables the specification of a wide range of vehicle strategies for executing platooning based on soft-constraints [27], a general algebraic framework for specifying optimization problems. That is, our model can accommodate several strategies including those expressed as classical, fuzzy and probability theories and their combination. For example, strategies for maintaining distances between vehicles that are both safe and fuel-efficient can be reduced to an optimization problem based on soft constraints.
- **Intruder Models:** Our second contribution consists of formal intruder models that subvert communication channels to carry out attacks. These intruder models are parametric on the intruder capabilities, i.e., the capability of either blocking messages from a communication channel or injecting messages into communication channels.
- **Automated Verification:** Our third contribution is the implementation of our models, both platoon and intruder models, in Maude [28], an efficient formal verification tool based on Rewriting Logic. Our specifications are executable. That is, users can automatically invoke Maude's search mechanisms to formally verify their platooning specifications for the verification of safety, e.g., vehicles not crashing, by considering security, e.g., in scenarios where an intruder may block or inject messages.

Using our formal framework, engineers can evaluate whether the proposed safety and security measures are sufficient to mitigate the considered attack scenarios. The model we developed for the Connected Car vertical is described in Section 5.3.5.

Chapter 4 Risk Discovery, Assessment and Management for Complex Systems of Systems (T5.3)

4.1 Context and Background

Figure 11 provides a high-level overview about actors and systems commonly involved in the development and build process of a given software project. Project maintainers and contributors commit source code changes to a versioning control system (VCS) like Git or SVN, which is hosted in an organization's own infrastructure or by providers like GitHub. Periodically or upon every commit, build processes compile and test the software project in question, whereby open source dependencies are downloaded from 3rd party package repositories like PyPI, Rubygems or npm. Once a new version of the respective software project is released, all relevant resources are packaged and uploaded to a private or public distribution site, which could be public package repositories (for open-source software libraries), private package repositories (for proprietary re-use components) or application stores like Google Play Store (for end-user applications).

Such development and build processes, with all its actors and systems, exist for every single software component ending up in a given end-user application. According to a recent report from GitHub[1], typical JavaScript (Node.js) applications depend in average on 10 direct and 683 transitive components. In other words, the security of a given Node.js application depends on the security posture of almost 700 other projects.

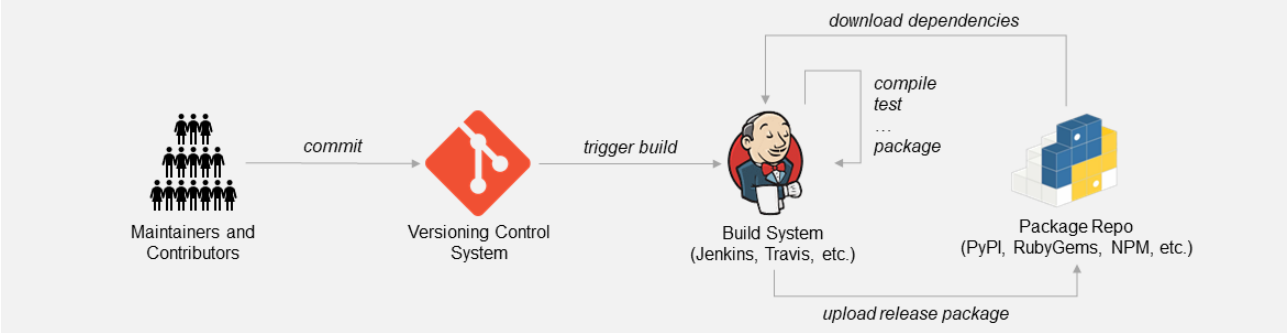

Figure 11: Actors and systems part of typical, open source-based software development

Figure 12 illustrates the focus of Task 5.3 regarding such development and build processes:

- On the one hand, security vulnerabilities can be accidently introduced by benign developers, when committing source code in their respective VCS. Such security-relevant bugs happen on a regular basis and affect any direct or indirect user of the respective component, who downloads affected component releases during their own development and build process. As mentioned above, a typical application depends on many upstream components (transitive dependencies), and the application developer must track known vulnerabilities of those dependencies in order to update to non-vulnerable component versions where necessary.
- On the other hand, attackers deliberately try to inject malicious code into open source components in order to infect downstream consumers. Those attacks became more prominent in the past, e.g., Sonatype noticed a 430% year-over-year growth of such supply chain attacks in a recent report from 2020[2] (even though on relatively low numbers).

[1] https://octoverse.github.com/static/2020-security-report.pdf

[2] https://www.sonatype.com/software-supply-chain-2020

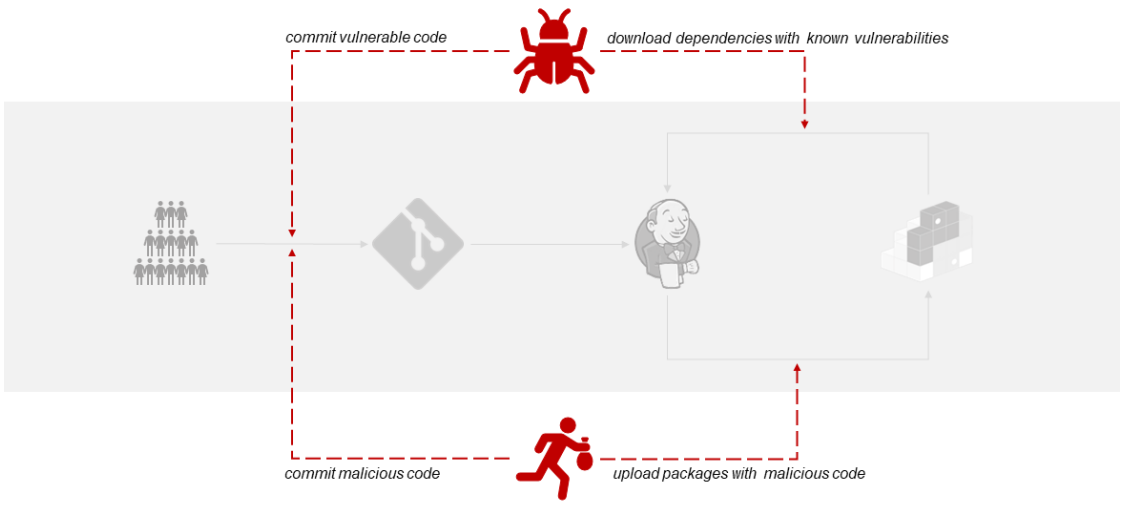

Figure 12: Security threats targeting such actors and systems

4.2 Technical Specifications

4.2.1 Overview

Several contributions developed in the context of the SPARTA CAPE program address the above-mentioned threats related to the security of software supply chains.

Figure 13 positions those contributions with respect to common development and build environments. Most contributions, e.g., Buildwatch (Section 7.3), Frama-C (Section 7.4) or Approver (Section 7.1), are executed as part of build processes, e.g., automated jobs executed by build servers such as Jenkins. Other contributions target package repositories such as Google's Play Store, e.g., the Logic bomb detection (Section 7.6), whereas others read information from versioning control systems such as Git, e.g., SafeCommit (Section 7.13). A detailed description of integrations and synergies of all contributions can be found in deliverable D5.3 [2].

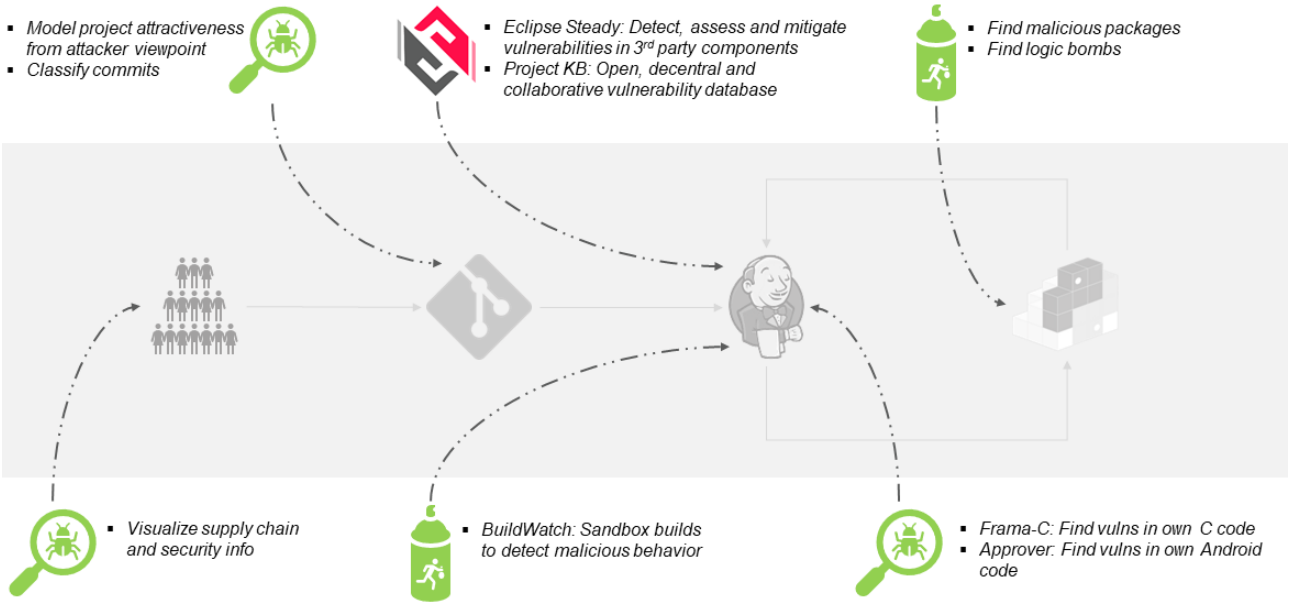

Figure 13: Positioning of Task 5.3 contributions

Those contributions that correspond to actual tools are listed in Table 4, and are comprehensively described in the respective subsections of Chapter 7. Contributions of other types, e.g., datasets or models, as well as background information are described in subsections 4.2.2 and 4.2.3.

Partner	Contribution	Tech. Spec.	Technologies Covered	Vertical
UBO	Buildwatch	Section 7.3	Agnostic	e-Government
CEA	Frama-C	Section 7.4	C	Connected Car
CINI	Approver	Section 7.1	Java (Android)	e-Government
SAP	Steady	Section 7.15	Java, Python	e-Government
SAP	Project KB	Section 7.10	Agnostic	e-Government
UNILU	Logic Bomb Detection	Section 7.6	Java (Android)	e-Government
UNILU	SafeCommit	Section 7.13	C/C++	Connected Car
UKON	Supply chain visualization	Section 7.18	Java, Python	e-Government

Table 4: Overview about tools extended/developed in the context of task 5.3

4.2.2 Known and Unknown Vulnerabilities

This section describes the co-training approach used by UNILU's tool described in Section 7.13, as well as SAP's work on fix commit identification. Both contributions relate to the use of artificial intelligence to automatically identify or classify commits in source code repositories as security relevant. This work is significant to accelerate the detection of new security vulnerabilities on the one hand (Section 4.2.2.1), and the detection of vulnerability patches on the other hand (Section 4.2.2.2). The former promises to reduce the number of security vulnerabilities in released software components, the latter is meant to accelerate and automate the curation of vulnerability databases, which suffer from deficiencies regarding coverage, quality and timeliness [29][30][31].

4.2.2.1 New approaches for commit-classification

According to the meeting on Oct 9th, the tool currently available and described in Section 7.13 does not reflect the latest research. Initially, the goal of UNILU was to develop a tool able to detect both vulnerability introducing commits and vulnerability fixing commits. However, thanks to several discussions with researchers from SAP in the context of SPARTA, UNILU researchers realized that SAP already works on the detection of vulnerability fixing commits. Rather than competing, both SAP and UNILU decided to join forces: Together they can propose a generic approach and tool aiming at detecting security-relevant commits, i.e., commits that either introduce or fix a vulnerability. UNILU will focus on the detection of vulnerability introducing commits and SAP will focus on the detection of vulnerability fixing commits.

A major issue with any vulnerability introducing commit detection endeavour is the lack of labelled data, i.e., a dataset in which samples are correctly labelled as vulnerability introducing commit or not. While researchers can collect many hundreds of thousands commits, acquiring even a modest dataset of known vulnerability introducing commits requires a massive effort.

One semi-supervised learning approach, called co-training and introduced by Blum and Mitchell [32] will be investigated in the course of SPARTA. On a Web page classification problem, Blum and Mitchell [32] used two classifiers in parallel to complete training sets with unlabelled data. They ended up with an error rate of just 5% based on both the page content and hyperlinks over a test set of 265 pages: only 12 pages labelled (3 as positives course-pages, 9 negatives) and around 800 unlabelled. They demonstrated that Co-Training achieved performances on this problem that was unmatched by standard, fully supervised machine learning methods. It is a technique that has industrially proven a reduction of false positives by a factor of 2 to 11 on specific element detection on a video [33], and for which conditions of maximum efficiency it induces were analysed [34].

Co-Training Principle: When trying to detect vulnerability introducing commits, an important point is that unlabelled commits are unlabelled not because they are not vulnerability introducing commits,

but because it is unknown whether they are vulnerability introducing commits. Arguably, in any large-enough collection of commits, it is reasonable to assume at least some of them are actually vulnerability introducing commits. The insight behind trying Co-Training with vulnerability introducing commits detection is the following:

- By building two preliminary and independent vulnerability introducing commit classifiers, the unlabelled commits predicted to be vulnerability introducing commits by both classifiers could be used to augment the training set. By repeating this step, it might be possible to leverage the vast space of unlabelled commits.

Description of the algorithm: [32] showed that the co-training algorithm works well if the feature set division of dataset satisfies two assumptions:

1. each set of features is sufficient for classification, and
2. the two feature sets of each instance are conditionally independent given the class.

Both vulnerability introducing commits features set and the alternate feature set can be split into two subsets of features: one based on code metrics, and one based on the commit message.

Previous work on security patches detection showed that, for the New Feature set, the two resulting feature subsets are independent, and thus satisfy the two main assumptions for Co-training [35].

Once these two assumptions are satisfied, the Co-training algorithm considers these two feature sets as two different, but complementary *views*. Each of them is used as an input of one of two classifiers used in Co-training: one focused on code metrics, and the other on commit messages. The algorithm is given three sets: a positive set, a negative set, and a set of unlabelled.

As shown in Figure 14, the training process is an iterative process in which each classifier (**h1** and **h2** on Figure 14) is initialized being just given the labelled inputs **LP**. From the whole set of unlabelled, a subset is randomly selected **U'**. At every round, each classifier is trained and chooses, from this subset of unlabelled commit, an arbitrary amount of commits to be added. The former training set and the selected commits form the new training set. The commits are confirmed to augment pseudo labelled classes (negatives and positives) based on the confidence of the classifier (distance from the hyperplane) and on the agreement of the other classifier. The new round starts by training the classifiers a new, based on the augmented just-labelled set. The process keeps going until we reach a predetermined size of label set. **U'** is refurbished every round by as many commits that were taken from it, using commits left from **U**.

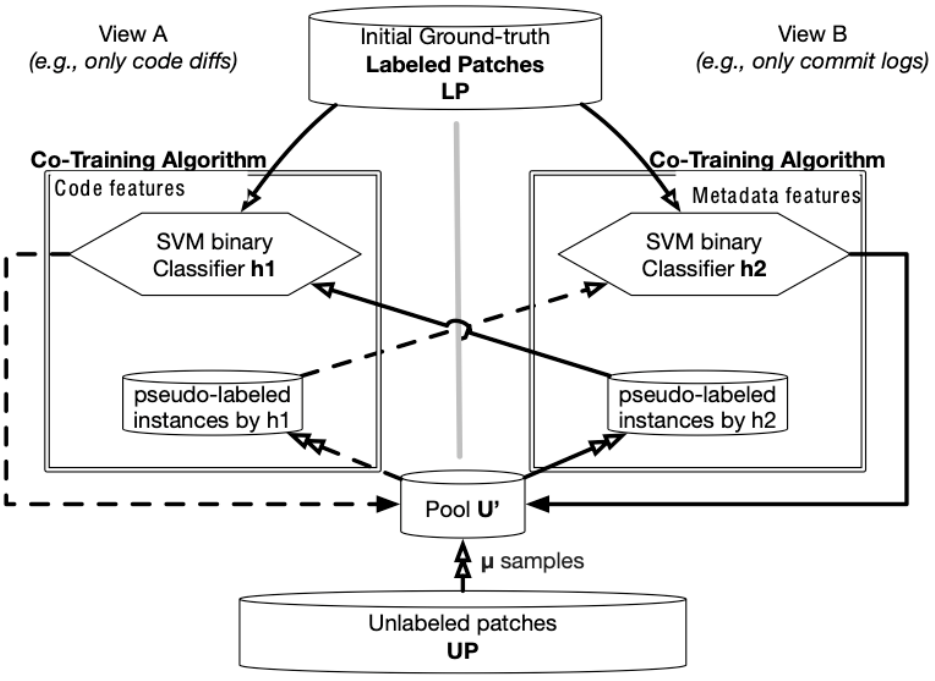

Figure 14: Co-Training (Figure extracted from [35])

4.2.2.2 Commit2Vec – Learning distributed representations of code changes

Deep learning methods have been proven successful in a variety of problems, such as image classification, natural language processing, speech recognition, and others. More recently there is a growing interest in using similar approaches to programming-language related tasks [39] [40] [41] [48], using code as the main input source.

To this end, a key prerequisite is the ability to represent the code (or code fragments) as a numerical vector (embedding), similarly to the word2vec [43] approach for natural language processing (NLP). Such vectoral representation should have the property of mapping similar instances of code elements onto close points in the embedding vector space. Using NLP methods to build representations of software code is meaningful, indeed, as empirically shown in [42]. Source code is characterized by similar statistical properties as natural language (naturalness hypothesis [37], which is not surprising considering that code is written and read by humans, in addition to being executable by machines). On the other hand, there are significant differences, since code is written in a programming language, which is a formal language: it presents minimal ambiguity, large re-use of identical “sentences”, and reduced robustness to small changes compared to natural language. In addition, the semantic units of text, as sentences or paragraphs, are typically relatively short, present a high level of locality, and they rarely used more than one time in the text. On the contrary, (sequences of) code statements or functions are clearly delimited, they may be used multiple times in different contexts, and present long range correlations (i.e., the semantic of a statement can be influenced by other statements in a somewhat distant part of the code).

For these reasons, beyond NLP-inspired methods, a number of representations that use the structural nature of code have been proposed, such as using data flowgraphs, control flow graph and abstract syntax trees, and used to perform tasks as variable and method naming [36] [40], clone detection [47], code completion [45] [46],summarization [38], and algorithm classification [44].

Due to the complexity of the code structure, training a deep learning algorithm to solve a code-related task needs a large amount of labelled data, which in many practical cases is not available. Ideally, we would like to build a low-dimensional representation using a task where a large amount of labelled data is available (or can be obtained in a relatively inexpensive way), and use the learned representation to solve a different (target) task, where fewer data point are available, but enough to finetune the model (transfer learning).

In the following, we propose a new model for representing code changes, called commit2vec, which, along the lines of [40], uses paths from the abstract syntax tree (AST) to build a representation of code changes.

Finding the best-suited representation of code in machine learning frameworks is an open research question. The survey in [37] classifies the representation of code into three main categories:

- token-level models treat code as a sequence of tokens in a similar way as traditional natural language processing (NLP) techniques represent text as a sequence of words in a given language
- syntactic models leverage on the underlying structural information of code through their abstract syntax tree (AST) representations
- semantic models represent code as a graph generalizing both token-level and syntactic models

The concept of word embeddings, made popular by the work in [43], allowed a breakthrough in many NLP-related tasks. Over the last few years, approaches in-spired on the same concept are emerging in the domain of source code analysis. The work in [41] presents a survey of different works that use the concept of embeddings at different granularities of code.

In this work, we introduce a method to represent source code changes (such as those contained in the commits of a source code repository). Differently from the approaches that represent a static snapshot of code [40], or that represent changes in small code fragments [48], we focus on

representing full commits, which can contain changes across multiples methods, classes, and even files.

Our method uses code2vec as a basic building block: for a given commit, we extract all the methods that are changed and use the same pre-processing steps as code2vec to extract a set of paths over the AST (context in the terminology of [40]); we then discard the contexts that are identical in the code before and after the commit, and use the remaining paths as the basis for the commit representation.

More precisely, let a code commit, C, be defined as a change in the source code of a given project in a set of files $f_i \in F$, where $i \in [1..I]$, where I is the number of files changed within C.

The concept of a commit implies a prior and a posterior version of files fi, which we denote as $f_{i,pre}$ and $f_{i,post}$ respectively.

Analogously to textual tokens in token-based representations, our model uses paths constructed traversing the AST of each method changed in C. Consistently with the terminology of [40], we call contexts the triplets of two terminal nodes and their connecting path on the AST. Let the union of all the contexts of the prior versions of all methods $m_{1..J,pre}$ in all files $f_{1..I,pre}$ in commit C be defined as $S_{pre}=\{p1,p2, ...,pk\}$ and the union of all the contexts of the posterior versions of all methods $m_{1..J,postin}$ all files $f_{1..I,postin}$ commit C be defined as $S_{post}=\{p1,p2, ...,pk\}$. We then define the set of contexts describing commit C as the symmetric difference between S_{pre} and S_{post}:

$$SC = S_{pre} \Delta S_{post} \equiv \{p: p \in S_{pre} \cup S_{post}, p/\in S_{pre} \cap S_{post}\}$$

Intuitively, the symmetric difference S_C between the two sets of contexts contains the contexts that have been changed in the commit C. S_C is the input provided to the neural network architecture that yields a distributed representation of the code changes performed in commit C. In order to generate meaningful representations, the neural network typically requires large amounts of data to be trained on. Unfortunately, in many applications the data available is not sufficient. In these cases, transfer learning techniques are applied, where the network is pre-trained on a similar task for which large amounts of data are available, often called the pretext task, and then fine-tuned on the target task using a smaller dataset.

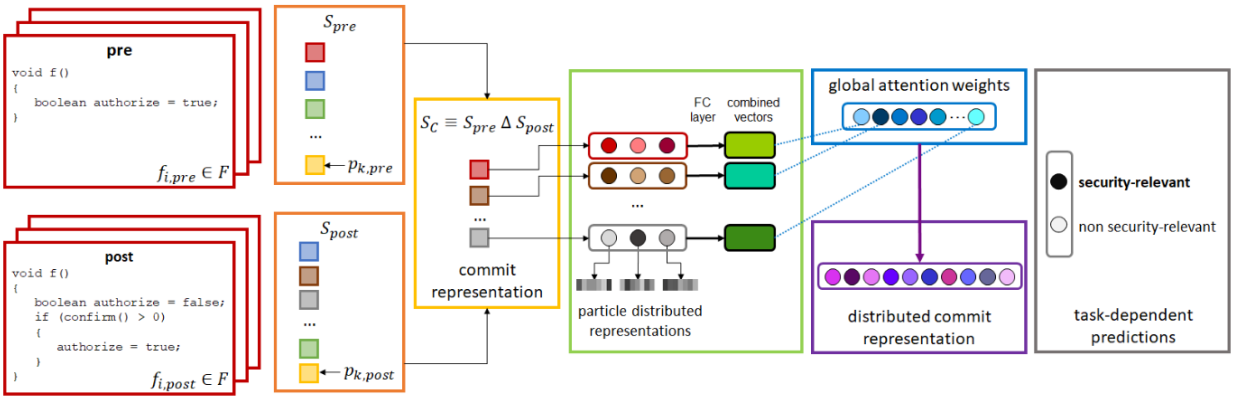

Figure 15: commit2vec method

In Figure 15 the prior (f_{pre}) and posterior (f_{post}) versions of all code-relevant files in a commit, C, are transformed into contexts through an AST-based code representation, generating both S_{pre} and S_{post}. The commit representation, SC, is computed as the symmetric difference between S_{pre} and S_{post} and is provided as the input to a neural network. In this diagram, the exemplified task is that of classification of security and non-security relevant commits.

4.2.3 Supply Chain Attacks

Figure 16 provides an attack tree illustrating various attack vectors that can be used by malicious actors to inject malicious code into legitimate open source projects. This version is a slightly modified version of the tree presented in deliverable D5.1[1] and has been published by SPARTA partners at DIMVA 2020, the 17th conference on the Detection of Intrusions and Malware & Vulnerability Assessment.

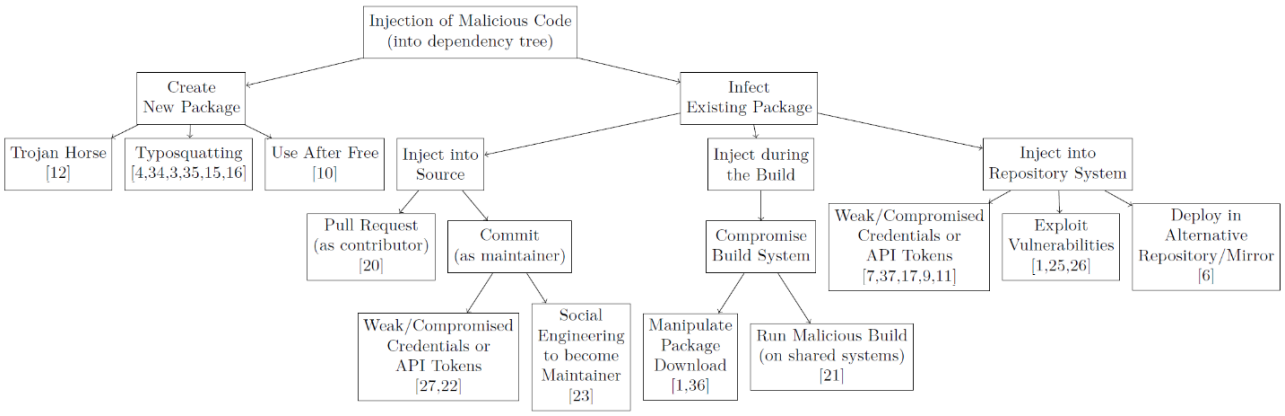

Figure 16: Attack tree to inject malicious code into dependency trees (taken from [49])

With respect to this attack tree, the technique explained in Section 4.2.3.3 addresses the nodes “Inject during the Build” and “Inject into Repository System”, both of which have the common characteristic that malicious code in a distributed package is not present in the respective source code repository.

The proposed detection technique compensates the lack of reproducible builds, which allows to verify that a given package has been produced from a given commit or tag in some versioning control system.

The model presented in Section 4.2.3.1 aims to determine (predict) the attractiveness of open source projects for attackers depending on various project features. As such, it covers all attack vectors, and aims to identify projects that deserve particular attention and protection.

4.2.3.1 Metrics for OSS components’ attractiveness to attackers

This section describes the development of a method aiming to evaluate open source software components in terms of their attractiveness to attackers. This method can reflect the way attackers target open source supply chains and select such components for malicious acts.

The overall idea is to identify metrics and information related to the factors that affect the selection of OSS components from possible attackers and produce an algorithm that allows to rate any OSS components with respect to these factors.

Nowadays, in view of the openness of information and within the context of interoperability and reusability of code and components, a lot of public repositories have been created, hosting packages, code and components which are shared between different types of software, suites, platforms, etc. Although this obviously contributes to the exchange and improvement of code development and, thus, of software engineering and development, it also offers illegal hackers and cybercriminals numerous ways of taking advantage of this. More specifically, since the publicly available components will, indeed, be used in several, different types of software, cybercriminals tend to attack these components aiming to inject malicious code into the software.

Therefore, several scientists, (security) organizations, agencies, institutions, etc. look for ways to detect these injections as easily and fast as possible. Some publications, such as [50] where a pragmatic approach to facilitate the impact assessment is presented, concern known vulnerabilities in open-source software libraries; their approach is independent of specific kinds of vulnerabilities or

programming languages and capable of delivering immediate results. Furthermore, [51] addresses the over-inflation problem of academic and industrial approaches for reporting vulnerable dependencies in OSS (Open-Source Software software). Other publications, such as [52], [53], [54],[55] and [56] concern the detection, assessment, and mitigation of vulnerabilities.

The aim of the methodology to be presented below is to set the foundations for the definition, quantification, and calculation of the "attackability", a notion that represents the probability of an OSS component to become a target of a cybercriminal.

Work Description

In this Section, we provide the definition of the concept of attackability as well as a description of our methodology for the quantification and calculation of this concept.

We propose the definition of attackability through probability theory. Let p denote the probability that a repository or package will be attacked. Obviously, it holds that:

$$0 \leq p \leq 1 \quad (1)$$

and henceforth, we will say that "the package /repository under consideration is attackable with probability p" or, simply, that its "attackability is (equal to) p".

Factors affecting attackability

In the absence of a concrete theory, we follow a probabilistic approach, and we seek to determine the factors, on which attackability depends. To this purpose, we employ a nonlinear regression technique, with the aid of which we will determine these factors as well as the dependence of attackability on these parameters; i.e. a mathematical formula, which shall be used for the calculation of the attackability.

First, we make a fundamental assumption; for the repositories / packages of interest there is neither a way to determine the vulnerabilities of the package, nor any prior knowledge about them. Additionally, the factors fall within the following principal categories:

- **Ownership***:* Type of owner the package belongs to (e.g. user or organisation). We assume that a repository owned by a user is likely to be more attractive as a target. The reasoning behind this is that organisations tend to adopt stricter security policies and enforce better security practices, at least the mandatory ones. Furthermore, organisations usually have dedicated departments or teams or partners, who make sure their packages / repositories stay up to date as concerns security, latest security patches against vulnerabilities. Moreover, organizations usually carry out more extensive testing; either with the aid of dedicated teams and experts or with the help of the developers, researchers, collaborators, etc.
- **Maturity***:* Package's age calculated using the timestamp of when the repository was created on the host. Intuitively, on the one hand, the more mature a package is, the more difficult dissuasive it is to attack; on the other hand, an old package that is not likely to be used may not be considered to be an attractive target. However, there are cases where abandoned packages were attacked, but we need to highlight that our approach aims to define a methodology, which will be able to compute the probability that a package will be attacked, based on specific assumptions.
- **Reach & Popularity***:* This category includes factors related to the popularity of the packages. Intuitively, the more popular a package/ repository is, the more attractive to attackers it is expected to be. This is because a more popular package / repository will probably allow attackers to affect a larger audience, contrary to a less popular one.
- **Activity**: a package / repository with more activity around it from the development team is expected to be more attractive to attackers, as it would be easier for the attacker(s) to inject the malicious code in the former (even pretending they are contributors / collaborators) or hide the malicious code more easily, given there is a large number of commits, issues, releases, etc.

- **Responsiveness**: Intuitively, the higher the responsiveness, the more difficult it will be for an attacker to attack. This is because responsiveness is considered to indicate that people frequently contribute to, work on, review the repository / package, so any changes are likely to be reviewed and checked soon enough.

Machine Learning Methodology

In the context of the development of a methodology, which will be suitable for the fast computation of the "attackability" of a package of interest, we present the concrete steps taken to measure the "attackability" of a package of interest.

To this purpose, we employ a nonlinear regression model of adaptive polynomial degree with the following associated Objective / Cost function:

$$I(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left\{w_i[y_i - f(x_i,\theta)]^{\ 2}\right\} + \frac{1}{2m}\sum_{j=j_0}^{j_f}\lambda_j\theta_j^2 \quad (2)$$

where:

m denotes the number of examples,

n stands for the number of features (predictors),

$w_i, i = 1,2,..,m$ represents the observation weights, chosen so that different observations in (2) have the corresponding influence on the model to be fitted,

$x_i, i = 1,2,..,m$ denote the predictors for observation i (selected features),

$\theta_i, i = 1,2,..,n$ represents the regression coefficients (so θ is a vector, the elements of which are θ_i),

$y_i, i = 1,2,..,m$ are the values of "attackability", associated with the respective features $x_i, i = 1,2,..,m$, and $\lambda_j, j = j_0,..,j_f$ (with $j_0 \geq 1, j_f \leq n$) stand for regularisation (scaling) factors, which can be used to control the contribution of a number of features and partially address overfitting.

Finally, $f(x,\theta)$ denotes the nonlinear regression model. So, we seek to solve the following minimisation problem:

$$\min_{\theta}\{I(\theta)\} \quad (3)$$

Thus, the solution of (3) with respect to θ will yield the desired approximation of "attackability". The list of possible (the algorithm will determine which ones shall be included in the final model) features to be employed by our model, so as to compute the "attackability" of a package under consideration is presented in Table 5 below.

Data	Description	Category	Intuition / rationale
Owner	Type of repo's owner: user or organisation	Ownership	More attractive to attackers in case of a user-owner, less attractive otherwise
Created Timestamp	Timestamp of when the repository was created on the host. It is used to calculate the age of the component	Maturity	Less attractive if the package is too old or too young, more attractive otherwise
Releases	Total number of releases for the package	Maturity	The higher the number of releases, the more attractive to attacks
Dependent Packages Count	Number of other packages that declare the package as a dependency in one or more of their versions	Reach & Popularity	The higher the dependent packages count, the more attractive to attackers the package

Data	Description	Category	Intuition / rationale
Dependent Repositories count	The total count of open source repositories that list the package as a dependency	Reach & Popularity	Same as in the previous one
Stars count	Number of stars on the repository	Reach & Popularity	The more popular, the more attractive
Subscribers count	Number of subscribers to all notifications for the repository	Reach & Popularity	The more popular, the more attractive
Forks count	Number of forks of the repository	Reach & Popularity	The more popular, the more attractive
Contributors	Number of unique contributors that have committed to the default branch.	Activity	The more the contributors, the more attractive to attackers it is expected to be
Commits	Total number of commits	Activity	The more the commits, the more attractive to attackers
Open Issues Count	Number of open issues on the repository	Activity	The more open issues, the more attractive to attacks it is expected to be
Issue_Created At	Calculate time needed to close an issue	Responsiveness	The higher the responsiveness, the less attractive it is expected to be

Table 5: List of possible features for the regression model

4.2.3.2 Dataset with malicious open source components

This section describes a public dataset with malicious open-source components used in real-world attacks, the methodology used to construct the dataset as well as plans to extend and use the dataset.

Methodology

The dataset comprises the subset of malicious packages used in real-world attacks for which the actual malicious code could be obtained (typically a compressed archive). The compilation took place between July 2nd and August 2nd, 2019 and was updated on 27th of January 2020. The programming languages JavaScript with its package repository npm, Java (Maven Central), Python (PyPI), PHP (Packagist) and Ruby (RubyGems), which are the most popular languages according to newly created GitHub repositories in 2018 [57], are covered by the dataset.

During that time, the vulnerability database Snyk[3], language-specific security advisories, and research blogs were reviewed to identify malicious packages and possible attack vectors. It must be noted that these sources solely mention the packages' names and affected versions, thus, the actual malicious code has to be downloaded from other sources. However, such malicious packages are typically not available anymore on standard package repositories of the respective programming language, e.g. npm or PyPI. Instead, where possible, they were retrieved from deprecated mirrors, internet archives, and public research repositories. If the code of a malicious package could be retrieved, it was analysed and categorized manually. This was done in order to confirm the packages' maliciousness, map them to the existing attack trees or extend them if necessary. The publication of malicious versions of a package are dated according to Libraries.io[4], a service that monitors package releases across all major package repositories. Advisories and public incident reports are used to date the public disclosure of the malicious package.

[3] https://snyk.io

[4] https://libraries.io/

Description

The dataset contains 174 packages and was compiled according to our methodology as described above. A total number of 469 malicious packages could be identified. Additionally, 59 packages were found that could be identified as proof of concept (published by researchers) and hence are excluded from further examination. Eventually, we were able to obtain at least one affected version for 174 packages. The rate of successful downloads of malicious packages for npm is 109/374 (29.14%), for PyPI 28/44 (63.64%), for RubyGems 37/41 (90.24%), and for Maven Central 0/10 (0.00%). All statements and statistics below refer to the set of downloaded packages. Please refer to [49] for the complete description of the dataset.

The dataset consists of 62.6% packages published on npm and hence are written forNode.js in JavaScript. The remaining packages were published via PyPI (16.1%, Python) and via RubyGems (21.3%, Ruby). Unfortunately, a malicious Java package targeting Android developers could not be downloaded. For PHP, we were not able to identify any malicious package at all.

Figure 17 visualizes the publication dates of the collected packages which range from November 2015 to November 2019. The publication and disclosure dates are identified according to the upload time of the package and the publication date of the corresponding advisory identifying the respective version as malicious. A trend for an increasing number of published malicious packages is apparent. While malicious packages for PyPI are known to date back to 2015 and since then are increasing, npm gained a massive amount of malicious packages in 2017, and malicious packages on RubyGems experienced a boom in 2019.

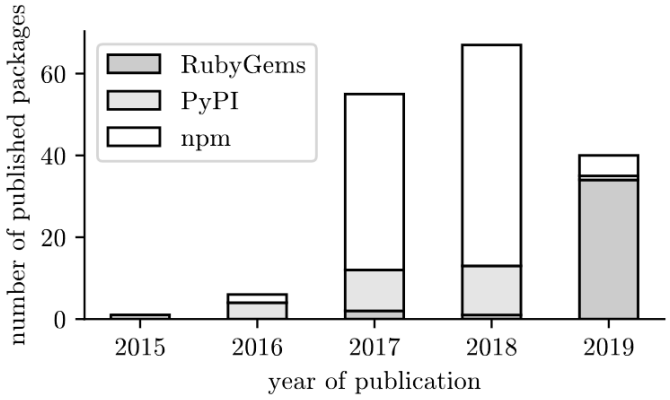

Figure 17: Publication dates of collected packages (from [49])

Figure 18 shows that on average a malicious package is available for 209 days (min=−1, max= 1,216, ρ= 258, x̃=67) before being publicly reported. A minimum of −1 days was reached for `npm/eslint-config-airbnb-standard/2.1.1` which was affected by `npm/eslint-scope/3.7.2`. Even though the infection of `npm/eslint-scope/3.7.2` was known, the package was still in use due to the developers' re-packaging strategy. The maximum of 1,216 days was reached by `npm/rpc-websocket/0.7.7` which took over an abandoned package and went undetected for a long period. In general, this shows that packages tend to be available for a longer period. While PyPI has the highest average online time, that period varies the most for npm, and RubyGems tends to detect malicious packages timelier.

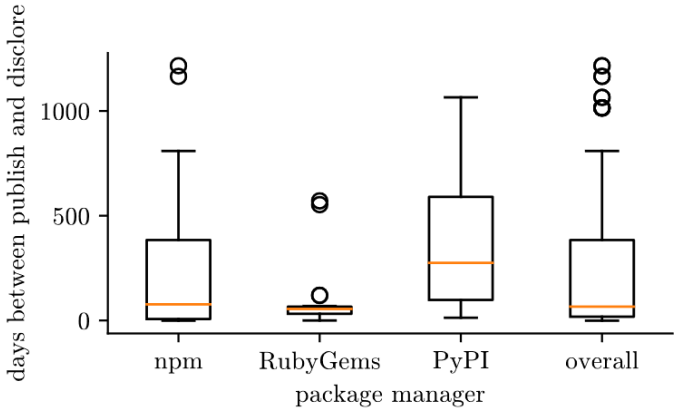

Figure 18: Temporal distance between date of publication and disclosure (from [49])

As shown in Figure 19, most packages aim at data exfiltration. Commonly, the data of interest is the content of `/etc/passwd`, `~/.ssh/*`, `~/.npmrc`, or `~/.bashhistory`. Furthermore, malicious packages try to exfiltrate environment variables (which might contain access tokens) and general system information. Another popular target (7 reported packages, 3 of them available in our dataset) is the token for the voice and text chat application Discord. A Discord user's account may be linked to credit card information and thus be used for financial fraudulence. Moreover, 34% of the packages function as Dropper to download second stage payload. Another 5% open a backdoor, i.e. reverse shell, to a remote server and await further instructions. 3% aim to cause a denial of service by exhausting resources through fork bombs and file deletion (e.g. `npm/destroyer-of-worlds/1.0.0`) or breaking functionality of other packages (e.g. `npm/load-from-cwd-or-npm/3.0.2`). Only 3% have financial gain as primary objective by for instance running a cryptominer in the background (e.g. `npm/hooka-tools/1.0.0`) or stealing cryptocurrency directly (e.g. `pip/colourama/0.1.6`). In addition, combinations of the above-mentioned objectives might occur.

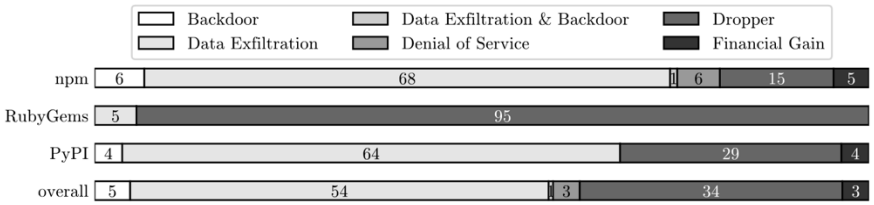

Figure 19: Primary objective of the malicious package per package repo and overall (from [49])

Malicious actors often try to disguise the presence of their code, i.e. hindering its detection by sight. In our dataset nearly the half of the packages (49%) employ some kind of obfuscation. Most often a different encoding (Base64 or Hex) is used to disguise the presence of malicious functions or suspicious variables such as domain names. A technique often used by benign packages to compress source code and thus save bandwidth is minification. However, this is a welcome opportunity for malicious actors to sneak in extra code which is unreadable for humans (e.g. `npm/tensorplow/1.0.0`). Another way to hide variables is to use string sampling. This requires a seemingly random string which is used to rebuild meaningful strings by picking letter by letter (e.g. `npm/ember-power-timepicker/1.0.8`). In one case the malicious functions are hidden by encryption. The package `npm/flatmap-stream/0.1.1` leverages AES256 with the short description of the targeted package as decryption key. That way, the malicious behaviour is solely exposed when used by the targeted package. Furthermore, combinations of the above-mentioned techniques exist.

Maintenance and use

The complete dataset is available for free on GitHub[5]. However, access will be granted on justified request only due to ethical reasons. The dataset is structured as follows: `package-manager/package-name/version/package.file`. Malicious packages are grouped by their originating package manager on the first level. Further, multiple affected versions of one package are grouped under the respective package's name. As example for the affected version of the well-known case of event-stream it is: `npm/event-stream/3.3.6/event-stream-3.3.6.tgz`.

After making the dataset public, it received several contributions from 3rd parties, including the security team of a well-known provider of Java-related open-source tooling. In the meantime, the dataset grew from 174 to 1,083 samples (as of December 7th).

Going forward, the open-source dataset will allow researchers within and outside of the SPARTA research project to study real-world supply chain attacks in order to develop detective and preventive safeguards.

The usefulness of such a public dataset is illustrated by three usages within the SPARTA CAPE program: The data will be one important input to train models that predict the attractiveness of open-source projects from an attacker perspective (see Section 4.2.3.1), thus, projects that require special attention and protection. Moreover, the dataset entries will be transformed into YAML statements according to the format of Project KB (see Section 7.10), which makes it possible to consume this data in automated scanners such as Eclipse Steady (see Section 7.15). Furthermore, the dataset was used to demonstrate the feasibility of Buildwatch (see Section 7.3).

4.2.3.3 Commit-based detection of malicious packages

This section presents an approach to detect malicious open source packages published on distribution platforms like PyPI (Python) or npm (Node.js), and which is based on the intuition behind reproducible builds[6]: it is suspicious if the code in the source code repository differs from the code in the artefacts distributed in the package repository. In this respect, we propose an approach to detect code injected into software packages by comparing their distributed artefacts (e.g., those in PyPI) with the source code repository (e.g., those in GitHub). The proposed approach can be used to detect injected code in typo squatting and hijacked packages.

For example, consider a typo squatting Python package `jeIlyfish`, discovered by Lutoma [58], which was persistent in PyPI for nearly a year until its detection on December 1, 2019. `jeIlyfish` mimicked the popular package `jellyfish` (the first L is an I) to steal SSH and GPG keys. Our technique processes the suspected `jeIlyfish` artefacts to identify the corresponding source code repository. Then we compare the file hashes and contents extracted from the artefacts with those obtained from the source code repository. Our tool detects two injected files: `setup.py`, and `_jellyfish.py` and reports several lines in `_jellyfish.py` to contain suspicious API calls for decoding and executing the malicious code.

As said, the approach compares distributed artefacts in package repositories (e.g., PyPI) and the source code repository (e.g., GitHub) to detect the injected code by the following steps:

1. For each package, we identify the source code repository by mining metadata properties (e.g., homepage).
2. We clone the repository and extract all the commits. For each commit, we check out each involved file, calculate the file hash, and collect the file content. The file hashes and contents are stored into a database.
3. We download each artefact of the package from the package repository, decompress it into files. For each file, we calculate the hash and collect the file content.

[5] https://github.com/dasfreak/Backstabbers-Knife-Collection

[6] https://reproducible-builds.org/

4. Then we compare the file hashes and contents from step (3) with those extracted from step (4). This comparison results in files (and their lines) whose hashes are not recorded in (differ from) the source code repository.
5. For the unknown lines, we check the presence of API calls (e.g., `urlopen`) and imports (e.g., `import os`) using regular expressions.

During the packaging process, the packaging tools (e.g., setuptools in Python) create new (benign) metadata files (e.g., METADATA, WHEEL), these files are specified in PEP 4277. Hence, we exclude such files from our analysis and focus on the differences in files containing executable code (e.g., .py, .js, .rb).

Chapter 5 Connected and Cooperative Car Cybersecurity Vertical Technical Specifications (Vertical 1)

5.1 Context and Background

The “Connected and Cooperative Car Cybersecurity” vertical (a.k.a. Connected Car Vertical) has as goal to advance the cyber-security of connected vehicles driving in platoon mode. A platoon is a sequence of vehicles as depicted by Figure 20, it is composed by a leader vehicle and a sequence of followers.

The increased interconnectivity between vehicles in the platoon and their increased level of autonomy raise both the attack surface of these systems and the degree of damage that intruders can cause. Indeed, cyber- attacks can exploit vulnerabilities in the available communication channels to cause catastrophic events, i.e., great human and material loss. For more details about the user requirements for the Platooning scenario, we refer the interested reader to D5.1 [1].

Each vehicle in the platoon communicates using dedicated communication channels. Moreover, each vehicle in the platoon possesses sensors, such as cameras, distance sensors, enabling a highly automated mode of operation. Indeed, when formed, the platoon requires only driver supervision.

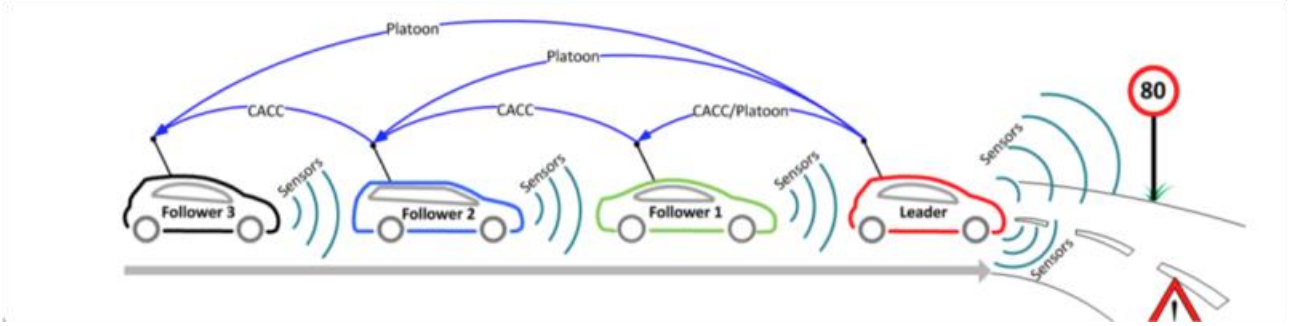

Figure 20: Platooning scenario

The use of communication channels and sensors enable the gap between vehicles to be greatly reduced. This means that a vehicle, such as Heavy-Duty Vehicles, can greatly profit from the wind shadow of the following vehicle, thus greatly reducing fuel consumption. Studies have shown that fuel consumption can be reduced by 17% in a platoon formation with reduced gaps between vehicles [59][11].

However, the reduced gap between vehicle has serious consequences to platoon safety. On the one hand, the reduced gap improves safety as it avoids vehicles that are not part of the platoon to move between two platoon vehicles. On the other hand, the reduced gaps increase the chance of accidents and potentially causing harm.

As a control measure, the platooning greatly relies on the communication channels to reduce the reaction time of vehicles in case of a vehicle ahead of the platoon reduces its speed or even needs to perform an emergency brake. Figure 21 illustrates the reduction of reaction time resulting from the use of the platooning communication channel [60]. The reaction time without relying on any automated function, such as advanced driving assistants, is greatly increased by the time that humans perceive that the vehicle ahead is reducing its speed and also by the time that it takes to react to this. The reaction time is reduced by using automated driving assistants placed in a vehicle, e.g., sensors. However, these functions still take time to determine that the vehicle ahead is starting to stop, as they need to identify the change in speed (or gap to the vehicle immediately in front).

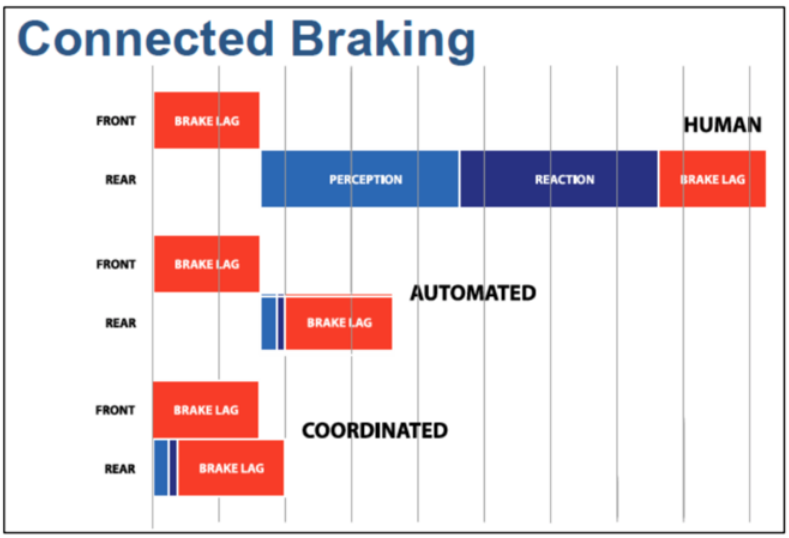

Figure 21: Illustration of the reduction of reaction time by using the platooning communication channels

By using the communication channels, on the other hand, the reaction time is greatly reduced as the vehicle in front can simply inform vehicles in the back that it is reducing its speed (and even to which speed), enabling the vehicles in the back to react almost immediately and also reducing their speed in order to avoid accidents. The delay caused by this communication is orders of magnitude lower than the delay caused by human/sensor perception.

However, the use of communication channels leads to security challenges. Indeed, as described in the literature [61] [62] [63], intruders can impersonate a vehicle and inject messages with false information about a vehicle speed and position to cause harm, namely, accidents. Intruders can carry out such attacks for financial motivations like, *e.g.*, to carry out ransom attacks or to steal the transported cargo. To increase the security of vehicle platooning against these types of attacks, countermeasures based on plausibility checks have been proposed. These plausibility checks cross check the consistency of the information received by a vehicle with, e.g., the data collected by its other sensors.

5.2 Scenarios

In the following sections, we describe five scenarios involving the security of vehicle platooning. Each scenario has been carefully selected so that they focus on different aspects for the safety and security of vehicle platooning, investigating how the considered CAPE tools can support the safety and security process.

- **Basic Scenario**: The first scenario's goal is to evaluate the process, from security analysis, requirements to implementation and verification and validation, for increasing the security of vehicle platooning when assuming a malicious intruder that can manipulate the communication channels.
- **Firewall updates**: The second scenario considers the update of firewall policies so to ensure safety and security in a continuous fashion.
- **Verification tooling**: The third scenario focuses on the verification tools that can be used to verify the security of vehicle platooning.
- **Safety and Security compliance assessment and certification**: The fourth scenario considers the generation of assurance cases for certification standards, such as the ISO 26262 and SAE J3061. These assurance cases contain the safety and security arguments and the evidence supporting these arguments.
- **Fault-injection and analysis of faulty scenarios**: The fifth scenario considers the impact of component faults for the safety and security of vehicle platooning.

The details on the implementation of these scenarios are collected in D5.3 [2].

5.2.1 Scenario 1: Basic Scenario

We consider a platoon using Cooperative Adaptive Cruise Control (CACC), with one leader and *n* followers, where new vehicles may join the platoon after a negotiation phase. We assume that the platoon vehicles navigate on a straight road, and that vehicles can communicate using peer-to-peer connections or by broadcasting messages. We also assume that all messages are signed using vehicles secret keys that cannot be guessed by intruders, and contain adequate measures to ensure freshness, such as using timestamps or nonces, to avoid replay attacks.

The goal of our intruder is to cause a crash between two legitimate vehicles. To this end, the intruder either injects false messages into the CACC communication channels or jams (i.e., blocks) legitimate messages from the CACC communication channels. The actual capability used by the intruder depends on the attack scenario. We consider scenarios where the intruder (1) injects false messages only, (2) blocks messages only, and (3) both injects and blocks messages.

To ensure that injected messages are valid, we assume that the intruder can obtain encryption keys from any vehicle in the platoon. The same assumption is considered by previous related work like, e.g., [64] and [65]. For simplicity, we assume that the intruder has obtained the leader's encryption key.

Given the leader's encryption key, the intruder makes valid connections with a target vehicle (i.e., a follower or a joining vehicle). For example, assume an attack scenario where both capabilities (i.e., injecting and blocking) are required. The intruder blocks all messages originated from the leader and injects (impersonating the leader) false messages to either followers or vehicles joining the platoon.

We describe next in more detail the type of attacks that we consider for this basic scenario.

Attack 1 (II-B): Injecting false messages to follower and blocking legitimate messages from leader

An intruder sends false position and speed values to a vehicle in order to cause a crash with the preceding vehicle. This attack works because CACC algorithms ensure that a vehicle maintains a desired distance from the preceding vehicle based on the received messages from other vehicles in the platoon (especially from the leader). The attack scenario is illustrated in Figure 22.

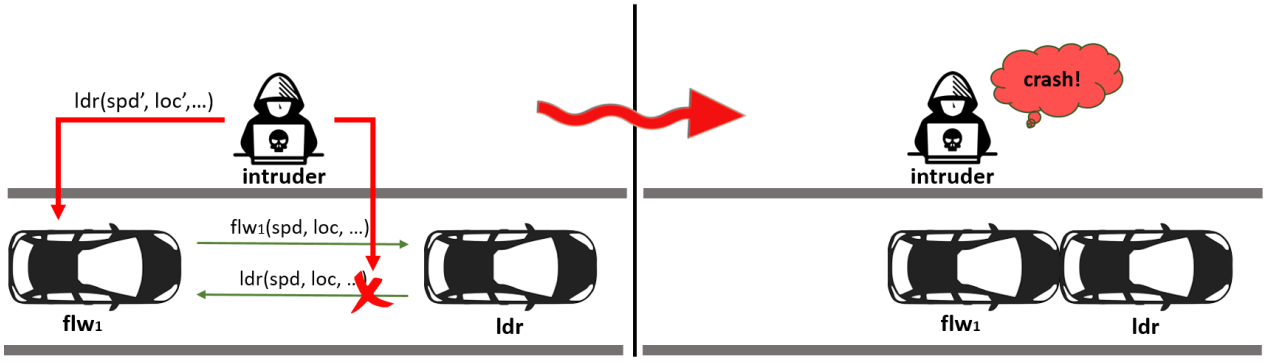

Figure 22: Attack1: Injecting false messages to follower and blocking legitimate messages from leader

This scenario is composed of two vehicles: a leader *ldr* and a follower *flw1*. Illustrated by the green arrows, such vehicles exchange information to ensure that *flw1* keeps a safe distance from *ldr*. The red cross illustrates that the legitimate messages from the leader are blocked by the intruder while the attack is in progress. Next, the intruder impersonates *ldr* to send high position and speed values to *flw1*. The follower *flw1* adapts its distance based on the high false values sent by the intruder. As a result, a crash between *flw1* and *ldr* is expected, as illustrated by the right-hand side of Figure 22.

To mitigate this attack, we propose a countermeasure based on **plausibility checks** for the information that is communicated between the platoon vehicles. These plausibility checks are

described in the protection profile and described in this deliverable (see Section 5.3.4). In a nutshell, the **countermeasure** works as follows: Whenever a vehicle receives a message with the speed of the preceding vehicle, the countermeasure checks it against the local history of measurements. The countermeasure is triggered if the incoming speed value deviates from given percentage w.r.t. the average of the last *n* speed values received by the vehicle.

Attack 2 (II-C): Slow-Injection of false messages

The goal of Attack 1 is a quick crash between two vehicles. To this end, the intruder injects extreme false position and speed values into the CACC communication channels. However, existing countermeasures (a.k.a plausibility checks) are able to detect such extreme values, and thus mitigate the attack. Attack 2 is a smarter variation of the previous attack in order to bypass existing countermeasures that checks whether incoming values highly deviate from the previous received ones. To this end, the intruder injects messages with false information into the CACC communication channels modifying the values of speed and position with a small increase rate after each message.

Attack 3 (II-D): Injection of false messages against joining vehicle

A new vehicle may join a platoon after a negotiation phase (a.k.a synchronization handshake) with the leader of the platoon. During this negotiation phase, the leader sends the platoon information to this vehicle, including the position and speed of the last vehicle, so that the joining vehicle can adapt itself to catch up to the platoon.

An intruder may impersonate the leader to send false information during this negotiation phase. For example, assume an attack scenario composed of two vehicles: the leader *ldr* of the platoon and a vehicle (*veh*) that wishes to join the platoon. The intruder may inject (as *ldr*) high position and speed values to *veh* during the negotiation phase, while blocking all messages originated from *ldr*. Eventually, *veh* crashes into *ldr*, as *veh* adapts its acceleration based on the received values.

Attack 4 (II-E): Injection of false emergency brake messages

Emergency brake is a safety-type message that may be triggered by any vehicle in the platoon to avoid crashes. For example, the leader may trigger an emergency brake if an obstacle is detected in its path. Then each follower receives an emergency brake message from the leader, and immediately actuates by stopping the vehicle.

An intruder, however, might take advantage of this situation to carry out attacks. Figure 23 illustrates an attack scenario using emergency brake messages. This scenario is composed of three vehicles: a leader (*ldr*) and two followers (*flw1*) and (*flw2*). The goal of this attack is a crash between *flw1* and *flw2*. To this end, the intruder injects a false emergency brake message to *flw1* only. This message results in a crash as *flw1* immediately stops and *flw2* keeps driving, yet following the previously received information (*e.g.*, position and speed).

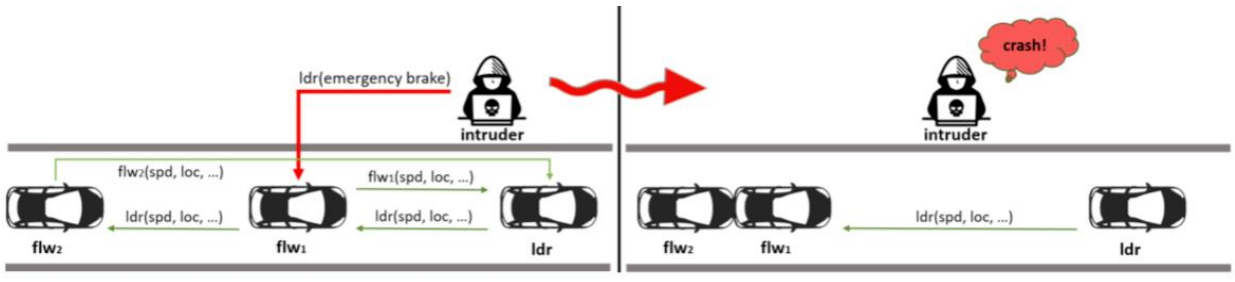

Figure 23: Injecting false emergency brake to follower

Attack 5 (II-F): Blocking legitimate emergency brake messages

Instead of injecting false emergency brake messages, the intruder may block legitimate emergency brake messages from the CACC communication channels in order to cause a crash. An attack scenario with this purpose is illustrated in Figure 24.

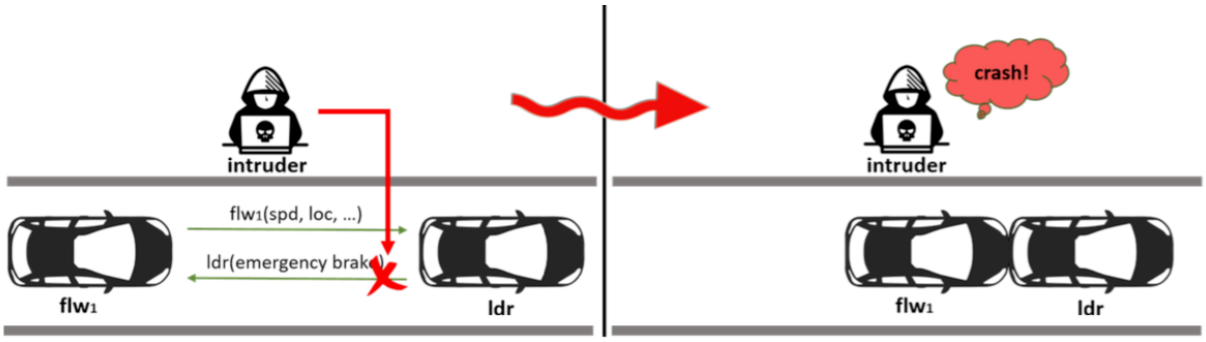

Figure 24: Blocking legitimate emergency brake from leader

The intruder monitors the channels till a legitimate emergency brake message is triggered by the leader (*ldr*). At this point, *ldr* stops the vehicle and the intruder blocks the message to avoid that any follower (*flw1*) can receive and trigger emergency brake as well. As a result, *flw1* keeps driving the vehicle till crashing into *ldr*.

Finally, the Connect Car basic scenario is complemented by the incorporation of a **dashboard,** a web page that allows to check the status of the platoon and launch cyber-attacks to the platoon members. The dashboard mock-up is depicted in Figure 25. The dashboard shows the following data for each platoon member:

- The unique identifier of the car.
- The current mode of the car in the platoon. We will manage only three modes: *leader*, *follower* and *emergency break*. Others, such as *joining to the platoon* or *leaving the platoon*, are not involved in our scenario.
- The current speed of the car.
- The distance gap with any obstacle behind detected by the distance sensor.
- The id and speed of the preceding vehicle in the platoon.
- The number of speed messages received that didn't pass any of the plausibility checks.

The dashboard will also have capabilities to launch three kind of cyber-attacks to the platoon members:

1. Injecting false speed messages from one car to another.
2. Blocking the legitim incoming messages to a car.
3. Hacking one of the sensors of a car (camera, speed or distance).

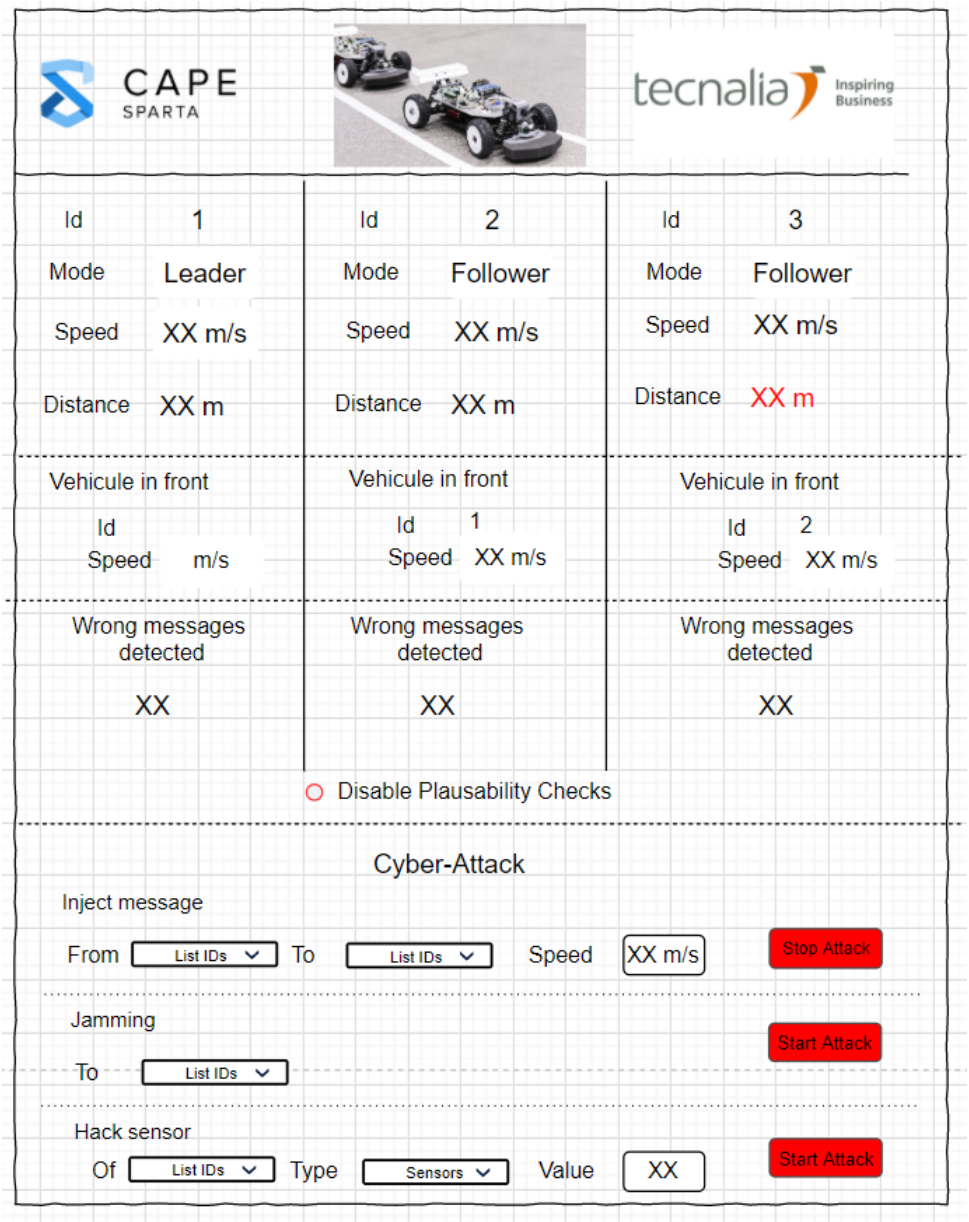

Figure 25: Dashboard mock-up (Platooning basic scenario)

5.2.1.1 Specificities for FTS Rovers

Figure 26 shows an FTS Rover. FTS rovers are composed of two Raspberry Pi devices, two ultrasonic sensors, one laser sensor, and one camera. The model of the Raspberry Pi devices is *Raspberry Pi 3 Model B Plus* and they are responsible, respectively, for running the code generated from the AutoFOCUS3 model, and for the lane detection using the camera. The Raspberry Pi devices communicate with each other (e.g., to exchange information on lane detection) through a WiFi network. Both the ultrasonic and the laser sensors are responsible for detecting the distance to the preceding FTS rover. The camera, as mentioned before, is responsible for detecting the lanes.

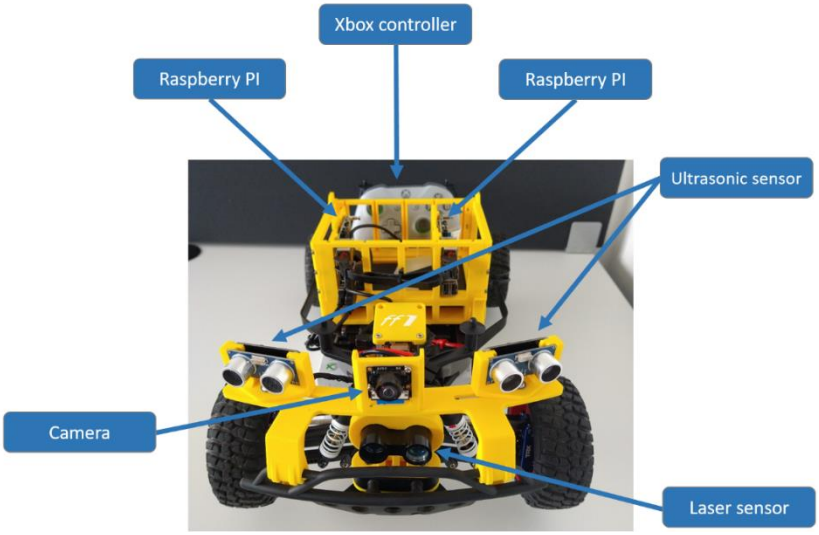

Figure 26: FTS Rover

FTS rovers may drive autonomously with the help of the sensors listed above and the exchanged messages (e.g., with speed values) between rovers. Those messages are exchanged through Ethernet network. When set to manual-drive mode, the FTS rover is remote-controlled by an Xbox controller connected through a WiFi network. Figure 27 illustrates two of the FTS rovers driving on a single lane circuit, marked with white lines.

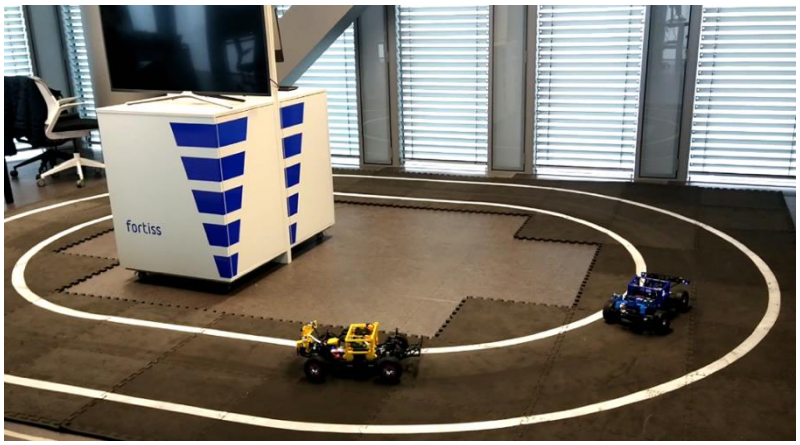

Figure 27: FTS rovers moving on the circuit

AutoFOCUS3 supports the automatic generation of C code from the component architectures specified in the AutoFOCUS3 model. Once the C code generation is completed, one can deploy the C generated code into the FTS rovers[7]. The deployment process is performed by (1) copying the C generated code into the FTS rover, (2) locally compiling the code on the rover, and (3) launching the executable code on the rover.

5.2.1.2 Specificities for TEC Rovers

Figure 28 shows the hardware components of TEC rovers:

- ADAS-ECU. An Odroid-XU4 board that will run all the autonomous-driving-functions as lane detection, lane keeping and the CACC generated by AUTOFOCUS3. All those functions need the inputs coming from the sensors. This board is also in charge of managing the rover communications with other rovers, through a WiFi network, and with the Vehicule-ECU, in order to manage the camera and the ultrasonic sensor.

[7] https://download.fortiss.org/public/projects/af3/help/ta/code_generation.html

- Vehicle-ECU to manage the speed and direction of the rover.
- Camera to capture images for the detection of the road to calculate the direction.
- Ultrasonic Sensor to obtain the distance of the preceding rover or the distance to any obstacles in the path of a rover.
- Encoders in each of the back wheels to measure the current wheel speed.
- Wifi Module to allow communications between rovers.
- Remote Control. It's possible to control the speed of a rover by its remote control and its direction by the ADAS, or vice versa. Even when the speed and direction of a rover are controlled by the ADAS, the vehicle remote control system still has to be used as a deadman-switch to enable the movement of the rover.

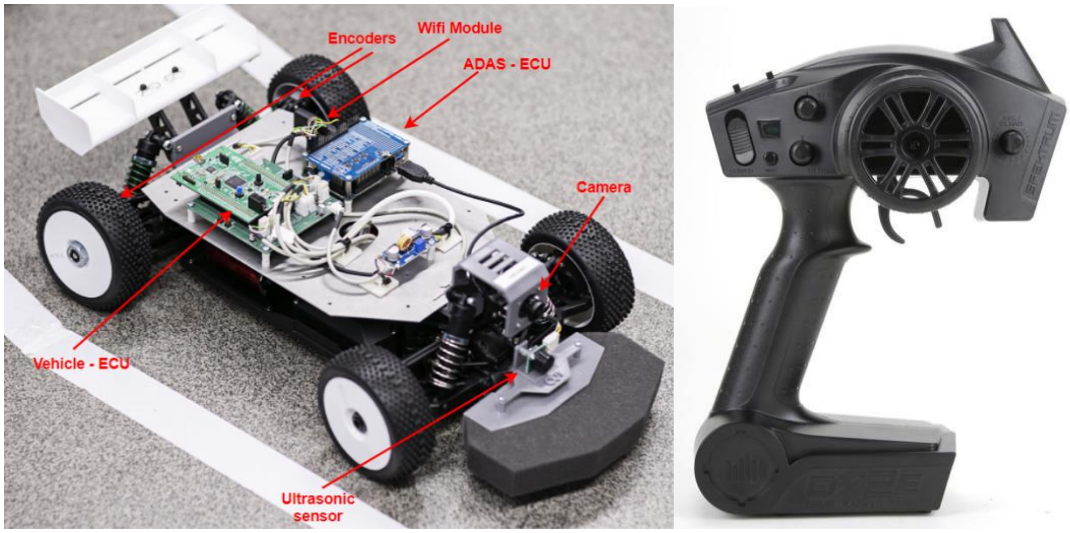

Figure 28: TEC Rover + Remote Control

TEC rovers may drive autonomously with the help of the above sensors on a single lane circuit of 3m x 2,5m (see Figure 29), marked with white lines separated approximately by 45cm.

Figure 29: TEC Rovers moving on the circuit

5.2.2 Scenario 2: Firewall updates

In this demonstration scenario, we consider the Basic Scenario as a basis and develop an Infrastructure to Vehicle (I2V) case study where continuous compliance can be maintained when security requirements are dynamic.

This section describes the analysis of two V2I scenarios in the context of Platooning: the firewall reconfiguration scenario and the firewall update scenario.

The context for these two scenarios is the “platoon gap adaptation” case study and a multi-layer platoon management platform from the ENSEMBLE (ENabling SafE Multi-Brand pLatooning for Europe) H2020 project [66] that aims for multi-brand vehicle platooning to improve fuel economy, traffic safety and throughput.

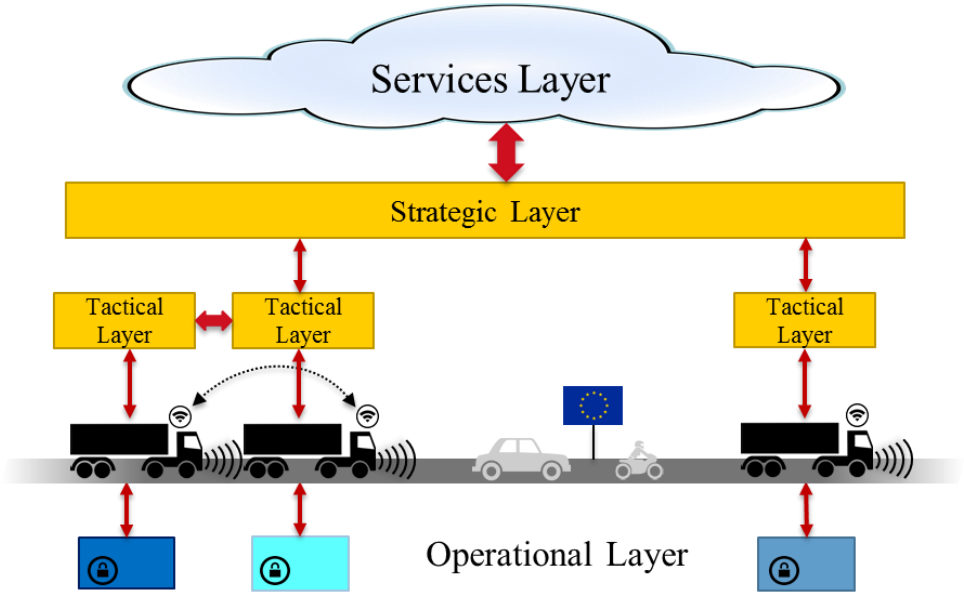

Figure 30: Layered architecture of ENSEMBLE

Figure 30 shows the multi-layer ENSEMBLE architecture for platoon management [66]. The layers are the following:

- Service layer: provides a platform for added value logistic services related to platooning.
- Strategic layer: planning of platoons based on vehicle types and optimisation with respect to fuel consumption, travel times, destination, and impact on highway traffic flow and infrastructure. It is also responsible for vehicle routing to enable platoon forming. Centralised traffic control centres communicate via long-range wireless communication with platoon vehicles and drivers.
- Tactical layer: coordinates platoon formation, operation and dissolution.
- Operational layer: controls actuators to accelerate, brake and steer to regulate inter-vehicle distance or velocity.

The “platoon gap adaptation” case study involves V2I communication between the platoon vehicles and the traffic control centres. In this case study the traffic control centres inform passing platoons of specific zone policies such as increased distances between vehicles, specific speeds or lateral positioning.

In the **V2I firewall reconfiguration scenario** platoons communicate with the traffic control centres via edge clouds distributed along the road network. As the platoon progresses it must change edge clouds to get the best network latency available. Each platoon leader that communicates with the traffic control centres is protected by a firewall. As the platoon progresses the firewall needs to be reconfigured when a new edge offers better QoS than the currently connected edge.

From the monitoring point of view only authorized firewall reconfigurations should be made. The firewall must be monitored for detecting firewall intrusions and unauthorized changes to the configuration that would allow an attacker to launch injection or jamming attacks [67] [68]. Assuming that the firewall has been certified with respect to a Common Criteria protection profile such as the

SafeSecPMM protection profile (see Section 5.3.4), as long as the firewall operates within the requirements defined in the protection profile, and includes the ability to reconfigure the firewall, there is no need to re-certify the firewall. However, the reconfigurations of the firewall must be monitored in order to verify that the certified requirements are respected.

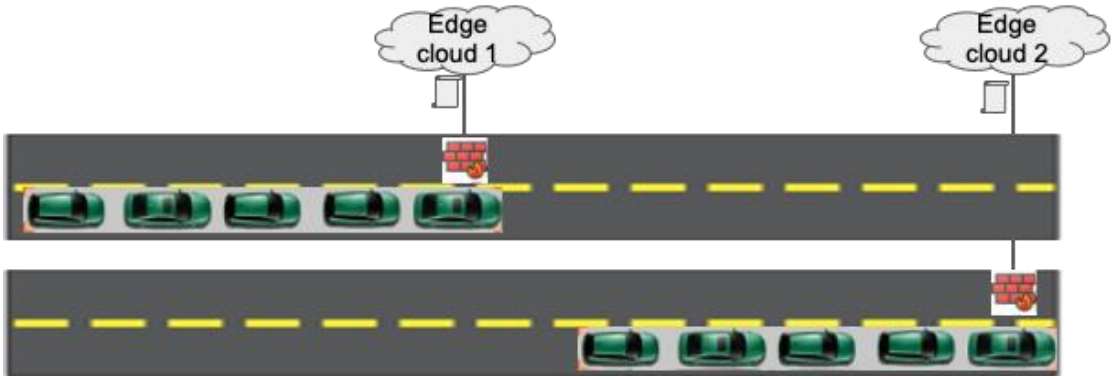

Figure 31: V2I firewall reconfiguration scenario

In the **V2I firewall update scenario**, a new version of the firewall is available and needs to be deployed on customer vehicles. The update is performed when vehicles are not being driven. The update process is described in the figure below.

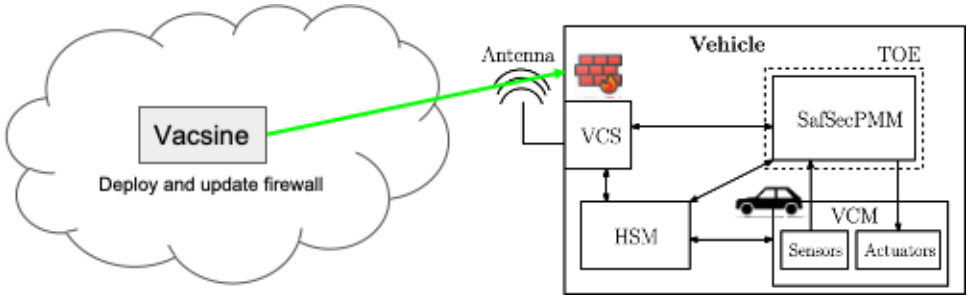

Figure 32: Firewall update scenario

The firewall update is orchestrated by the VaCSIne tool that is running in a Cloud and has agents deployed in the cars. The firewall shows that the firewall protects the SafeSecPMM from external communication threats. From a certification point of view the new firewall version must be analysed with respect to the SafeSecPMM protection profile to determine if some certified requirements are impacted. If some certified requirements are impacted then the new firewall version must be re-certified. In this case an incremental certification [69] can be performed.

The infrastructure informs driving vehicles of zone policies, those can state for example increased distances between vehicles, speed recommendations, increased security requirements, etc. Upon notification of a new zone policy, the platoon will coordinate to apply it. When the platoon leaves the zone, the policy is invalidated and the platoon will reconfigure itself accordingly. We assume the platoon is running in a default mode and without error, and that the platoon and I2V communications are secure.

The demonstration is composed of a platoon of rovers connected to a cloud using edge infrastructure nodes that have various security requirements. When the platoon enters the zone controlled by a given edge node, it receives a new security policy. For our purposes, the change of edge node is based on the strength of WIFI signal between the edge node and the platoon. The new security policy can be more or less strict than the previous one. We consider an edge node with a default security policy where security considerations are kept minimal, and another edge node that necessitates enhanced security functions such as hardening of the firewall with stricter network access rules. To ensure that the platoon security configuration satisfies the new zone security policy, the platoon is scanned for vulnerabilities by the edge node. Vulnerability scan reports and audit logs

of the security operations are collected, made available for the next steps of the continuous certification process and displayed in a basic web dashboard.

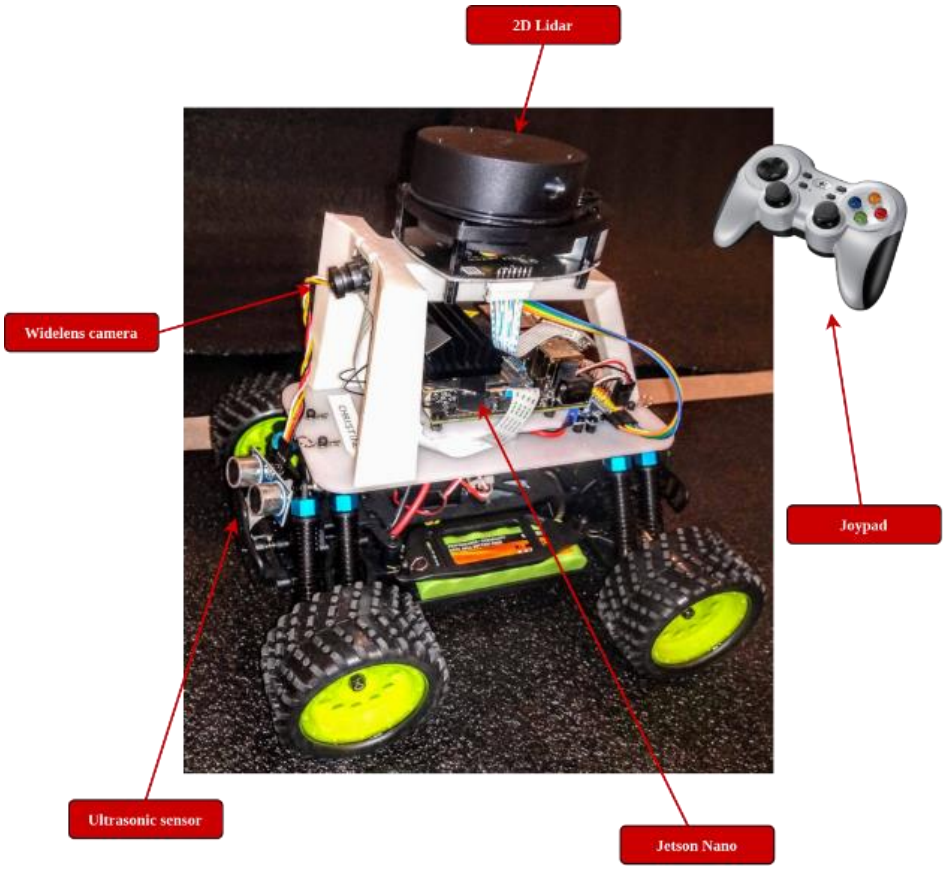

Figure 33: CETIC Donkey Car rovers

CETIC rovers are based on the Donkey Car[8] platform. They can use a Raspberry Pi to drive the rover autonomously, but we chose the Jetson Nano Development Board[9] instead to leverage its improved GPU capabilities. The rovers use a camera for lane detection, ultrasonic sensors and a 2D Lidar for distance detection. The rovers can also be controlled manually using generic joypad controllers over WIFI, for example when doing autonomous driving training. The edge infrastructure consists of single board computers playing the role of WIFI access points that have associated zone security policies. The cloud consists of a container orchestration platform deployed inside a private or public cloud.

5.2.3 Scenario 3: Verification tooling

We consider the Basic Scenario described in Section 5.2.1. as a basis. Taking into account the functions of this scenario the following Verification tooling setup has been proposed.

The penetration testing scenario comprises the following inputs:

- Analysis of the attack's surfaces and protocols to be checked as part of the TOE as defined in the Protection Profile.
- Preparation of a set of HW tools to interact with the surfaces found in the rovers' demo.
- Preparation of a set of scripts or manual procedures to be run over the HW tools.
- Follow the directives and standards of the AVA_VAN family of the CC standard [19].

[8] https://www.donkeycar.com/

[9] https://developer.nvidia.com/embedded/jetson-nano-developer-kit

Two sets of Rovers (FTS and TEC) are considered with the following architectures and attack's surfaces (see Figure 34). CACC protocol and plausibility checks work thanks to the WiFi protocol. The ultrasonic sensors are used as distance sensors and the camera is used for lane detection.

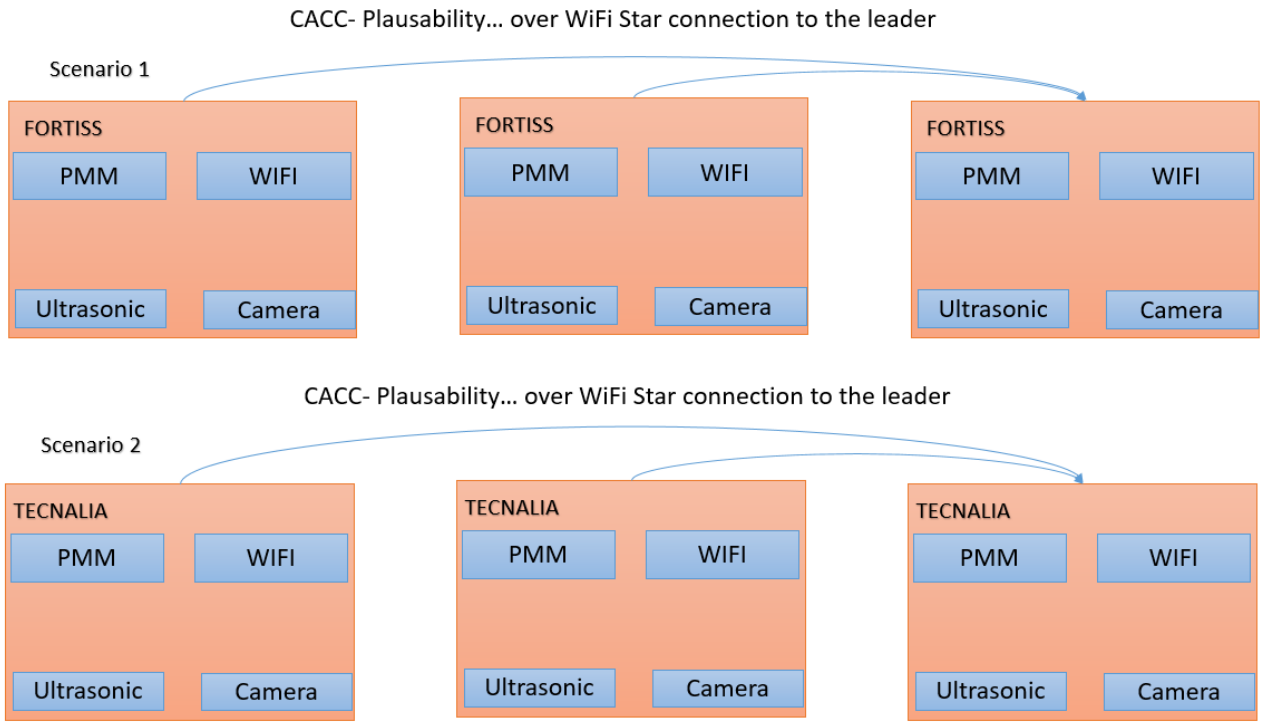

Figure 34: Set of architectures to be tested

In order to perform the verification activities, the output of some of the tools considered in the SPARTA CAPE project are going to be used:

- AF3 block diagrams/functionality diagrams (see Section 7.2)
- Maude verification outputs (see Section 7.7)
- Code and System Architectures (EUT will provide a questionnaire to be filled by TEC and FTS with the different protocols used)
- Others: Firewalls attacks are going to be considered, due to the firewall update architecture planned in the Scenario 2 (see Section 5.2.2).

Another important point is to prepare a set of HW tools that will be used to attack the attack's surfaces identified. These tools include a basic USB Alfa Wifi Antenna to perform attacks on Wifi interfaces and protocols, ultrasound sensors or other devices which are able to provoke misfunction of the cameras.

Figure 35: Set of HW tools to be used for the penetration testing

Due to the COVID-19, the possibilities of performing penetration testing on-site are difficult because a set of rovers is in Munich (FTS), another one is in Bilbao (TEC) and the penetration testers are in Barcelona (EUT). A possible mitigation plan which is under study is to use Raspberrys Pi and a basic set of tools which are connected to them and allow remote access to the Raspberry on each of the scenarios. It's a kind of complicated setup but the situation requires it.

On top of this HW tools the following kind of scripts and manual procedures are considered:

- Scripts of protocols (known vulnerabilities, as defined in AVA_VAN) WiFi, TLS, etc.
- Scripts of the Platooning CACC (with inputs of Maude Verification, etc.).
- Scripts involving also sensors.
- Scripts on the firewall update protocols.

The quantity of scripts and surfaces to attack will depend on the quantity of time available to perform the analysis, beginning on the most characteristics of the platooning main application and ending in the less related attacks. It's important to mention as well that some attacks require a "step by step exploration" instead of a basic script.

5.2.4 Scenario 4: Safety and Security compliance assessment and certification

For the Scenario 4, the Basic Scenario described in Section 5.2.1 has considered as a basis. The OpenCert management tool (see Section 7.9) will be applied helping the Safety and Security engineer in the whole assurance processes of the Connected Car vertical life cycle.

As it was mentioned in the deliverable D5.1 [1], Safety and Security standards has been considered, particularly *ISO 26262 "Functional Safety Road Vehicles"* for functional safety and *SAE J3061 "Cybersecurity Guidebook for Cyber-Physical Vehicle System"* for cybersecurity. OpenCert will support knowledge management about these two standards. The "Standards & Regulations Information Management**"** activity group supports knowledge management about standards, regulations and interpretations, in a form that can be stored, retrieved, categorized, associated, searched and browsed. The activities involved in this group are intended to be shared among various assurance projects. Figure 36 shows in a graphical way the ISO 26262 standard modelled in OpenCert, where the different parts of the ISO 26262 are represented with boxes and, their activities, sub-activities and requirements are defined inside them.

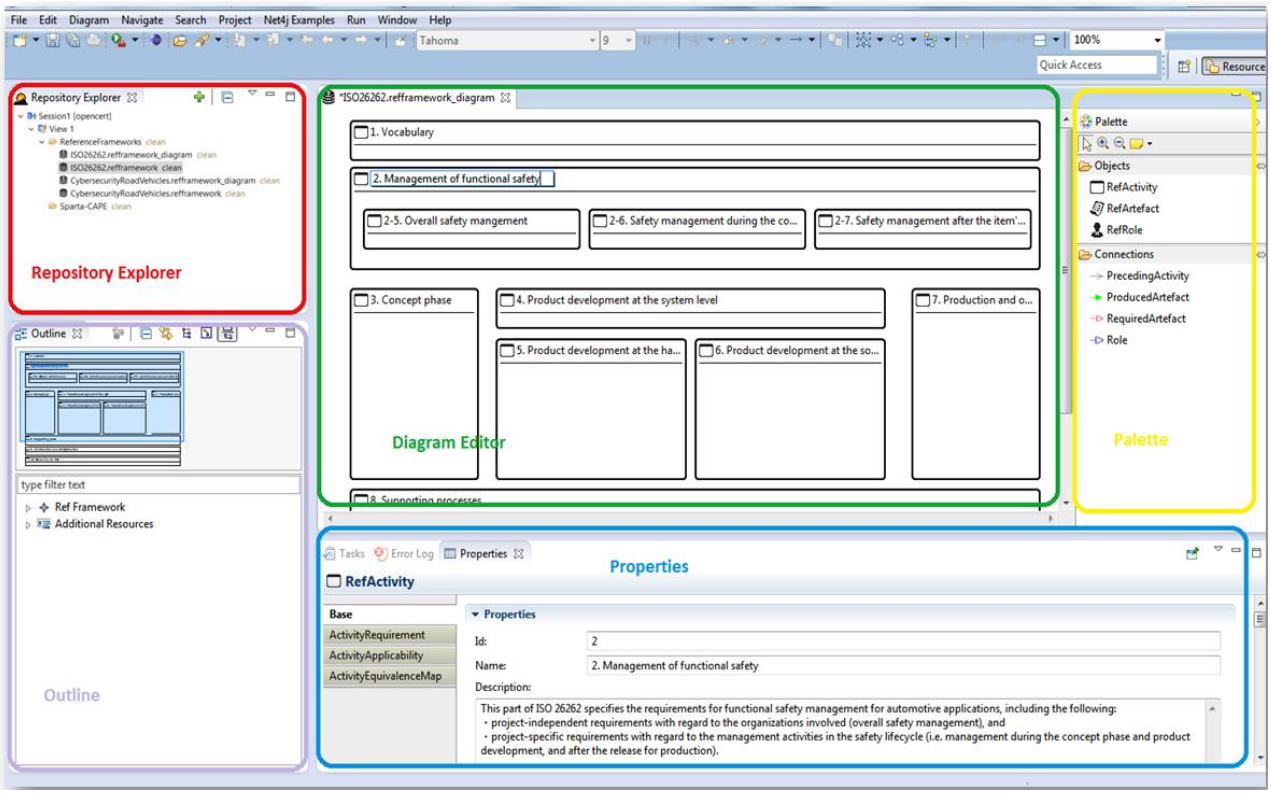

Figure 36: OpenCert – Modelling the ISO 26262 standard

OpenCert will assist the entire assurance process of the platooning scenario, on the one hand, the Safety assurance process following the ISO 26262 and on the other hand, the Security assurance process regarding SAE J3061. Both assurance processes will be constantly connected to each other to be sure that there are not incongruences between them. This is the main activity group called “Assurance Project”, where assurance process management, and global monitoring of the compliance with standards, assurance cases and evidence management are performed.

Assurance Case Management functionality is a feature which manages argumentation information in a modular fashion as it is shown in Figure 37. Assurance cases are a structured form of an argument that specifies convincing justification that a system is adequately safety and secure for a given application in an environment. Assurance cases are modelled as connections between claims and their evidence.

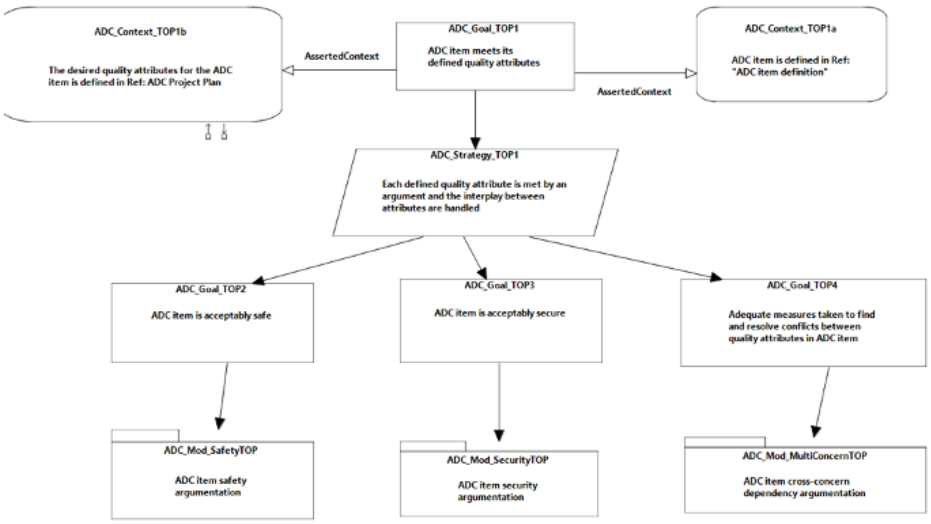

Figure 37: OpenCert - Assurance Case example

The Evidence Management functionality deals with the specification of the artefacts that are used as evidence in an Assurance Project. The artefacts can have specific properties and can be the result generated from external tools (e.g. results of a test case) stored in OpenCert. All these aspects are managed throughout an artefact's lifecycle, which can include changes to an artefact and evaluations (e.g. about the completeness of a document). Figure 38 shows an example of how the results generated are stored.

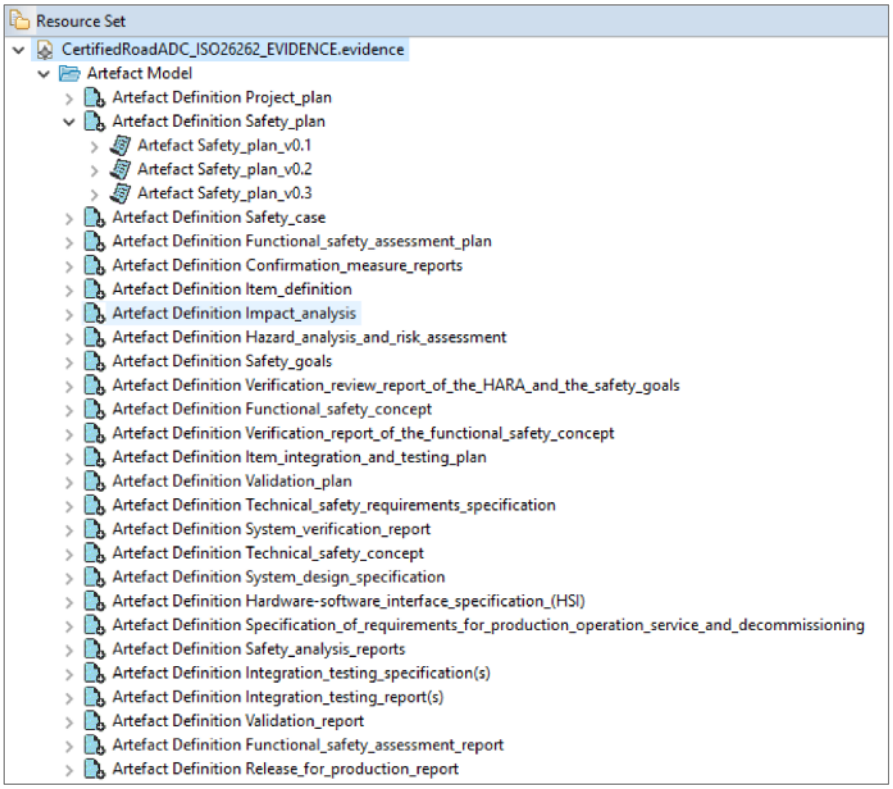

Figure 38: OpenCert - Evidence management feature

To conclude, the development of all the above mentioned activities related to the OpenCert tool will be carried out and documented in detail within the deliverable D5.3 [2].

5.2.5 Scenario 5: Fault-injection and analysis of faulty scenarios

For the Scenario 5, the Basic Scenario described in Section 5.2.1 has considered as a basis. The Sabotage tool (see Section 7.12) will be used to simulate how a fault, originated from a random hardware fault or cyber-attack, can affect the vehicle behaviour by changing the velocity to an abnormal value.

Each vehicle in the platooning has different control measures integrated. In this scenario, the sensor-based plausibility check will be evaluated. This plausibility check verifies the incoming sensor signals (ultrasonic sensor) and the speed received from the preceding vehicle via WIFI and detects faulty or missing signals (see Section 5.3.4.3).

The developed sensor-based plausibility check is used to validate the correctness of the speed received from the preceding vehicle and to compare it with the distance sensor. By adding different faults into the plausibility check algorithm, the engineer can verify that it assesses safety and security requirements. The effectiveness (detection and/or recovery of errors) of the measures can be analysed to know if it is needed, for example, to modify the algorithm or to add sensor redundancy.

Simulation-based fault injection contains remarkable benefits. For instance, it allows high observability and controllability of the experiments without corrupting the original design.

The Sabotage framework is the responsible to automatically inject faults into the system and compute the results. First, the Workload Generator creates the functional inputs to be applied to the system. More specifically, it is the responsible of the following subtasks:

- selecting the system model under test, in this case, the sensor-based plausibility check;
- choosing the operational scenario from an and environment scenario library (different inputs will be chosen to evaluate the algorithm); and
- configuring the fault injection experiments. This includes creating the fault list and deciding the read-out or observation points (signal monitors).

Second, we configure the Fault Injector. The fault list is used to produce a Faulty system only in terms of reproducible and prearranged fault models by including saboteur blocks (S-functions). Fault models are characterised by a type (e.g. *frozen, stuckat0, delay, invert, oscillation* or *random*), target location, injection time triggering, and duration. In order to create a Faulty System, the Fault Injector injects an additional saboteur model block per fault entry from the Fault List. Moreover, the injected block is fulfilled with information coming from a fault model template library. Saboteurs are extra components added as part of the model-based design for the sole purpose of Faulty injection experiments.

After performing the configuration of the fault injection scenarios and creating the required amount of Faulty systems, the Monitor invokes the simulator. It tracks the execution flow of the fault free (Golden) system and Faulty simulations. The Monitor compares Golden and Faulty SMUT (switch matrix under test) results by the data analysis activity. The pass/fail criterion of the tests, which was established by the designer. This pass/fail criterion will help the engineer to adjust and/or modify the plausibility check algorithm if it is necessary.

To conclude, the development of all the above mentioned activities related to the Sabotage tool will be carried out and documented in detail within the deliverable D5.3 [2].

5.3 Technical Specifications

In the following sections we describe how some methodologies and tools described in Chapter 3 and Chapter 7 respectively have been applied in the technical specifications of the Connected Car use case, in particular those related to Safety Analysis, Security Analysis, Trade-off analysis, Requirements engineering and Security/Safety by design.

5.3.1 Safety Analysis

Following the ISO 26262 standard guidelines, we have applied the HARA methodology to identify possible hazards in the Connected Car use case. The results of the resulting HARA were illustrated in the deliverable D5.1[1]. The ISO 26262 highly recommends applying methodologies such as FMEA, which points out potential failures to identify possible failure causes with the aim of reducing or removing the hazards impact.

In the Platooning scenario, potential risks will be evaluated in a new platooning design, thus, a Design FMEA has been developed. A DFMEA should begin with the development of information to understand the system, subsystem or component being analysed and the definition of their functional requirements and characteristics. To do that, a block diagram and a functional requirements list is recommended.

The main functional requirements of the Platooning VeloxCars for the Connected Car use case are as follows:

- Each vehicle’s camera should get an image each 10 milliseconds.
- Each car should detect the two continuous white lines that delimit the lane.
- Each vehicle should be kept on the lane.
- Each vehicle should calculate a trajectory of 10 points to keep on the lane.
- The leader vehicle shall drive to the established speed.

- The follower vehicles shall drive to the speed calculated by the CACC.
- Each follower vehicle should be connected with the leader via WIFI.
- Each vehicle should exchange messages with heart-beat data and sensoring data with the others every 10 milliseconds.
- Each follower vehicle should maintain a safety distance with the previous car.
- The leader should stop in case of detecting obstacles in the trajectory.

The comprehensive DFMEA table has been included in the Chapter 11, as an annex of this document. At this moment, an impact analysis has been done to identify potential failure modes in the platooning system. However, the actions to reduce and/or avoid these failures have not been implemented yet, therefore, the last columns of the DFMEA table related to that implementation are still empty. They will be reported on the deliverable D5.3 [2] after the recommended actions have been applied.

5.3.2 Security Analysis

5.3.2.1 Goal Oriented Analysis of the Firewall Reconfiguration and Update Scenario

Following the KAOS goal-oriented requirements engineering methodology described in Section 3.2.3 a partial analysis of the connected Car vertical has been made.

Figure 39 describes some of the platooning goals. One of the important goals of a platooning system is to maintain a safe distance between vehicles in a platoon. This goal is decomposed into three different cases: maintaining a safe distance while joining, leaving and being a member of a platoon. While in a platoon, speed needs to be adapted in time to maintain a safe distance with vehicles in front and back using braking and acceleration actions.

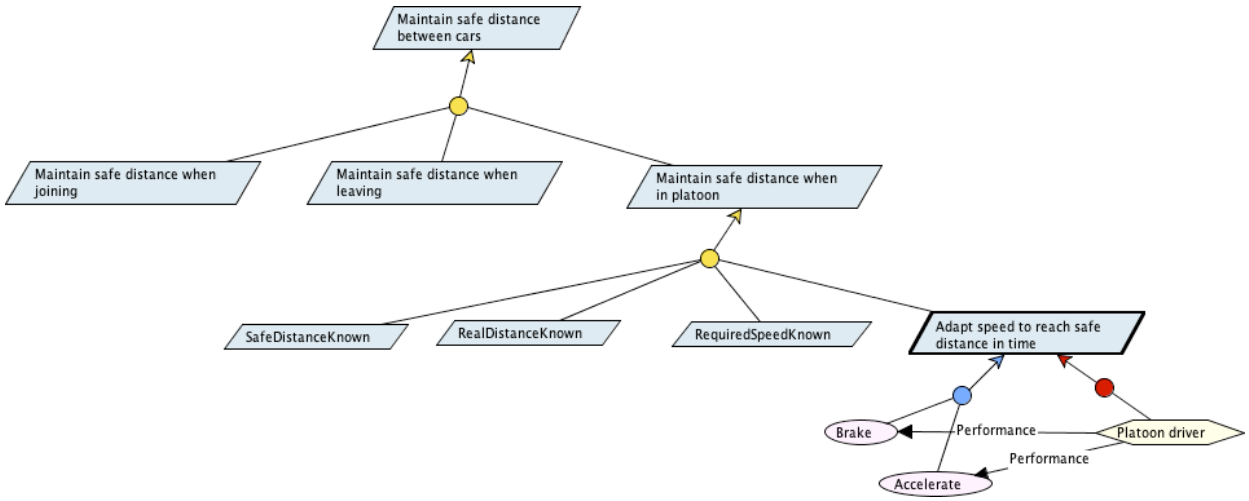

Figure 39: High-level platooning goals

Attackers may have as objective to achieve an unsafe distance between vehicles in a platoon to provoke an accident. Figure 40 shows an attack tree on the goal “Maintain safe distance when in platoon”, where false information about the speed and distance with respect to other vehicles is injected by an attacker.

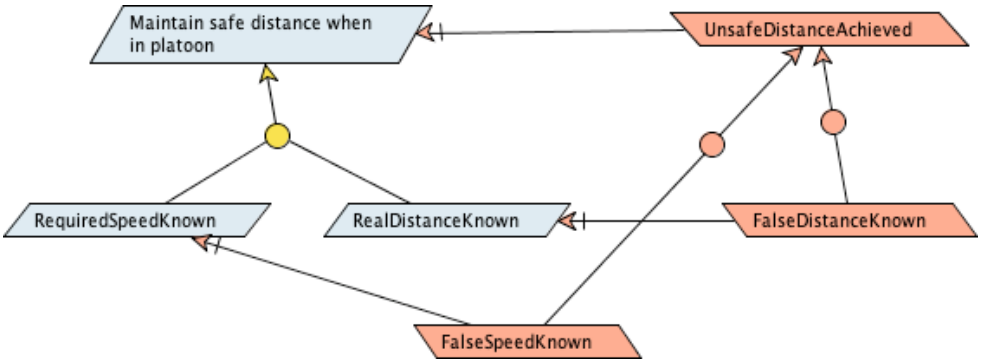

Figure 40: Fragment of high-level obstacles to platooning

The two automotive scenarios defined in Section 5.2.2 are analysed below.

Analysis of Firewall Reconfiguration Scenario

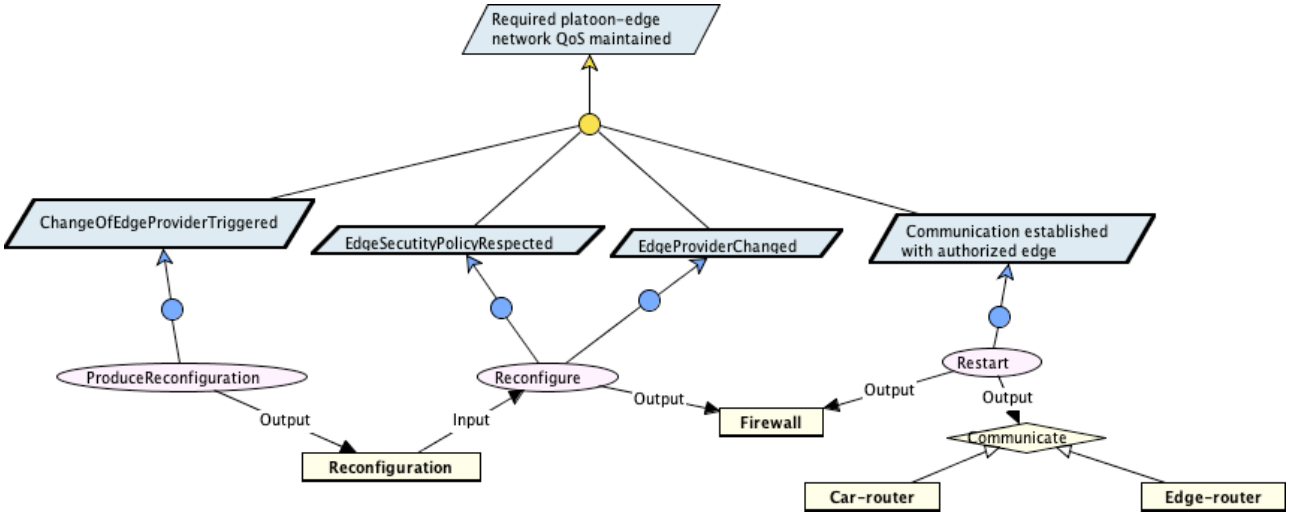

Figure 41: Firewall reconfiguration main goals and operations

In the V2I firewall reconfiguration scenario platoons communicate with the traffic control centres via edge clouds distributed along the road network. As the platoon progresses it must change edge clouds to get the best network latency available. Figure 41 shows the goal model for maintaining platoon to edge communication QoS when the platoon is moving. The high-level latency objective is decomposed into the following sub-goals:

- ChangeOfEdgeProviderTriggered: when a new edge provider that provides better QoS is identified, then the process of changing edge providers must be started.
- EdgeSecurityPolicyRespected: each edge provider has his own security policy, that must be propagated to the platoon vehicles to allow them to communicate with the new edge, including firewall reconfigurations.
- EdgeProviderChanged: Once the new security policy is known it must be communicated to the platoon vehicles so that firewalls can be reconfigured.
- CommunicationEstablishedWithAuthorisedEdge: Once firewalls are reconfigured with the security policies for the new edge provider, vehicle to edge communication can resume.

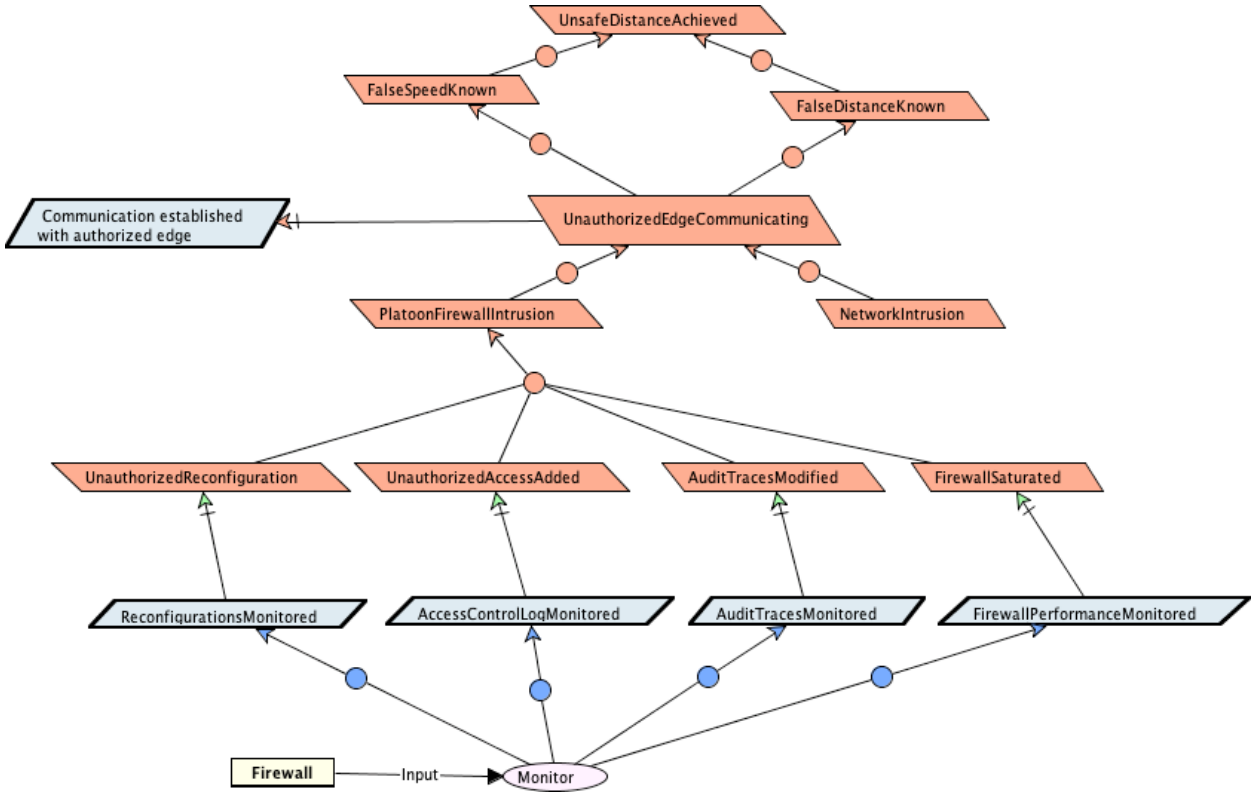

Figure 42: Firewall reconfiguration main obstacles

Figure 42 shows an attack tree on the goal “CommunicationEstablishedWithAuthorizedEdge”. It details several threats to firewalls if the firewall is compromised. This includes unauthorized reconfigurations of the firewall allowing providing unauthorized accesses, modifying audit traces, or saturating the firewall (DDoS). The figure also shows countermeasures to these threats that rely on monitoring of the firewall configuration.

Analysis of Firewall update Scenario

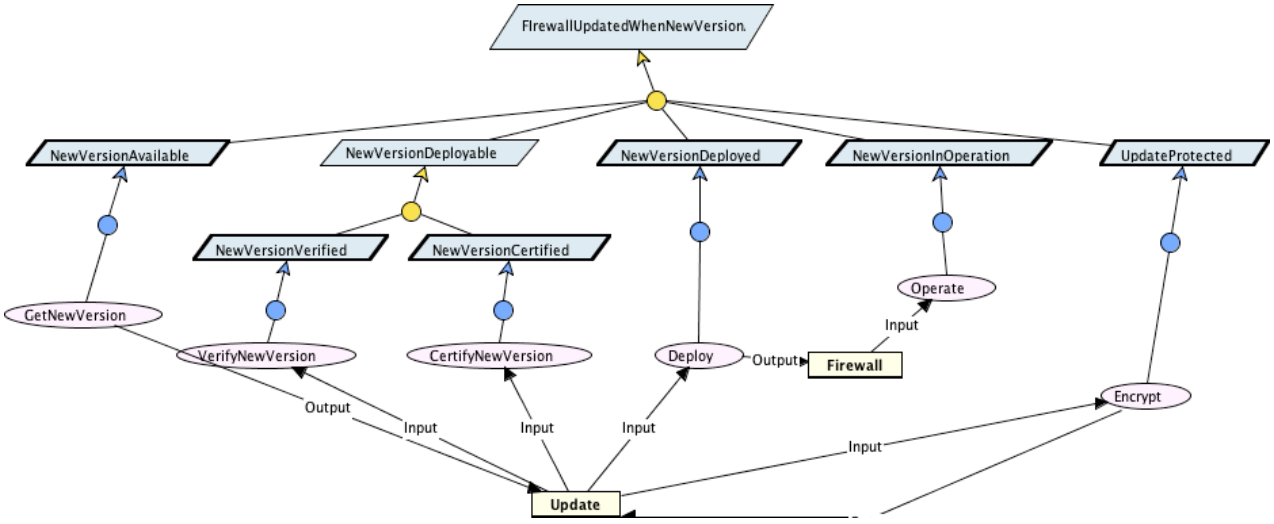

Figure 43: Firewall update main goals

In the V2I firewall update scenario, a new version of the firewall is available and needs to be deployed on customer vehicles. The above goal model shows the update process:

- NewVersionAvailable: the availability of a new version of the firewall triggers an update process.

- NewVersionVerified: the new version of the firewall is verified with respect to firewall requirements, and especially any certified requirements.
- NewVersionCertified: if some certified requirements are impacted then the new version must be re-certified
- NewVersionDeployed: once the new firewall version has been certified if necessary, it can be deployed on the target vehicles.
- NewVersionInOperation: once new firewall versions have been deployed, they can be put into operation
- UpdateProtected: the firewall updates must be protected during the whole process.

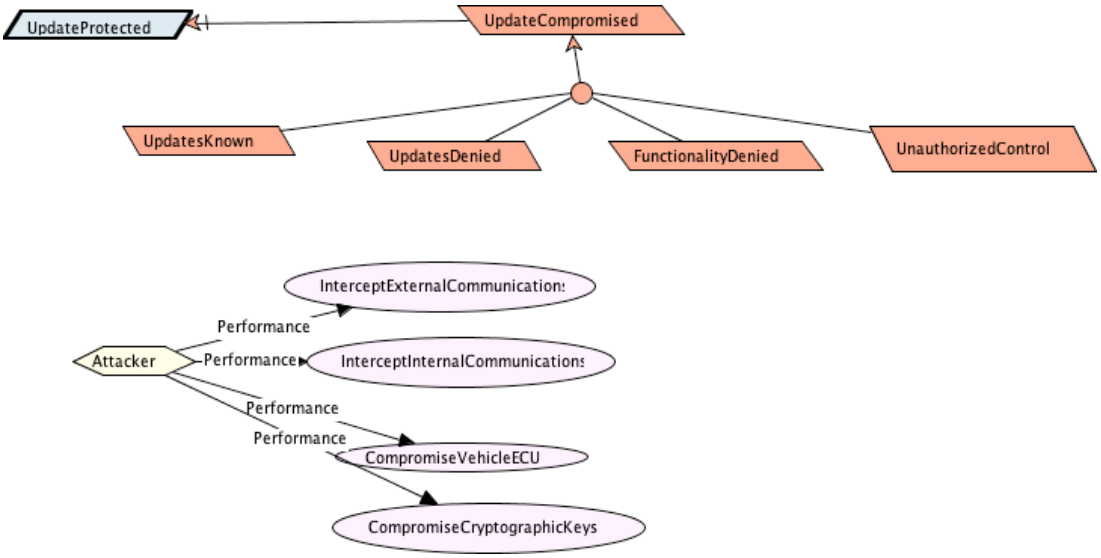

Figure 44: Firewall update main obstacles and attacker capabilities

Figure 44 shows an attack tree on the goal “UpdateProtected”. The attack tree lists some attacks on firewall updated:

- UpdatesKnown: Attackers gain unauthorized access to software updates in order to reverse-engineer software firmware and steal intellectual property from the vehicle manufacturer.
- UpdatesDenied: Attackers aim ro prevent updates so that can exploit existing vulnerabilities.
- FunctionalityDenied: By compromising updates attackers aim to prevent firewalls from functioning correctly.
- UnauthorizedControl: By compromisiung updates, attackers want to modify platoon behaviour.

5.3.3 Trade-off Analysis

Our vision is to build an incremental development process for system safety and security assurance cases using automated methods that incorporate safety and security reasoning principles. We provide safety reasoning principles with safety patterns used during the definition of system architecture for embedded systems. We specify these principles using logic and logic programming as they are suitable frameworks for the specification of reasoning principles as knowledge bases and using them for automated reasoning [70].

Our main contributions are threefold:

- **Domain-Specific Language (DSL):** We propose a DSL for safety reasoning with safety patterns. Our DSL includes (1) architectural elements, both functional components and

logical communication channels; (2) safety hazards including guidewords used in typical analysis, *e.g.*, erroneous or loss of function; and (3) a number of safety patterns including n-version programming, safety monitors, and watchdogs.

- **Reasoning Principles:** We specify key reasoning principles for determining when a hazard can be controlled or not, including reasoning principles used to decide when a safety pattern can be used to control a hazard. These reasoning principles are specified as Disjunctive Logic Programs [71] based on the DSL proposed.
- **Automation**: We illustrate the increased automation enabled by the specified reasoning principles using the logic programming engine DLV [76]. Our machinery enables two types of automated reasoning: (1) Controllability: which hazards can be controlled by the given deployed safety patterns and which hazards cannot be controlled; (2) Safety Pattern Recommendation: which safety patterns can be used and where exactly they should be deployed to control hazards that have not yet been controlled.

We validate our machinery with two examples of safety-critical embedded systems taken from the connected car example. The first example is an *Adaptive Cruise Control* (ACC) system installed in a vehicle to adapt its speed in an automated fashion without crashing into objects in front and at the same time trying to maintain a given speed. The second example is a *Battery Management System [4]* responsible for ensuring that a vehicle battery is charged without risking it to explode by, *e.g.*, overheating. Our machinery infers a number of possible solutions involving different safety patterns that can be used to control identified hazards.

While the details of our approach can be found in our papers [74][75], we illustrate with the Battery Management System (BMS) the types of trade off analysis reasoning that can be performed with our machinery.

Consider the BMS system architecture depicted in Figure 45 responsible for controlling a rechargeable electric car battery [76].

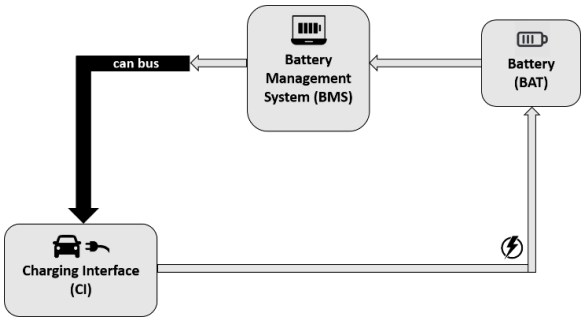

Figure 45: Battery Management System (BMS) functional architecture

The BMS is a critical system as harm, *e.g.*, battery explosions, may occur if it does not compute the charging state of the battery correctly. The BMS's main functions are the charging interface (CI) that represents the interface at the charging car station. This interface is triggered while recharging the battery (BAT) of the car. BMS receives relevant information (*e.g.*, voltage and temperature values) from BAT so that it can compute the charging state of BAT. Depending on the state of BAT, BMS sends signals of activation or deactivation of the external changer to CI. These signals are sent though a can bus.

To address the safety of the BMS, safety analyses are carried out to determine main hazards. The main hazard is:

HM: The BAT is overcharged leading to its explosion.

We identify one erroneous hazard H1 that may lead to HM. The word erroneous is used by safety engineers to describe hazards: erroneous is used when a function is working but not correctly.

- H1: The CI sends charging signals when BAT is fully charged

The following faults may lead to H1:

- H1.1: Erroneous BMS: The BMS sends wrong signals to CI
- H1.2: Erroneous CAN: The can bus sends wrong signals to CI

Moreover, a main threat to the BMS is that an attacker uses the public interfaces of the BMS, namely the CI, to trigger a battery explosion. For example, since the CI has access to the can bus, an attacker may inject a message in the can bus through the CI to "start charging". Since the can bus delivers messages to all connected components, including the CI, the CI receives back this message and starts/continues to recharge, although it may have received previously a message from the BMS to stop doing so.

To control the hazards above and mitigate the attack just mentioned, safety and security engineers place safety and security architectural patterns, such as safety monitors and firewalls. The placement of such patterns can be inferred by our machinery. In particular, it infers the placement of patterns, a safety monitor and a firewall as depicted in Figure 46.

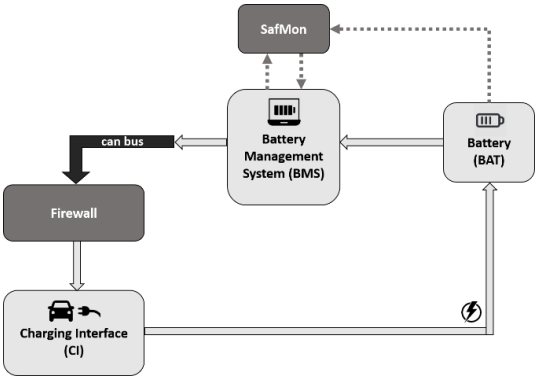

Figure 46: BMS architecture with a safety monitor and a firewall

However, our machinery is also able to infer that the main hazard HM is not yet controlled. This is because the firewall placed by the security reasoning has an impact on safety. If the firewall wrongly omits a message sent by the BMS to stop recharging, the hazard HM may still occur (with an unreasonable probability).

We can use our machinery to infer ways to correct this problem by placing additional patterns. Our machinery infers the architecture depicted in Figure 47 that adds a Voter. The Voter serves the purpose of creating a redundant path in the architecture that is used to ensure that the message from the BMS to stop the charging of the battery is enforced.

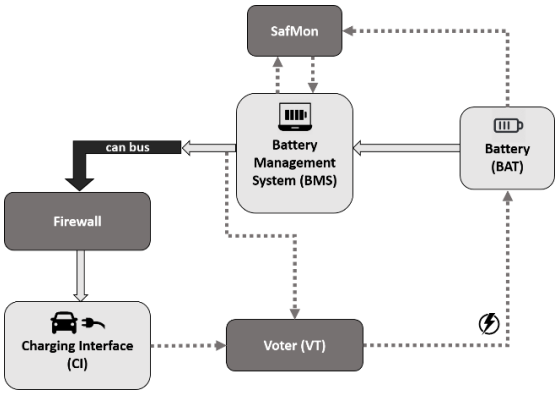

Figure 47: BMS architecture with an additional Voter

5.3.4 Requirements Engineering

In the context of the Platooning scenario (Vertical 1), we have defined the safety and security requirements of the scenario in the same Common Criteria protection profile. A protection profile (PP) is an implementation-independent statement of security needs for a TOE type [17].

The TOE, Safety and Security Platooning Management Module (**SafSecPMM**), is used to ensure the safe and secure operation of vehicle platoons, e.g., to avoid vehicle collisions leading to human and material damages. The TOE has an interface towards the Vehicle Communication Device (VCS), the Hardware Security Modules (HSM), if HSM is available and is not directly integrated in VCS, and the Vehicle Control Module (VCM).

The TOE receives data from the VCS, using HSM to decrypt any encrypted message, or to check the integrity of messages. The TOE also uses sensing data available in the VCM, such as information about the distance to any object, speed and localization. The sensor information from the VCM may be signed by HSM to guarantee communication integrity.

Moreover, based on the data collected, the TOE communicates necessary data to other vehicles and stationary deployments through the VCS. Communication may be signed/encrypted using HSM. The TOE also sends commands to the VCM actuators, to guarantee the safe and secure operation of the vehicle and the platoon, such as commands setting the speed and the direction vector. Figure 48 illustrates the interface of the TOE with the VCS, HMS and VCM.

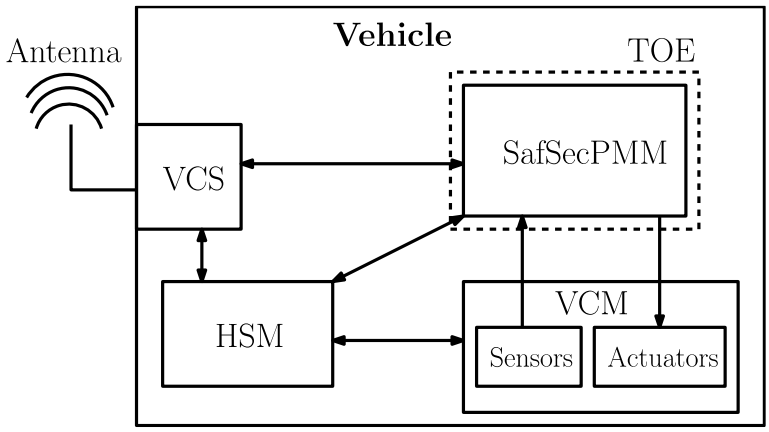

Figure 48: TOE Interfaces

Chapter 12 (Annex B) includes the full contents of the PP for a Safety and Security Platooning Management Module (SafSecPMM). This module addresses cyber-attacks on a formed platoon that exploit the communication and sensing interfaces of a vehicle by sending incorrect information about the state of the world, e.g., wrong speed, position of vehicles in the platoon. Such attacks can lead to honest vehicles to potentially make wrong decisions that may affect the safety of passengers, e.g., accelerate when it should not, thus placing the platoon in an unsafe state.

The PP of the SafSecPMM incorporates security countermeasures and other security features to increase the robustness of the platooning behaviour, provides accountability information for this behaviour, and contains security measures to protect its own assets.

As platooning security is a relatively new subject, the PP document shall be considered as a “living document” that shall be extended in future version to consider other types of vulnerabilities and threats, specifying other requirements to SafSecPMM to address these vulnerabilities and threats.

In the next sections we provide an overview of the main contents of the PP. For more details on the document, we refer the interested reader to Chapter 12.

5.3.4.1 Threats

Table 6 shows the threats have been identified against the TOE.

Name	Threat against TOE
Communication Data Spoofing (T.COM_SPF)	The attacker may inject data in the communication channel by, for example, carrying out replay attacks. For another example, if the attacker possesses valid secret encryption keys, then Sybil attacks can cause the vehicle to infer that there is a vehicle that does not actually exists.

Name	Threat against TOE
VCM Data Spoofing (T.SEN_SPF)	The attacker can carry environmental attacks that may confuse sensors causing the vehicle to perform incorrectly, e.g., accelerate the vehicle placing its passengers in danger.
Communication DoS (T.COM_DOS)	The attacker can carry out denial of service attacks on the communication channels used.
VCM DoS (T.VCM_DOS)	The attacker may deny the service of a sensor by, for example, covering the lenses of a camera/lidar used to infer objects.
SafSecPMM Software Tamper (T.SW_TAMPER)	The attacker may tamper the software installed in the TOE causing the attacker to control the vehicle.
Exploit Service Defects (T.SW_DEFECTS)	The attacker may take advantage of a TOE malfunction/defect of the Platooning Management Service.
Tamper Accountability Data (T.ACC_TAMPER)	The attacker may tamper the accountability data, thus avoiding attacks that have being carried to be accounted for.
Attack Software Update Mechanism (T.SW_UPDATE)	The attacker may attack the mechanisms used by vehicles to update the TOE software to install malwares or other malicious software.
Attack system access (T.ECU_ACCESS)	The attacker may get unauthorized access to the vehicle ECU via network getting the control of the vehicle, e.g., Brute-force password attack.

Table 6: Threats against TOE

5.3.4.2 Security Objectives

Table 7 shows the set of security objectives that the TOE should achieve in order to solve its part of the problem.

Security Objective	Description
OT.VCS_DATA	The TOE shall provide periodically to the VCS data about the vehicle, e.g., speed, direction, position. This data shall reflect the actual state of the vehicle.
OT.INCORRECT_VCM_DATA	The TOE shall be able to detect when data incoming from the VCM is incorrect, i.e., it differs from the actual state of the world.
OT.INCORRECT_VCS_DATA	The TOE shall be able to detect when data incoming from the VCS is incorrect, i.e., it differs from the actual state of the world.
OT.SENSOR_FAIL	The TOE shall be able to guarantee the safety of the vehicle even if a sensor fails, either due to an attack or due to some component failure.
OT.COMM_FAIL	The TOE shall be able to guarantee the safety of the vehicle even if a communication channel fails, either due to a DoS attack or due to some component failure, e.g., by going to a safe-mode and informing the driver.
OT.VCM_DATA	The TOE shall provide periodically to the VCM data to be consumed by the VCM actuators. The data shall be used to ensure the safety of the vehicle, e.g., keep the platoon lane and a safe distance to all other vehicles.
OT.TOE_SELF-PROTECTION	The TOE shall be able to protect itself and its assets from manipulation including physical and software tampering. Moreover, the following is assumed by the TOE: • Messages outgoing from the VCS shall be digitally signed by the HSM. • The digital certificate of messages in incoming flows from the VCS shall be checked by the HSM.

OT.ACCOUNTABILITY	The TOE shall provide accountability for all the decisions made that affect the behaviour of the vehicle. The TOE shall provide proof of the integrity and origin in order any message stored in the memory can be said to be genuine with high confidence.

Table 7: Security Objectives for the TOE

Table 8 shows the security objectives for Operational Environment, i.e. the set of statements describing the goals that the operational environment should achieve.

Security Objective	Description
OE.SECURE_COMM	The TOE operational environment shall implement protections for the integrity, authenticity and confidentiality of the data exchanged between vehicles and between vehicles and stationary deployments.
OE.CORRECT_IMP	The TOE operational environment shall ensure that the TOE software does not have defects, such as, software bugs that can be exploited by the attacker, e.g., to carry-out buffer overflows, badly configured access control.
OE.INTEGRATION	Appropriate technical and/or organisational security measures shall be in place during platform integration phase.
OE.TOE_ACCESS	The TOE environment shall implement security measures to ensure that the TOE is only accessible from the VCS and the VCM by deploying measures for authenticity and access control.
OE.VCM_SEN_FAIL	The VCM must be able to detect when a sensor has failed and inform the sensor fail to the TOE whenever this occurs.
OE.VCS_CMM_FAIL	The VCS must be able to detect when a communication link to other vehicles/infrastructure stations fails and inform which link failed to the TOE whenever this occurs.

Table 8: Security Objectives for the Operational Environment

5.3.4.3 Security Functional Requirements

Table 9 shows a summary of the SFRs that have been elicited for the TOE. The development of Coonected Car vertical must comply with all of them. For more details on the definition of the security requirements, we refer the interested reader to Chapter 12.

Functional Class	Security Functional Requirements
Platoon Management Module (PMM)	**Information Flow (PMM_IF):** PMM_IF.1.1 Maintain heart-beat data (vehicle identifier, speed, direction, geo-position, timestamp) to VCS PMM_IF.2.1 Maintain heart-beat data from VCS PMM_IF.3.1 Maintain incoming emergency brake PMM_IF.4.1 Maintain outgoing emergency brake PMM_IF.5.1 Maintain data from VCM PMM_IF.6.1 Maintain data to VCM
	Plausibility Checks (PMM_PC): PMM_PC.1.1 Data passes all VCS plausibility checks PMM_PC.2.1 Data passes all VCM plausibility checks PMM_PC.3.1 Inform on failed plausibility checks
	VCS History-based Plausibility Checks (PMM_VCS-HPC): PMM_VCS-HPC.1.1 Maintain heart-beat data history PMM_VCS-HPC.2.1 Heart-beat message consistent to the history PMM_VCS-HPC.3.1 Emergency brake consistent to the history

	VCS Sensor-based Plausibility Checks (PMM_VCS-SPC): PMM_VCS-SPC.1.1 Maintain distances history PMM_VCS_SPC.2.1 VCS message consistent to distances history PMM_VCS_SPC.3.1 Emergency brake consistent to distances history
	VCS Timestamp-based Plausibility Checks (PMM_VCS-TPC): PMM_VCS-TPC.1.1 Consult the TOE vehicle internal clock PMM_VCS-TPC.2.1 Message freshness
	VCM History-based Plausibility Checks (PMM_VCM-HPC): PMM_VCM-HPC.1.1 Maintain sensor data history PMM_VCM-HPC.2.1 Sensor message consistent to the history
	VCM Timestamp-based Plausibility Checks (PMM_VCM-TPC): PMM_VCM-TPC.1.1 Consult the TOE vehicle internal clock PMM_VCM-TPC.2.1 Message freshness
Protection of the TSF (FPT)	**Availability of exported TSF data (FPT_ITA):** FPT_ITA.1.1 Inter-TSF availability within a defined availability metric
	Confidentiality of exported TSF data (FPT_ITC) FPT_ITC.1.1 Inter-TSF confidentiality during transmission
	Integrity of exported TSF data (FPT_ITI) FPT_ITI.1.1 Inter-TSF detection of modification FPT_ITI.1.2 Inter-TSF verify integrity
	Fail secure (FPT_FLS) FPT_FLS.1.1 Failure with preservation of secure state
Communication (FCO)	**Non-repudiation of origin (FCO_NRO):** FCO_NRO.1.1 Generate evidence of the origin of the data FCO_NRO.1.2 Relate the evidence of origin with the originator FCO_NRO.1.3 Verify the origin of the data
Identification and Authentication (FIA)	**User authentication (FIA_UAU)** FIA_UAU.2.1 User authentication before any action FIA_UAU.3.1 Detect use of authentication data that has been forged FIA_UAU.3.2 Detect use of authentication data that has been copied FIA_UAU.6.1 Re-authenticating
	User identification (FIA_UID) FIA_UID.1.1 No action allow before the user is identified. FIA_UID.1.2 Successful user identification before any action
Resource Utilization (FRU)	**Fault tolerance (FRU_FLT)** FRU_FLT.1.1 Degraded fault tolerance

Table 9: TOE Security Functional Requirements

5.3.4.4 Security Assurance Requirements

The Security Assurance Requirements (SARs) are a description in a standardized language of how the TOE is to be evaluated. Table 10 shows the security assurance requirements that have been chosen for the TOE. They comprise the ATE and the AVA classes.

Assurance Class	Assurance Component Name	Rationale
ATE:Tests	Analysis of coverage	ATE_COV.1
	Testing: basic design	ATE_DPT.1
	Functional testing	ATE_FUN.1

Assurance Class	Assurance Component Name	Rationale
	Independent testing – sample	ATE_IND.1
AVA: Vulnerability assessment	Focused vulnerability analysis	AVA_VAN.1

Table 10: Security Assurance Requirements

5.3.5 Security/Safety by Design

We have applied the methodology described in Section 3.2.6 for the Basic Scenario of the Connected Car Vertical enabling the security and safety by design of platoon scenarios. We detail in the following section the formal specification framework enabling engineers to use program precise mathematical models about the behaviour of vehicles in platoons and evaluate whether they possess enough mechanisms to perform safely, even in the presence of intruders.

5.3.5.1 Formal Specification Framework for Vehicle Platooning using C-ACC

In this section we specify a formal model that can be used for the safety and security analysis of scenarios considered for Vertical 1. Our framework is constructed on the soft-agents model [76] which is rewriting a logic framework for the specification and verification of (autonomous) cyber-physical systems. The framework, which can be found at [76], is implemented in the rewriting logic language Maude [77]. It provides the general machinery (data-structures, functions, sorts) for the specification of the behaviour of agents, e.g., agent capabilities and effects of actions. The semantics of how the system evolves is specified by a small number of rewrite rules defined in terms of the general machinery.

Figure 49 depicts the general architecture of a soft-agent, or simply agent. An agent has its own local knowledge base that contains, e.g., its current perceived speed, position, and direction of the other agents. Further data may be obtained by sensing the environment or by sharing of information between agents through communication channels. Using its local knowledge base, the agent decides which action (α) to perform according to its different concerns specified as a soft constraint (optimization) problem. For example, if the distance to the vehicle in front is too great, the fuel consumption concern kicks in and attempts to reduce it by accelerating. Similarly, if the distance is dangerously short, then the safety concern kicks in and attempts to increase it by decelerating. As soft constraints subsume other constraint systems, e.g., classical, fuzzy and probabilistic, it is possible to formally specify a wide range of decision algorithms.

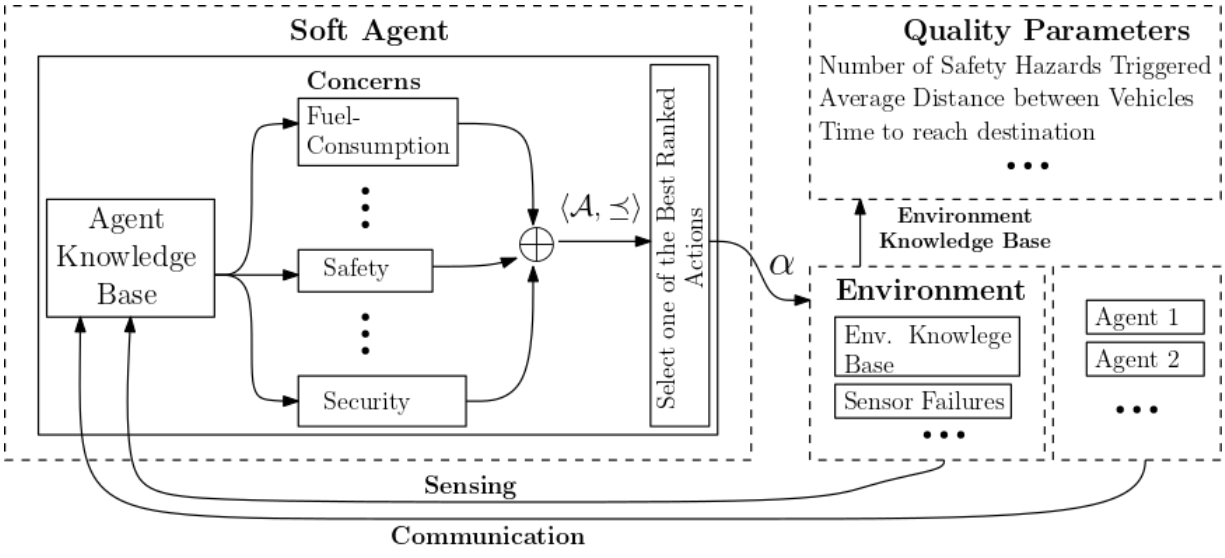

Figure 49: Soft-Agent Architecture

We instantiated the general framework (data structures, sorts, types, soft constraints) provided by the Soft Agents framework for specifying platoon scenarios, enabling their formal verification. While the complete implementation can be found at [78], we describe some of this machinery below.

Knowledge Base: Vehicles have a local knowledge base (*lkb*). It represents the vehicle's view of the world, e.g., the speed and position of itself and of the other vehicles. Formally, a vehicle knowledge base is composed by a set of grounded facts, *p*, i.e., facts not containing variable symbols, of the form *p*, or associated with a timestamp, *p@t*, where *t* is natural number. We list the main facts below. We assume that each vehicle has a unique identifier written *id*.

- *clock(t)* denotes that the current time is *t*.
- *atloc(id,pos) @ t* denotes that the vehicle *id* has at time *t* the position of value *pos*. We assume that vehicles navigate on a straight road. Therefore, *pos* is a value representing the position on this road.
- *speed(id,spd) @ t* denotes that the vehicle *id* has at time *t* the speed of value *spd*.
- *maxAcc(id,acc)* denotes that the vehicle *id* can accelerate (and for simplicity also decelerate) at any time with value *acc*.
- *platoon(idL,[id1,...,idn]) @ t* denotes that at time *t*, the platoon led by *idL* has the sequence of follower vehicles *id1,...,idn*.
- *mode(id,md) @ t* denotes that the vehicle *id* at time *t* is in mode *md* which include:
    - *nonplatoon* when all the vehicle's platooning functionalities are not active, i.e., the vehicle is driven by a human driver
    - *leading()* when the vehicle leads a platoon
    - *following(idL)* when id is following the platoon led by *idL*
    - *emergency()* when *id* is in emergency brake mode
    - *fuseRear(idL,idB)* when *id* is in the process of joining platoon led by *idL* and shall join be behind vehicle *idB*.

Sensors: A vehicle is equipped with three sensors *locS*, *speedS* and *gapS*. They measure, respectively, the vehicle's location, speed and the gap to the vehicle immediately ahead. As we illustrate below, at each tick, vehicles use these sensors to query the environment knowledge base and update the vehicle's local knowledge base. While it is not the focus of this work, it is possible to evaluate the robustness of agents with respect to sensor faults as described in [79].

Communication Channels and Protocols: We assume that vehicles may communicate using peer-to-peer connections or by broadcasting messages. Based on this assumption, we have implemented several protocols for platooning including:

- Heartbeat from Follower to Leader (HFL): A follower vehicle sends periodically a (time-stamped) message to the leader with information such as its current speed and position.
- Heartbeat from Leader to Follower (HLF): The platoon leader sends periodically a message to each follower with information of all vehicles in the platoon such as their speeds and positions.
- Emergency Brake: Any vehicle in the platoon may broadcast an emergency brake message informing that it is activating its emergency brakes.
- Heartbeat from Joining Vehicle to Leader (HJL): A vehicle that wants to join a platoon sends a heartbeat to the platoon leader, such as its current position and speed.
- Heartbeat from Leader to Joining Vehicle (HLJ): The platoon leader sends to the vehicle that is joining the platoon information, such as the position and speed of the last vehicle in the platoon.

Actions: Vehicles decide to accelerate or decelerate. Since there may be infinitely many possibilities of acceptable speeds (for safety and fuel efficiency), we abstract actions by using facts of the form

act(id,vmin,vmax) denoting a set of actions of changing id's speed to values between *vmin* and *vmax*. Actions are evaluated with a value that is the result of a soft constraint problem specification described next.

Soft Constraints: The evaluation of possible actions is done by taking into account the vehicle's concerns specified as a soft constraint problem. To evaluate our verification machinery, we implemented a strategy that depends on the vehicle's mode.

- When in *following* mode, a vehicle has two main concerns: *Fuel-Saving* and *Safety*. The former attempts to close the gap to the vehicle immediately in front, while the latter attempts to keep a safe distance to the vehicle immediately in front. These are specified by the knowledge items *safe* and *fuel*. Our machinery uses these two parameters to determine which (set of) actions are the most highly ranked. This is accomplished by attempting to satisfy both concerns, safety and fuel-saving. If this is not possible, then safety is given priority over fuel-saving.
- When in *emergency* mode, the vehicle has only the concern of stopping the vehicle.

Intruder Model: An intruder can impersonate an honest vehicle, listening to messages, injecting messages, and blocking message from communication channels between vehicles. These capabilities enable us to carry out similar verification done for safety, but now considering a malicious intruder, i.e. an attacker entering into the system. Our intruder model is similar to [80], for the security verification of Industry 4.0 applications, in that the intruder model is parametrized by its capabilities. Here we consider two capabilities: injecting messages signed by honest participants and blocking specific messages from communication channels (see Figure 22):

- Message Injection (INJ): The intruder may choose at any moment of a system execution to inject the first message, *msg*, in its list of messages *msgList*. This results in the injection of *msg* to its destination in the system configuration system, and the list *msgList* is updated by deleting *msg*.
- Blocking (BLK): The vehicles in *ids* are jammed during the whole attack execution. This means that all outgoing messages of a vehicle *ids* are blocked.

Our model is parametric w.r.t. the intruder capabilities. It requires little effort to include other capabilities to the intruder model. For example, it is possible to add capabilities where the intruder tampers, i.e., modifies messages sent by vehicles; or periodically sends messages from a set of messages, instead of in a list; or only starts blocking a message after some particular time has elapsed.

We have applied this model for the verification of some of the vehicle platooning scenarios. These are detailed in D5.3 [2].

5.4 Assessment tools pipeline

For each demonstration scenario, we have identified the V-model steps (security/safety/certification) that will be covered by the continuous assessment pipeline, and which tool intervenes in each step. Figure 50 illustrates an example continuous assessment pipeline using CAPE tools in the context of the Connected Car vertical.

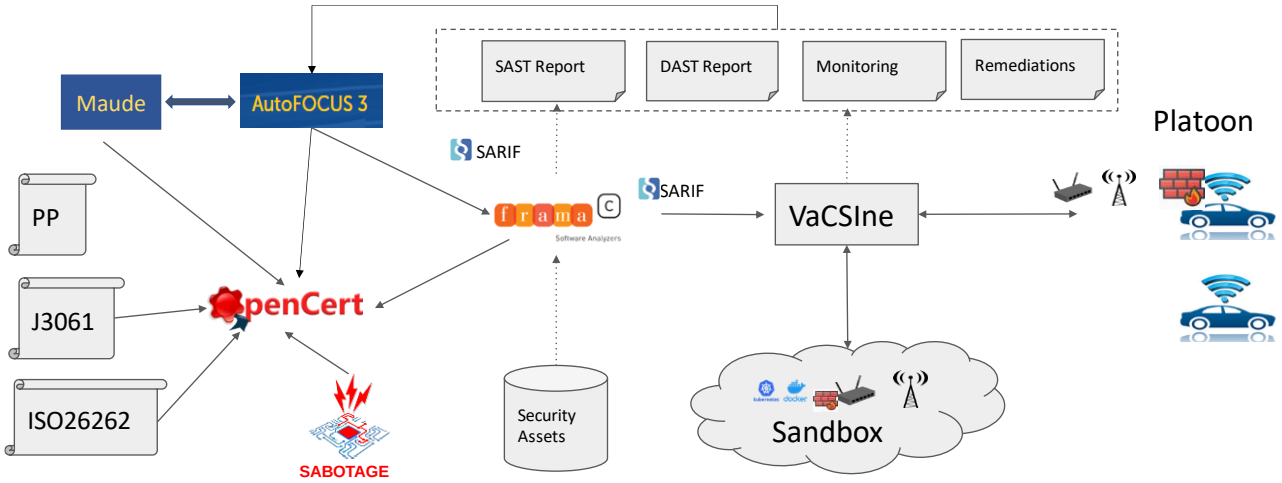

Figure 50: A sample CAPE continuous assessment process for the Connected Car vertical

The following tables describe the various assessment pipelines that have been developed in the context of the demonstration scenarios for the CACC vertical. They provide a summary of the integration between the various tools in each scenario by detailing the relevant V-Model step for each tool and the associated inputs and outputs.

Scenario title	**Connected Car Vertical**
Description	Evaluate the process, from security analysis, requirements to implementation and verification and validation, for increasing the security of vehicle platooning when assuming a malicious intruder that can manipulate the communication channels
V-Model step: SW design, SW development	• Tool name: **AutoFOCUS3** • Input: AF3 model (incl. logical architecture and behaviour) for the Platooning scenario • Output: C code for the architecture and behaviour specified in the AF3 model
V-Model step: Development, Unit testing	• Tool name: **Frama C** • Input: AF3 models of the Platooning scenario • Output: SARIF report for the analysed code
V-Model step: Verification and Validation	• Tool name: **Maude** • Input: AF3 model for the Platooning scenario. capabilities of intruder. • Output: Evidence supporting the security of the AF3 model with respect to the intruder model.
V-Model step: Technical safety concept and verification design	• Tool name: **Sabotage** • Input: Mathematical model in Matlab/Simulink (external Tool); Fault List. • Output: Report of the simulation results
V-Model step: All V-Model	• Tool name: **OpenCert** • Input: AF3 models of the Platooning scenario, SARIF report for the analysed code; Safety & Security standards (ISO26262, SAE J3061); Protection Profile; Report of the simulation results from Sabotage. • Output: Assurance Case, including Safety and Security arguments.

Table 11: Connected Car vertical pipeline

Scenario title	Connected Car Vertical - Scenario 2 - Firewall update
Description	Orchestration and validation of the adaptive security response to changing security requirements.
V-Model step: SW and HW component design, secure design	• Tool name: **SCAP workbench** • Input: Protection profile • Output: SCAP policy
V-Model step: Acceptance testing, function verification, CS assessment	• Tool name: **OpenSCAP Base** • Input: SCAP policy • Output: SCAP report
V-Model step: Operations, CS assessment	• Tool name - **VaCSIne** • Input: SCAP policy, SCAP report, operation logs • Output: Remediation orchestration logs

Table 12: Connected Car vertical, scenario 2 pipeline

Scenario title	Connected Car Vertical - Scenario 3 – Verification tooling
Description	Penetration testing tools and methods to perform vulnerability assessment of the Fortiss and Tecnalia Rovers.
V-Model step: Architecture and System Design, Security by design	• Tool name: Set of HW tools, scripts and exploration • Input: Output of AVA_VAN report (iteration loop) • Output: System Design, Security by Design (iteration loop)
V-Model step: SW and HW development, CS assessment	• Tool name: Set of HW tools, scripts and exploration • Input: Output of AVA_VAN report (iteration loop) • Output: System Design, Security by Design (iteration loop)
V-Model step: Acceptance testing, function verification, CS assessment	• Tool name: Set of HW tools, scripts and exploration • Input: Protection profile, AF3, Maude • Output: AVA_VAN report
V-Model step: Operations, CS assessment	• Tool name: Set of HW tools, scripts and exploration • Input: Protection profile, AF3, Maude • Output: AVA_VAN report

Table 13: Connected Car scenario vertical, scenario 3 pipeline

5.5 Adoptability

In the context of the Connected Car Platooning scenario, we have developed assets that will be made publicly available, and thus may be used by users such as industry partners. These assets are:

- **Protection profile**: We have written a protection profile document for a safety and security platooning management module. This document includes a list of requirements that shall be implemented by platoon members. This document can serve as a starting point for users interested in the safety and security of CACC platoons.

- **Model**: We have specified executable models for CACC platoons. These models include: (a) CACC functionalities, e.g., that maintain distance between vehicles that are both safe and fuel-efficient; (b) communication protocol between platoon members, i.e., how platoon members exchange messages between each other; and (c) defences based on plausibility checks to mitigate injection attacks. From our models, one can automatically generate C code using AutoFOCUS3.

In addition, our **labs** (FTS and TEC) may serve as transfer institutes for users. That is, we can offer technical consultation on all issues relating to the assets mentioned above, including consultation on how to adapt the elements specified in the model as well as on how to deploy the generated C code into rovers. We may also offer research consultation on such assets, e.g., future research directions on how to improve the assets developed in this project.

Chapter 6 e-Government Services Vertical Technical Specifications (Vertical 2)

6.1 Context and Background

The “Complex System Assessment Including Large Software and Open-Source Environments, Targeting e-Government Services” vertical (a.k.a. e-Government services vertical) has as goal to improve the cyber-security of the innovative authentication solutions based on the usage of the Italian national electronic identity card (CIE). The (cryptographic capabilities of the) CIE ensures a high level of assurance of the resulting authentication. This vertical leverages the collaboration in the context of a Joint Lab between Fondazione Bruno Kessler (one of the institutions part of the SPARTA partner CINI) and Instituto Poligrafico e Zecca dello Stato (IPZS), the Italian State Mint and Polygraphic Institute. Indeed, IPZS handles the production of the identity cards in Italy and the shipment of the CIE to the Municipalities.

As detailed in D5.1 [1], the CIE-based Italian Identification Scheme for accessing services envisages two mutual authentication use cases: a so-called “desktop” use case, in which the user uses his/her CIE with a workstation equipped with a RF smart card reader and the so-called “Middleware CIE”, and a “mobile” use case. As just mentioned, the desktop use case requires citizens to own an external RF smart card reader, bring it with them at every authentication, and install the middleware. To avoid this burden, another use case, called “hybrid”, has been recently proposed. In the “hybrid” use case, citizens use the CIE to authenticate themselves onto an online service from their personal computer’s browser, by using a “companion app” (say CIE ID APP) on their smartphone as a card reader. A QR-code shown in the browser allows the CIE ID APP to collect the authentication information from the browser. Given that we expect that the “desktop” use case based on the middleware is going to be progressively dismissed, being replaced by the “hybrid” use case based on the CIE ID APP, the role of the CIE ID APP will be more and more central. For this reason, in the context of SPARTA, we currently focus on the “mobile” use case, and we will evaluate in the next future whether the “desktop” use case deserves further security analyses.

The “mobile” use case, shown in the diagram in Figure 51, is based on the use of a smartphone to interact with the CIE (through the NFC interface) as an authentication tool to gain access to a service of the Public Administration. In detail, this identification scheme envisages that the user accesses a service provided by a service provider through the browser of his/her smartphone and selects the mode of access via CIE. When authenticating using the CIE, he/she is then redirected to the CIE ID APP, that performs the authentication mechanism through the CIE with the CIE ID SERVER component, hosted by the Italian Ministry of Interior.

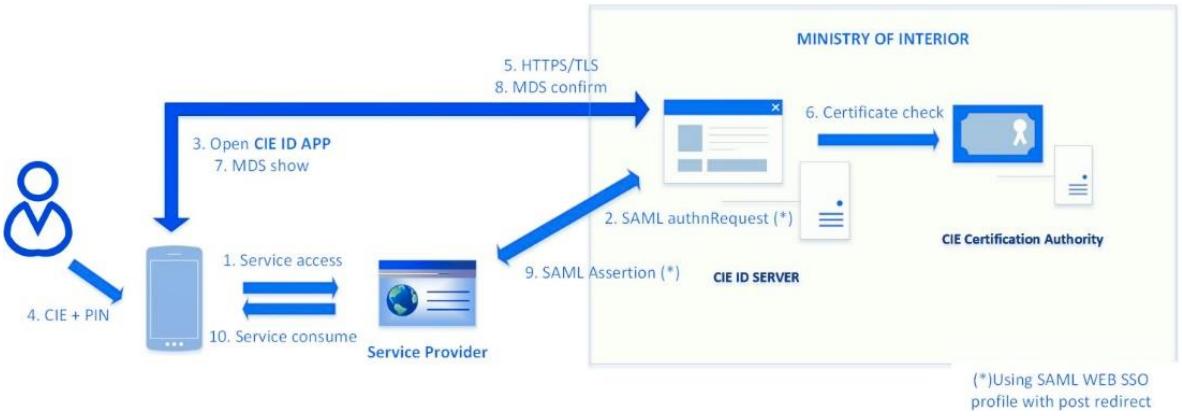

Figure 51: The mobile use case of Vertical 2

The detailed steps of the procedure shown in the diagram in Figure 51 are described below:

1. To be able to access the services of the Public Administration, the user needs to be authenticated. To this purpose, the service provider sends a SAML authentication request (through the construct <*AuthnRequest*>) to the component CIE ID SERVER.

2. The CIE ID SERVER component requests the user to use his/her CIE to authenticate him/herself and sends a notification to the mobile terminal which triggers the launch of the CIE ID APP.
3. By following the instructions shown on the app, the user presents the CIE to the smartphone's NFC reader and enters his/her PIN.
4. Having verified the correctness of the PIN, a secure HTTPS/TLS channel is created between the CIE ID APP and the component CIE ID SERVER.
5. From the secure session the latter verifies the validity of the digital certificate associated with the user by contacting the Authentication CA of the Ministry of the Interior and retrieves the minimal attributes relative to the user from the certificate.
6. On the CIE ID APP the user views the attributes that will be sent to the service provider.
7. The user authorises the transmission of the attributes displayed.
8. The component CIE ID SERVER redirects the user to the service provider by sending an assertion of successful authentication, including attributes, to the latter.
9. The service provider grants access to the service which takes place through the browser of the mobile terminal used in step 1.

6.2 Scenarios

The main goal of these demonstration scenarios is to show how the cutting-edge security analysis tools developed in the context of SPARTA and the novel paradigms for continuous security assessment (DevSecOps) in the context of task 5.3 will contribute to increase the overall security of the e-government service. The mobile use case includes several components. Each software component must be carefully implemented in order to avoid security issues. For the demonstration of vertical 2 we identified two main components, depicted in Figure 52.

- CIE ID App, and
- SAML IdP on the CIE ID SERVER.

Figure 52: Components in the scope of the demonstrations

In both demonstration scenarios two main actors are involved: the developer of each component and the security analyst.

The developer is expected to push a commit on the code repository that contains the source of the component (either the CIE ID APP or the CIE ID SERVER) and is connected to the DevSecOps pipeline. This operation must trigger the proper CAPE assessment tools involved in the scenario and return the feedback concerning the security assessment to the developer. This feedback might be different according to the peculiarities of each tool. For instance, the developer could find the list of security vulnerabilities in the issue tracker or receive a complete report.

At the same time, the security analyst should be notified about the vulnerabilities introduced by the developer, being able to access the security report of each CAPE assessment tool. Some tools

should provide a GUI which allows the security analyst to properly analyse the issues and receive support in their mitigation. In some cases, security issues in the scenarios are not introduced by a change in the source code, but they are simply due to the discovery of novel vulnerabilities in the existing code. It is thus expected that the vulnerability databases used by the tools are kept up to date, and the execution of the tools is triggered on a periodic basis.

As already mentioned in D5.1, the current software development process involves four main environments:

- a development environment and a testing environment hosted by FBK (part of the SPARTA partner CINI); and
- pre-production and production environments hosted by the Italian Ministry of the Interior.

The demonstration scenarios of the vertical 2 involve the development and testing environments, where the preliminary versions of the components are developed and tested (before being migrated on the Italian Ministry of the Interior servers).

The development and testing environments consist of a Gitlab platform hosted by FBK, and cloud-hosted Azure virtual machines.

Gitlab provides:

- a version control system (Git-repository), storing the source code of CIE ID APP and SAML IdP; and
- issues tracking and continuous integration and deployment pipeline.

The Azure virtual machines, running Linux distributions (Ubuntu) and supporting the Docker technology, are used to:

- host the deployed services for functional and security testing purposes, and
- run some of the CAPE assessment tools.

The technical details on how we integrated the CAPE assessment tools in the aforementioned development and testing environments, by following a continuous integration and DevSecOps approach, are provided in D5.3 [2]. In the next sections we provide more details about the demonstration scenarios for the CIE ID APP, and the SAML IdP.

6.2.1 Scenario for the CIE ID App

The user uses his/her CIE with an Android smartphone equipped with NFC interface alongside the CIE ID APP to authenticate with the CIE ID SERVER. It is thus extremely important to leverage methodologies and techniques for the automatic security analysis and risk evaluation of the CIE ID APP. A security flaw in the authentication App could lead to severe security issues, allowing a malicious user to authenticate on behalf of the victim.

For instance, to enhance the usability and the security of the solution, FBK has recently extended the app to support biometric authentication based on fingerprint recognition. The proper usage of the libraries offered by Android is a key requirement to ensure the proper level of security.

The CIE ID APP is an Android app developed in Kotlin. The source code is stored in Gitlab. In the context of the SPARTA project, CINI has extended the Gitlab environment in such a way to use the continuous integration functionalities offered by Gitlab to automatically build the apk file.

The details about this extension and the integration with the CAPE assessment tools are provided in D5.3 [2].

6.2.2 Scenario for the SAML IdP

The SAML IdP is deployed leveraging **Shibboleth**[10]. Shibboleth is among the world's most widely deployed federated identity solutions, connecting users to applications both within and between organizations. Shibboleth provides several software components, each of them is open source.

In the context of Vertical 2, the basic version provided by Shibboleth has been customized to support the interaction between the SAML IdP and the mobile application which communicates with the CIE.

The SAML IdP source code is stored in Gitlab. FBK has recently extended the Gitlab environment in such a way to use the continuous integration functionalities offered by Gitlab. Every time a git commit is pushed on the repository, the source code is automatically built and deployed (using Apache Maven) on an Azure virtual machine. It provides a running version of CIE ID SERVER, which is accessible for functional and security testing.

6.3 Technical Specifications

In the following sections we describe the Security Analysis requirements that are relevant for the security analysis of the CIE ID APP and he SAML IdP provider. To cope with the specific security requirement of both the CIE ID APP and SAML IdP server, we relied on the state-of-the-art standards and frameworks, like the NIST SP 800 series[11] and the OWASP ASVS[12], that were specifically tailored on the mobile and web domains.

6.3.1 Security Analysis of the CIE ID App

6.3.1.1 Vulnerability and Risk Assessment of the CIE ID App

The security evaluation of the CIE ID APP involves the verification of a set of security requirements that enable the assessment of the security posture of the mobile app. In the context of Vertical 2 we relied on the OWASP Mobile Application Security Verification Standard (OWASP MASVS) and the OWASP Mobile Security Testing Guide[13] (MSTG) to identify the security requirements shown in Table 9.

ID	OWASP MOBILE TOP 10	Description
SecR1	M3	Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app.
SecR2	M7	The app is signed and provisioned with a valid certificate, of which the private key is properly protected.
SecR3	M7	The app has been built in release mode, with settings appropriate for a release build (e.g., non-debuggable).
SecR4	M7	Debugging code and developer assistance code (e.g., test code, backdoors, hidden settings) have been removed. The app does not log verbose errors or debugging messages.
SecR5	M7	Debugging symbols have been removed from native binaries.
SecR6	M7	The app only requests the minimum set of permissions necessary.

[10] https://www.shibboleth.net/

[11] https://csrc.nist.gov/publications/sp800

[12] https://owasp.org/www-project-application-security-verification-standard/

[13] https://github.com/OWASP/owasp-mstg

ID	OWASP MOBILE TOP 10	Description
SecR7	M2 M5 M7	The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.
SecR8	M2	The app removes sensitive data from views when moved to the background.
SecR9	M7	The app detects, and responds to, the presence of a rooted or jailbroken device either by alerting the user or terminating the app.
SecR10	M3	The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards.
SecR11	M7	JavaScript is disabled in WebViews unless explicitly required.
SecR12	M3 M7	The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted.
SecR13	M3 M7	The app only depends on up-to-date connectivity and security libraries.
SecR14	M2 M7	No sensitive data is shared with third parties unless it is a necessary part of the architecture.
SecR15	M2 M7	No sensitive data should be stored outside of the app container or system credential storage facilities.
SecR16	M7	The app does not export sensitive functionality through IPC facilities, unless these mechanisms are properly protected.
SecR17	M7	No sensitive data is included in backups generated by the mobile operating system.
SecR18	M7	A WebView's cache, storage, and loaded resources (JavaScript, etc.) should be cleared before the WebView is destroyed.
SecR19	M3	The app either uses its own certificate store, or pins the endpoint certificate or public key, and subsequently does not establish connections with endpoints that offer a different certificate or key, even if signed by a trusted CA.
SecR20	M7 M9	Obfuscation is applied to programmatic defenses, which in turn impede de-obfuscation via dynamic analysis.
SecR21	M3	A WebView's cache, storage, and loaded resources (JavaScript, etc.) should be cleared before the WebView is destroyed.

Table 14: CIE ID App Security Requirements

To evaluate the aforementioned security requirements, we will rely on the tools **Approver** (see Section 7.1) and **TSOpen** (see Section 7.6) by implementing the DevSecOps pipeline that will be described in the Deliverable 5.3 [2].

Approver will check all the vulnerabilities listed in Table 14. Regarding TSOpen, the tool aims at detecting logic bombs in Android apps. However, in the context of this scenario, TSOpen will particularly take care of the dependencies leveraged by the app. In other words, TSOpen will be used to detect logic bombs in the app, including the dependencies used by the app. We remind that logic bombs are mechanisms used by malicious apps to evade detection techniques. Typically, an attacker uses logic bomb to trigger the malicious code only under certain chosen circumstances (e.g., only at a given date) to avoid being detected by the Security Analysis of the CIE ID APP.

6.3.2 Security Analysis of the SAML IdP

6.3.2.1 Software Verification Methods and Vulnerability Assessment for the SAML IdP

Eclipse **Steady** (see Section 7.15) will be used to assess the presence of known security vulnerabilities affecting any of the dependencies of the specific version of Shibboleth integrated in the scenario. In addition, Eclipse Steady will be used to detect whether the code implemented to

customize the solution depends on open-source components with known vulnerabilities, to collect evidence regarding the execution of vulnerable code, and to provide update recommendations.

Eclipse Steady addresses one of the OWASP Top 10 security risks for Web applications, namely the use of components with known vulnerabilities[14], which has the characteristics detailed in Figure 73.

Threat Agents / Attack Vectors		Security Weakness		Impacts	
App. Specific	Exploitability: 2	Prevalence: 3	Detectability: 2	Technical: 2	Business ?
While it is easy to find already-written exploits for many known vulnerabilities, other vulnerabilities require concentrated effort to develop a custom exploit.		Prevalence of this issue is very widespread. Component-heavy development patterns can lead to development teams not even understanding which components they use in their application or API, much less keeping them up to date. Some scanners such as retire.js help in detection, but determining exploitability requires additional effort.		While some known vulnerabilities lead to only minor impacts, some of the largest breaches to date have relied on exploiting known vulnerabilities in components. Depending on the assets you are protecting, perhaps this risk should be at the top of the list.	

Figure 53: Characteristics of A9:2017 - Using Components with known Vulnerabilities (from OWASP)

Due to its code-centric approach, Eclipse Steady can provide the following security functionalities:

- It detects constructs, e.g., methods and constructors of a Java classes that have been subject to known vulnerabilities, no matter through which Java archive those constructs are distributed, provided their signature remains unchanged.
- It detects whether the construct body is equal (or closer) to the vulnerable or the fixed version, according to the commit(s) that were used to fix the respective vulnerability, and which are maintained in Steady's vulnerability database (directly or through Project KB).
- It detects if such constructs are invoked during the execution of automated unit and integration tests or manual tests.
- It detects whether such constructs are part of the call graph that is built starting from the constructs of the application under analysis or starting from the traces collected during test execution.

The latter two analyses are part of the so-called reachability analysis, which is used to prioritize findings. The reachability analysis is necessary, as not all code of all open source dependencies is used in a given application context, a phenomenon often called software bloat.

Eclipse Steady is going to be integrated in the development process of the SAML IdP, leveraging the continuous integration techniques developed in the context of task 5.3.

To complement Eclipse Steady, the **SafeCommit** tool, also called Commit Classifier, (see Section 7.13) will be used to automatically detect vulnerability introducing commits (also referred as patches for sake of simplification) in Continuous Integration Ecosystem. SafeCommit is built on top of AI techniques relying on innovative features and advanced patch representation learning.

Systematically and automatically identifying commits that introduce a vulnerability once a commit is contributed to a code base is of the utmost importance: (1) To reduce the number of vulnerabilities in a software code base; and (2) To incite maintainers to quickly reject the relevant changes. The proposed tool aims at being integrated into real-world software maintenance and usage workflows. The objective is to carry out a live study in order to collect practitioner feedback for iteratively improving the tuning of the research output, towards an effective technology transfer.

[14] https://owasp.org/www-project-top-ten/2017/A9_2017-Using_Components_with_Known_Vulnerabilities

6.3.2.2 Mitigating Software Supply Chain Attacks against SAML IdP

The **Buildwatch** tool (see Section 7.3) will be used to analyse **Shibboleth** that serves as basis of SAML IdP for possible software supply chain attacks. To this end, multiple versions of Shibboleth will be run through the dynamic analysis to determine a common base line. Subsequently, newer versions of Shibboleth can be compared to this base line in order detect unusual changes in behaviour. It will be evaluated how much manual effort remains after the analysis.

6.3.2.3 Risk Assessment of the SAML IdP

The **NeSSoS** risk assessment tool (see Section 7.8) will be used to ensure that all relevant threats affecting the SAML IdP are covered. The main advantage of applying NeSSoS tool is that it provides a holistic evaluation of the network (i.e., considers all aspects of cyber protection), helps to identify the areas which lack security, and provides an instrument to justify investments in the implemented security controls. This tool complements the other tools, focussing on specific security aspects, by providing, though high level, but a holistic overview of cyber security of the considered target.

In scope of this scenario, we will comprehensively analyse the security practices and controls implemented to protect the server, estimate losses for possible threat occurrences, and suggest security controls which can be strengthened to improve protection.

6.4 Assessment tools pipeline

To assess the security of the CIE ID APP and the SAML IdP, we deployed two DevSecOps pipelines. In Figure 54 and Figure 55, we report the DevSecOps pipelines for the CIE ID APP and the SAML IdP, respectively.

The first DevSecOps pipeline relies on Approver (CINI) and TSOpen (UNILU) to evaluate the security and risk requirements for the CIE ID mobile app.

The second DevSecOps pipeline relies on Eclipse Steady and Project KB (SAP), SafeCommit (UNILU), Buildwatch (UBO), NeSSoS (CNR), and VI (UKON) to evaluate the security and risk requirements of the SAML IdP. The details concerning the integration of the tools will be provided in D5.3 [2].

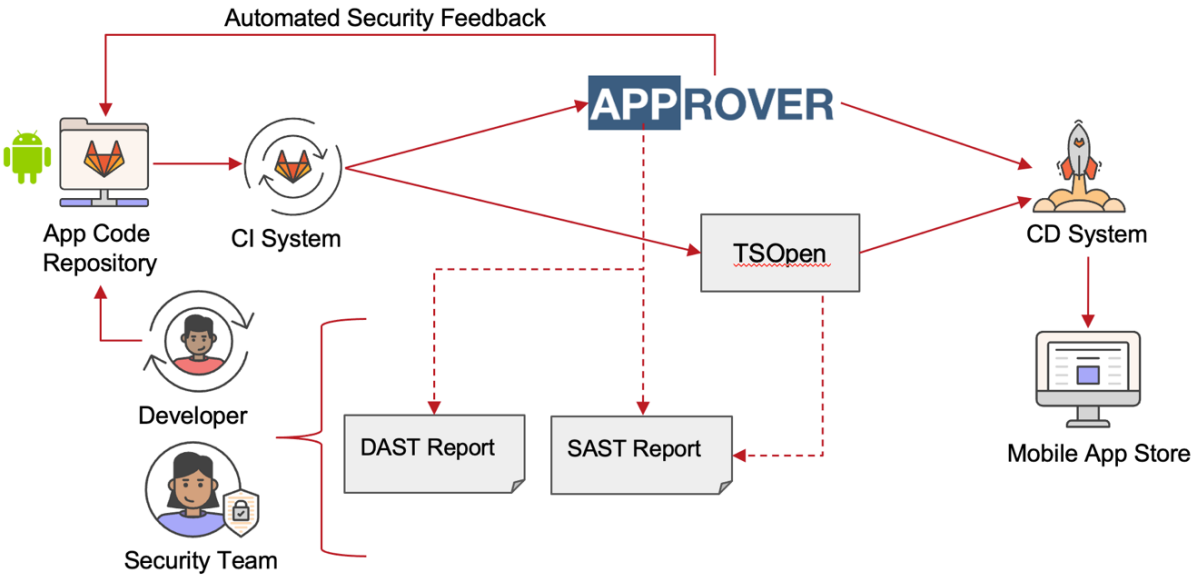

Figure 54: E-gov DevSecOps pipeline CIE ID App

Scenario title	**EGovernment – Mobile**
Description	Demo for the CIE ID APP
V-Model step: Design (from unit testing to acceptance testing)	• Tool name: **TSOpen** • Input: the APK file built from the CIE ID APP source code • Output: an HTML report with the vulnerability assessment and a list of security issues in the GitLab repository. See Section 7.6.4
V-Model step: Development Process	• Tool name: **Approver** • Input: the APK file built from the CIE ID APP source code • Output: a PDF report with the vulnerability assessment and a list of security issues in the GitLab repository. See Section 7.6.4

Table 15: E-gov DevSecOps pipeline CIE ID App

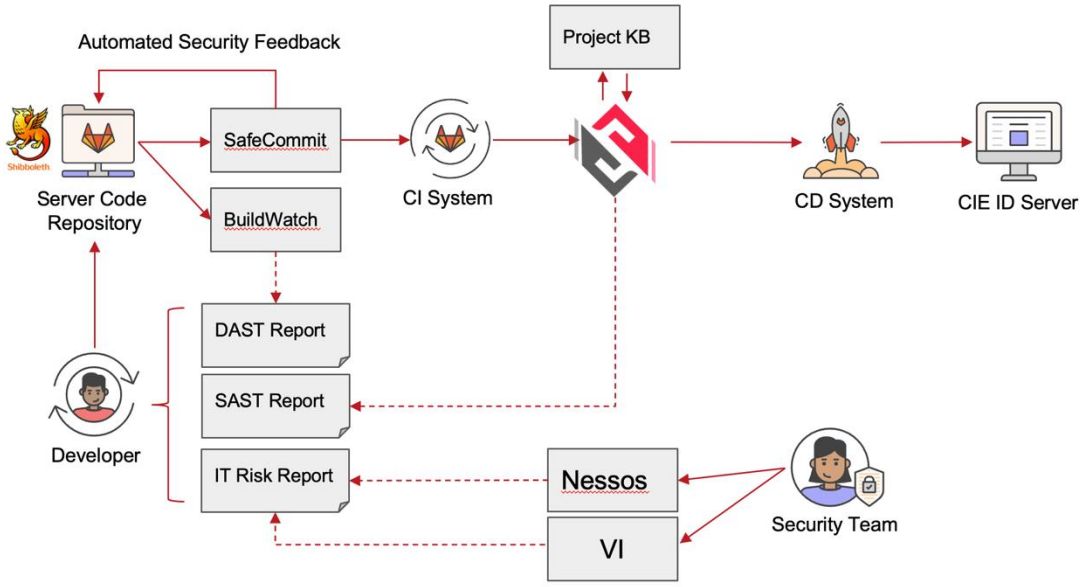

Figure 55: E-gov DevSecOps pipeline SAML IdP Server

Scenario title	**EGovernment – Mobile**
Description	Demo for the SAML IdP
V-Model step: Design (from component design to deployment) and Operations	• Tool name: **Eclipse Steady** • Input: a clone of the Git source code repository of the SAML IdP • Output: an HTML report highlighting open-source dependencies with known vulnerabilities (if any), including a reachability assessment used for issue prioritization See Section 7.15.4
	• Tool name: **Project KB** • Input: YAML statement for a demo vulnerability in one of SAML IdP's dependencies, available in a public or private Git repository • Output: an entry in Steady's vulnerability database that can be used for actual repository scans See Section 7.10.4

V-Model step: Application Development	• Tool name: **BuildWatch** • Input: the list of used packages in the server code • Output: the report with Cyber observables of all packages See Section 7.3.4
V-Model step: Security Analysis, Verification and Validation	• Tool name: **Visual investigation of security information (VI)** • Input: security vulnerabilities discovered by Steady and Project KB • Output: user feedback from domain experts about the usability and usefulness of the design. The demonstrator is used to confirm the applicability of the design. See Section 7.18.4
V-Model step: Risk Management process at the global level	• Tool name: **NeSSoS** • Input: input of the Security Analyst • Output: an HTML report with the simulation results See Section 7.8.4
V-Model step: Software Development (of the libraries used by the application)	• Tool name: **SafeCommit** • Input: source code of the SAML IdP Server • Output: assessment report containing security vulnerabilities introduced in the security commit See Section 7.13.4

Table 16: E-gov DevSecOps pipeline SAML IdP Server

6.5 Adoptability

The two DevSecOps scenarios we deployed for vertical 2 are representative instances of complex systems. Thus, the assets we have developed will be made publicly available and may be used by end-users willing to include the CAPE assessment tools in their pipeline and perform a security assessment of their complex systems. The assets we provide include mainly documentation, software, and services.

Concerning the Documentation, we provide the instructions to:

- install and configure the open-source tools;
- include all the selected tools in the DevSecOps; and
- interact with the tools (e.g., through a GUI) to retrieve the information about vulnerabilities and use them as a support to mitigate the issues.

The tools included in the DevSecOps are heterogeneous. The software we provide is aimed at simplifying the integration and the interaction among the tools. It mainly includes:

- the source code of the open-source tools;
- the tools developed as microservices using Docker technology; and
- the scripts we used to integrate each tool in the DevSecOps (both the tools installed locally, and the ones offered as online services). These scripts are specific for the environment of the vertical (based on Gitlab) but can be used as references and are easily adapted to other environments.

Finally, we provide some services and support, namely:

- the tools that are not free are offered as a service and can be invoked through APIs; and
- we share our expertise acquired during the deployment of the vertical. This experience can be also a topic for future research directions on how to improve the assets.

Chapter 7 Technical Specifications of the CAPE Assessment Tools

This Chapter includes the technical specifications related to the implementation of the tool prototypes that have developed in the context of the tasks T5.1 (see Chapter 2), T5.2 (see Chapter 3) and T5.3 (see Chapter 4), and will be validated in T5.4 by their integration in the CAPE vertical use cases: Connected Car (see Chapter 5) or e-Government (see Chapter 6).

Table 1 gives and overview of the complete list of tools. The descriptions of the tools have been significantly improved and expanded from D5.1[1], while keeping a similar formalism to facilitate their understanding.

Each of the tools provides a detailed technical specification, describing the internal functions of the tool, and including the following subsections:

- **Requirements description**
  - Use cases: description of the use cases in the relevant case studies.
  - User requirements: description of the certification requirements, and when possible related to a compliance standard.
  - Software requirements: list of software requirements of the tool.
- **Functional Specifications**
  - Description of the components that the tool consists of.
  - Description of the tool architecture where components are presented, in order to define a detailed tool roadmap.
- **Development roadmap**
  - The roadmap relates to the use cases and the architecture, it also describes how the proposed architecture will be realized.
  - The development activities identified in the roadmap will be reported in D5.3 [2].
- **Software verification and validation plan**
  - List of methods for verifying the software requirements.
  - The verification of requirements will be reported in deliverable D5.3 [2] and demonstrated upon their completion in D5.4 [3].

7.1 Approver (RAA) – CINI

Approver is an automatic toolkit for the in-depth, fully automatic security analysis of mobile applications. Approver automatically detects, evaluates and provides comprehensive reports explaining the security risks hidden in the mobile applications. The key features include, but are not limited to:

- Advanced Application Analysis based on state-of-the art static analysis techniques
- Automated Security Policy Verification, ensuring that mobile apps comply with security requirements and regulations
- Risk Score and Reports. Detailed, per-app risk reports that summarize the security concerns of the analysed applications.

The state of the art of security application analysis comprises several research tools to analyze Android applications (e.g., SCanDroid [81], CHEX [82], DroidChecker [83], DroidSafe [84], AppAudit [85], VanDroid [86]) or to monitor applications behavior (e.g., ProfileDroid [87], CopperDroid [88], Intellidroid [89]). However, these tools have some limitations, they have been built in order to perform a single type of analysis, they do not provide any type of app aggregation and they generally lack comprehensive reporting capabilities.

More information about Approver is available at: https://approver.talos-sec.com

7.1.1 Requirements Description

7.1.1.1 Use cases

Table 17 shows an update of the Use Cases that were defined for the Approver tool in D5.1.

Use Cases	No change
UC1 Detect security vulnerabilities in Android applications packages	X
UC2 Detect security vulnerabilities in Android applications during development and suggest mitigations	X

Table 17: Approver - Update of Use Cases specifications

7.1.1.2 User Requirements

Table 18 and Table 19 show an update of the User Requirements that were defined for the Approver tool in D5.1.

User Requirements	Add	Comments
UR1.1 Detailed Security Report of the application package	X	Missing in D5.1
UR2.1 Security issues in the DevSecOps pipeline	X	Missing in D5.1

Table 18: Approver - Update of User Requirements specifications

UR1.1	Detailed Security Report of the application package
Description	The tool needs to provide a detailed security evaluation report of the Android application package under test. To this aim, the security analyst needs to access to a detailed security report that describes the overall security risk score, and the detected security vulnerabilities.
Actors	Security Analyst
UR2.1	Security issues in the DevSecOps pipeline
Description	During the development phase, both security analysts and developers need to have an overview of the security posture of the application under development. To this aim, the Approver tool needs to provide a DevOps plugin that can be triggered during the development phase and that can provide a detailed list of security vulnerabilities identified in the source code.
Actors	Security Analyst, Developer

Table 19: Approver – Changes in User Requirements specifications

7.1.1.3 Software Requirements

Table 20 and Table 21 show an update of the SW Requirements that were defined for the Approver tool in D5.1.

Software Requirements	No change	Add	Comments
SR1.1 Implementation of Approver CI Plugins	X		
SR1.2 Enhancement of the SAST vulnerability module		X	Missing in D5.1

Table 20: Approver - Update of SW Requirements specifications

SR1.2	Enhancement of the SAST vulnerability module
Description	The Approver tool contains modules to detect security vulnerabilities in the application source code or in the application binaries. However, the rise of new vulnerabilities and technologies poses a great importance in the accuracy and the customizability of a vulnerability analysis module. To this aim, during the project we will develop a brand new SAST vulnerability analysis module that seamlessly support the addition of new vulnerabilities as security plugin, thus facilitating the constant update of the overall tool.
Actors	Security Analysts, Developers
Basic Flow	The SAST vulnerability module is triggered once an application is submitted to the Approver system. The module computes the list of vulnerabilities, its severity and the suggested countermeasures.

Table 21: Approver – Changes in SW requirements specifications

7.1.2 Functional Specifications

At high-level, Approver is composed of a set of modules for both Static Analysis (SAST) (see Figure 56) and Dynamic Analysis (DAST) (see Figure 57).

Each module, developed as a microservice using Docker technology, enables a different security analysis and is managed by an orchestration layer. Besides, each module exposes a set of RESTful APIs. The modules for SAST are in charge of analysing the application package according to its content. Examples of implemented SAST analysis include vulnerability analysis, permission analysis, and string analysis. Instead, the DAST modules aim to install the application package in a testing environment and evaluate the security of the application during the execution. Examples of DAST analysis include network analysis, API monitoring and filesystem monitoring.

Finally, Approver provides a web front-end that allows to i) view the detailed results of each application analysed, ii) download all the artefacts produced during the analysis, and iii) download a security report which contains all the identified issue.

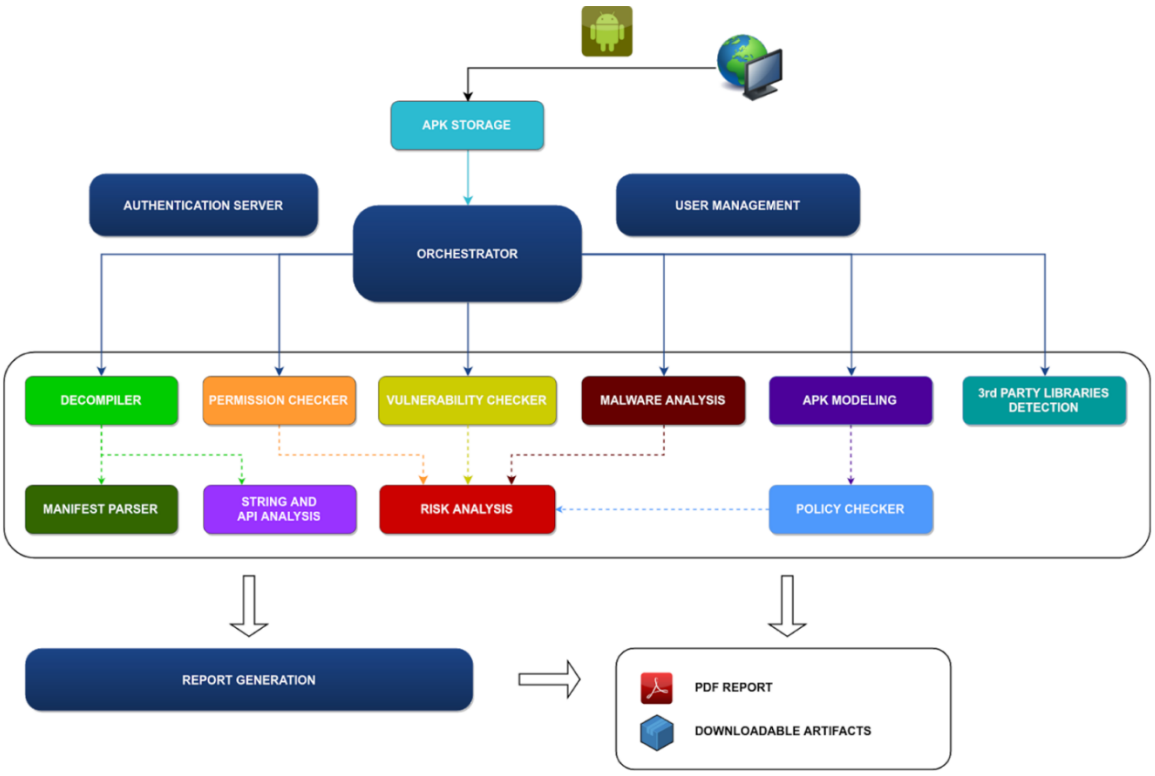

Figure 56: Approver - SAST Modules

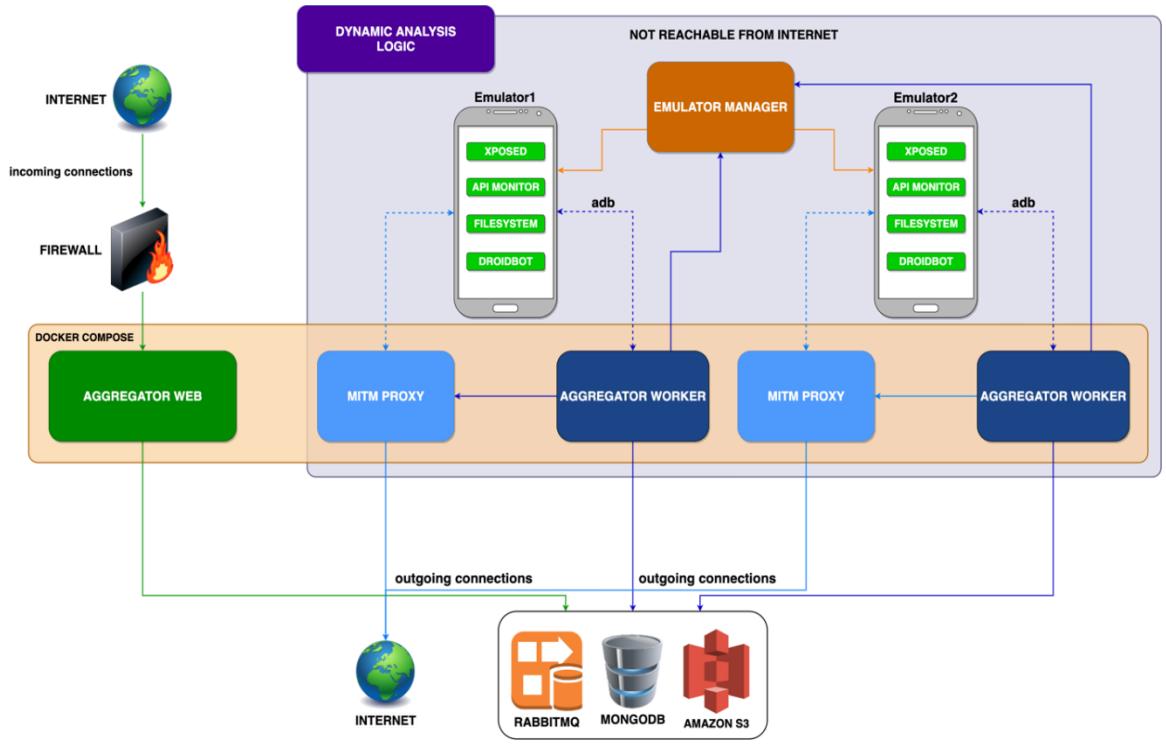

Figure 57: Approver - DAST Modules

7.1.3 Development roadmap

The high-level development roadmap of Approver is to implement the software requirement SR1 within 2020-21 such that it can be demonstrated at project end, as explained in Table 23.

Use Case	Architecture components	Realisation	Involved partners
UC1	Approver RAA	Enhancement of the tool security analysis capabilities	CINI
UC2	Approver RAA	Implementation of the DevSecOps plugins	CINI

Table 22: Approver– Development Roadmap

We will develop the following functionalities:

- An enhancement of the SAST Vulnerability Analysis module. In details, the new module will support the addition of new vulnerability patterns using a plug-n-play approach. The core module collects all the vulnerability patterns and executes the corresponding checks to provide the overall report of the findings. Each vulnerability will include a CVSSv3 score, a description, an OWASP Top 10 risk category, and suggestion for the remediation.
- A set of plugins for the integration of the Approver tool with DevSecOps pipelines to automatically analyse apps during the development process. The first plugin will allow the integration of Approver with the Gitlab CI/CD process.

7.1.4 Software verification and validation plan

SR id	Description	Verification method	Demonstration scenario
SR1.1	Submit the app source to the DevSecOps plugin	Check if scans succeed and the tool successfully reports the security vulnerabilities to the issue tracker	e-Government (Vertical 2)
SR1.2	Scan a mobile app package	Check if scans succeed and findings are correct	e-Government (Vertical 2)

Table 23: Approver - Demo scenarios and verification methods

Submit the app source to the DevSecOps plugin

Input: The source code of an Android application package.

Output: A list of security vulnerabilities in the issue tracker of the DevSecOps pipeline.

Test Procedure:

To test the plugin, the developer is expected to push a commit on the code repository that contains the application source and is connected to the DevSecOps pipeline. The Approver plugin is expected to build the app package and to send it to the analysis backend.

After the analysis is completed, the developer can check in the issue tracker a list of issues that represent the security vulnerabilities contained in the app. At the same time, the security analysis can access to the same report on the Approver web interface.

Scan a mobile app package

Input: An Android application package (APK).

Output: A per-app security report.

Test Procedure:

To test the new SAST vulnerability analysis module, the Security Analyst is expected to submit an Android application package through the Approver web interface. The new SAST module is expected to analyse the binaries of the app and send the result to the backend collector to generate the security report.

After the analysis is completed, the Security Analyst can access to the security report on the Approver frontend and download a PDF version of the report.

7.2 AutoFOCUS3 (AF3) – FTS

AutoFOCUS3 is a Model-Based Engineering Tool that supports Safety Analysis using Goal Structure Notation Models; Security Analysis using Attack Defense Tree Models; Textual and Structured Requirements Engineering; Architecture modelling using hierarchical component structure; Design Exploration Methods; Requirements traceability; Executable semantics; Automatic Code Generation; Hardware Deployment Mapping; and Code deployment.

AutoFOCUS3 provides features that cover most of the phases of the V-model for the development of safety-critical embedded systems: Safety cases as GSN [94], security analysis as attack defence trees, requirements engineering and traceability, design-space exploration, formal verification, automatic code-generation and deployment. In particular, AutoFOCUS 3 enables the use of the most efficient solver (e.g., SMT) for model-based design-space exploration techniques that scale in realistic use cases. Indeed, a state-of-the-art satisfiability modulo theories (SMT) solver, namely Z3 [91], is used to compute such solutions [90]. While other tools such as Papyrus [92] with Moka allow the execution of models based on the fUML [93] Semantics, AUTOFOCUS3 integrates all the modules into a unified software. This has a significant impact on the verification for either testing or

formal verification [90]. AutoFOCUS can be downloaded as a stand-alone application. It is part of Eclipse Foundation. More information about AutoFOCUS is available at: https://af3.fortiss.org/

7.2.1 Requirements Description

7.2.1.1 Use cases

Table 24 shows an update of the Use Cases that were defined for the AF3 tool in D5.1.

Use Cases	No change
UC1 Support the Safety and Security compliance assessment and certification of the platooning scenario	X
UC2 Architecture Modelling for Vertical 1	X

Table 24: AF3 - Update of Use Cases specifications

7.2.1.2 User Requirements

Table 25 shows an update of the User Requirements that were defined for the AF3 tool in D5.1.

User Requirements	No change
UR1 Certifications, such as those used by the automotive industry, e.g., ISO 26262, have been taken into account in several projects involving AutoFOCUS.	X

Table 25: AF3 - Update of User Requirements specifications

7.2.1.3 Software Requirements

Table 26 and Table 27 show an update of the SW requirements that were defined for the AF3 tool in D5.1.

Software Requirements	Add	Comments
SR1.1 C-ACC Safety and Security Co-Validation	X	Missing in D5.1
SR1.2 TARA and HARA modelling	X	Missing in D5.1

Table 26: AF3 - Update of SW Requirements specifications

SR1.1	C-ACC Safety and Security Co-Validation
Description	Cooperative Adaptive Cruise Control (C-ACC) is used by vehicles to improve safety and fuel-efficiency in vehicle platoon. This is because C-ACC enables the safe reduction of the gap between vehicles as vehicles can quickly adapt their state and react to emergency by relying on the information communicated through the communication channels. However, attackers can also exploit these communication channels to cause harm, such as vehicle crashes. We have proposed adequate countermeasures based on plausibility checks. We are developing in the Model-Based Tool AF3 the implementation of C-ACC behaviour. Our starting point is an existing model for platooning using only ACC and extending it to support C-ACC. We are also implementing plausibility checks. We validate using AF3's simulation machinery the impact of the introduction of security countermeasures to safety. The behaviour is also being implemented in the formal verification tool *Maude* (see Section 7.7) to enable the verification of security properties. This means that after the safety and security of C-ACC is evaluated using simulation, one can also use formal verification techniques to provide further evidence about the safety and security of C-ACC.

Actors	AF3, Maude
Basic Flow	AF3 implementation -> AF3 Simulation -> Maude Formal Verification -> AF3 Code Generation
SR1.2	TARA and HARA modelling
Description	TARA and HARA are used to guide the generation of evidence supporting the safety and security of systems. Moreover, safety and security attempt to control threats and hazards by implementing countermeasures and control mechanisms. It is important to understand how these activities impact the general safety and security. However, typical textual descriptions do not enable the automation required to build this understanding. Models, such as *Attack Defence Trees* and *Goal Structure Notation* (GSN) Models, provide structure to these analyses that enable automation. For example, it is possible to extract information contained in GSN models and extract Attack Trees. Similarly, it is possible to understand the impact of security countermeasures in the safety arguments built. We are extending AF3 with the machinery enabling users to create models for TARA and HARA, including attack defence trees and goal structured notation. On a second direction, we are developing a domain specific language for the specification of safety and security analysis, which is used by logic programming engines to enable further automation, such as to determine whether all hazards and threats are adequately handled, automatically suggest solutions for any pending hazard or threat, and automatically carry out trade-off analysis.
Actors	AF3, Logic Programming
Basic Flow	AF3 -> Logic Programming

Table 27: AF3 – Changes in SW requirements specifications

7.2.2 Functional Specifications

AF3 is organized into several Java Plug-ins (more than 20). Each plug-in is responsible for some particular feature. For the SPARTA project, we are developing the AF3 Security plug-in. It contains features, such as threat analyses using Attack Defence Trees, and algorithms for extracting security relevant information from safety analysis.

Table 28 depicts the key developments to be carried out in SPARTA:

- Modelling HARA and TARA of the platooning scenarios using, respectively, Attack Trees and GSN models. This is done by relying on the existing machinery in AF3, namely the Safetycases Plug-In. However, we also extend existing Attack Trees by using the developed algorithms for extracting security relevant information from GSN Models. Finally, we infer the confidence on the combined safety and security assessments based on the trade-off analyses.
- The AF3 Security plug-in has been developed to enable the modelling of attack defence trees, algorithms for the co-analysis of safety and security, and includes domain specific language for security and safety.
- Moreover, the security plug-in uses directly machinery developed in the AF3 Component plug-in, implementing the component model elements available in AF3, and with the SafetyCases plug-in, implementing the machinery for specifying Goal Structure Notation Models.
- Finally, the machinery implemented in the Security plug-in enables the use of the formal verification tool Maude (see Section 7.7) and Logic programming engines to carry out further analysis. This integration is not automatic, however, denoted by the dashed arrows.

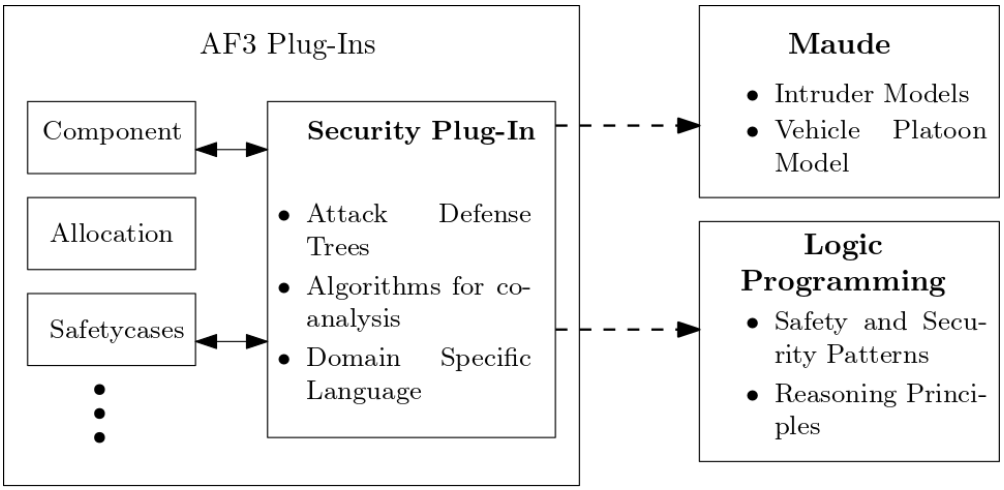

Figure 58: AF3 Security Plug-In and its interaction with other AF3 Plug-Ins and external tools

AF3 has also been used for modelling the logic used by the fortiss Rovers (in Vertical 1, see Chapter 5). We are currently refining these models to accommodate countermeasures that have been proposed in SPARTA. We are implementing the plausibility checks described in Section 5.3.4.3.

7.2.3 Development roadmap

Use Case	Architecture components	Realisation	Involved partners
UC2	AF3	Based on the models available for the fortiss rovers, we are currently implementing in AF3 the countermeasures proposed for Vertical 1 (Demo 1) with the Connected Car.	FTS
UC1	AF3, Logic Programming	Implementation of knowledge bases with safety and security co-analysis techniques as logic programs.	FTS
UC1	AF3	Implementation of quantitative evaluation of safety models written in Goal Structured Notation and Attack Defence Trees.	FTS

Table 28: AF3 – Development Roadmap

We will develop the following functionalities:

- Algorithms for the automated construction of Attack Defence Trees from GSN Models.
- Methodologies for safety and security trade-off analyses. We are implementing two different methods:
  - The first one is based on the architecture of the solution. We will develop a domain specific language with the key aspects to consider in both safety and security, including architectural information, such as components, communication channels, typical safety and security architectural patterns. These are then fed to a logic programming engine together with general reasoning principles. The logic programming engine returns results, such as analysis on which hazards and threats are controlled and mitigated, which patterns could be used to improve the design, and the trade-offs of the proposed safety and security measures.
  - The second type of analysis goes into the behaviour of components. We take as input the behaviour specification expressed as AF3 specification, and feed it to *Maude*, a formal verification tool (see Section 7.7). We have implemented several intruder models for the evaluation of the security of platoon vehicles. Maude then searches whether intruders can attack communication channels to cause safety problems.

7.2.4 Software verification and validation plan

SR id	Description	Verification method	Demonstration scenario
SR1.1	We use the machinery developed to evaluate safety and security of the architectures used for Cooperative Adaptive Cruise Control	Simulations-Based, Formal Verification-Based	Connected Car (Vertical 1)
SR1.2	We use models in Goal Structure Notation (GSN) and Attack Defence Trees for modelling, respectively, the safety and security analysis of the Platooning scenario	Safety and Security Co-Validation	Connected Car (Vertical 1)

Table 29: AF3 – Demo scenarios and verification methods

Simulation-Based verification process

Input: AF3 model with the implementation of C-ACC, the Maude specification corresponding to the AF3 model, and Test-Scenarios. The test scenarios have been selected to cover all the implemented features.

Output: Simulation Results.

Test Procedure:

For each input scenario, we proceed as follows:

1. We configure the parameters of the scenario by configuring the parameters of the AF3 model.
2. We execute the simulation machinery in AF3 with the AF3 model configured to reflect the given scenario.
3. We evaluate the simulation by checking whether the simulation results correspond to the results expected by the scenario.
4. If so, then the scenario verification is considered a success, otherwise a failure.

Formal Verification-Based verification process

Input: AF3 model with the implementation of C-ACC, the Maude specification corresponding to the AF3 model, and Test-Scenarios. The test scenarios have been selected to cover all the implemented features.

Output: Formal Verification Results.

Test Procedure:

For each input scenario, we proceed as follows:

1. We configure the Maude implementation with the parameters provided by the scenario.
2. We use Maude to search for undesired states that would disagree with the expected results in the scenario.
3. We carry out search until a timeout is reached.
4. If an undesired state is reached, then the scenario verification is considered a failure; otherwise it is considered a success.

Verification of C-ACC Safety and Security Co-Validation

The validation procedure is done by the implementation of different features for the modelling of attack defence trees and goal structured notation. These features are to be tested using as input the analysis carried out for the Platooning scenario.

We consider the modelling successful if all the safety and security analysis can be modelled using the Attack Defence Trees and Goal Structure Notation implemented.

Input: Safety and security analysis.

Output: Attack Defence Trees and GSN Models.

Test Procedure:

1. For each safety analysis, we model it as a safety case in the form of a GSN model.
2. We extract an attack tree from the GSN model constructed in the previous step.
3. We complement the attack tree with the security analysis provided as input.
4. The co-validation is considered successful if all the provided safety and security analysis can be modelled as Attack Defence Trees and GSN models.

7.3 Buildwatch (BW) – UBO

Buildwatch is a tool to monitor the interaction of a software with the host operating system during the development phase of a software project. This comprises forensic artifacts, e.g., files created/read or network connections established. It does so by providing a virtual environment, a sandbox, for tasks occurring during development of a software project. Based on the emitted artifacts, an assessment for changed behaviour is possible. Analysis may be conducted within a Continuous Integration process.

The use of Continuous Integration (CI) is a common practice now. Automated security testing of software is considered state of the art and bundled under the principle of “DevSecOps”. Buildwatch can extend this by leveraging well-established state of the art techniques of dynamic analysis used for malware analysis. In the field of malicious open-source software components only a few methods based on anomaly detection using dynamic analysis exists. Most often a heuristic check is carried out to detect suspicious characteristics.

More information about Buildwatch is available at: https://dl.acm.org/doi/10.1145/3407023.3409183

7.3.1 Requirements Description

7.3.1.1 Use cases

Table 30 shows an update of the Use Cases that were defined for the BW tool in D5.1.

Use Cases	No change
UC1 Build Host State Introspection	X

Table 30: Buildwatch - Update of Use Cases specifications

7.3.1.2 User Requirements

Table 31 and Table 32 show an update of the User Requirements that were defined for the BW tool in D5.1.

User Requirements	Add	Comments
UR1.1 Integration	X	Missing in D5.1
UR1.2 Review	X	Missing in D5.1

Table 31: Buildwatch - Update of User Requirements specifications

UR1.1	Integration
Description	Buildwatch needs to be integrated into a CI platform. Hence, the sandbox environment needs to be set up and a custom job must be implemented in the specific continuous integration platform, used during development. If a custom dependency format is used (e.g. Docker) the Differ needs to be calibrated.

Actors	Software developers & testers
UR1.2	Review
Description	After each build, a set of forensic artifacts is presented to the user, to be reviewed.
Actors	Software developers & testers

Table 32: Buildwatch – Changes in User Requirements specifications

7.3.1.3 Software Requirements

Table 33 shows an update of the SW requirements that were defined for the BW tool in D5.1.

Software Requirements	No change
SR1 Process Automation	X
SR2 Version Control	X

Table 33: Buildwatch - Update of SW Requirements specifications

7.3.2 Functional Specifications

The Buildwatch system consist of three Parts (depicted in Figure 59):

- The Monitor
- The Reporting Module
- The Diff Tool

The Buildwatch Sandbox is based on the Cuckoo Sandbox [95] which has experimental support for Linux-based guest systems. Hence, the *monitor* is based on the Cuckoo agent. A software repository, including a build job description, is submitted to the sandboxed environment. The job is executed and resulting system calls are captured using the `systap` interface of the Linux kernel.

Recorded system calls are passed to the *reporting module* for interpretation. The *reporting module* computes an abstraction based on cyber observable objects [96].

The *diff tool* allows the computation of differences between two of these reports. The comparison must be conducted in an object position (in terms of order) independent manner. Further is must filter observables which will be emitted and show differences during every change. These may include temporary files or files whose name include the version number string in their name.

In order to use the Buildwatch Sandbox in a continuous integration pipeline supported development process, the required interfaces must be added.

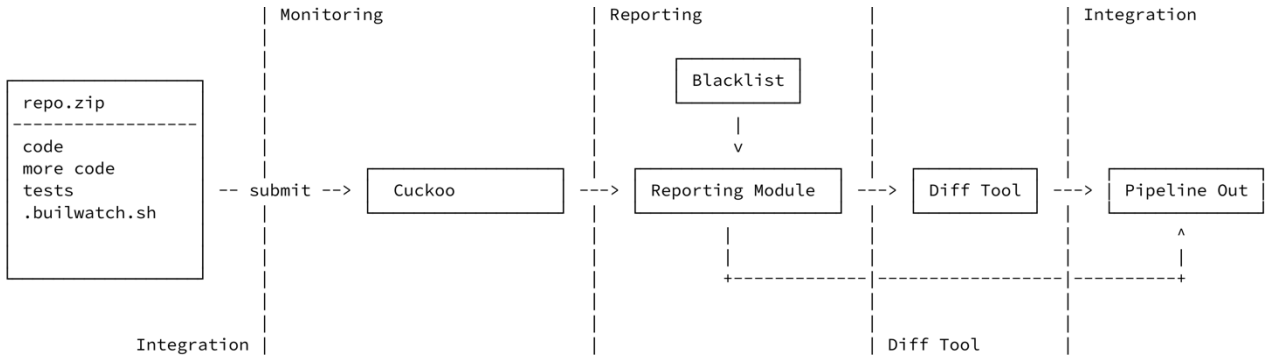

Figure 59: Architecture of Buildwatch, a CI extension for dynamic analysis

7.3.3 Development roadmap

Use Case	Architecture components	Realisation	Involved partners
UC1	Monitor	Extend Cuckoo monitoring capabilities for Linux-based guests.	UBO
UC1	Reporting Module	Implement a custom reporting module that ingests the Cuckoo reported data and aggregates them to cyber observable objects.	UBO
UC1	Diff tool	Implement a script that computes the differences between two Buildwatch reports.	UBO
UC1	Integration	Implementation of the interfaces to ingest a CI job and report the result.	UBO

Table 34: Buildwatch – Development Roadmap

We will develop a security tool for software development which is independent of the actual CI tool employed. To this end, we will leverage Cuckoo, a popular automated sandbox for dynamic analysis of potential malicious software. Most often, build processes are run on Linux. Cuckoo, however, is optimized towards Microsoft Windows and hence certain extensions are required to support Linux (Component: Monitor).

Furthermore, we will create a reporting module that summarises recorded artifacts during the build process of the software (Component: Reporting Module). In order to reduce manual inspection of artefacts, we will (1) store previously encountered artifacts that already have been accepted as benign and (2) are statistically common for the currently analysed software project (Component: Diff Tool). These will be removed from the report that is presented to the developer by the CI job.

To ease integration, we will decouple Buildwatch from the actual employed CI tool by providing an API the CI job can communicate with (Component: Integration). It will accept the source code together with build instructions. After automated analysis and selection of relevant atifacts the CI job can pull the results of Buildwatch and present them to the developer.

7.3.4 Software verification and validation plan

SR id	Description	Verification method	Demonstration scenario
SR1	Ingest a common build dependency	Check that a report comprises all cyber observable objects created or modified by the build process of the software	e-Government (Vertical 2)
SR2	Compute difference between two versions	Two Versions are built in the Buildwatch Sandbox two times each. The differences are computed between all four reports. The computation yields no result between builds of the same version but computes the same differences on reports of different versions.	e-Government (Vertical 2)

Table 35: Buildwatch – Demo scenarios and verification methods

Ingest a common build dependency

This test will show that malicious packages are detectable using this method.

Input: Known malicious packages that show their behaviour during installation and two prior benign versions as their counterparts.

Output: Cyber observables of all packages.

Test Procedure:

1. For each package the observable sets emitted during installation are recorded by Buildwatch.
2. Differences between the two prior benign packages are compared.
3. Differences between the last prior package and the malicious package are compared.

The test is successful if there is a noticeable difference in observables in the second comparison.

Compute difference between two versions

This test will reduce the number of observables that have to be reviewed.

Input: Packages form PyPi and npm.

Output: Diff of the cyber observables of the packages.

Test Procedure:

1. Submit a package to Buildwatch two times.
2. Use the Diff Tool to compute the difference between the two analysis runs of the same package. The test is successful if the comparison yields the result of equivalency of these runs.

7.4 Frama-C (FC) – CEA

Frama-C is a platform for C code analysis based on formal methods. It is collaborative and open source. The tool is comprised of several modular parts capable of performing code transformations, safety and security analyses, and program proofs. A common specification language (ACSL) allows the exchange of results.

One of the main analysers provided by Frama-C is the Eva plug-in, based on abstract interpretation, which enables proving the absence of certain classes of runtime errors such as buffer overflows, invalid pointer dereferencing, and arithmetic errors, all of which may lead to security vulnerabilities.

Open standards such as SARIF15 (Static Analysis Results Interchange Format) are essential to maximize cooperation and reuse between code analysis tools, and between code analysis and other parts of the development cycle. For instance, the static analysis tool evaluation (SATE)16 proposed by NIST accepts (and will recommend in future editions) the SARIF format. Collaborative code analysis tools are strongly encouraged to support this format to help leverage evidence gathered at the code level to other parts of the development and validation cycle. Being an offline text format (produced at the end of an analysis), SARIF is also useful as artifact produced in a continuous integration process.

Considering the assessment of code analyses based on formal methods, [97] reported on studies indicating the difficulty of communication between those writing the code and those verifying it. [98] also reported about the importance of communication between different roles; formal methods rely on very specific assumptions and hypotheses concerning the code, and it is fairly easy to overlook them during repeated iterations of the development cycle. Providing explicitly means for stating these assumptions and an automatic means of enforcing them is necessary to avoid gaps in the process. An audit mode tailored for this purpose is useful for both manual and automatic assessments of the security-related properties which the system must preserve, and therefore a useful feature in the context of SPARTA.

More information about Frama-C is available at: https://frama-c.com

[15] https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html

[16] https://samate.nist.gov/SATE6ClassicTrack.html

7.4.1 Requirements Description

7.4.1.1 Use cases

Table 36 shows an update of the Use Cases that were defined for the Frama-C tool in D5.1.

Use Cases	No change
UC1 Runtime errors and vulnerability identification via static analysis	X
UC2 Code audit accelerated by a value analysis	X

Table 36: Frama-C - Update of Use Cases specifications

7.4.1.2 User Requirements

Table 37 shows an update of the User Requirements that were defined for the Frama-C tool in D5.1.

User Requirements	No change
UR1.1 Quasi-automatic analysis configuration	X
UR1.2 Exchangeable analysis results	X
UR2 Audit-centred analysis exploration and report	X

Table 37: Frama-C - Update of User Requirements specifications

7.4.1.3 Software Requirements

Table 38 shows an update of the SW Requirements that were defined for the Frame-C tool in D5.1.

Software Requirements	No change
SR1.1 CI-based set of parametrization options + example use cases	X
SR1.2 Standardized output format	X
SR2 ”Audit” mode	X

Table 38: Frama-C - Update of SW Requirements specifications

7.4.2 Functional Specifications

Frama-C is a platform for C code analysis based on formal methods. It is comprised of several modular parts, which include code transformations, safety and security analyses, and a graphical interface to explore results and perform semi-interactive proofs.

In CAPE, CEA’s focus is to improve one of the main analysers of the Frama-C platform, called Eva, a value analysis based on abstract interpretation. It performs an automatic, whole-program static analysis which outputs an extensive list of possible runtime errors. Eva also provides information about each program variable at each statement, for all possible executions, easily accessible via a graphical interface. The current architecture of Frama-C/Eva is presented in Figure 60.

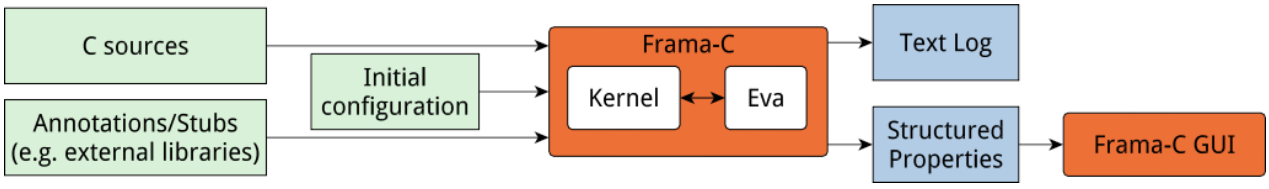

Figure 60: Frama-C/Eva’s current architecture

Given the two use cases related to Frama-C/Eva, there are two main modes of usage of the analyser:

- **CI mode (automatic)**: Eva is used as a static analysis tool, similarly to a code sanitizer, during a build process. A fast, automatic analysis is required, outputting data for a continuous integration process.
- **Audit mode (interactive)**: Eva is used to augment the auditor's understanding of the code, complementing but not replacing human expertise during an assessment. Frama-C's graphical interface provides the set of all possible variable values, plus code navigation possibilities, providing points-to and aliasing information, and evaluation of arbitrary expressions.

Concerning the automatic use mode, Frama-C/Eva has been historically developed for in-depth analyses of safety-critical code bases developed using a traditional process, with few revisions and a long assessment period. In CAPE, the transition to a CI-based analysis with rapid assessments imposes changes to its architecture, as illustrated in Figure 61.

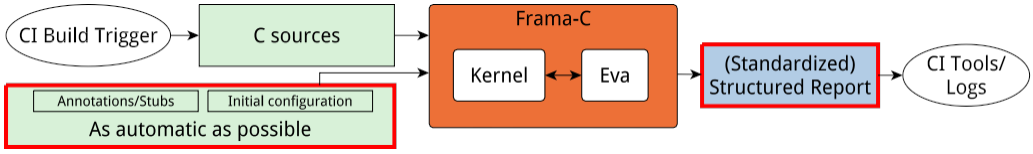

Figure 61: Frama-C/Eva's architecture for CI builds

For the audit mode, the goal is to complement automatic analysis and to support external assessments taking into account the environment, subject to changes. Figure 62 highlights the differences with respect to the existing architecture.

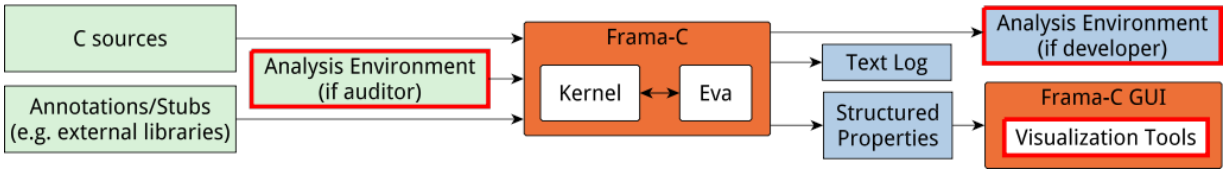

Figure 62: Frama-C/Eva's architecture for audits

7.4.3 Development roadmap

Use Case	Architecture components	Realisation	Involved partners
UC1	Frama-C kernel	Simplify/automate parsing and initial setup	CEA
UC1	Markdown-Report plug-in	Produce outputs in standardized format (SARIF)	CEA
UC2	Frama-C kernel and GUI	Produce environment summaries and check their conformance	CEA

Table 39: Frama-C – Development Roadmap

For UC1, there are two distinct developments: the first part consists in the streamlining of the initial usage of Frama-C which, due to the complexities of the C language, requires a substantial amount of information and setup. The development and improvement of analysis templates, as well as a standardization of defaults, allows for a simpler process based on examples. The introduction of new Frama-C options and helper scripts will further help this process.

For the second part of UC1, the generation of up to date SARIF reports will enable tools conforming to this standard to read the output of Frama-C. This development requires updating the format produced by Frama-C to the latest standard version, then adjusting the output to make it deterministic and as complete as possible. Some new Frama-C features are required to provide necessary data for SARIF reports, such as the full set of command-line options used in the analysis.

For the second use case, UC2, the main features are the sets of environment information, and programmatic reports of the analysis itself. Once these are made available to the user by implementing new runtime options, they will be output as extra analysis results. The final steps of the development consist in incorporating, either in the command-line or in the graphical interface, validators for this information which will allow auditors (and users alike) to quickly identify unusual parametrizations.

7.4.4 Software verification and validation plan

SR id	Description	Verification method	Demonstration scenario
SR1.1	CI-based configuration and use cases	Check applicability and usability on a set of existing code bases	Set of open-source code bases in the Connected Car scenario (Vertical 1)
SR1.2	Standardized output format (SARIF)	Feed output to other tools compatible with SARIF	Integration in the CI pipeline produced in T5.3 in the Connected Car scenario (Vertical 1)
SR2	Audit-mode outputs and validation as inputs	Modify outputs and re-feed them as inputs to check conformance	Set of open-source code bases in the Connected Car scenario (Vertical 1)

Table 40: Frama-C – Demo scenarios and verification methods

Check applicability and usability on a set of existing code bases

Input: Open-Source-Case-Studies Git repository, Frama-C Docker image for CI.

Output: CI artifacts and reports after each build.

Test Procedure: For each open source case study in the OSCS Git repository, we add a CI configuration file, following the documented convention, and taking into account special behaviour as needed for each case.

Then, we run the CI tool, which uses the Frama-C Docker image, to check that Frama-C is able to run the analysis and produce the required artifacts and reports.

Feed output to other tools compatible with SARIF

Input: SARIF reports produced by Frama-C on a few different sample programs.

Output: Manual inspection of format acceptance by SARIF-compatible tools.

Test Procedure: We run Frama-C on some sample programs (short and large code bases, with different kinds of properties) and output SARIF reports for the analysis.

Then, we feed these reports into the SARIF-multitool (a command line-based tool) and check that it validates them as syntactically correct. We then import the reports with the SARIF Viewer plug-in of VS Code, which performs a similar syntactic validation, but also allows checking the usable output: whether messages are informative, locations are correctly mapped, and alarms are signalled as expected.

Modify outputs and re-feed them as inputs to check conformance

Input: Textual result of audit-related Frama-C options.

Output: Pass/fail result based on Frama-C audit options.

Test Procedure: We run the Frama-C tool on the test cases from Open-Source-Case-Studies Git repository, adding audit-related options which produce textual information related to the audit process, such as the implicit hypotheses of an analysis which could lead to an incomplete verification. These options produce some machine-readable output intended for an auditor. We then re-run Frama-C under the “auditor mode”, using the provided input, to verify that the tool reports the expected information. We also try modifying the tool options to “conceal” some important information, and check that the tool reports the discrepancy.

7.5 Legitimate Traffic Generation system (LTGen) – IMT

LTGen (Legitimate Traffic Generation system) is an unreleased network traffic generation and network environment construction system. It relies on a feature-based model extracted from a captured traffic to generate an arbitrary network topology including agents mimicking application clients that will reproduce a background traffic similar to the capture, as realistically as possible.

LTGen has the ability to successfully reproduce traffic captures from well-known traffic dataset sources (e.g., MAWILab), and overwhelm intrusion detection systems under test, so as to elicit false positives or system failures.

Assessing the performance of intrusion detection systems has often been performed by measuring the attack detection accuracy, i.e., the ability of the detector to correctly classify in the presence of both legitimate and malicious data. Many approaches even only test the attack coverage, that the ability of the classifer to recognize attacks in the sole presence of malicious data [99].

In the case of network intrusion detection, datasets are made of network traffic captures, which were generated in a more or less automated way. Sadly, the number of datasets is quite low, making their diversity questionable and their ageing quite problematic [100]. Thus, instead of relying on a small number of static datasets, practictioners may resort to dynamically generating more diverse datasets for assessment purposes.

Traffic data generation has been extensively studied in the literature, following 3 main methods, namely (i) traffic replay, (ii) traffic modelling, and (iii) user behaviour modelling. The first approach often leverages the above-mentioned ageing datasets and requires adapting the replayed captures to the assessment environment. The second models traffic from existing traces by leveraging statistical distributions, such as IP spatial distribution, inter-session start times and session duration for D-ITG [101] or file sizes, inter-connection times and the number of active flows for Harpoon [102].

Our work actually falls into the third category where behaviour patterns of actual users are mimicked to generate background traffic using real services and protocols, as it was the case for constituting the infamous DARPA datasets.

It extends this approach by combining the real generation of traffic with the modelling of previous traces as a way to define a reference model.

Data-driven generation is re-emerging with the advent of machine- and deep-learning where neural networks, such as autoencoders [103] or generative adversarial networks [104], have demonstrated their ability to generate convincing models. While these works are quite promising, they do not generate traffic traces but only feature vectors. Our approach will merge both efforts in features generation and traffic generation to provide traffic traces that look similar to real network captures. Additionally, we will investigate how these new generative methods could be used for adversarial training, by generating new malicious traces from existing ones.

7.5.1 Requirements Description

7.5.1.1 User Cases

Table 41 shows an update of the Use Cases that were defined for the LTGen tool in D5.1.

Use Cases	No change
UC1 Synthetic traffic generation from existing traces	X
UC2 Attack traffic mutation	X

Table 41: LTGen - Update of Use Cases specifications

7.5.1.2 User Requirements

Table 42 shows an update of the User Requirements that were defined for the LTGen tool in D5.1.

User Requirements	No change
UR1.1 Availability of network traffic trace	X
UR1.2 Privacy-preserving traffic generation	X
UR2.1 Availability of network intrusion traffic	X
UR2.2 Interpretability of results	X

Table 42: LTGen - Update of User Requirements specifications

7.5.1.3 Software Requirements

Table 43 shows an update of the SW Requirements that were defined for the LTGen tool in D5.1.

Software Requirements	No change
SR1.1 Metrics to measure realism	X
SR1.2 Anonymization functions	X
SR2.1 Metrics to measure malice	X
SR2.2 Mutation functions	X

Table 43: LTGen - Update of SW Requirements specifications

7.5.2 Functional Specifications

The proposed tool, LTGen, is constituted of two main modules:

- a network traffic parser to process captured traffic inputs, and
- a network traffic generator to generate traces for IDS/SIEM evaluation.

In CAPE, IMT aims at improving the parser to extract new features that will enable a more faithful modelling of traffic traces. By reliably modelling real traffic traces, we believe that we will be able to generate more realistic network traffic. The models learned from a single traffic trace allow the generator to reproduce traffic for this particular trace. One particular challenge is the feasibility of producing full-fledged traffic traces from a few model features. Future developments aim at using an autoencoder to learn the traffic features found in traffic captures.

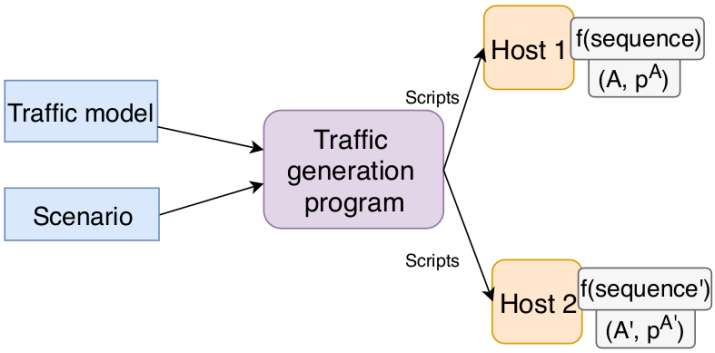

Figure 63: Architecture of LTGen generation module

The traffic generator of LTGen takes two inputs as shown in Figure 63: (1) the model features learned from the parser and (2) a network topology description (also called scenario). The network

description is written in YAML, a text-file format, to be consumed by the Heat[17] orchestration engine of the OpenStack virtualization platform. This description includes a section for each subnet, and lists the hosts within the subnet, along with their IP addresses. This file is currently obtained manually from a quick analysis of the traces. One improvement could be to have the parser generate it automatically.

The second input, the list of model features, is generated by the parser and supports the following features:

- **Time interval**. Indicates the period during which a specific flow is generated at a specified throughput
- **Average throughput**. Indicates the amount of traffic data, in terms of bytes per second, for a flow generated in the network
- **Distribution of services**. Lists the main application protocols and their corresponding weights, that is the ratio of the traffic volume (in terms of bytes) over the total amount of traffic.

LTGen currently supports four main protocols, namely HTTP(S), IMAP, SMTP, and FTP. From these inputs, the LTGen generator launches a network construction module to create an environment with the topology specified in the description. As show in Figure 64, after confirming that the environment has been constructed successfully (steps 1 and 2), it processes the traffic features and triggers the generation of the synthetic flows through scripts orchestrating the launched hosts (see Figure 63). At the same time, it records the generated traffic at the switches (step 5). When the generation is complete, the records are processed, and the extracted traffic features are reported back to the user (steps 6 and 7). Finally, it cleans up the environment, to be available for the next run (steps 8 and 9).

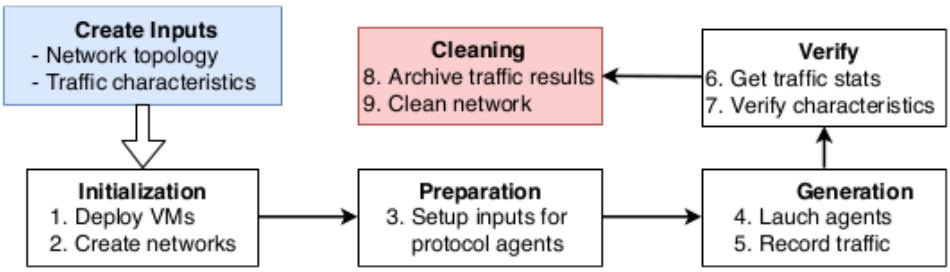

Figure 64: LTGen Workflow

A second objective of our approach is to generate malicious vectors able to challenge the systems under evaluation (IDS, SIEM). IMT will develop a generative adversarial network-based (GAN) approach. Using a GAN, we aim at improving concurrently two aspects of the generated traffic: its realism and its malice, so that it becomes difficult for the system under evaluation to discriminate real, legitimate traffic from the malicious, synthetic one.

Finally, a human interface module should summarize the results of the test and assist the tester in identifying the weaknesses of the system under evaluation to make recommendations on how to improve it.

7.5.3 Development roadmap

Use Case	Architecture components	Realisation	Involved partners
UC1	Parser	Design new features set to extract	IMT
UC1	Generator	Design and implement autoencoder-based feature generator and translator	IMT

[17] https://docs.openstack.org/heat/latest/

Use Case	Architecture components	Realisation	Involved partners
UC1	User interface	Produce explainable outputs for IDS/SIEM evaluation	IMT
UC2	Generator	Design and implement a GAN-based attack traffic generator	IMT
UC2	User interface	Produce explainable outputs for IDS/SIEM evaluation	IMT

Table 44: LTGen – Development Roadmap

The first use case (UC1) will require a couple of developments to improve the state of the art. First, we need to design a set of features able to capture dynamically network traffic behaviours and develop the according collection tools to extract them. It will enable to break away from the statistical approach currently employed, which is unable to capture the changes of network behaviours over time. To achieve the dynamic learning of the network traffic behaviours, an autoencoder-based approach will be developed to support the generation of new traffic features. Such features are not sufficient for traffic generation, and will therefore be translated using a new component, the translator, that takes as input a learned model and outputs a traffic generator configuration. The current traffic generator has been described previously in Section 7.5.2. Finally, the user interface should integrate additional information in order to highlight the results of the evaluation of the IDS/SIEM, with respect to the generated traffic, in particular, false positives.

The second use case (UC2) will develop a new component as an adversarial generator, able to evade intrusion detection. The design will need to consider how the malice can be adequately represented in the generation model, and how to concretely generate it, which is an emerging research issue. The GAN-based approach will complement the legitimate traffic generation in constituting a mixed dataset, necessary to a complete intrusion detection evaluation. Similarly to the UC1, UC2 requires that the evaluation results be highlighted in the user interface, with respects to false negatives.

7.5.4 Software verification and validation plan

LTGen is being developed as a standalone tool within CAPE, and hence does not interact with other components during verification and validation.

SR id	Description	Verification method	Demonstration scenario
SR1.1	Metrics to measure realism	Check the metrics against real traffic traces	Test set of real traffic traces
SR1.2	Anonymization functions	Assess the privacy of pro- cessed traces	Privacy impact assessment (PIA)
SR2.1	Metrics to measure malice	Check the potential damage to a target system	Set of target systems
SR2.2	Mutation functions	Measure mutation ratio	State-of-the-art mutation metrics

Table 45: LTGen – Demo scenarios and verification methods

Testbed-Based verification process

Input: Real traffic traces, target deployment scenario (topology). A set of diverse traffic traces and deployment scenarios should demonstrate the ability of the tool to reproduce traffic realistically, and to tailor it to different environments.

Output: Generated traffic, evaluation results, Privacy Impact Assessment.

Test Procedure: For each input scenario, we proceed as follows:

1. We build a scenario file (topology description) for OpenStack Heat and pre-process the traffic capture.
2. We execute LTGen over these inputs which will launch the deployment of the scenario and trigger traffic generation.
3. We record the generated traffic.
4. We compute several metrics to assess (1) the realism of the synthetic traffic, and (2) the privacy impact with respect to the original trace.
5. If the traffic is deemed realistic AND does not harm privacy, then the generation is considered to be successful.

7.6 Logic Bomb Detection (TSOpen) – UNILU

TSOpen is an open-source tool able to statically detect logic bombs mechanisms in Android applications. Logic bombs are mechanisms used by malicious apps to evade detection techniques. Typically, an attacker uses logic bomb to trigger the malicious code only under certain chosen circumstances (e.g. only at a given date) to avoid being detected by the analysis. The goal of TSOpen is to detect such logic bombs. The approach used to perform the detection is fully static and combine multiple techniques such as symbolic execution, path predicate reconstruction, path predicate minimization, and inter-procedural control-dependency analysis. In a first version, TSOpen will focus on detecting triggers related to time, location and SMS.

From a more technical point of view, TSOpen is developed over Flowdroid, a static analysis tool, which provides a useful model of the Android Framework on which one can easily apply algorithms.

Researchers have been fighting against logic bombs for decades on desktop applications. However, not much work has been provided in the literature to cope with the logic bomb problem in the Android ecosystem. Mainly, related-works provide approaches able to detect sensitive triggers [105] [106]. Besides, triggering code under certain circumstances can also be used for good. Indeed, Zeng et al. [107] have presented an approach to detect app repackaging using special triggers.

TriggerScope [105] is a fully static analysis tool that relies on symbolic execution and path predicate recovery to automatically reveal certain types of sensitive triggers. More recently, Dark hazard was presented as a hybrid approach combining static analysis to find trigger of interest as well machine learning techniques to detect Hidden Sensitive Operations using not an SVM classifier. They do not focus on malicious behaviour but are able to reveal sensitive behaviour triggered under specific circumstances.

Our prototype, TSOpen is, likewise TriggerScope, a fully static analysis tool built on the idea of tweaking some parameters of the analysis to make it more efficient.

More information about TSOpen is available at: https://github.com/JordanSamhi/TSOpen

7.6.1 Requirements Description

7.6.1.1 Use cases

Table 46 shows an update of the Use Cases that were defined for the TSOpen tool in D5.1.

Use Cases	No change
UC1 Detecting hidden malicious code	X

Table 46: TSOpen - Update of Use Cases specifications

7.6.1.2 User Requirements

Table 47 and Table 48 show an update of the User Requirements that were defined for the TSopen tool in D5.1.

User Requirements	Add	Comments
UR1 Security report on the presence of logic bomb mechanism	X	Missing in D5.1

Table 47: TSOpen - Update of User Requirements specifications

UR1	Security report on the presence of logic bomb mechanism
Description	The tool provides a detailed security report on the presence of logic bombs in an Android application under test. In the best case, the tool is also able to pinpoint the malicious piece of code which is “protected” by the logic bomb.
Actors	Security Analyst

Table 48: TSOpen – Changes in User Requirements specifications

7.6.1.3 Software Requirements

Table 49 and Table 50 show an update of the SW Requirements that were defined for the TSOpen tool in D5.1.

Software Requirements	No change	Add	Comments
SR1 A standalone command line tool	X		
SR2 Trigger database	X		
SR3 Precise Data Flow tracking		X	Missing in D5.1

Table 49: TSOpen - Update of SW Requirements specifications

SR3	Precise Data Flow tracking
Description	To detect logic bomb, the tool needs to perform data-flow tracking in order to follow sensitive information flow. This task is challenging, especially in the Android ecosystem where communication between components are performed via intent and specific ICC methods (e.g., startActivity).
Actors	TSOpen Users
Basic Flow	• Download TSOpen from: https://github.com/JordanSamhi/TSOpen • Follow the instructions in the README file to build it • Run the tool with the options available • Analyse the results

Table 50: TSOpen – Changes on SW requirements specifications

7.6.2 Functional Specifications

TSOpen is developed over Flowdroid which provides a useful model of the Android Framework on which one can easily apply algorithms. Figure 65 provides an overview of the tool. First, an inter-procedural control flow graph from Flowdroid is retrieved on which TSOpen applies a symbolic execution in order to retrieve the semantic of objects of interest. Then simple predicates are retrieved during the block predicate recovery to annotate the Inter-Procedural Control-Flow graph (ICFG). The annotated ICFG is then used to retrieve the full path predicate of every instruction. A predicate minimization algorithm is then applied in order to rule out false dependencies. Afterwards, a first decision is taken during the predicate classification step to get suspicious predicates. Finally, a control dependency step is applied in order to take the decision regarding the suspiciousness of the potential logic bomb under study.

The TSOpen tool consists of a standalone executable Java archive file (jar). It has to be executed with the command line or in scripts.

Dataflow in Android application is challenging due to its inner functioning. Indeed, Android apps rely on Inter-component communication (ICC) to share data, switch from one User Interface to another, perform background tasks or start other applications. Those behaviours can be performed thanks to special ICC methods (e.g. *startActivity*, etc.). Then, for precise dataflow tracking it must be taken into account to have a precise model, the state-of-the-art got interesting and developed many tools (IccTA, Amandroid, Droidsafe, etc.) to overcome this limitation.

However, ICC can also be performed with non-standard (atypical) methods that are not modelled by the state-of-the-art, e.g. *sendTextMessage* which can trigger another component with the help of *PendingIntent* Objects. We then propose a tool called RAICC (Revealing Atypical Inter-Component communication) which is able to overcome this limitation by modelling 111 methods systematically gathered in the Android Framework.

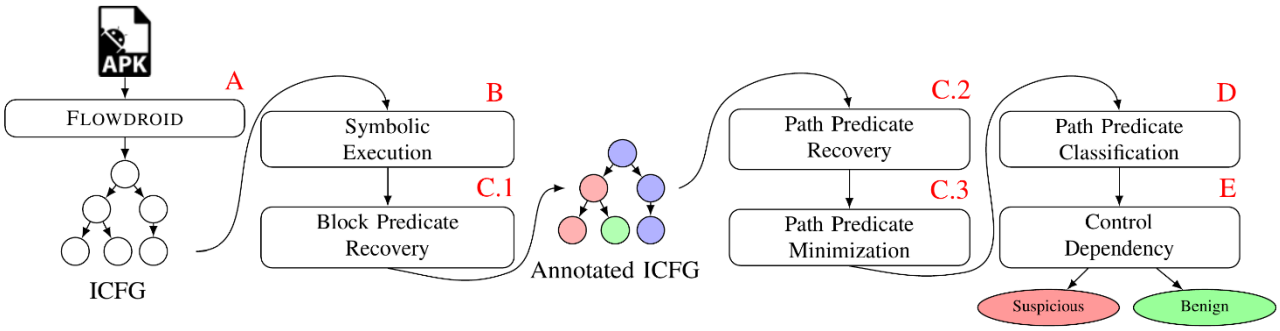

Figure 65: Overview of Logic Bomb Detection (TSOpen)

7.6.3 Development roadmap

Use Case	Architecture components	Realisation	Involved partners
UC1	Detection of hidden malicious code	Build the prototype to detect logic bombs	UNILU

Table 51: TSOpen – Development Roadmap

TSOpen is a tool that is composed of several modules that need to be developed separately, therefore we will follow the following approach for the development of the logic bomb detector in Android apps:

1. We will first set up the environment with Soot and Flowdroid which are responsible for the data-flow model generation and the Inter-Procedural Control Flow Graph generation (ICFG).
2. We will then develop the symbolic execution engine and log the values modelled to test the prototype.
3. The ICFG will then be annotated by tagging the statements with simple block predicates that need to be passed to reach a specific statement.
4. Each statement will then be annotated by the full path-predicate (entire formula) to reach it and this formula will be minimized applying well-known Boolean algorithms to remove false dependencies on guarded blocks.
5. The predicate will be classified according to the symbolic execution output and the tag given for the variables in the formula.
6. Finally, we will perform a control dependency step that checks if the blocks dominated by a potential sensitive predicate contain a call to a sensitive API.

Once the prototype is fully developed, we will test it and evaluate the results. Eventually we will perform large-scale study to assess the tool in the wild.

The approach is fully static, which means it is prone to high false-positive rates, therefore next year we plan to address the challenge by improving the current approach to reduce the false-positive rate. More specifically, we plan to implement precise taint analysis to find potential entry-point to logic bombs. Also, as detecting malicious behaviour in the code guarded by an “if” statement reduces to detection malicious code in the entire applications, we plan to test an anomaly detection scheme to detect potential logic bombs.

7.6.4 Software verification and validation plan

SR id	Description	Verification method	Demonstration scenario
SR1	Standalone command line tool	Check if the tool works properly with right dependencies	Use the tool with the command line
SR2	Trigger database	Check if the database contains correct triggers	Connect to the database (e-Government scenario)
SR3	Precise Data Flow tracking	Manually check reported data flow paths on sample data	Use a benchmark (e-Government scenario)

Table 52: Logic Bomb Detection – Demo scenarios and verification methods

Command line tool

Input: Android app.

Output: Result of logic bomb detection.

Test Procedure: For each input scenario, we proceed as follows:

- We package the TSOpen tool
- We set all the parameters needed to run it on specific app
- We execute the detection
- Given the output, we verify if it runs correctly or if a further dependency is needed
- If it runs correctly, we verify the result: presence of logic bomb or not
- It not, we resolve the dependency.

Trigger database

Input: Trigger database.

Output: Clean database or not.

Test Procedure: For each input scenario, we proceed as follows:

- We connect to the database
- We manually verify if the triggers in the database are correct
- If it is the case, the database is clean
- If not, we have to correct it.

The database entries are given after manual verification of output of logic bomb detection.

Precise Data Flow tracking

Input: Android app

Output: Dataflow path

Test Procedure: For each input scenario, we proceed as follows:

We execute RAICC base on arbitrary sources and sinks and we manually verify its output. If the app contained a data flow path using an atypical method, the flow should be found from a source to a

sink. If the data flow path does not use an atypical method, it should not detect it. Therefore, we developed 20 benchmark apps to verify its precision, each app being an input.

7.7 Maude (MAU) – FTS

Maude is a formal verification tool based on Rewriting Logic, a language for distributed systems. Maude can be used to formally verify distributed systems by using its search engine. A number of frameworks have been developed over Maude. For example, the framework Soft-Agents enables the specification and verification of robust autonomous agents. Other frameworks have been built for security verification of industry 4.0 applications [108]. These models enable the symbolic verification of systems using symbolic intruder models. In SPARTA, we are developing models in Maude for the specification and verification of platooning scenarios, in particular, countermeasures and intruder models that can enable the verification of such systems using Maude.

Maude is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications [109]. The key novelty of Maude is that it supports rewriting logic computation, besides supporting equational specification and programming. Another key distinguishing feature of Maude in comparison to other languages like CafeOBJ [112] and ELAN [111] is its systematic and efficient use of reflection, a feature that makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming and metalanguage applications [110].

More information about Maude is available at:
http://maude.cs.illinois.edu/w/index.php/The_Maude_System

7.7.1 Requirements Description

7.7.1.1 Use cases

Table 53 and Table 54 show an update of the Use Cases that were defined for the Maude tool in D5.1.

Use Cases	Add	Comments
UC1 Formal Security Verification of platooning SafeSec module	X	Missing in D5.1

Table 53: Maude - Update of Use Cases specifications

UC1	Formal Security Verification of platooning SafeSec module
Description	By modifying or spoofing messages, an intruder can confuse vehicles' embedded systems and cause accidents. We propose the specification and formal verification of countermeasures proposed to mitigate attacks.
Actors	Security and Verification Engineer
Basic Flow	• Modelling of platooning in Maude • Propose Intruder Models • Use Maude to verify proposed countermeasures

Table 54: Maude – Changes in Use Cases specifications

7.7.1.2 User Requirements

Table 55 and Table 56 show an update of the User Requirements that were defined for the Maude tool in D5.1.

User Requirements	Add	Comments
UR1.1 Automated Formal Security Assessment of Cyber-Physical Agents	X	Missing in D5.1

Table 55: Maude - Update of User Requirements specifications

UR1.1	Automated Formal Security Assessment of Cyber-Physical Agents
Description	Specification of countermeasures and intruder model capabilities.
Actors	Security and Verification Engineer

Table 56: Maude – Changes in User Requirements specifications

7.7.1.3 Software Requirements

Table 57 and Table 58 show an update of the SW Requirements that were defined for the Maude tool in D5.1.

Software Requirements	Add	Comments
SR1 Maude Software	X	Missing in D5.1

Table 57: Maude - Update of SW Requirements specifications

SR1	Maude Software
Description	Maude software
Actors	Security and Verification Engineer
Basic Flow	• Install Maude (available from http://maude.cs.illinois.edu/w/index.php/The_Maude_System#General_Maude_Information) • Download the specification of the platooning scenario • Execute Maude with a given security query

Table 58: Maude – Changes in SW requirements specifications

7.7.2 Functional Specifications

Maude is formal framework for the modelling of distributed systems and the verification of its properties. Currently it is in version 3.0. It uses as underlying foundations Rewriting Logic. It is a logic that is suitable for the specification of concurrent systems, as it can express stateless behaviour in the form of equational theory, and stateful behaviour in the form of rewriting rules. A number of systems are based on Maude, including systems for the formal verification of security protocols, real-time systems, biological systems.

For an overview of the Maude tool, we refer the reader to the Maude home page[18].

In particular, Maude has been used to specify the framework Soft-Agents [113] which is a framework for the specification and verification of Cyber-Physical systems.

[18] http://maude.cs.illinois.edu/w/index.php/The_Maude_System

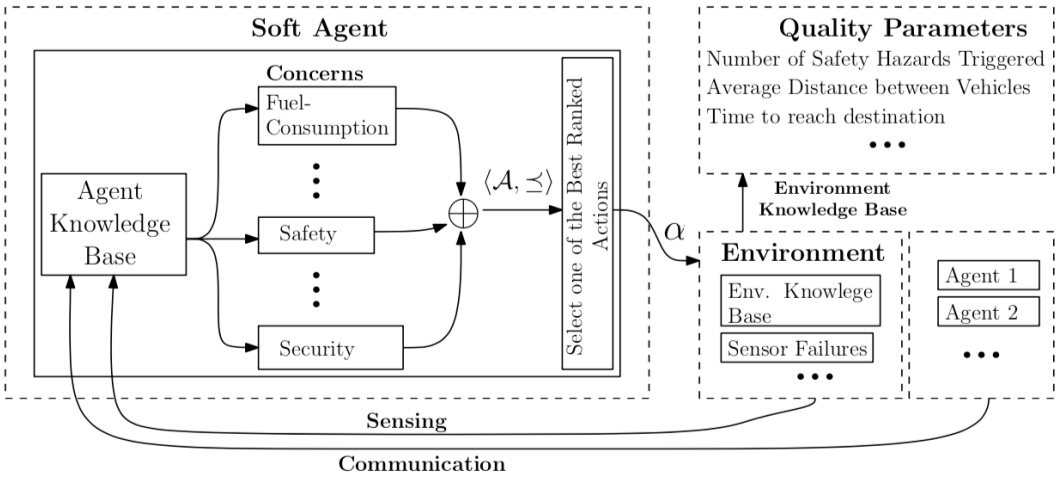

Figure 66: Soft-Agent Framework Architecture in Maude

Figure 66 depicts the general architecture of a soft-agent, or simply agent. An agent has its own local knowledge base that contains, e.g., its current perceived speed, position, and direction of the other agents. Further data may be obtained by sensing the environment or by sharing of information between agents through communication channels. Using its local knowledge base, the agent decides which action to perform according to its different concerns specified as a soft constraint (optimization) problem [114]. For example, if the distance to the vehicle in front is too great, the fuel consumption concern kicks in and attempts to reduce it by accelerating. Similarly, if the distance is dangerously short, then the safety concern kicks in and attempts to increase it by decelerating. As soft constraints subsume other constraint systems, e.g., classical, fuzzy and probabilistic, it is possible to formally specify a wide range of decision algorithms.

7.7.3 Development roadmap

Use Case	Architecture components	Realisation	Involved partners
UC1	Maude specifications for the Platooning scenario	Maude specification based on the framework Soft-Agents [115]	FTS

Table 59: Maude Tool – Development Roadmap

We will take the following steps for the development of a verification framework for Vertical 1 (Platooning) using Maude and the existing Soft-Agent architecture.

- We will develop a domain specific language with the alphabet specific to Vertical 1.
- We will specify in Maude the soft constraints used for governing the behaviour of vehicles in a platoon. This means that vehicles consider at least two concerns, the reduction of fuel consumption and safety.
- We will implement intruder models specifying the intruder's capabilities that include the injection of messages in the vehicle communication channels and the jamming of communication channels.
- We will implement countermeasures, e.g., plausibility checks.
- We will implement evaluation scenarios and verification problems, such as, determining whether an intruder can cause vehicles to crash.

7.7.4 Software verification and validation plan

SR id	Description	Verification method	Demonstration scenario
SR1	Use Maude to formally verify platooning modules	• Discover an attack • Evaluate countermeasure	Connected Car (Vertical 1)

Table 60: Maude Tool – Demo scenarios and verification methods

Maude will be evaluated in its capacity to automatically find attacks and its capacity in providing evidence on the security of systems, in particular, the security of platooning systems.

Verification method

Input: The model of the platoon behaviour specified in Maude, including countermeasures. The capabilities of intruders. The scenarios to be verified.

- Some scenarios are taken from the literature with attacks that have been found. The intention is to validate whether the Maude machinery can discover these attacks in an automated fashion. Moreover, we also consider scenarios that implement countermeasures.

Output: Evidence supporting the security of the given models with respect to the specified intruder models.

Test Procedure: For each test scenario,

- We configure the Maude model to correspond to the scenario and configure the intruder to possess the specified capabilities.
- For a given timeout, we search using Maude's search engine for a bad situation, e.g., a vehicle crash that can be caused by the intruder by bypassing existing countermeasures.
- If an attack is found that is expected, then we say that Maude succeeded to discover an attack. Otherwise, we consider it to have failed.

7.8 NeSSoS Risk Asessment tool (RA) – CNR

NeSSoS Risk assessment tool is a free to use on-line service with the main goal to provide a simple and quick facility for cyber risk self-assessment. The tool requires two types of input: information about security measures and information about key assets of the enterprise. When all inputs are provided, the tool estimates the expected annual losses for every relevant threat and a total one. The output is to be available when the input information is correctly provided.

Risk assessment is an essential and well-recognised practice to ensure that all security risks are taken into account and adequate treatments are implemented. There are a number of approaches to risk assessment [116] [117] [118] [119] [120] and they often require significant time and effort (as well as knowledge) to conduct risk assessment properly. This is especially challenging for SMEs which are often short in resources and cyber security knowledge. The NeSSoS tool (provided as a free service) simplifies the process allowing the users to conduct basic risk self-assessment (without relying on external cyber risk experts). The tool realises quantitative risk assessment, in contrast to the majority of other risk assessment methods using qualitative analysis, which helps to roughly estimate expected losses due to cyber events. Another advantage of the tool is that it also helps to optimise future expenditure using the cost-benefit facility of the tool.

More information about NeSSoS is available at:
https://www.cybersecurityosservatorio.it/en/Services/survey.jsp

7.8.1 Requirements Description

7.8.1.1 Use cases

Table 61 shows an update of the Use Cases that were defined for the NeSSoS tool in D5.1.

Use Cases	No change
UC1 Evaluation of e-government risks	X

Table 61: NeSSoS - Update of Use Cases specifications

7.8.1.2 User Requirements

Table 62 shows an update of the User Requirements that were defined for the NeSSoS tool in D5.1.

User Requirements	No change
UR1.1 Identification of risks and relevant security controls	X
UR1.2 Continuous risk assessment/certification	X

Table 62: NeSSoS – Update of User Requirements specifications

7.8.1.3 Software Requirements

Table 63 and Table 64 show an update of the SW Requirements that were defined for the NeSSoS tool in D5.1.

Software Requirements	No change	Add	Comments
SR1 A stand-alone on-line tool	X		
SR2 Identification of (additional) countermeasures		X	Missing in D5.1
SR3 Continuous assessment		X	Missing in D5.1

Table 63: NeSSoS - Update of SW Requirements specifications

SR2	Identification of (additional) countermeasures
Description	The tool is able to propose additional countermeasures to strengthen the security. The selection of these countermeasures is performed in a cost-efficient way.
Actors	Analyst, system owner
Basic Flow	First, risk assessment is performed with NeSSoS tool. Then, the analyst provides the cost limit and adjust the input values for the algorithm (if required). The tool evaluates various options and proposes a list of suggested security controls.
SR3	Continuous assessment
Description	Risk is re-assessed on the fly once objective information on the system settings is provided from verification/monitoring module.
Actors	The system, verification/monitoring tool.
Basic Flow	After risk assessment with the tool, NeSSoS is set to wait for updates. A verification/monitoring tool performs its analysis and sends the result to the NeSSoS tool. The NeSSoS tool “translates” the results of the analysis into values for risk assessment (e.g., attack probabilities) and re-evaluates the risk assessment results. The results are provided to the interested entity.

Table 64: NeSSoS – Changes in SW requirements specifications

The NeSSoS tool has been upgraded with the possibility of selecting additional countermeasures. Now, not only can a client evaluate its risks, but also look for possible improvements to increase its protection and decrease risks.

The work is currently on the integration of the risk assessment tool with real-time information coming from some verification modules. Once this information is added to the tool, the risk, previously computed only using the inputs from the client, is re-evaluated taking into account the monitored information.

7.8.2 Functional Specifications

The NeSSoS tool consists of the following components:

- **User interface**. A web-based GUI for the user to insert the information about the system, as well as for receiving the results of risk assessment.
- **Risk computation unit**. The core unit which computes (and re-computes) the risks and identifies suggested countermeasures.
- **Communication unit**. A unit that manages communication of machine-readable information (e.g., receiving it from a monitoring module and sending it to a risk consumer module).
- **Database**. A database with the expert knowledge stored and used for simplifying the analysis.

In short, the tool is to work as follows (see Figure 67). A user (e.g., Risk Analyst) enters the required data (the information about available security controls, key cyber assets and expected impact). The *user interface* passes this information to the *risk computation* unit, which, with the help of the knowledge stored in the database, identifies relevant threats and compute risk levels. This information is provided to the user through the user interface. If the tool is to be used for continuous assessment, the *risk computation* unit triggers the communication unit to send the aggregated risk information in a machine-readable format to any risk consumer module (e.g., a tool working on behalf of risk analyser). At this point, a *monitoring* module must be set up (based on the information previously generated by the NeSSoS tool (e.g., credentials and tokens for access, the ID for the system under evaluation, etc.). Once the monitoring module provides the up to date information about the state of security practices to the communication unit, risk is re-computed and the updated risk results are provided to the *risk consumer* module.

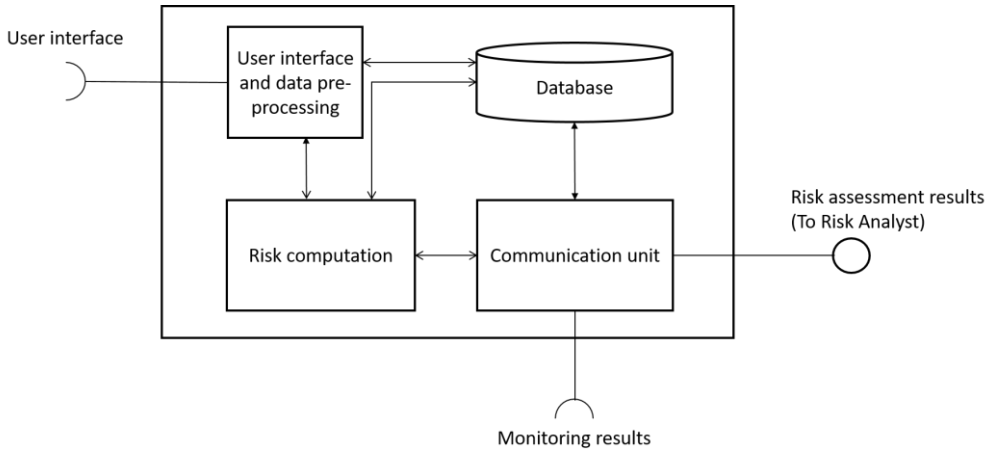

Figure 67: NeSSoS - Risk Assessment Architecture

7.8.3 Development roadmap

Use Case	Architecture components	Realisation	Involved partners
UC1	NeSSoS tool	We are implementing the NeSSoS tool to evaluate cyber security risks of the system and propose additional security controls.	CNR

Table 65: NeSSoS – Development Roadmap

The tool will provide the following functionality:

- Calculation of risks per threats using the input provided by the user (i.e., information about assets and security controls implemented)
    - The user is asked to answer a questionnaire containing questions on various aspects of security (based on ISO 27002 standard) The user should also provide the list of the key assets, their amount and expected loss in case of confidentiality, integrity or availability loss.
- Selection of the set of additional controls which more effectively reduce the overall risk and fit the budget limit.
    - The user is asked to provide the budget limit to be spent on additional security controls. The tool will automatically select the set of additional controls which once installed will reduce the risk better than any other set. This is a type of the cost-benefit analysis provided by the tool. The cost of controls is pre-defined but could be adjusted by the user if needed.
- Monitoring the correctness of the declared input (with external tools) and updating risk values according to the objective information
    - Risk computation is based on the information provided by the user. This information could be wrong, imprecise or not up to date. Monitoring tools installed in the assessed system should provide the objective information to the NeSSoS tool about the current state of security controls, and the corresponding measures in the risk assessment will be made.

7.8.4 Software verification and validation plan

SR id	Description	Verification method	Demonstration scenario
SR1	We develop the tool as an on-line service available through human-friendly GUI.	Simulation-based	e-Government (Vertical 2)
SR2	We develop a functionality based on security configuration optimisation by a Genetic Algorithm.	Random (Monte-Carlo) verification Simulation-based	e-Government (Vertical 2)
SR3	We implement machine-accessible interfaces which allow accessing and receiving (also at run-time) risk values.	Machine accessibility simulation	e-Government (Vertical 2)

Table 66: NeSSoS Tool – Demo scenarios and verification methods

Simulation-Based verification process

Input: The input is to be provided by the system owner (or by an analyst on behalf of the system owner). The input includes: the detailed information about implemented security controls and the parametrised list of assets.

Output: Simulation Results.

Test Procedure: The procedure is as follows:

1. We provide the parameters required by the NeSSoS tool.
2. We execute the NeSSoS tool to compute risk values.
3. We evaluate the results by checking if the results correspond to the expected ones by the scenario.
4. If the values are considered acceptable this is counted as success, otherwise we investigate the problem and adjust the weights accordingly.

Random (Monte-Carlo) verification process

Input: In addition to the parameters required for risk computation, the budget limit is provided, as well as the costs of controls are verified and updated by the user.

Output: Simulation Results.

Test Procedure: The procedure is as follows:

1. We insert budget limit to the tool.
2. The tool generates a set of additional controls, which are considered as the “best” now.
3. We select several other sets of controls withing the provided budget
4. The risk assessment is performed with the NeSSoS tool to compute risk values.
5. If the result (the overall risk + the overall cost of controls) is lower than the one predicted for the “best” set previously, the tool fails to detect the optimal set. Otherwise, we count this as success.
6. The procedure is repeated more than 20 times. If the failure rate is less than 1 out of 10 the tool passes the test.
    a. The GA algorithm is an approximate method, i.e., it may fail to find the global minimum.

Machine-accessibility simulation verification process

Input: This test will focus on machine-to-machine interaction. For this testing some modules simulating monitoring results flow are to be developed. First the user provides the data to set up the risk assessment practices. Then, the monitoring tool starts providing generated “monitoring results”. Another simulated module is required to receive updated risk levels.

Output: Simulation Results.

Test Procedure: The procedure is as follows:

1. We provide input data to the NeSSoS tool to conduct risk assessment.
2. Several sequences of monitored data are pre-set, which can be split into the following two sets:
    a. “Good” sequence: consisting of the monitored values corresponding to the input data from step 1 (or better).
    b. Bad sequence: consisting of the monitored values which provide the evidence that some of the controls from the input from step 1 are not in place (or do not function as declared).
3. The module simulating a monitoring engine starts providing data through the available interface.
4. Risk level is re-computed and sent to a simulated receiving module.
5. The risk assessment is performed with the NeSSoS tool to compute risk values.
6. The test is considered to be passed if:
    a. The NeSSoS tool provides the updates to the simulated receiving module as specified (e.g., once in an hour).

b. The risk values do not change if the “good” sequence of inputs is provided.
c. The risk values worsen if the “bad” sequence of inputs is provided.

The result returned by the tool should be evaluated by the system owners whether they see them as valid predictions or not.

7.9 OpenCert (OC) – TEC

There have been several attempts to synergise safety and security as assurance qualities for mission-critical cyber-physical systems. Several models exist, which seek to demonstrate the extensibility of the “failure engineering” approach which underpins system safety assurance to a “threat engineering” approach for assuring security. Work undertaken at the US Software Engineering Institute, Carnegie-Mellon University [121] proposed a model of conceptual commonalities between safety and security. The SAFSEC model [122] proposed a similar series of commonalities. The principal driver here is to support the reuse of evidence produced for the assurance of the system in terms of one of the criteria– perhaps with minimal changes – to support an assurance claim relating to the other criterion. For example, both safety and security rely on cause-effect models, such as fault trees or attack trees. Such reuse offers considerable cost-time benefits, if successfully achieved. OpenCert includes a Common Assurance & Certification Metamodel (CACM) [123] to resolve the inconsistencies in terminology across the target domains, to facilitate mappings – where possible – between assurance concepts across standards and to support informed reuse of safety/security assets within and across domains.

OpenCert is an open product and process assurance/certification management tool to support the compliance assessment and certification of Cyber- Physical Systems (CPS) spanning the largest safety and security-critical industrial markets, such as aerospace, space, railway, manufacturing, energy and health. OpenCert supports a number of features, including Standards & Regulations Information Management, Assurance Project Management concerned with the development of assurance cases and evidence management, Cross/intra-domain Reuse of assurance assets, Compliance Management, and Modular and Incremental Certification.

OpenCert can be downloaded as a stand-alone application. It is part of the Eclipse Foundation.

More information about OpenCert is available at: https://www.eclipse.org/opencert/

7.9.1 Requirements Description

7.9.1.1 Use cases

Table 67 shows an update of the Use Cases that were defined for the OpenCert tool in D5.1.

Use Cases	No change
UC1 Support the Safety and Security compliance assessment and certification of the platooning scenario	X

Table 67: OpenCert tool - Update of Use Cases specifications

7.9.1.2 User Requirements

Table 68 and Table 69 show an update of the User Requirements that were defined for OpenCert in D5.1.

User Requirements	Add	Comments
UR1.1 Digitalization of the standards	X	Missing in D5.1
UR1.2 Application of the standards	X	Missing in D5.1
UR1.3 Addition of evidences and Safety/Security trade-off	X	Missing in D5.1

Table 68: OpenCert - Update of User Requirements specifications

UR1.1	Digitalization of the standards
Description	Digitalization of Safety and Security standards in a graphical way.
Actors	Safety and Security Engineer
UR1.2	Application of the standards
Description	Compliance with the standard in all phases of the life cycle
Actors	Safety and Security Engineer
UR1.3	Addition of evidences and Safety/Security trade-off
Description	Inclusion of the results of standards requirements and comparison between safety and security on these results
Actors	Safety Engineer, Security Engineer

Table 69: OpenCert – Changes in User Requirements specifications

7.9.1.3 Software Requirements

Table 70 and Table 71 show an update of the SW Requirements that were defined for OpenCert in D5.1.

Software Requirements	Add	Comments
SR1 Create a Reference Framework for the ISO 26262 Safety standard and the SAE J3061 Security standard	X	Missing in D5.1
SR2 Create an Assurance Project	X	Missing in D5.1
SR3 Add evidences to the Assurance Project	X	Missing in D5.1
SR4 Create Assurance Case (trade-off Safety/Security)	X	Missing in D5.1

Table 70: OpenCert - Update of SW Requirements specifications

SR1	Create a Reference Framework for the ISO 26262 Safety standard and the SAE J3061 Security standard
Description	Digitalization of the ISO 26262 Safety standard and the SAE J3061 Security standard.
Actors	Safety Engineer and Security Engineer
Basic Flow	-
SR2	Create an Assurance Project
Description	The engineer selects the relevant parts of a standard depending on the criticality level or applicability level.
Actors	Safety Engineer, Security Engineer
Basic Flow	SR1→SR2
SR3	Add evidences to the Assurance Project
Description	The engineer can manage all the evidences in an Assurance Project by doing traceability management and impact analysis.
Actors	Safety Engineer, Security Engineer
Basic Flow	SR2→SR3
SR4	Create Assurance Case
Description	The engineer arguments, supported by evidences, that a system is acceptable safe and/or secure for a specific application in diverse scenarios, thus, the Safety Case will allow to perform co-assessment between safety and security.

Actors	Safety Engineer, Security Engineer
Basic Flow	SR2→SR4

Table 71: OpenCert – Changes in SW requirements specifications

7.9.2 Functional Specifications

At high-level, OpenCert is divided in 8 functional groups, as shown in Figure 68, where the functional groups that are involved in the Connected Car Platooning scenario (Vertical 1) are marked with the SPARTA's project logo.

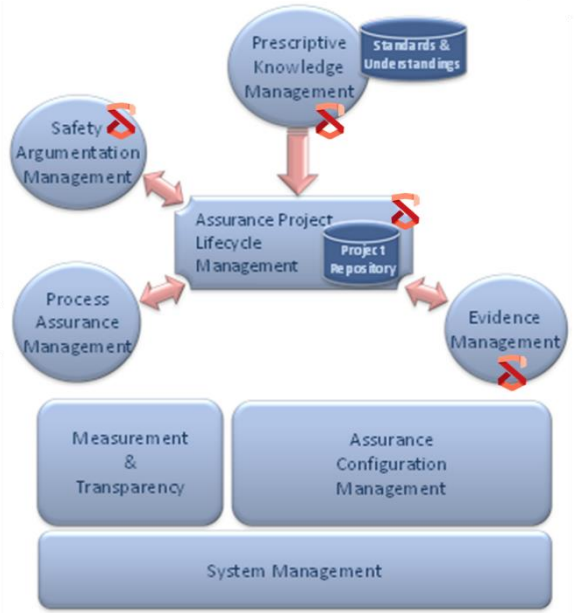

Figure 68: Functional decomposition for the OpenCert platform

Table 72 summarizes the definition of each of the OpenCert functional groups:

Functionality Group	Description
Prescriptive Knowledge Management	Functionality related to the management (edition, search, transfer, etc.) of standards information as well as any other information derived from them, such as interpretations about intents, mapping between standards, etc. This functional group maintains a knowledge database about "standards & understandings", which can be consulted by other OpenCert functionalities.
Assurance Project Lifecycle Management	This functionality factorizes aspects such as the creation of safety assurance projects locally in OpenCert and any project baseline information that may be shared by the different functional modules. This module manages a "project repository", which can be accessed by the other OpenCert modules.
Safety Argumentation Management	This group manages argumentation information in a modular fashion. It also includes mechanisms to support compositional safety assurance, and assurance patterns management.
Process Assurance Management	This functionality group handles every activity related to the specification, execution and validation of safety assurance processes in connection with engineering processes. It also manages compliance information related to functional safety standards. This module should be integrated with process-related tools managed by companies (ALM/PLMs, process workflows, etc.)

Functionality Group	Description
Evidence Management	This module manages the full life-cycle of evidences and evidence chains. This includes evidence traceability management and impact analysis. In addition, this module is in charge of communicating with external engineering tools (requirements management, implementation, V&V, etc.)
Measurement and Transparency	This is an infrastructure functional module. It supports metrics and estimation management related to information from the other modules.
Assurance Configuration Management	This is an infrastructure functional module. This includes functionality for traceability management, change management and impact analysis.
System Management	Includes generic functionality for security, permissions, reports, etc.

Table 72: OpenCert Functional groups

7.9.3 Development roadmap

OpenCert will be applied in its current version and with its current functionalities in the Connected Car Platooning use case, specifically in scenario 4 (see Section 5.2.4).

OpenCert will be used on the whole life cycle to help comply with the two standards (ISO 26262 and SAE J3061), including the evidences that will be collected and stored in a structured way. Finally, based on the evidence stored, it will be verified through arguments that the system is acceptable from both a Safety and Security perspective.

Use Case	Architecture components	Realisation	Involved partners
UC1	OpenCert	Connected Car vertical, scenario 4. Based on the digitalization of the standards proposed for Vertical 1 and the creation of Assurance projects with evidence management and Assurance cases.	TEC

Table 73: OpenCert – Development Roadmap

7.9.4 Software verification and validation plan

SR id	Description	Verification method	Demonstration scenario
SR1	Use OpenCert to create an Assurance Case that presents safety and security arguments.	Verification by means of the scenario defined in Section 5.2.4 and in D5.3, and also by the CAPE tools integration pipeline (Section 5.4)	Connected Car (Vertical 1), scenario 4
SR2			
SR3			
SR4			

Table 74: OpenCert – Demo scenarios and verification methods

Scenario-based verification process

Input: Digitalization of the Safety/Security standards, Assurance project with evidences.

Output: Assurance Case.

Test Procedure: The verification process will check that an Assurance Case is created by adding argumentations using the Goal Structuring Notation (GSN) in a graphical notation for presenting the structure of. The Assurance Case acts primarily as a communication means to describe how a particular claim has been shown to be true by means of evidence.

7.10 Project KB (KB) – SAP

Project KB represents an open and collaborative knowledge base with code-level information about security vulnerabilities in open source projects. It comprises a human- and machine-readable data format to express so-called statements, a dataset of several hundreds of statements for known vulnerabilities as well as the necessary tooling to create, publish and consume those statements to/from one or multiple dataset.

The overall motivation for Project KB has been sketched in deliverable D5.1 [1] as part of Eclipse Steady. In summary, Project KB addresses the problem that there are no public databases with code-level information about open source vulnerabilities. Existing vulnerability databases are either proprietary with limited access, or do not contain information required to link vulnerabilities to the actual code base of the affected project.

In contrast to private datasets, public ones available to the computer science and computer security research community can broadly foster the development of innovative solutions. This particularly applies to the domain of machine learning and is underlined by the increased recognition of such datasets by top research conferences such as the IEEE/ACM International Conference on Mining Software Repositories (MSR), which offer dedicated dataset tracks.

In comparison to previous work on public vulnerability databases [124] [125], Project KB adopts PURL[19] as a means to uniquely identify (non-)affected component versions, introduces the notion of conflicts that inevitably occur in distributed maintenance scenarios and offers a tool to facilitate the creation, publication and consumption of vulnerability information. Similarly to [126], Project KB links vulnerabilities to the actual code of the affected open source component in order to support automated program analyses. While Project KB has been introduced in D5.1 in the context of Eclipse Steady, it became clear during the development of the fundamental concepts related to an open and distributed vulnerability database, that it makes sense to continue this effort independently. Eclipse Steady is just one of many potential downstream users of such a database with security-related information about open source software.

Accordingly, Project KB is now described in a dedicated section of this deliverable. What remains in the section of Eclipse Steady is the development of a component consuming the information from Project KB.

Note that Project KB comprises the definition of a human- and machine-readable plain-text format to express so-called vulnerability statements, a tool called kaybee to create, publish and consume such statements, and a dataset with hundreds of statements for known vulnerabilities in open source projects.

Project KB has been open sourced itself and is maintained at: https://github.com/sap/project-kb.

7.10.1 Requirements Description

7.10.1.1 Use cases

Table 75 and Table 76 show an update of the Use Cases that were defined for the KB tool in D5.1.

Use Cases	Add	Comments
UC1 Create and share vulnerability information about an open source project	X	Missing in D5.1
UC2 Consume vulnerability information about one or more open source projects	X	Missing in D5.1

Table 75: Project KB – Update of Use Cases specifications

[19] https://github.com/package-url/purl-spec

UC1	Create and share vulnerability information about an open source project
Description	Actors with code-level information about open source vulnerabilities, thus, fix commits or (non)affected library identifiers, share this information in the form of statements with other interested parties (public or private).
Actors	Security researcher, Maintainer of open source project
Basic Flow	Actors create statements, manually or with help of the tool, and publish it to a public or private Git repository.
UC2	Consume vulnerability information about one or more open source projects
Description	Actors download statements from one or more public or private repositories and, if there are multiple statements for the same vulnerability, merge them into a consolidated statement.
Actors	Security researcher, developer
Basic Flow	Actors use the tool to consume download (clone) statements, and to merge them in their local file system.

Table 76: ProjectKB – Changes in Use Cases specifications

7.10.1.2 User Requirements

Table 77 and Table 78 show an update of the user requirements that were defined for the KB tool in D5.1.

User Requirements	Add	Comments
UR1.1 Create statements with security information in a standardized format	X	Missing in D5.1
UR1.2 Publish statements to a public or private repository	X	Missing in D5.1
UR2.1 Download and merge statements from one or multiple repositories	X	Missing in D5.1
UR2.2 Transform statements into other formats required by downstream users	X	Missing in D5.1

Table 77: Project KB - Update of User Requirements specifications

UR1.1	Create statements with security information in a standardized format
Description	The tool must support a standardized, human-readable format for security information about open source projects, so-called statements. To avoid overlap with existing standards and databases, the focus is on information about fix commits and affected libraries. The tool must support the creation and validation of such statements.
Actors	Security researcher, Maintainer of open source project
UR1.2	Publish statements to a public or private repository
Description	The tool must allow the upload and sharing of statements to public or private repositories such that other people can consume the information manually or programmatically.
Actors	Security researcher, Maintainer of open source project
UR2.1	Download and merge statements from one or multiple repositories
Description	The tool must support the download of statements from one or multiple, public or private repositories. If statements from different repositories have the same identifier, it must be possible to define a merge strategy.
Actors	Developer, Operator of Eclipse Steady
UR2.2	Transform statements into other formats required by downstream users
Description	It must be possible to transform downloaded (and potentially merged) statements into other formats, e.g., XML, in order to facilitate the consumption of security information by downstream users such as Eclipse Steady.
Actors	Developer, Operator of Eclipse Steady

Table 78: Project KB – Changes in User Requirements specifications

7.10.1.3 Software Requirements

Table 79 and Table 80 show an update of the SW requirements that were defined for the KB tool in D5.1.

Software Requirements	Add	Comments
SR1 Human-and machine-readable plain-text format	X	Missing in D5.1
SR2 Digital signature	X	Missing in D5.1
SR3 Public and private repositories	X	Missing in D5.1
SR4 Versioning	X	Missing in D5.1

Table 79: Project KB - Update of SW Requirements specifications

SR1	Human-and machine-readable plain-text format
Description	Statements must be human-and machine readable. It must be possible to create and modify statements using standard text editors.
Actors	Consumers and producers of statements
Basic Flow	Producers use standard text editors to create or modify statements, consumers read through statements in the browser and locally (after cloning repositories to the local file system)
SR2	Digital Signature
Description	It must be possible to sign changes to statements (creation/modification/deletion) such that consumers get assurance about the authorship of changes.
Actors	Consumers and producers
Basic Flow	Statement producers sign modifications using their private key. Statement consumers verify statement signatures using public key certificates. Potentially, statement consumers can skip the processing of statements that cannot be verified, or which come from unknown authors.
SR3	Public and Private Repositories
Description	To implement a distributed database with vulnerability information, it must be possible to publish to (and consume from) multiple statement repositories. Moreover, it must be possible to have public repositories, whose content is accessible to everyone, and private repositories, whose content is only accessible to a limited audience.
Actors	Consumers and producers
Basic Flow	Producers publish statements to one or more public and private repositories. Consumers read statements from one or more public and private repositories.
SR4	Versioning
Description	Statements must be versioned such that dates and authors of initial contributions and subsequent changes can be tracked.
Actors	Consumers and producers
Basic Flow	Producers and consumers consult the version history of statements in order to understand who created and modified statements and what exactly has been changed, e.g., the list of fix commits or the list of affected packages

Table 80: Project KB – Changes in SW requirements specifications

7.10.2 Functional Specifications

So-called statements express the knowledge (belief) of the issuer about a particular vulnerability in an open source project. Each statement issuer can maintain their own repository of statements, of which they keep full control. Alternatively, multiple issuers can share repositories. Consumers decide which issuers to trust and how to reconcile potential conflicts that can arise from consuming statements with identical identifiers from multiple sources.

Requirement SR1 has been implemented by defining a YAML format to store the following vulnerability information about open source projects:

- One or more textual descriptions (node `text`), similar to the description of CVE/NVD vulnerabilities
- One or more links to Web pages with additional information (node `links`)
- Fix commits through the reference of commit hashes in given source code repositories (node `fixes`), including the possibility to group commits, e.g., for different release branches.
- Vulnerable and non-vulnerable artefacts (node `artifacts` and its subnodes `id`, `reason`, `affected`). Note that it is possible to make positive and negative assertions about the affectedness of artefacts, thus, it is possible to state that given artefacts are not vulnerable. This is different from other standards such as the NVD, which only enumerates affected versions. The artefacts are identified using the PURL specification[20], which gains traction in open source ecosystems. The advantage of using PURL, in contrast to CPE identifiers used in CVEs, is that they map unambiguously to package identifiers in different open source package repositories such as npm or PyPI.

Note that the primary focus of this format is on fixing commits and affected artefacts, which are not at all or insufficiently covered by existing vulnerability databases such as the NVD. This focus allows to link vulnerabilities to the respective source code and the standards and formats used by developers to identify open source packages.

On the other side, the format does not include information that is well-covered already, e.g., CVSS severity ratings.

The following YAML statement illustrates the format at the example of vulnerability CVE-2014-0054 (additional affected artefacts have been omitted):

```
vulnerability_id: CVE-2014-0054

notes:

- text: 'The Jaxb2RootElementHttpMessageConverter in Spring MVC in Spring Framework before 3.2.8
and 4.0.0 before 4.0.2 does not disable external entity resolution, which allows remote attackers
to read arbitrary files, cause a denial of service, and conduct CSRF attacks via crafted XML, aka
an XML External Entity (XXE) issue.  NOTE: this vulnerability exists because of an incomplete fix
for CVE-2013-4152, CVE-2013-7315, and CVE-2013-6429.'

fixes:

- id: DEFAULT_BRANCH

  commits:

  - id: 1c5cab2a4069ec3239c531d741aeb07a434f521b

    repository: https://github.com/spring-projects/spring-framework.git

  - id: edba32b3093703d5e9ed42b5b8ec23ecc1998398

    repository: https://github.com/spring-projects/spring-framework.git

artifacts:

- id: pkg:maven/org.springframework/spring-web@4.3.0.RELEASE
```

[20] https://github.com/package-url/purl-spec

```
  reason: Reviewed manually
  affected: false
- id: pkg:maven/org.springframework/spring-oxm@3.1.2.RELEASE
  reason: Assessed with Eclipse Steady (AST_EQUALITY)
  affected: true
```

Requirements SR2-SR4 have been implemented by relying on the Git versioning control system.

As of version v1.7.9, Git can be used to GPG sign individual commits (SR2), which is important to ensure the identity of statement contributors. Note that the creation and verification of signatures is only optional.

Git repositories can be public or private (SR3), depending on where the Git server is hosted and how access is managed. Public repositories can be used to share vulnerability information with the general public. Private repositories can be used either to store complementary private information about public statements, e.g., descriptions or affected packages, or to store private statements about internal, non-public components.

The use of Git as underlying infrastructure also supports the versioning of statements (SR4). Changes to YAML statements are recorded with timestamp and author, and version differences can be analysed using Git's convenient diff functionality.

With the YAML format briefly described above and Git as underlying infrastructure, the kaybee tool will support the publication and consumption of security statements as illustrated in Figure 69.

Security researchers or project maintainers can create YAML statements and publish them to public or private Git repositories (`kaybee create` and `git add/commit/push`). Multiple contributors can use the same or different repositories.

Consumers need to configure the kaybee tool in order to specify and prioritize one or multiple sources (Git repositories) containing YAML statements. With `kaybee pull`, statements of all sources are copied to local replicas such that they can be merged using different conflict resolution strategies, e.g., considering the priority of the respective source or the digital signature of the statement author. Finally, the aggregated and potentially reconciled statements reside in a local folder.

Those statements can be exported to other formats, e.g., XML or Steady (`kaybee export`). The export is based on a simple templating mechanism such that the consideration of new target formats does not require any coding. The export to Steady, for instance, results in a bash script that checks-out all fix commits, creates some metadata data and, eventually, calls the kb-importer component developed as explained in Section 7.15.

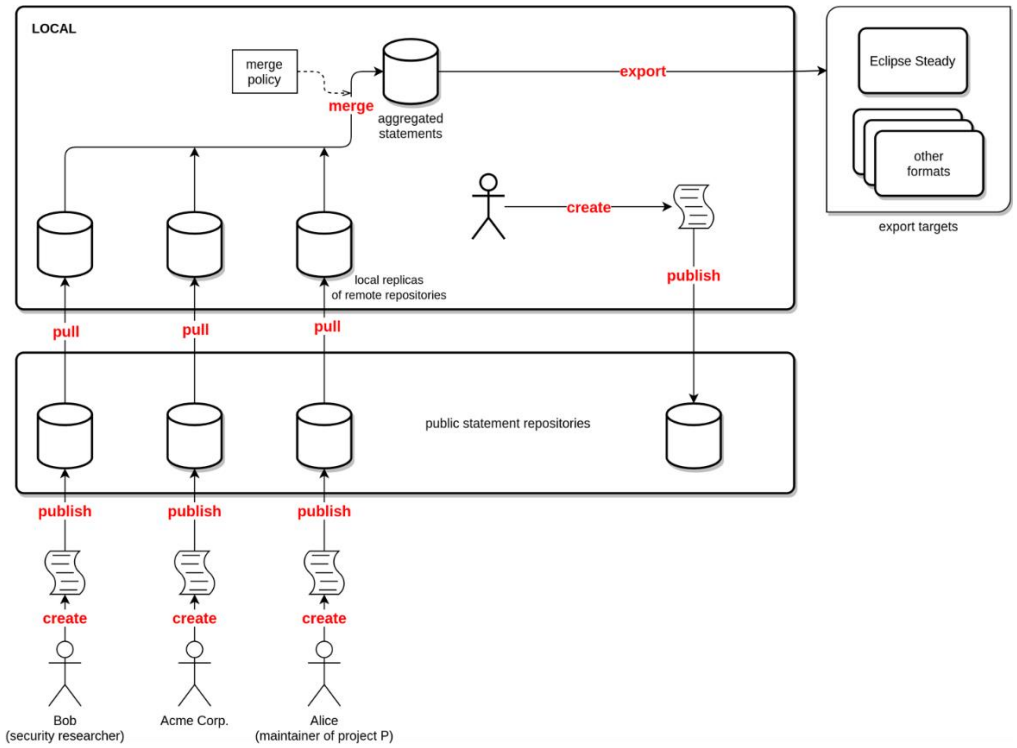

Figure 69: Project KB: Use-cases

7.10.3 Development roadmap

The format and the kaybee tool have already been implemented and open-sourced on GitHub[21]. Moreover, several hundreds of YAML statements have been published in a dedicated branch[22].

The verification of digital signatures and the use of signatures for content selection will be implemented in 2021 (SR2). Finally, the dataset with known supply chain attacks[23], maintained by the University of Bonn and SAP, will be used to generate corresponding YAML statements in Q1 2021.

7.10.4 Software verification validation plan

Integration tests will be done in the context of the e-Government use-case (on top of automated unit tests part of the actual tool). A test scenario and a test environment (comprising one or more Git repositories with test statements) must be created such that statements can be pulled from sources, merged, exported and loaded in the Steady database, which will be invoked during the CI/CD pipelines.

SR id	Description	Verification method	Demonstration scenario
SR1	Plain-text format	Integration test according to defined scenario	e-Government (Vertical 2)
SR2	Digital Signature		
SR3	Public and private repositories		
SR4	Versioning		

Table 81: Project KB – Demo scenarios and verification methods

[21] https://sap.github.io/project-kb/

[22] https://github.com/SAP/project-kb/tree/vulnerability-data

[23] https://github.com/dasfreak/Backstabbers-Knife-Collection

SR1-SR4 – Statement Publication and Consumption

Input: Commit in the source code repository of one of the dependencies of the SAML IdP, e.g., the GitHub repository of the Bouncycastle Java Cryptographic APIs[24].

Output: Aggregated statement in the local file system.

Test Procedure:

A random commit in the source code repository of one of the dependencies of the SAML IdP will be used to create a sample YAML statement. This YAML file will be signed and pushed to a test branch of some private Git repository (SR1-SR3).

Then, taking the perspective of a consumer, the statement will be pulled from this Git repository, the signature will be verified and the statement will be merged using the Project KB tool such that it is contained in the set of aggregated statements, which then can be exported into other formats.

During statement creation, different versions can be created and compared using the Git diff functionality (SR4).

7.11 Risk Assessment for Cyberphysical interconnected infrastructures (MRA) – NCSR

MRA is a stand-alone tool to introduce the cyber-physical elements in the critical infrastructure risk assessment. It can be used as a continuous high-level risk identification and appraisal on how systemic and cyber related risks can have an impact of the infrastructure's operation and service levels. MRA has been conceived as a flexible and customizable approach that is applicable on single infrastructure components and assets and also expandable to include the impacts on interconnected assets and domino effects.

Cyber Security Risk Assessment (CSRA) framework have main a mainstream practice ever since the exponential introduction of the cyber world in critical infrastructures. (CSRA) is the cornerstone element for risk-informed policies in CI [127], usually trying to provide evidence based responses to the following questions: a) Where is the origin and characteristics of threat, and how this may evolve over time?, b) What is the time and place and magnitude of occurrence of a cyber/physical event? d) What are systemic vulnerabilities across different dimensions (including governance and insiders)? e) What is the likelihood of a cyber / physical event? f) What are the expected or estimated service disruptions? g) How to establish risk-based defences?

Several existing frameworks and standards exist in the field such as the National Institute of Standards and Technology (NIST) Cybersecurity Framework (CSF), Cyber Security Evaluation Tool (CSET®), Cybersecurity Capability Maturity Model (C2M2), International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC) Standard 31010. [128] documented cyber threats to smart grid domains (e.g., distribution grid management, advanced metering infrastructure, demand response, etc.). In 2017, European standardization bodies published a report that identified the information assets and considered them in the risk assessment as part of mapping dependencies to vulnerabilities [129]. In the report, smart grid asset management is mapped based on domain (e.g., generation, transmission) and zone (e.g., process, field, station, etc.). In another report [130] the expert group categorized the assets based on their protection needs and classified them into two groups: smart cyber assets (e.g., advanced metering infrastructure or AMI, intelligent electronics devices or IED, supervisory control and data acquisition or SCADA, etc.) and grid cyber assets (energy management system or EMS, distribution management system or DMS, communication link, etc.).

[24] https://github.com/bcgit/bc-java

7.11.1 Requirements Description

7.11.1.1 Use cases

Table 82 and Table 83 show an update of the Use Cases that were defined for the MRA tool in D5.1.

Use Cases	No change	Modify	Comments
UC1 Cyber-attack with cascade impacts	X		The work has been focused on the impacts from the cyber to the physical domain with focus on the impacts on the service levels of the infrastructure
UC2 Continuous risk quantification		X	Added UC3 (below)

Table 82: MRA - Update of Use Cases specifications

UC3	Assets Attractiveness Assessment
Description	This module adds the assets attractiveness as a component of the cyber-physical risk assessment. Attractiveness is used to de-compose the likelihood element of risk.
Actors	Risk Managers / Security Officials
Basic Flow	The module takes elements from the profile of the case study and, specifically the identified assets and potential vulnerabilities. It proceeds with the estimation on the “attractiveness” of the identified assets as potential sites of attack, cumulatively accounting for possible different types

Table 83: MRA – Changes in Use Cases specifications

7.11.1.2 User Requirements

Table 84 shows an update of the User Requirements that were defined for the MRA tool in D5.1.

User Requirements	No change
UR1 Security profile of domain	X

Table 84: MRA - Update on User Requirements specifications

7.11.1.3 Software Requirements

Table 85 shows an update of the SW Requirements that were defined for the MRA tool in D5.1.

Software Requirements	No change	Comments
SR1 MRA stand-alone tool	X	Scripts presently in python language

Table 85: MRA - Update on SW Requirements specifications

7.11.2 Functional Specifications

The MRA tool is built by the following components (see Figure 69):

- A **user interface** that allows user to input information about the infrastructure, its assets and interconnections, their properties and potential vulnerabilities and other needed ancillary input.
- A **modelling component** that performs a cascade analysis of the assets and estimates risk.
- A **display element** that transfer outputs to users.
- A **database** storing all required / processed / produced information.

In brief, the tool works as described in the following lines. Users enter the required inputs (infrastructure assets, properties, interconnections, safeguards, potential impacts), which are stored in the database. The software passes the data to the modelling component and identifies:

- MRA.ID1. Identification of Potential threats.
- MRA.ID2. Determination of Attractive assets/processes
- MRA.ID3 Vulnerability Assessment
- MRA.ID4 Interconnections and potential cascade effects
- MRA.ID5 Analysis of Impacts (in the cyber and physical domains), including cascading effects
- MRA.ID6 Risk Assessment

This information is fed back to the user through the display element. If the tool can be extended for continuous risk assessment, the interfaces need to be customized to allow inputs from a machine-readable format.

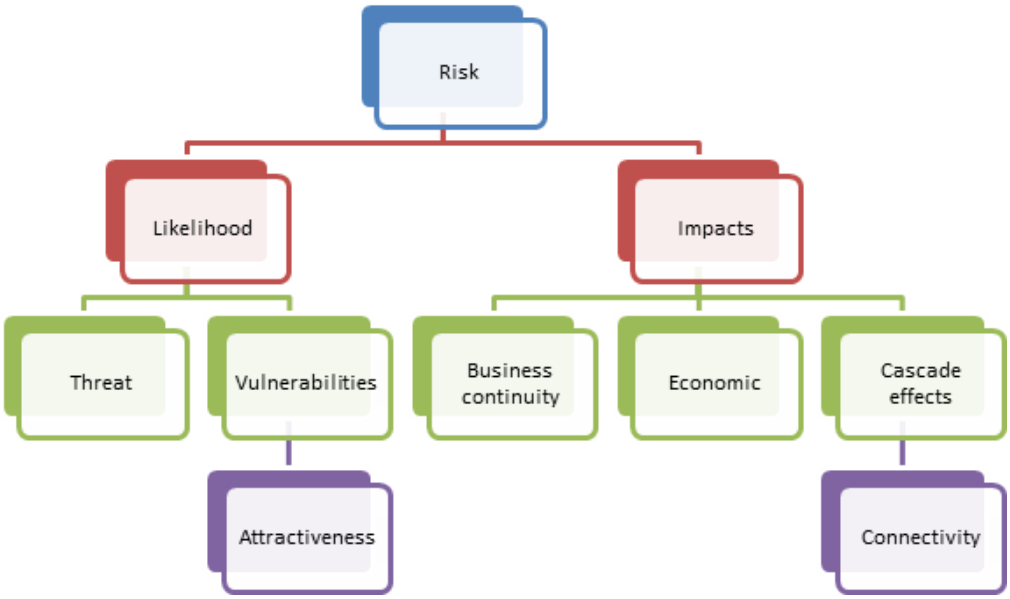

Figure 70: MRA domain elements

7.11.3 Development roadmap

The design, implementation and preliminary validation have been implemented. Initially NCSRD efforts has been placed on transforming the existing multi-hazard risk assessment framework of NCSRD to include the cyber-physical domain. Secondly, the development focus of the framework has been shifted to establish the asset attractiveness as a way of identifying the most “juicy” target in infrastructure assets. The attractiveness element has been selected as a simple, yet highly effective indicator of prioritizing assets at risk in highly dynamic environments.

The development phase included several different evaluations of potential indicators that are under consideration still to-date. Specific asset attributes linked to information from global vulnerabilities databases (e.g. Common Vulnerabilities and Exposures - CVE®) and infrastructures properties are tested.

A theoretical framework has been established, which is followed by programming in different platforms (python and initial version in Excel). The first version will be presented in python code in Q2 of 2021.

Use Case	Architecture components	Realisation	Involved partners
UC1	Interface, database, modelling component, display	Use the tool to define risks in cyber-physical systems	NCSRD
UC2	Infrastructure risk attractiveness modelling component	Use the tool and define infrastructure attractive assets as targets	NCSRD

Table 86: MRA – Development Roadmap

7.11.4 Software verification and validation plan

The MRA verification method within SPARTA will be conducted as a two-step process, involving the stand-alone version. The utilised data will come from existing datasets within NCSRD augmented with the Security profile of the project verticals.

Step1: Use data of NCSRD infrastructure assets, from the ongoing NCSRD internal security vulnerability assessment process, that expand in the cyber-physical domain. Check that the developed software when fed with reference NCSRD internal data properly assigns the identified attractiveness level.

Step2: Expand attractiveness definition to capture the unique challenges of "vertical 1 (platooning)" using the established security profile (D5.1). Provide first internal validation with subject matter experts from NCSRD and then second verification with external experts (e.g. project partners, national cybersecurity experts).

SR id	Description	Verification method	Demonstration scenario
SR1	Stand-alone tool	Check if the MRA tool provides the risk levels as identified. Use the tool through the web interface.	Connected Car (vertical 1) - security profile

Table 87: MRA Tool – Demo scenarios and verification methods

7.12 Sabotage (SB) – TEC

Sabotage is a model-driven and simulation-based fault injection tool built upon the FARM model [131] that allows to accomplish an early evaluation dependability evaluation of safety-critical systems. The FARM model is an effective way to characterize a fault injection environment. The FARM sets constitute the major attributes that can be used to fully characterized fault-injection (faults F, activations A, readouts R and derived measures M). Given the FARM model, a fault injection campaign is a collection of experiments, each requiring the injection of a fault f from the set F while the system is exercised with an activation trajectory a selected from A in a workload w from W. The set of measures M is obtained elaborating the set of readouts R gathered during each experiment.

One of the techniques with more relevant benefits is the so-called Simulation-based Fault Injection which allows full observability and controllability. To get meaningful and accurate FI experiment results, a representative fault model is required. Different types of faults can appear depending on its nature during the system design process or during its operational life.

This technique is novel technique where hardly any research has been done. It is beneficial to use simulation technologies before the construction of physical models, as the build-up of virtual model concepts need fewer resources than the preparation of a physical prototype. These techniques also highly recommended across the verification and validation phases of the V-Cycle development process.

The Sabotage tool is based on Eclipse combined with Matlab/Simulink and can be used in an early assessment of safety-critical in different areas such as automotive or robotics. The integrated Simulation fault injection technique allows the construction of a simulation model of the system under analysis. Thanks to this simulated system the verification and validation is achieved during its early development phases. The framework sets up, configures, executes and analyses the simulation

results. The tool includes a fault model library and it is possible to connect to virtual environments such as a virtual vehicle or a robot.

More information about Sabotage is available at: https://www.cyberssbytecnalia.com/node/271

7.12.1 Requirements Description

7.12.1.1 Use cases

Table 88 and Table 89 show an update of the Use Cases that were defined for Sabotage in D5.1.

Use Cases	Modify
UC1 Fault-injection and analysis of faulty scenarios with simulation	X

Table 88: Sabotage - Update of Use Cases specifications

The definition of the UC1 has been updated as follows:

UC1	Fault-injection and analysis of faulty scenarios with simulation
Description	The Sabotage tool will be applied in the Platooning scenario (see Section 5.2.5). It will be used to simulate how a fault, originated from a random hardware fault or cyber-attack, can affect the vehicle behaviour by changing the velocity to an abnormal value. Each vehicle has integrated different measures, e.g. plausibility checks, thus, Sabotage will verify those requirements and measures to ensure the system implements appropriate mechanisms to prevent the violation of the safety properties. The effectiveness (detection and/or recovery of errors) of the measures can be analysed by injecting different faults in the developed plausibility checks.
Actors	Safety Engineer
Basic Flow	The following steps would be followed: • Model the countermeasure, e.g. plausibility check • Define the different faulty scenarios • Perform Simulation-based Fault Injection • Verify if the mechanisms are correctly implemented and if enough level of safety has been achieved. If not, do the necessary model modifications and perform the simulation again as many times as needed.

Table 89: Sabotage – Changes in Use Cases specifications

7.12.1.2 User Requirements

Table 90 and Table 91 show an update of the User Requirements that were defined for Sabotage in D5.1.

User Requirements	Add	Comments
UR1.1 Define the different faulty scenarios	X	Missing in D5.1
UR1.2 Perform Simulation-based Fault Injection	X	Missing in D5.1
UR1.3 Verification and Validation	X	Missing in D5.1

Table 90: Sabotage - Update on User Requirements specifications

UR1.1	Define the different faulty scenarios
Description	The engineer can define different test cases with one or more faults in each of them specifying the trigger time, duration and value of every single fault.
Actors	Safety Engineer

UR1.2	Perform Simulation-based Fault Injection
Description	Once the test cases are filled, the safety engineer runs the simulations.
Actors	Safety Engineer
UR1.3	Verification and Validation
Description	The engineer can visualize all the simulations and compare to each other to analyse the behaviour of the safety mechanisms and/or secure countermeasures.
Actors	Safety Engineer

Table 91: Sabotage – Changes in User Requirements specifications

7.12.1.3 Software Requirements

Table 92 and Table 93 show an update of the SW Requirements that were defined for Sabotage in D5.1.

Software Requirements	Add	Comments
SR1 Configuration of fault injection experiments	X	Missing in D5.1
SR2 Automation of the experiments	X	Missing in D5.1

Table 92: Sabotage - Update of SW Requirements specifications

SR1	Configuration of fault injection experiments
Description	Sabotage helps to specify different failures within a model-based system design performed in Eclipse environment using the Eclipse modelling framework (EMF) in combination with Massif[25], which converts MathLab Simulink models to EMF, and supports the specification of failures with an intuitive fault list.
Actors	EMF, Massif
Basic Flow	-
SR2	Automation of the experiments
Description	The template language Xtend is applied to generate Matlab and C code. Xtend technology includes a template language to generate code. As explained in D5.1 [1], Sabotage creates the fault-free simulation and one or more faulty simulations. The Xtend technology is employed to export the resulting C code that generates each fault, Matlab code to create a fault-free and a faulty system, and Matlab code to execute the experiments and visualise the results. Xtend allows the creation of code replacing the dynamic areas of the template with information from a metamodel.
Actors	EMF, Xtend
Basic Flow	SR1→SR2

Table 93: Sabotage – Changes in SW requirements specifications

7.12.2 Functional Specifications

At high-level, Sabotage is divided into three functional groups: Workload Generator, Fault Injector and Monitor (see Figure 71).

[25] https://github.com/viatra/massif

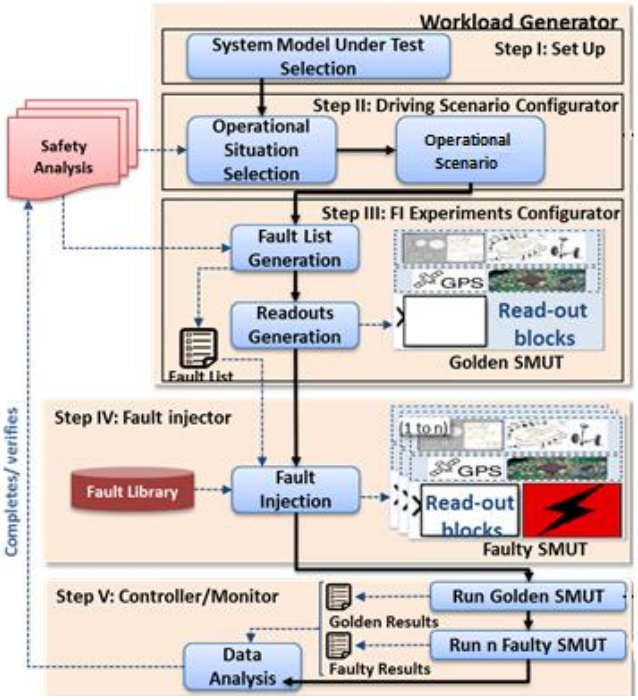

Figure 71: Sabotage functional groups

Table 94 summarizes the definition of each of the Sabotage functional groups.

Functionality Group	Description
Workload Generator	This block selects the system, chooses the most appropriate scenario, which represents the operational situation, and configures fault injection experiments. The basis for specifying the operational situations are driven by safety analysis. Afterwards, the fault injection experiments configuration gives the designer the possibility of creating the fault list and selecting where to monitor fault injection experiments by including signal monitors or readout blocks. The main strategy is to identify a representative and optimal fault subset to reproduce target system malfunctions or failure modes.
Fault Injector	The fault list is used to produce a Faulty system only in terms of reproducible and prearranged fault models by including saboteur blocks. Fault models are characterised by a type (e.g. omission, frozen, delay, invert, oscillation or random), target location, injection triggering (e.g. time), and duration. In order to create a Faulty system, the Fault Injector injects an additional saboteur model block per fault entry from the Fault List. Moreover, the injected block is fulfilled with information coming from a fault model template library. Saboteurs are extra components added as part of the model-based design for the sole purpose of Fault Injection experiments.
Monitor	After performing the configuration of the fault injection scenarios and creating the required amount of Faulty systems, the Monitor invokes the simulator. It tracks the execution flow of the fault free system and Faulty simulations. The Monitor compares fault free system and Faulty system results by the data analysis activity.

Table 94: Sabotage functional groups

7.12.3 Development roadmap

Sabotage will be used in its current version and with its current functionalities in the Platooning scenario (Vertical 1.

Use Case	Architecture components	Realisation	Involved partners
UC1	Sabotage MatLab	Connect Car vertical, scenario 5. Based on the countermeasures defined for Vertical 1 that have been developed by Tecnalia.	TEC

Table 95: Sabotage – Development Roadmap

The Sabotage tool will be applied in the Platooning scenario (Vertical 1). Some different simulations will be deployed on a sensor-based plausibility check algorithm adding several faults originated from random hardware fault or cyber-attacks, to see how it can affect the vehicle behaviour by changing the velocity to an abnormal value. Each simulation will be elaborated through a fault list which contains some saboteurs and signal monitors. The results reflect the effectiveness (detection and/or recovery of errors) of the plausibility check helping to verify the corresponding requirement defined in the Vertical 1.

7.12.4 Software verification and validation plan

SR id	Description	Verification method	Demonstration scenario
SR1 **SR2**	Simulation-based Fault injection and analysis of faulty scenarios	Verification by means of the scenario 5 defined in Section 5.2.5 and in D5.3.	Connected Car vertical, scenario 5

Table 96: Sabotage – Demo scenarios and verification methods

Scenario-based verification process

Input: Fault list.

Output: Effectiveness of the plausibility check.

Test Procedure: The verification of the configuration of experiments will be done using a called Fault List. The Fault list will include the definition of fault locations, fault injection times, fault durations, and the input data for the system.

The automation of experiments will be done using Xtend technology, which is a template language specialized generating code, in this case, in MatLab code to execute the experiments and visualise the results.

7.13 SafeCommit (SF) – UNILU

The SafeCommit tool, also called Commit Classifier, aims at automatically detecting vulnerability introducing commits (also referred as patches for sake of simplification) in Continuous Integration Ecosystem. SafeCommit is built on top of AI techniques relying on innovative features and advanced patch representation learning. Systematically and automatically identifying vulnerability introducing patches once a commit is contributed to a code base is of the utmost importance: (1) To reduce the number of vulnerabilities in a software code base; (2) To incite maintainers to quickly reject the relevant changes. The proposed tool aims at being integrated into real-world software maintenance and usage workflows. The objective is to carry out a live study in order to collect practitioner feedback for iteratively improving the tuning of the research output, towards an effective technology transfer.

The possibility of automatically finding vulnerabilities in code bases has long been identified by researchers as a worthy investigation target. Related works rely on various type of techniques such as static analysis, symbolic execution, dynamic analysis, machine learning, etc. However, only few approaches have been proposed to detect vulnerabilities at commit level [132][133][134].

VCCFinder [134] is a seminal approach in the literature and probably the most popular approach that builds on machine learning to automatically detect whether an incoming commit will introduce some vulnerabilities. VCCFinder has brought two key innovations: (1) VCCFinder was the first approach where the focus is made on code commits, which are “the natural unit upon which to check

whether new code is dangerous" allowing to implement early detection of vulnerabilities just when they are being introduced; (2) the wealth of metadata on the context of who wrote the code and how it is committed is leveraged together with the code analysis to refine the detection of vulnerabilities. SafeCommit is built on this idea by proposing a new feature set as well as a new technique to overcome the problem of unbalanced datasets.

7.13.1 Requirements Description

7.13.1.1 Use cases

Table 97 shows an update of the Use Cases that were defined for the SafeCommit tool in D5.1.

Use Cases	No change	Remove
UC1 Vulnerability Introducing Commit/Patch	X	
UC2 Vulnerability Fixing Commit/Patch		X

Table 97: SafeCommit - Update of Use Cases specifications

We decided to not consider the use case related to the detection of commits that fix vulnerabilities. This decision has been motivated by the fact that another SPARTA partner (SAP) has already developed such a tool (this tool is not listed in this document). Rather than competing, both SAP and UNILU decided to join force. Together they can propose a generic tool aiming at detecting security relevant commits, i.e., commits that either introduce or fix a vulnerability.

7.13.1.2 User Requirements

Table 98 and Table 99 show an update of the User Requirements that were defined for the SafeCommit tool in D5.1.

User Requirements	Add	Comments
UR1.1 Software developer commit checking	X	Missing in D5.1
UR1.2 Repository maintainer commit checking	X	Missing in D5.1

Table 98: SafeCommit - Update of User Requirements specifications

UR1.1	Commit Security Relevance Checking
Description	A software developer checks if his/her commit introduces a vulnerability in the repository code base.
Actors	Software developer
Basic Flow	Just before committing their modifications (i.e. a commit) into a code base (i.e., a version control repository such as GIT), developers can check if their modifications introduce a vulnerability. In this way, SafeCommit allows to avoid the introduction of vulnerabilities at the very early stage of software development.
UR1.2	Repository maintainer commit checking
Description	A repository maintainer checks if the commit of a developer does not contain any vulnerability before propagating the commit into the repository.
Actors	Repository maintainer
Basic Flow	In a typical scenario, a developer proposes changes bundled as a software patch by pushing a commit (i.e., patch + description of changes) which is analysed by the project maintainer, or a chain of maintainers, who eventually reject or apply the changes to the master branch. With SafeCommit, maintainers will be immediately informed that a vulnerability introducing commit has been proposed, and thus, they can reject the change.

Table 99: SafeCommit – Changes in User Requirements specifications

7.13.1.3 Software Requirements

Table 100 and Table 101 show an update of the SW Requirements that were defined for the SafeCommit tool in D5.1.

Software Requirements	Add	Comments
SR1.1 High precision and recall	X	Missing in D5.1

Table 100: SafeCommit - Update of SW Requirements specifications

SR1.1	High precision and recall
Description	SafeCommit should ensure both: • High precision, a predicted vulnerable commit should actually be a vulnerability introduction commit • High recall, most of the vulnerability introducing commits should be detected
Actors	Researcher

Table 101: SafeCommit – Changes on SW requirements specifications

7.13.2 Functional Specifications

Note that SafeCommit will be developed in the course of the SPARTA project.

SafeCommit will use a machine-learning based approach as described in Figure 72**:** Overall SafeCommit Process. In particular, SafeCommit will address a binary classification problem of distinguishing vulnerability introducing patches from other patches. As any classification problem, well-labelled datasets are more than welcome. To develop SafeCommit, the first main step will consist in building such datasets ("Ground Truth" in Figure 72**:** Overall SafeCommit Process). Then, we will investigate the possibility to consider a combination of text analysis of commit logs and code analysis of commit changes diff to catch security patches. To that end, the idea is to proceed to the extraction of "facts" from both text and code, and then perform a feature engineering by assessing the efficiency of the proposed features for discriminating security patches from other patches ("Features Set" in Figure 72**:** Overall SafeCommit Process). Then, we will build a prediction model ("Classifier" in Figure 72**:** Overall SafeCommit Process) using machine learning classification techniques.

As an add-on, we will investigate a specific learning approach named Co-Training, which has shown convincing results in situations where the training datasets are un-balanced. Finally, one major success criteria of SafeCommit is its ability of supporting the work of developers/maintainers in distributed software development. Once prediction models are learnt, we will assess their efficiency by performing extensive empirical studies in real development environments.

ID	code-fix	ID	security-sensitive
F1	#commit files changed	S1	#sizeof added
F2	#loops added	S2	#sizeof removed
F3	#loops removed	S3	S1-S2
F4	F2-F3	S4	S1+S2
F5	F2+F3	S5-S6	Like S1-S2 for continue
F6-F9	Like F2-F5 for *if*	S7-S8	Like S1-S2 for break
F10-F13	Like F2-F5 for Lines	S9-S10	Like S1-S2 for INTMAX
F14-F17	Like F2-F5 for Parenthesized expression	S11-S12	Like S1-S2 for goto
F18-F21	Like F2-F5 for Boolean operators	S13-S14	Like S1-S2 for define
F22-F25	Like F2-F5 for Assignements	S15-S18	Like S1-S4 for struct
F26-F29	Like F2-F5 for Functions call	S19-S20	Like S1-S2 for offset
F30-F33	Like F2-F5 for Expressions	S21-S24	Like S1-S4 for void

ID	text
W1-W10	Most recurrent top 10 word

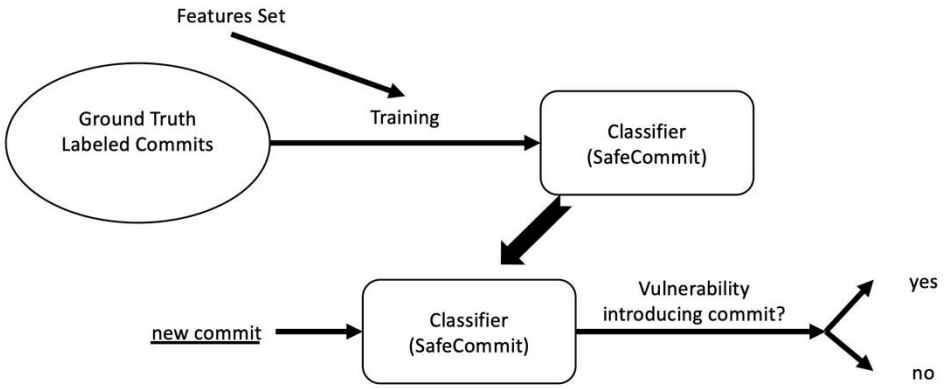

Figure 72: Overall SafeCommit Process

7.13.3 Development roadmap

Use Case	Architecture components	Realisation	Involved partners
UC1	Ground Truth, features set and classifier	Build a ground truth dataset, propose features from code and text, build the prediction models.	UNILU

Table 102: SafeCommit – Development Roadmap

A first prototype will be released in December 2020. To reach this objective, we will address the following steps:

- Collect enough commits and label them to build a ground truth
- Propose features
- Develop the classifier
- Perform extensive experiments to test the classifier
- Add the co-training module
- Evaluate the performance of the co-training module

7.13.4 Software verification and validation plan

SR id	Description	Verification method	Demonstration scenario
SR1.1	Compute performance scores by leveraging the ground truth	Check if the performance scores are high enough	Deploy SafeCommit and Run on the ground truth
SR1.1	Assess SafeCommit in practical settings	Check if SafeCommit is able to detect vulnerabilities in open source libraries used in Vertical 1	Deploy SafeCommit and Run on a git Repository of open-source libraries of Vertical 1

Table 103: SafeCommit – Demo scenarios and verification methods

Compute performance scores by leveraging the ground truth

Input: The ground truth (i.e., the labelled commits)

Output: Classification performance score

Test Procedure:

We follow a classical machine-learning assessment process. We will consider for instance using ten-folds cross validation and compute precision, recall and F1 metrics.

Assess SafeCommit in practical settings

Input: Large open-source repositories such as Linux

Output: Assessment report on this “in the wild” experiment

Test Procedure:

By considering commit history from large open-source repositories, mimic the behaviour of software developers. Check if at the time of a commit, this commit can be detected as vulnerability introducing commit.

7.14 SideChannelDefuse (FS) – CNIT

As anticipated in D5.1 [1], the main motivation for developing an assessment and countermeasure tool for side channel vulnerabilities comes from the fact that there is a new generation of side channel attacks which have raised suspects on the validity and trustiness of CPU operations. Popular attacks have raised attention to the public interest such as Spectre [135], Meltdown [136], and Foreshadow [137] [138].

We need an assessment towards these new forms of threats. To this extent there is very limited work in the literature since most of it is for discovering such vulnerabilities rather than finding a way to systematically assess and obstruct the presence and impact of such vulnerabilities.

For these reasons, we have augmented the Foreshadow assessment tool described in D5.1 along two different directions. We recall that, originally, the tool in D5.1 was designed as a stand-alone tool which, when manually started, was able to detect whether a system (with particular focus on virtualized environments) was vulnerable to side channel attacks, such as Foreshadow-VMM.

With respect to the above initial design, we have extended the tool by (1) turning it into a continuous assessment tool, and by (2) supplementing it with reactive mitigation capabilities.

Specifically, if the new SideChannelDefuse integrated tool detects (still manually) that the system is vulnerable, it can activate a continuous kernel-level system-wide detection mechanism which allows to detect whether some application (also running in a virtual machine) is carrying out a side-channel attack. This detection is continuous, in the sense that the (host) operating system kernel based detection mechanism is always on, while introducing a minimal overhead in the system. It is system-level, in the sense that it monitors all applications running in the system.

If the SideChannelDefuse tool detects that a (virtualized) application is trying to carry out a side-channel attack, that application is deemed as suspected. At this stage, the tool can activate per-application mitigation mechanisms, the goal of which is to reduce the likelihood that the application can exfiltrate data using the attack.

The overall resulting tool is able to detect Foreshadow-VMM attacks, as well as other attacks such as meltdown, spectre, or XLate-family attacks.

The SideChannelDefuse tool will be distributed as open source software. The public repository is not yet available.

7.14.1 Requirements Description

7.14.1.1 Use cases

Table 104 and Table 105 show an update of the Use Cases that were defined for the SideChannelDefuse tool in D5.1.

Use Cases	Add	Remove	Comments
UC1 Assessment of L1-TF Vulnerability		X	This feature is now not necessary anymore since the tool evolved in a self-contained new continuous instrument
UC2 Assessment of cache-based vulnerabilities	X		Missing in D5.1
UC3 Mitigation of cache-based vulnerabilities	X		Missing in D5.1

Table 104: SideChannelDefuse - Update of Use Cases specifications

UC2	Assessment of cache-based vulnerabilities
Description	The Cloud Infrastructure owner can perform an automatic detection of malicious processes on some VM, which try to exfiltrate information from the system using side channel attacks.
Actors	• Cloud Owner • Cloud Infrastructure • Infected VM
Basic Flow	The monitoring patch is installed on the guest operating system. The side-channel attack is launched on an infected VM. The tool detects the application as suspected and enforces proper mitigation actions to prevent information leakage.
UC3	Mitigation of cache-based vulnerabilities
Description	The Cloud Infrastructure owner can automatically mitigate side channel attacks on an Infected VM.
Actors	• Cloud Owner • Cloud Infrastructure • Infected VM
Basic Flow	Once the tool detects the application as suspected, it then enforces proper mitigation actions to prevent information leakage.

Table 105: SideChannelDefuse - Changes in Use Cases specifications

7.14.1.2 User Requirements

Table 106 and Table 107 show an update of the User Requirements that were defined for the SideChannelDefuse tool in D5.1.

User Requirements	Add	Remove	Comments
UR1 Assess the presence of the vulnerability		X	This feature is now not necessary anymore since the tool evolved in an automatic self-contained new continuous instrument.
UR2 Patch the kernel with the assessment-mitigation module	X		Missing in D5.1

Table 106: SideChannelDefuse - Update of User Requirements specifications

UR2	Patch the kernel with the assessment-mitigation module
Description	The monitoring tool (in the form of a patched kernel) shall be installed in the guest operating system.
Actors	Cloud Owner

Table 107: SideChannelDefuse – Changes in User Requirements specifications

7.14.1.3 Software Requirements

Table 108 and Table 109 show an update of the SW Requirements that were defined for the SideChannelDefuse tool in D5.1.

Software Requirements	Add	Modify	Comments
SR1 Linux OS support		X	
SR2 Assessment metrics	X		Missing in D5.1
SR3 Mitigation strategies	X		Missing in D5.1

Table 108: SideChannelDefuse - Update of SW Requirements specifications

SR1	Linux OS support
Description	In the current state of the tool, it has been implemented only to support Linux environments.
Actors	• Cloud Infrastructure • Cloud owner
Basic Flow	The cloud owner has to patch the Linux kernel in order to use the functionalities of the tool. No other software solutions are supported at this stage.
SR2	Assessment metrics
Description	The tool relies on metrics which are based on models that account for typical hardware usage patterns (with respect to the memory hierarchy) proper of applications trying to exfiltrate information by means of side-channel attacks.
Actors	Entirely automated
Basic Flow	The tool continuously runs in the background and evaluates its metrics in order to understand if a covert channel is currently on going.
SR3	Mitigation Strategies
Description	The tool enforces mitigation strategies in order to defend the host against side-channel attacks. Since the detection is fallible due to a degree of uncertainty, it does not take any destructive action with respect to the running process.
Actors	Entirely automated
Basic Flow	When the tool detects a side-channel attack, it triggers per-application mitigation mechanisms, in order to reduce the likelihood that the application can exfiltrate data.

Table 109: SideChannelDefuse – Changes in SW requirements specifications

7.14.2 Functional Specifications

In its previous iteration, the tool ran under Linux Kernel 4+ and targeted the KVM hypervisor. We fetched the output of the `cpuinfo` file to search if the l1tf flag is present inside the reported CPU bugs within the available microcode.

If the flag was present, the CPU checked, in the very same way, the presence of Intel proprietary Simultaneous Multithreading Technology on the target system. If so, the system is vulnerable to a cross-thread Foreshadow-VMM attack. At this point, the tool proceeded to assess the covert-channel

performance and error rate. To do so, it instantiated two identical Virtual Machines running on KVM. The two VMs were running a plain version of Linux 4+.

Once the two VMs were up, the attacker VM mounted the Foreshadow-VMM attack, reading a pre-defined number of bytes from an a-priori known memory location in its address space. This attacker's guest (virtual) memory location maps to a host virtual memory location, which in turn maps to a victim's virtual memory location. The tool, by manipulating the host memory mapping, managed to clash the three virtual addresses to the same host's physical address. Since the attacker knows what string is expecting from the reading process, it is possible to calculate the error rate of the covert-channel and the throughput of the latter.

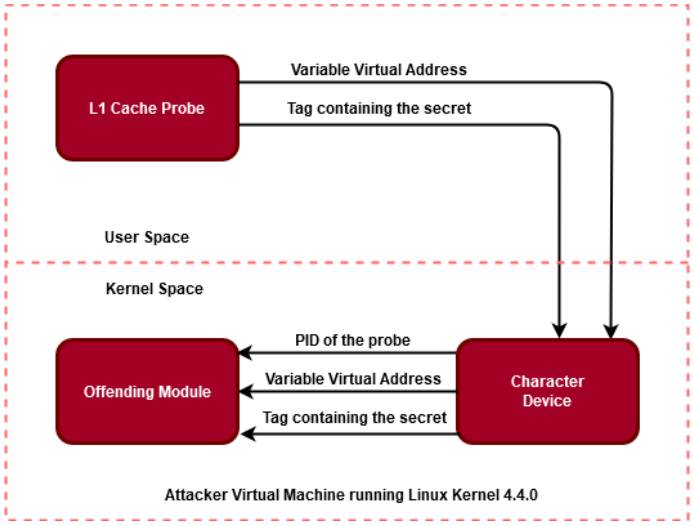

Figure 73: Architectural Diagram of the previous static FS assessment

In its new version the tool can now detect multiple side-channel attacks, relying on a patched Linux kernel. Detection is carried out at kernel level with a lightweight overhead, relying on hardware performance monitor units (PMUs) and dedicated interrupt handlers.

The tool obtains measures related to usage patterns of the caching subsystem on the machine, by properly configuring PMUs. These measures are associated with each running process. Thanks to the reliance on KVM, also virtualized applications are tracked with a proper granularity. Measures are managed so as to remove interference from the kernel itself.

These measures are then aggregated into higher-level metrics, the goal of which is to reduce the likelihood that benignware is incorrectly classified as a malicious application. These metrics are based on models which account for typical hardware usage patterns proper of applications trying to exfiltrate information by means of side-channel attacks. In this way, the tool is agnostic to the actual attack, trying to detect that a covert channel has been put in place and is currently being used.

To further reduce false negatives, the tool works using a sliding-window approach: the observation period is divided into time slots, which are observed over time. This allows discriminating among different execution phases, i.e. the tool is also able to detect malware which is running the attack only in a certain (reduced) timespan with respect to its overall lifetime - this is also the scenario of a non-malicious application infected with a side-channel based malware payload.

In order to avoid false positives, we have introduced a scoring system. The process's score will vary during execution as follows:

- the score is increased by α if the results of the comparison between metrics and thresholds show a behaviour similar to a side-channel attack;
- the score is decremented by β if the metrics don't detect any abnormal situation.

If the score reaches the value of a threshold γ, then the process becomes suspected. α, β and γ are tuneable hyperparameters of our model. Once a process becomes suspected, this information is

stored in the process' PCB (*Process Control Block*), to account also for more sophisticated attacks which could exploit `fork()` calls to jeopardize the detection system.

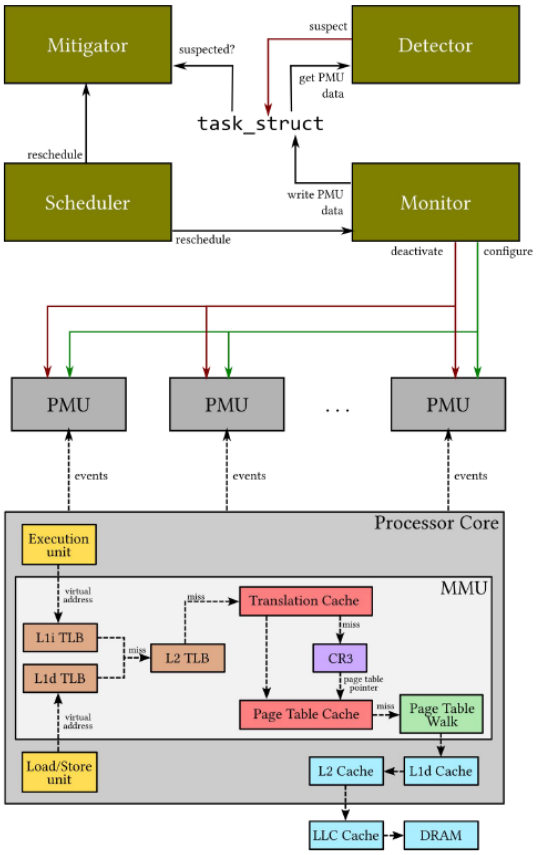

Figure 74: Updated architectural diagram for SideChannelDefuse

7.14.3 Development roadmap

Use Case	Architecture components	Realisation	Involved partners
UC1	Assessment Module	Implement a custom kernel patch that does a continuous assessment of different side channel attacks relying on hardware performance monitors	CNIT
UC2	Mitigation Module	Implement a per-application mitigation mechanism against detected vulnerabilities	CNIT

Table 110: SideChannelDefuse – Development Roadmap

We will develop the following functionalities:

- A continuous active detection strategy and algorithms based on hardware performance monitor units and dedicated interrupt handlers to assess the presence of Meltdown, Spectre, and side-channel attacks in general.
- A continuous active mitigation strategy, capable to cover the vulnerabilities that we previously discussed.

The tool will be developed as a self-contained subsystem working as a patch to the Linux kernel with no inputs and no outputs.

7.14.4 Software verification and validation plan

SideChannelDefuse is being developed as a standalone tool within CAPE, and hence does not interact with other components during verification and validation. The tool is preinstalled at the kernel level and it continuously monitors processes activities. By doing this, it can detect cache attacks but also deploy mitigation strategies on the fly. It follows that rather than integrating it in a pipeline, the most obvious way to use it is to install it and then run other tools on top of the patched kernel.

SR id	Description	Verification method	Demonstration scenario
SR1	Patch Linux Kernel in order to start the tool	Check if the tool is correctly installed and running in the kernel	Stand Alone tool
SR2 **SR3**	Continuously scan and eventually mitigate attacks in the system.	Check if the tool correctly reports on going side-channel attacks and mitigate their behaviour	Stand Alone tool

Table 111: SideChannelDefuse – Demo scenarios and verification methods

Patch Linux Kernel in order to start the tool

Input: The tool itself.

Output: None.

Test Procedure: The cloud owner is expected to install and load the patch into its cloud host. After that, the owner can check if the tool is correctly mounted into the system using standard Linux terminal commands (`lsmod`).

Continuously scan and eventually mitigate the applications behavior in the system

Our detection mechanism, as well as the aforementioned mitigations, have been implemented at kernel-level in Linux, and has been exercised on multiple processors of the x86 family. It is an indication of the viability of using HPCs as building blocks for articulated detection mechanisms, and for devising strategies where the setup of security-oriented patches can be put in place on a dynamic and per-process basis - rather than paying the cost of these patches by default when any process is active.

Input: None.

Output: Per-application values on `/proc/pid`

Test procedure: After having loaded the patched kernel in the guest system, the tool continuously assesses if there is a side-channel attack ongoing, and eventually mitigates its malicious effects. The cloud owner can check the output in `/proc/pid` in order to determine what applications have been suspected as malicious.

7.15 Steady (VA) – SAP

Steady supports software development organizations in regard to the secure use of open-source components during application development. As such, Steady addresses the OWASP Top 10 security risk A9, Using Components with Known Vulnerabilities, which is often the root cause of data breaches. Steady analyses Java and Python applications in order to:

- detect whether they depend on open-source components with known vulnerabilities,
- collect evidence regarding the execution of vulnerable code in a given application context (through the combination of static and dynamic analysis techniques), and
- support developers in the mitigation of such dependencies.

There exist several free [139] and commercial tools [140], [143], [144], [145] for detecting vulnerabilities in OSS components. [142] shows that Steady's approach outperforms state-of-the-art tools with respect to vulnerability detection. Though [145] claims to perform static analysis to

eliminate false positives, there is no public description of their approach available. OWASP Dependency Check [139] is used in [146] to create a vulnerability alert service and to perform an empirical investigation about the usage of vulnerable components in proprietary software. The results showed that 54 out of 75 of the projects analyzed have at least one vulnerable library. However, the results had to be manually reviewed, as the matching of vulnerabilities to libraries showed low precision. Alqahtani et al. proposed an ontology-based approach to establish a link between vulnerability databases and software repositories [140]. The mapping resulting from their approach yields a precision that is 5% lower than OWASP Dependency Check. All these approaches and tools differ from Steady in that they focus on vulnerability detection based on metadata, and do not provide application-specific reachability assessment nor mitigation proposals.

Steady is part of Eclipse Foundation and can be downloaded as a stand-alone application.

More information about Steady is available at:
https://projects.eclipse.org/projects/technology.steady

7.15.1 Requirements Description

7.15.1.1 Use cases

Table 112 shows an update of the Use Cases that were defined for the Steady tool in D5.1.

Use Cases	No change
UC1 Detect, assess and mitigate dependencies with known vulnerabilities in application projects	X
UC2 Detect dependencies with known vulnerabilities in open source projects and suggest mitigations	X

Table 112: Steady - Update of Use Cases specifications

7.15.1.2 User Requirements

Table 113 and Table 114 show an update of the User Requirements that were defined for the Steady tool in D5.1.

User Requirements	Add	Remove	Comments
UR1 Reduce number of unclassified findings	X		Missing in D5.1
UR2 Share efforts related to the maintenance of vulnerability databases		X	Facilitate the creation and sharing of information about vulnerabilities in open source software. Moved to Project KB, see Section 7.10.

Table 113: Steady - Update of User Requirements specifications

UR1	Reduce number of unclassified findings
Description	Reduce the number of cases where Steady cannot automatically establish whether the body of a given method is equal (or closer) to the vulnerable or fixed version. Today, such cases require human intervention and effort.
Actors	Developer, Security Analyst

Table 114: Steady - Changes in User Requirements specifications

7.15.1.3 Software Requirements

Table 115 and Table 116 show an update of the SW Requirements that were defined for the Steady tool in D5.1.

Software Requirements	No change	Modify	Comments
SR1 Comparison of Java source code and bytecode	X		Addresses UR1 and consists of finding (or creating) an intermediate representation that can be created from source and compiled code, and which serves as the basis for comparisons and distance metrics.
SR2 Implementation of a light-weight scan client	X		
SR3 Shared vulnerability database		X	Addresses UR2, and requires the definition of a data model, merge strategies and related tooling. The definition of a data model, merge strategies and related tooling has been moved to Project KB, see Section 7.10. What remains is a new component kb-importer, which takes the data of Project KB as input in order to populate Steady's vulnerability database.

Table 115: Steady - Update of SW Requirements specifications

SR3	Shared vulnerability database
Description	Steady must load vulnerability information from Project KB in an automated fashion.
Actors	N.A. (entirely automated)
Basic Flow	Upon installation and at regular timeframes, Steady uses Project KB to update its database with known vulnerabilities.

Table 116: Steady – Changes in SW requirements specifications

7.15.2 Functional Specifications

At high-level, see Figure 75, Steady comprises a number of client-side scan tools that analyse a given application, either manually or as part of automated build processes (`plugin-maven`, `plugin-gradle`, `cli-scanner`). Analysis results are uploaded to (and persisted by) a RESTful component called `rest-backend`, which is one out of several components that run server-side, e.g., in private or public clouds. The components `frontend-apps` and `frontend-bugs` are HTML5 applications rendered by a browser and used by end-users to consume the analysis results. The remaining components, `patch-analyser` and `rest-lib-utils` are related to the analysis and processing of commit information (of open source projects) and packages available on public or private package repositories.

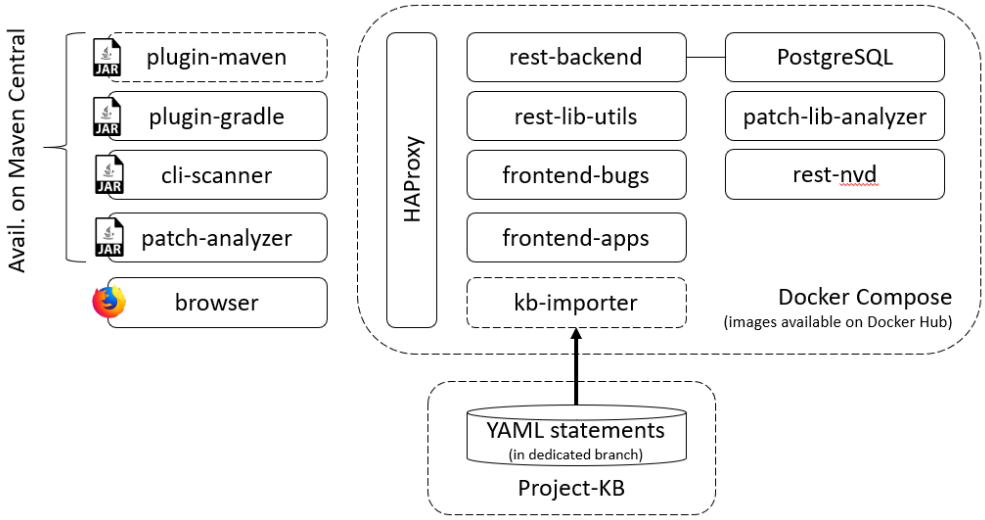

Figure 75: High-level architecture of Steady - Components created or modified by SR1-3 are highlighted with dotted borders

With respect to Figure 75, SR1 will be implemented by modifying the component `plugin-maven`. More specifically, the plugin will be extended by the additional goal `checkcode`[26].

Upon invocation, manually or during automated pipeline builds, the goal analyses all unconfirmed vulnerable dependencies, which are pairs of (vulnerability v, library l). They correspond to Java archives that contain constructs known to be affected by a given vulnerability, but for which it could not be clarified whether the construct body is equal (or closer) to the vulnerable or the fixed version. This happens for Java archives not known to Maven Central, or for archives on Maven Central without a corresponding source code artefact, e.g., Uber JARs. These cases appear as orange hourglasses in the report and application frontend.

In more detail and as explained in Figure 76, for every unconfirmed vulnerable dependency (v, l), the analysis consists of extracting the questionable constructs from the Java archive (JAR) and building their abstract syntax trees (AST). These ASTs are compared with the ASTs of the corresponding constructs in other Java archives previously assessed as vulnerable or fixed. Only if all questionable constructs correspond to constructs in either fixed or vulnerable archives, the unconfirmed vulnerable dependency is also set to fixed or vulnerable.

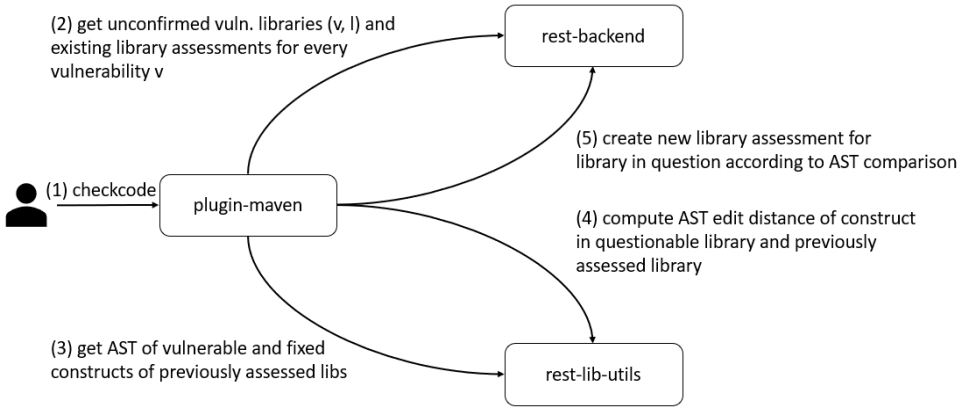

Figure 76: Eclipse Steady: Plugin goal "checkcode"

With respect to Figure 75, SR2 will be implemented by reducing the footprint of the Docker Compose environment such that it can also run locally, which has the big advantage of giving flexible deployment options to users: users can decide to run the entire Steady solution locally, e.g., for

[26] https://eclipse.github.io/steady/user/manuals/analysis/#analyze-unconfirmed-vulnerabilities-checkcode

testing and demonstration purposes. Alternatively, users can stick to the existing deployment model, which is especially interesting for larger software development organizations with many development projects and central compliance and security teams.

The footprint reduction will be realized by omitting components that are not necessary in local deployments, e.g., `frontend-apps` and `frontend-bugs`. Where necessary, information provided in those Web applications will be made available in a self-contained HTML report created through the existing plugin goal `report`[27].

Moreover, the local deployment shall happen as transparent as possible. Ideally, the local Docker Compose environment is downloaded, configured and started automatically when users invoke the scan clients, esp. `plugin-maven`. Corresponding Docker images are already available on Docker Hub[28], however, their lifecycle has to be managed through the interaction of Steady with a local Docker client.

SR3 has been modified since the writing of D5.1. Formerly, it was planned to develop an open and distributed vulnerability database as part of Steady. However, it turned out that such database has its own raison d'être, and that Steady is just one of potentially many downstream users. As such, it was decided to continue the development of this database as Project KB (see Section 7.10).

What remains with Steady is a new component `kb-importer`, which consumes information from Project KB in order to populate Steady's vulnerability database. `kb-importer` is a Java stand-alone application, which reads and processes vulnerability information from the file system, e.g., Java source code and vulnerability metadata, and calls the REST API of the component `rest-backend` in order to persist the information. A preliminary version of `kb-importer` has already been released[29], and a corresponding Docker image to facilitate its use and support automation is under development.

7.15.3 Development roadmap

Use Case	Architecture components	Realisation	Involved partners
UC1	All	Integration of Steady in CI/CD pipeline	CINI/FBK
UC2	All	Integration of Steady in CI/CD pipeline	CINI/FBK

Table 117: Steady – Development Roadmap

The software requirements SR1 and SR3 have been mostly completed and are available in the respective open-source repositories on GitHub. Minor adjustments and additions may be made throughout the tests described in Section 7.15.4. SR2 will be developed in the first half of 2021 such that its functionality is available and can be tested before project end.

7.15.4 Software verification and validation plan

SR id	Description	Verification method	Demonstration scenario
SR1	Bytecode comparison	Check if an unclassified finding can be resolved through the execution of the new plugin goal "checkcode"	e-Government (Vertical 2)
SR2	Light-weight scan client	Run light-weight Docker Compose environment and monitor resource consumption and performance	Independent

[27] https://eclipse.github.io/steady/user/manuals/analysis/#create-result-report-report

[28] https://hub.docker.com/search?q=eclipse%2Fsteady&type=image

[29] https://eclipse.github.io/steady/vuln_db/manuals/kb_importer/

SR id	Description	Verification method	Demonstration scenario
SR3	Shared vulnerability database	Check the initial and delta load of vulnerabilities from Project KB into Steady's database	e-Government (Vertical 2)

Table 118: Steady – Demo scenarios and verification methods

SR1 – Bytecode comparison

Input: Artificial Java archive with vulnerable code, added as dependency to the e-Government application.

Output: Correct classification as vulnerable after running the plugin goal.

Test Procedure: An artificial Java archive A with vulnerable bytecode will be created and installed in the local m2 Maven repository of the build environment. The vulnerable bytecode will be taken from an existing, publicly available open source Java artefact P, which has been previously assessed as vulnerable in the context of the e-Government application. The e-Government application will declare a new dependency on A.

Since the artificial artefact does not exist in the public Maven repository, neither as bytecode nor source code, the execution of Steady's analysis goal "app" will yield an unclassified finding for A. The execution of the new analysis goal "checkcode", however, should succeed, since the bytecode contained in the artificial artefact A can be compared to the bytecode of the already classified archive P.

SR2 – Light-weight scan client

Input: Publicly available Docker Compose file, Maven artefacts and Docker images.

Output: Successful scan performed with the local Docker Compose environment plus metrics about resource consumption and performance.

Test Procedure: The invocation of one or more Maven plugin goals will download and install Steady's Docker Compose environment on the local test machine. This environment will be used for several subsequent scans using the "app" analysis goal of Steady's Maven plugin. Both installation and scans shall be monitored in terms of resource consumption and performance.

SR3 – Shared vulnerability database

Input: Statements from Project KB.

Output: Populated Steady.

Test Procedure: The installation of Steady's Docker Compose environment shall automatically trigger the initial and delta load of vulnerability statements from Project KB into Steady's PostgreSQL database. This load will be tested as part of the e-Government scenario and can be checked by querying the database or consulting a dedicated Web application (`frontend_bugs`). The delta load can be tested by creating a new statement after the initial load in a given Docker Compose environment has happened, in a dedicated test branch of Project KB or a test repository (see Section 7.10.4).

7.16 SysML-Sec (TTool) – IMT

TTool (pronounced "tea-tool") is a toolkit dedicated to the edition of UML and SysML diagrams, and to the simulation and formal verification (safety, security, performance) of those diagrams. TTool supports several UML profiles: AVATAR, DIPLODOCUS and SysML-Sec.

SysML-Sec covers all development stages, including requirements, faults and attack trees, system-level hardware / software partitioning with automated design space exploration, embedded software design, software deployment, and finally code generation. Main diagrams can be formally verified against safety, security and performance properties. When formal verification induces combinatory

explosion, fast simulation helps having a better confidence in the system. Last but not least, TTool can generate test sequences.

The main originality of TTool / SysML-Sec relies its ability to support the design and formal verification of both safety and security aspects from the same input (SysML) models, while covering development cycles from requirements until code generation. Many other design methodologies handle the complete design flow of embedded systems, including design space exploration, and prototype code generation, such as [147][148][149]. [150] is a development environment with extensions so it can be customized for different domains. They all support modelling requirements and systems, and offer model-checking including simulation and formal verification capabilities. Unlike our toolkit, they also do not model or verify security properties.

While AADL takes safety and performance requirements into account during design [151], it also has been extended for modelling security for access control both in its hardware partitioning and software-based communications [152]. SecureUML targets the design and analysis of secure systems by adding mechanisms to model role-based access control [153]. The security model of TTool rather focuses on protecting against an external attacker instead of access control. UMLSec [154] features a rather complete framework addressing various stages of model-driven secure software engineering from the specification of security requirements to tests, including logic-based formal verification regarding the composition of software components. However, UMLSec does not take into account the HW/SW Partitioning phase necessary for the design of e.g. IoTs, nor the relation between safety and security.

More information about TTool is available at: https://ttool.telecom-paris.fr/

7.16.1 Requirements Description

7.16.1.1 Use cases

Table 119 and Table 120 show an update of the Use Cases that were defined for TTool in D5.1.

Use Cases	Add	Comments
UC1 Formal Security Verification of platooning SafeSec module	X	Missing in D5.1

Table 119: TTool - Update of Use Cases specifications

UC1	Formal Security Verification of platooning SafeSec module
Description	TTool intends to be used to verify that the defined architectures respect the safety and security requirements that can be verified from a high-level model.
Actors	System engineer
Basic Flow	Capture safety and security requirements, model fault and attack trees, perform the architecture design, including a model of hardware components, verify safety and security properties.

Table 120: TTool – Changes in Use Cases specifications

7.16.1.2 User Requirements

Table 121 and Table 122 show an update of the User Requirements that were defined for TTool in D5.1.

User Requirements	Add	Comments
UR1.1 Automated and formal security verification of digital systems from high-level models	X	Missing in D5.1

Table 121: TTool - Update of User Requirements specifications

UR1.1	Automated and formal security verification of digital systems from high-level models
Description	The main idea is to make high-level models of the (platooning) system taking into account both functional and architectural aspects, and then to perform in an automated way security evaluation.
Actors	System architects and verification engineers

Table 122: TTool – Changes in User Requirements specifications

7.16.1.3 Software Requirements

Table 123 and Table 124 show an update of the SW Requirements that were defined for the TTool tool in D5.1.

Software Requirements	Add	Comments
SR1 C-ACC Safety and Security Mapping and Verification	X	Missing in D5.1
SR2 Inter-relations between safety and security aspects	X	Missing in D5.1

Table 123: TTool - Update of SW Requirements specifications

SR1	C-ACC Safety and Security Mapping and Verification
Description	Cooperative Adaptive Cruise Control (C-ACC) is used by vehicles to improve safety and fuel-efficiency in vehicle platoon. This is because C-ACC enables the safe reduction of the gap between vehicles as vehicles can quickly adapt their state and react to emergency by relying on the information communicated through the communication channels. However, attackers can also exploit these communication channels to cause harm, such as vehicle crashes. We have proposed adequate countermeasures based on plausibility checks. We are going to develop in TTool a model-based view of the C-ACC architecture and functions, with a focus on safety and security aspects. By “architecture”, we mean the corresponding hardware platform, while functions refer to unmapped functional elements: the latter will be mapped onto the hardware components. The models will be done in collaboration with the models of AutoFOCUS3 (AF3). Models (e.g. mapping of functions over the hardware) will then be verified against safety and security requirements.
Actors	TTool, AF3
Basic Flow	1) Download and install TTool and AF3 2) Open the model of the system specification in AF3 and in TTool 3) Enrich the model of TTool by exporting the model of AF3 4) Perform safety or security verification in TTool, and backtrace results in TTool and AF3.
SR2	Inter-relations between safety and security aspects
Description	Ensure exchanges with external tools, e.g. AF3
Actors	Develop engineers
Basic Flow	Definition of necessary exchanges, program input / outputs.

Table 124: TTool – Changes in SW Requirements specifications

7.16.2 Functional Specifications

Concerning SR1, all necessary hardware components that are not present in TTool will be specified and then added to the TTool framework. This addition will not impact the verification framework of TTool, so new components will be defined upon the existing components.

For SR2, we will have to define the format for diagram and view exchanges. Backtracing verification results to the complementary tool (e.g., verification performed in TTool, result displayed in AutoFOCUS3) will also be studied by first defining which verification aspects could be exchanged, and then an exchange format will be defined.

In both cases, the following method will be applied. From the system specification and attack and fault trees, a functional view, an architectural view and then a mapping will be built. Then, verification will be performed, both from the functional view and the mapping view. Again, results will be backtraced to model, and updates will be proposed. An update could typically be a security countermeasure.

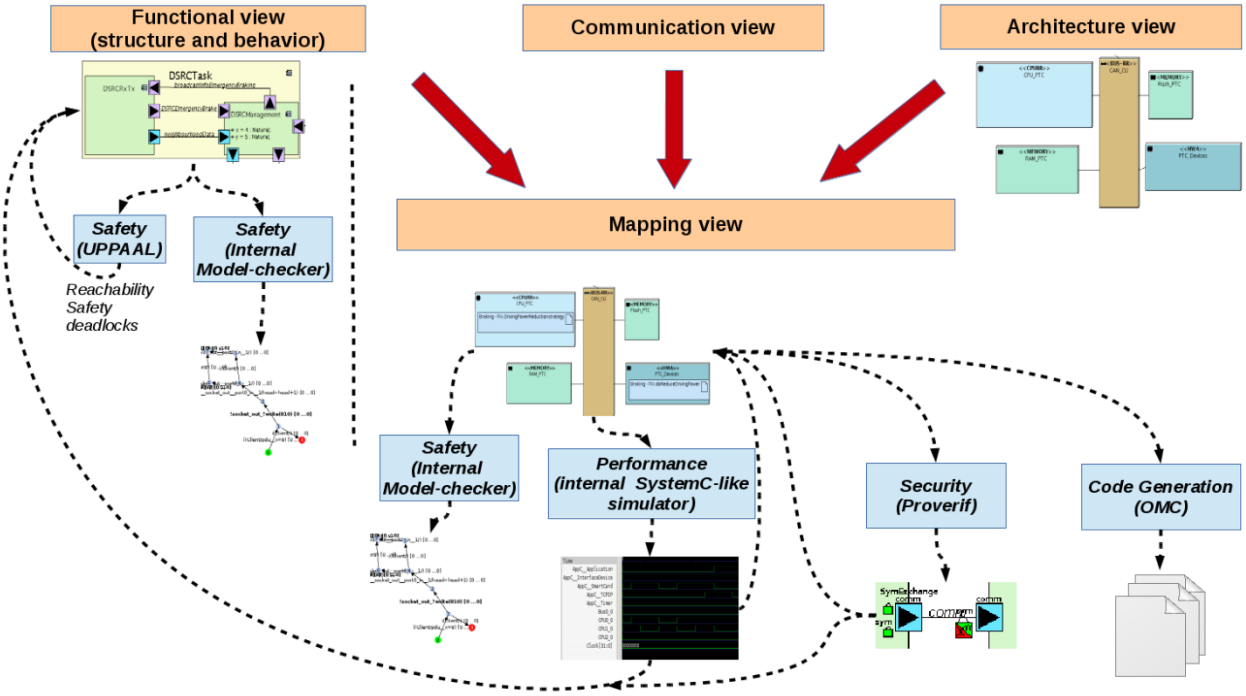

Figure 77: TTooL modules

7.16.3 Development roadmap

Use Case	Architecture components	Realisation	Involved partners
UC1	Hardware and software aspects will be taken into account. They will be built both from the system specification and from already developed code.	Hardware / software partitioning models, formal verification of safety and security properties.	IMT, FTS

Table 125: TTool – Development Roadmap

7.16.4 Software verification and validation plan

SR id	Description	Verification method	Demonstration scenario
SR1	Use TTool to verify the platooning system at a high level of abstraction	• Capture the digital platform at a high-level of abstraction • Look for possible attacks with formal verification • Study countermeasures	Connected Car (Vertical 1)

SR id	Description	Verification method	Demonstration scenario
SR2	Use TTool to verify the platooning system taking into account both software and hardware aspects	• Capture the digital platform at a high-level of abstraction • Look for possible attacks with formal verification. This security verification takes into account hardware aspects (e.g. access to buses, firewalls, etc.) • Study countermeasures	Connected Car (Vertical 1)

Table 126: TTool – Demo scenarios and verification methods

SR1 - Verify the platooning system at a high level of abstraction

Input: AF3 model

Output: Countermeasures resulting from the verification process

Test Procedure: We intend to import a (functional) model from AF3, to execute a safety verification with our internal model-checker and discuss with AF3 developers how we could inject the verification results in AF3. We will do the same for security properties: import of AF3 model in TTool, security verification with ProVerif, backtracing to AF3. In the scope of SR1, only very small model parts will be addressed because only the exchange of information is at stake here.

SR2 - Verify the platooning system taking into account both software and hardware aspects

Input: AF3 model

Output: Countermeasures resulting from the verification process

Test Procedure: We intend to show that several relevant properties of the system – safety properties, security properties – can be proved from TTool models of the platooning system, using the simulators and model-checkers integrated in TTool. Here, we intend to take into account both software and hardware aspects (which is not the case in SR1).

7.17 VaCSInE (VCS) – CETIC

VaCSIne is an open-source security orchestration, automation and response tool that provides adaptive security for distributed systems. It relies on continuous monitoring of Cloud and Edge systems to define, evaluate and apply automated countermeasures such as firewalls, intrusion detection systems, honeypots or quarantining. The automated response is triggered by changes to security requirements, indicators of compromise, incidents and vulnerabilities. The efficiency and speed of countermeasures deployment is evaluated in automatically provisioned sandbox environments that shadow the target Cloud/Edge systems. Those sandboxes provide observability and scalability for the training and maintenance of security response strategies.

Mobile Edge Computing and Fog introduces additional security challenges, for example the need to satisfy security requirements in the presence of unreliable networks or when low latency in the security response is critical. This resilience can be ensured by continuous monitoring of the system, prompt detection of anomalies and remediation in an autonomous way [155] [156] [157]. Security orchestration has to take into account those Edge and Fog specificities to avoid being the single point of failure [158]. In the context of autonomous cars for example, simulation and machine learning model training can be done in part in the Edge and in the Cloud [159].

The VaCSIne source code and documentation is hosted in the public git repository: https://github.com/cetic/vacsine .

7.17.1 Requirements Description

7.17.1.1 Use cases

Table 127 and Table 128 show an update of the use cases that were defined for the VaCSInE tool in D5.1.

Use Case	No change	Modify	Comments
UC1 Enforce security policy on the edge infrastructure based on certification criteria	X		
UC2 Continuous self-assessment for adaptive security with service function chaining		X	Further definition of the interactions with continuous certification pipeline

Table 127: VaCSInE - Update of Use Cases specifications

UC2	Continuous self-assessment for adaptive security with service function chaining
Description	Monitor and detect: Ensure the edge infrastructure is protected through an automated reconfiguration of the service function chains. This can involve adding/removing or updating existing security functions.
Actors	Security Officer
Basic Flow	The intrusion detection triggers a firewall re-configuration, remediation is checked against the system's security policy (derived from certification criteria) and applied to the security functions protecting the system. The resulting modifications to the firewall (configuration logs) are monitored and provide an input to the continuous certification process. Those logs can be used as evidence that the configuration has taken place and as input for further security certification tools further down the certification process. In case the changes to the firewall configuration require a re-certification, for example following a major version change of the security service, a new iteration in the certification process is started.

Table 128: VaCSInE – Changes in Use Cases specifications

VaCSInE will demonstrate how to ensure continuous assessment of edge systems by developing adaptative security mechanisms based on security policies derived from certification requirements.

7.17.1.2 User Requirements

Table 129 shows an update of the User Requirements that were defined for the VCS tool in D5.1.

User Requirements	No change
UR1 Minimal network attack surface	X

Table 129: VaCSInE - Update of User Requirements specifications

7.17.1.3 Software Requirements

Table 130 and Table 131 show an update of the SW Requirements that were defined for the VCS tool in D5.1.

Software Requirements	Add	Comments
SR1 – Orchestration of the security policy	X	Missing in D5.1
SR2 – Observability of the security policy orchestration	X	Missing in D5.1

Table 130: VaCSInE - Update of SW Requirements specifications

SR1	Orchestration of the security policy
Description	The tool should provide orchestration capabilities to manage the target system security policies. When a security policy changes, the system needs to be reconfigured accordingly. For example, the level of security of a system can be increased by restricting its attack surface through stricter firewall rules that allow only a minimal set of selected ports to be open.
Actors	Security officer
Basic Flow	1) A modified security policy is applied on the system 2) The system's security services are reconfigured to satisfy the policy
SR2	Observability of the security policy orchestration
Description	The orchestration of the target system's security policy should be observable to detect failures and their reasons. The execution status of policy changes and their effects (success, duration, logs, …) should be monitored, alerts should be triggered when failures happen.
Actors	Security officer
Basic Flow	1) A modified security policy is applied on the system, which results in error 2) Information on the failure (error logs) is made available, an alert is sent to the security officer 3) A modified security policy is applied to the system, which results in a successful reconfiguration of the services 4) Execution logs of the remediation are registered as compliance evidence

Table 131: VaCSInE – Changes in SW requirements specifications

7.17.2 Functional Specifications

Vacsine is composed of several modules that are deployed in Cloud and Edges infrastructures (see Figure 78):

- The **Federated Security Controller** provides federated management of the security remediation on the target system. It is a consolidated view of the remediations history and states across the various edges and clouds. This controller relies on a registry of the *security policies* of its federated infrastructure, a *remediation registry* containing templates and workflows of security remediations and *security monitoring* information such as remediation execution logs, results of vulnerability scans, threat indicators, etc.
- A **Security Agent** is deployed on each edge and cloud, it provides security remediations based on the detection of various events and the matching of those events to *remediation workflows*. Agents can operate in autonomous mode, this provides a quicker response time to events happening in the edge they are deployed on, and continued operation in case the edge-cloud connexion is degraded. The *edge datastore* contains a local version of the security policies, remediations registry and security monitoring information.
- Vulnerability remediation in the form of **security services** such as firewalls, intrusion detection systems or honeypots that are triggered by changes to security requirements, threat indicators, incidents and vulnerabilities.
- **Remediation sandboxes** to test remediation workflows in a dedicated environment before applying them or training new remediation strategies.

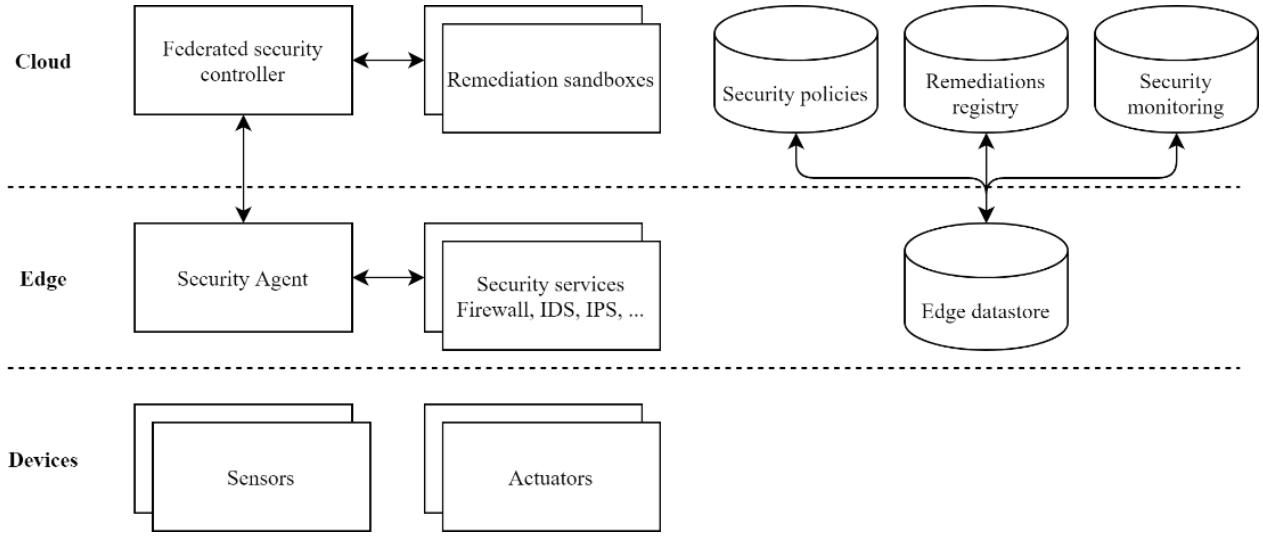

Figure 78: VaCSnE modules

7.17.3 Development roadmap

Use Case	Architecture components	Realisation	Involved partners
UC1	Security agent, security policies, remediations registry	Connected Car (Vertical 1) – scenario 2	CETIC
UC2	Federated security controller, security monitoring	Connected Car (Vertical 1) – scenario 2	CETIC

Table 132: VaCSInE – Development Roadmap

Both use cases will be validated in D5.4 following the T5.1 roadmap. The range of remediations will be extended to more security services such as honeypots, and we will be applying the security orchestration in the case study where a new vehicle joins a platoon.

7.17.4 Software verification and validation plan

SR id	Description	Verification method	Demonstration scenario
SR 1	Orchestration of the security policy	Vulnerability assessment	Connected Car (Vertical 1), scenario 2
SR 2	Observability of the security orchestration	Log analysis	Connected Car (Vertical 1), scenario 2

Table 133: VaCSInE – Demo scenarios and verification methods

SR 1 - Orchestration of the security policy

Input: security policy, target system description

Output: verified remediation execution

Test procedure: For each input security policy, we proceed as follows:

1. create a test sandbox containing an image of the target system
2. analyse the security policy and deduce a remediation workflow
3. apply the remediation to the test sandbox
4. check that the security requirements of the security policy are satisfied in the test sandbox

5. apply the remediation to the target system
6. check that the security requirements of the security policy are satisfied in the target system.

SR 2 - Observability of the security policy orchestration

Input: remediation workflow, target system description

Output: remediation logs

Test procedure: For each remediation workflow, we proceed as follows:

1. apply the remediation workflow to the system
2. check the remediation workflow execution status, this includes execution logs for each step of the remediation. Those logs should contain details on the execution for traceability such as start time, duration, informative and error messages.

7.18 Visual Investigation of security information (VI) – UKON

Assessing the security of software is a central challenge in CAPE. The assessment of individual software applications can significantly impact organizations' software projects, such as detecting vulnerabilities (e.g., CSV's) in widely used third-party software packages. The visual assessment of such software vulnerabilities in the context of a whole large software development organizations can help to identify, explore, and interpret the security status of entire organizations. There is currently no visual interface that presents an overview of a whole software organization's security situation, to the best of our knowledge. Therefore, we decided to design and develop a visualization from scratch based on partner tools (Eclipse Steady) as part of the work package.

The visual investigation (VI) of security information for larger software development organizations supports the visual analysis of individual software components' security status and evaluating the associated risk posed by their own and third-party components. The implemented demonstrator (Vulnerability Explorer) uses the outputs of the Eclipse Steady software (SAP) and allows to explore organization-wide picture of dependencies between the components as well as their exposures (vulnerabilities). The developed web demonstrator aims to increase confidence in a whole organization's security by presenting and exploring its exposure, including internal and external dependencies. The vulnerability explorer provides a complete overview of the software organization and investigates and prioritizes critical vulnerabilities.

7.18.1 Requirements Description

7.18.1.1 Use cases

Table 134 and Table 135 show an update of the Use Cases that were defined for the VI tool in D5.1.

Use Cases	Modify	Comments
UC1 Visual Investigation of Large Software Organizations	X	Following discussions with potential users from SAP, we have extended the characterization of the use case.

Table 134: VI tool - Update of Use Cases specifications

UC1	Visual Investigation of Large Software Organizations
Description	The automatically detected known vulnerabilities in large software organizations such as the Eclipse Foundation are presented and explored.
Actors	• Software developers • Software testers • Project managers

UC1	Visual Investigation of Large Software Organizations
Basic Flow	A software developer, tester, or project owner provides a project; the tool then depicts automatically detected known vulnerabilities in the component, in the dependencies to internally developed packages, as well as external third-party libraries. The stakeholder can also change the perspective and investigate which open source components are used frequently and explore their respective dependencies.

Table 135: VI tool – Changes in Use Cases specifications

7.18.1.2 User Requirements

Table 136 shows an update of the User Requirements that were defined for the VI tool in D5.1.

User Requirements	No change
UR1 Increase confidence in analysed systems	X
UR2 Multi-source levels of analysis	X
UR3 Information representation	X
UR4 Vulnerability prioritization	X
UR5 Interdependence analysis	X

Table 136: VI tool - Update of User Requirements specifications

7.18.1.3 Software Requirements

Table 137 shows an update of the SW Requirements that have been defined for the VI tool in D5.1.

Software Requirements	No change
SR1 Web Application Prototype	X

Table 137: VI tool - Update on SW Requirements specifications

7.18.2 Functional Specifications

The developed demonstrator is called the **SPARTA Vulnerability Explorer** and utilizes the Eclipse Steady API (see Section 7.15). The displayed data are the scanned package results of a whole software organization. In this case, the open-source Java packages of the Eclipse Foundation were crawled exemplarily for the demonstrator.

The demonstrator back-end was implemented in Java and the front-end with state-of-the-art web technologies. The Java back-end facilitates an in-memory database that accesses and stores the data from the Eclipse Steady API. The interactive visualizations are implemented using the JavaScript library D3 (Data-Driven Documents). The main input files for the demonstrator are either Java Maven projects or Python packages.

Figure 79 shows a high-level architecture of the SPARTA Vulnerability Explorer tool.

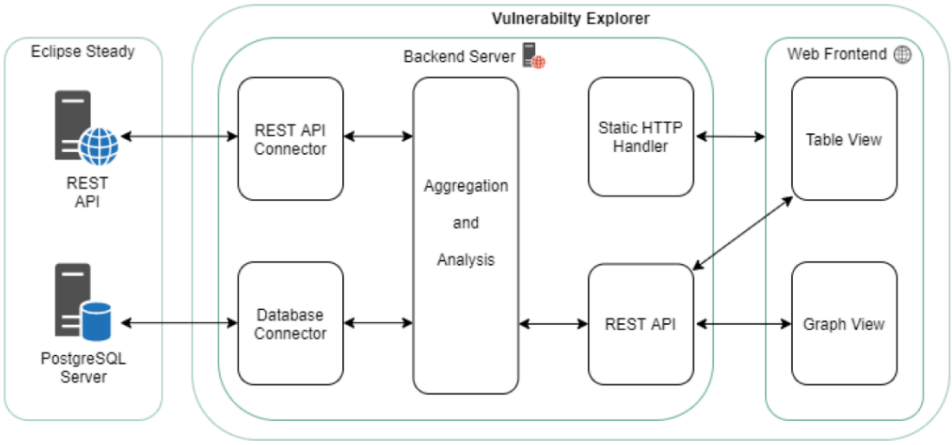

Figure 79: High-level architecture of the SPARTA Vulnerability Explorer

The SPARTA Vulnerability Explorer has two main views. The first one is a tree view to investigate search and filter all vulnerabilities which were detected in the whole software organization (see Figure 80). The second one is a graph view that enables to visually explore the dependencies between libraries to get an overview of the interrelationships in the software organization (see Figure 81).

7.18.3 Development roadmap

Use Case	Architecture components	Realisation	Involved partners
UC1	Interactive visualization prototype and Eclipse Steady	Build the prototype to get the vulnerabilities information from Eclipse Steady	UKON, SAP

Table 138: VI Tool – Development Roadmap

We implemented the following features and functionalities:

- A Java back-end which enables to access and preprocess the package scan results of the Eclipse Steady API.
- Interfaces to explore the security status of software organizations. We have designed and implemented two interfaces in close collaboration with domain experts.

The tree view (see Figure 80) depicts the whole software organization with their repositories, modules, libraries, and bugs in a tabular view. The vulnerability information is displayed in the various columns, for instance, a heatmap shows the distribution of all CVSS scores for all vulnerabilities in a repository. The tree view can be sorted, searched, and filtered using a filter panel. The tree view can also be used to investigate CVE in the whole software organization.

We will extend the tree view to include open source packages to allow analysts to overview the used packages and their potential impact on the whole software organization.

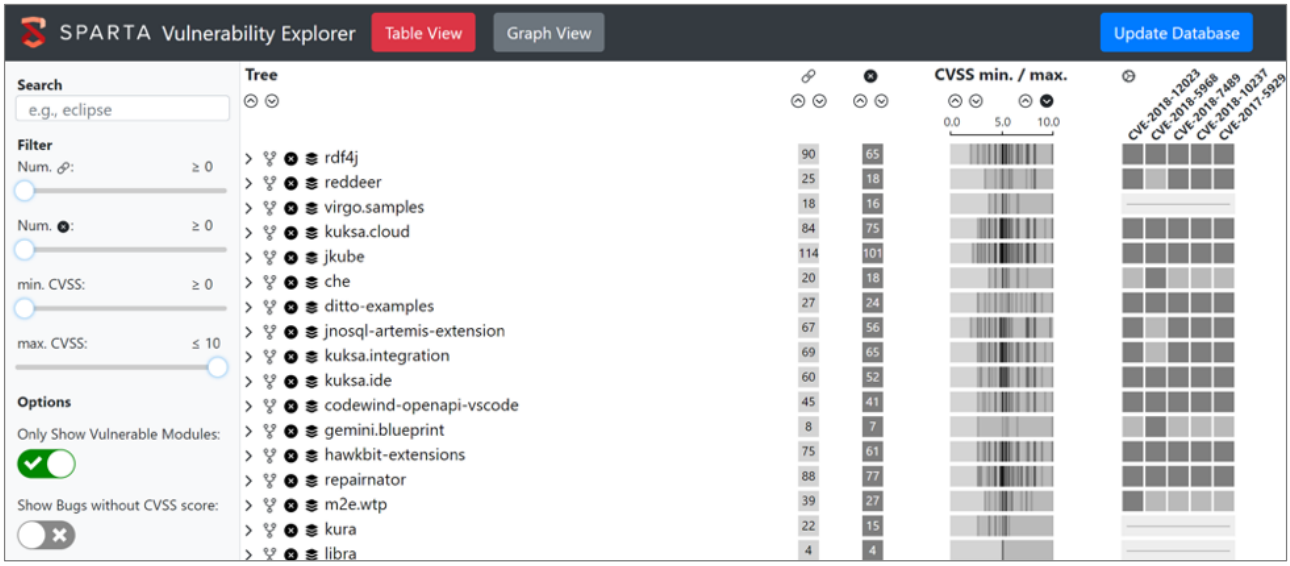

Figure 80: The tree view of the Vulnerability Explorer

The graph view (see Figure 81) displays the dependency structure from an ego-centric perspective to a particular repository or module. The visualization depicts a directed acyclic graph to allow analysts to visually explore dependencies between repositories to understand how various exposures affect the whole software organization.

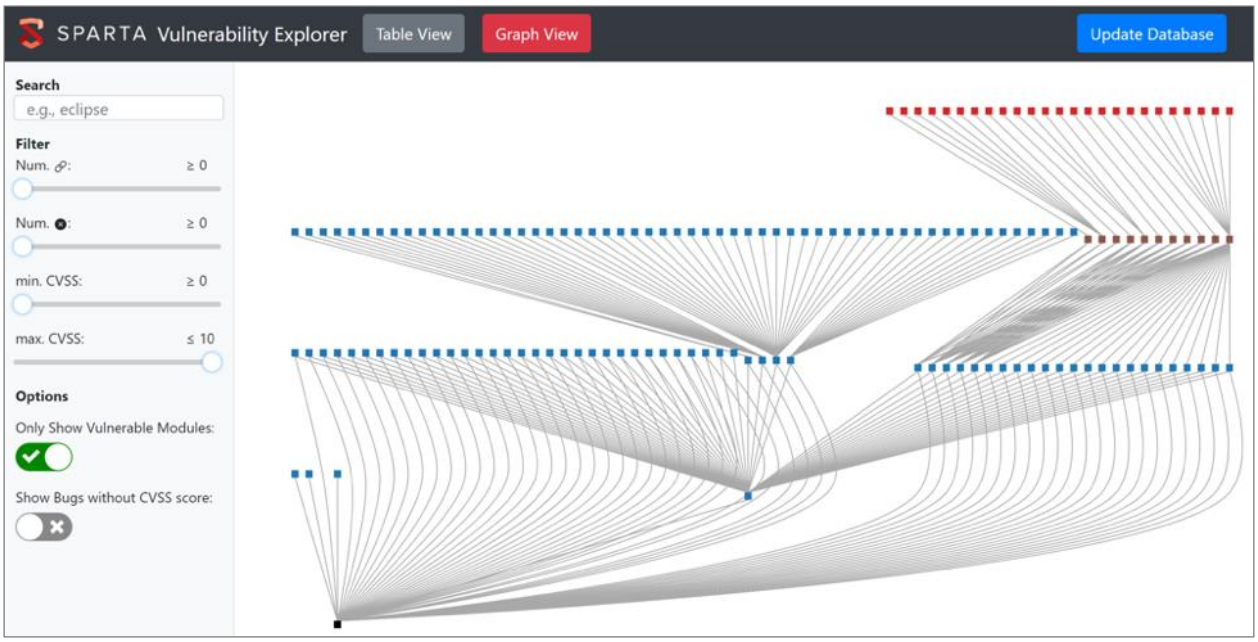

Figure 81: The graph view of the Vulnerability Explorer

7.18.4 Software verification and validation plan

We evaluated the designed interface and the demonstrator with domain experts from SAP in interviews. We conducted expert interviews with potential users with think-aloud protocols to capture the user requirements and further suggestions that we incorporated into the user interfaces' design. For example, we added more perspectives upon the software organization that enables analysts to investigate open source components in the whole organization.

SR id	Description	Verification method	Demonstration scenario
SR1	Web Application Prototype	Display projects developed in software organizations (e.g., Eclipse Foundation)	Visually investigate the Eclipse Foundation projects in the e-Government scenario (Vertical 2)

Table 139: VI tool – Demo scenarios and verification methods

Verification method

Input: The demonstrator will display the crawled Eclipse Foundation open-source project, with all vulnerabilities that Eclipse Steady detected. The usage scenario is in the e-government vertical and highlights how the demonstrator can be used to identify central exposures of packages in whole software organizations.

Output: User feedback from domain experts about the usability and usefulness of the design. The demonstrator is used to confirm the applicability of the design.

Test procedure:

The demonstrator will be used to visually explore the Eclipse Foundation to identify the number of critical vulnerabilities and their organization-wide dependencies. We will conduct expert interviews with the Eclipse Foundation members to get further insight into the usability of the demonstrator and collect more user requirements. The expert interviews will be think-aloud protocols to capture usability and additional suggestions, such as new user requirements.

Chapter 8 Summary and Conclusion

D5.2 is the second deliverable of the CAPE program and includes contributions for each task and vertical in the context of the CAPE program. It reports the work that has been conducted by the CAPE partners over the last 12 months on defining technical specifications for the development of the assessment tools and the demonstrators.

Regarding Task 5.1, in Chapter 2 we have continued the development of the cybersecurity assessment tools identified in the framework for continuous assessment and certification. These assessment tools, which are thoroughly described in Chapter 7, cover different aspects of assessment and are sometimes dependent on specific technologies. When designing an engineering process, it is thus necessary to identify the useful assessment tools and integrate them into a development process. In the SPARTA project, we have explored the use of continuous integration methods and tools to orchestrate the loose coupling of the framework tools. We have applied DevSecOps approaches to integrate security activities in the different DevOps phases. To increase the coverage of the V-Model phases, we have also integrated existing tools (Maude, OpenSCAP, etc.) into the DevSecOps processes. For orchestrating the continuous assessment and certification, some of the framework tools have developed connectors to continuous integration services such as Gitlab CI or GitHub Actions. Assessment tools with the connectors can more easily be integrated into a DevSecOps process. While building the first version of the prototypes, the Technology Readiness Level (TRL) of most of the framework tools has been improved.

Regarding Task 5.2, in Chapter 3 we have described both safety and security analysis for the CACC platoon scenario such as FMEA for safety and attack defence trees for security. We have described a new methodology for trade-off analysis between safety and security solutions. This methodology has led to two publications [74] [75] . We have described a protection profile for a safety and security platoon management module. This protection profile was the basis for designing the CACC platoon on AutoFOCUS3. Finally, we have assessed the security of CACC platoons by means of formal verification. This work has led to one publication [63].

Regarding T5.3, in Chapter 4 we have produced a comprehensive overview about open source supply chain attacks by reviewing recent, real-world supply chain attacks and creating a systematic attack tree. This work as well as the design and development of countermeasures led to a number of publications [49] [160] [161] [162] and a public dataset supporting future research[30]. The relevance of this work is also demonstrated by the attack on SolarWinds' build infrastructure, which led to the distribution of malicious software updates to more than 18,000 SolarWinds customers, including US and European Government and private sector organizations. Moreover, we have progressed in regard to extending the set of tools that will be integrated into automated CI/CD pipelines, and – going forward – it will be important to ensure that those tools do significantly impact on pipeline performance and availability. Also, we have progressed with regard to developing AI models to automatically classify source code commits as security relevant, and to model the attractiveness of open source projects for attackers in order to identify projects that require special security measures.

Chapter 5 and Chapter 6 cover the specification of our two CAPE use cases, the "Connected Car and the e-Government verticals. These two vertical use cases are particularly representative of the cybersecurity issues that modern digital systems are facing. Both use-cases are thoroughly described and analysed, in order to provide a strong and common vision of the validation and demonstration activities to be developed in deliverable 5.4 [3].

Regarding the Connected Car vertical, we have described five scenarios involving the security of CACC platoons: 1) Basic scenario, evaluated the security of CACC platoons by means of formal verification and experiments. More specifically, it evaluated the effectiveness of injection attacks against CACC platoons as well as the effectiveness of plausibility countermeasures against such attacks; 2) Firewall updates scenario extended the basic scenario and developed an I2V case study to investigate how to maintain continuous compliance when security requirements are dynamic; 3) Verification tooling scenario focused on verification tools and scenarios to evaluate the security of

[30] https://github.com/cybertier/Backstabbers-Knife-Collection

the basic scenario by means of penetration tests; 4) Safety and security compliance assessment and certification scenario considered the generation of assurance cases for certification standards, in particular the management tool OpenCert is considered to assist assurance process of the CACC platoon; and 5) Fault-injection and analysis of faulty scenarios described the first steps towards investigating the impact of component faults for the safety and security of CACC platoons, in particular the Sabotage tool is considered to simulate how a fault can affect the vehicle behaviour. We have also deployed a continuous assessment pipeline using CAPE tools in the context of the Connected Car vertical.

Concerning the e-Government vertical, namely the innovative authentication solutions based on the usage of the Italian national electronic identity card, we have provided the details about the identified demonstration scenarios for the CIE ID APP and the SAML IdP, including the involved actors. We have selected the CAPE tools and defined the corresponding security requirements they are able to evaluate. To assess the security of the CIE ID APP and the SAML IdP, we have deployed two DevSecOps pipelines. Finally, we have specified the assets we can provide to allow end-users to include the CAPE assessment tools in their pipeline and perform a security assessment of their complex systems.

The technical details on how we integrated the CAPE assessment tools in the development and testing environments are provided in D5.3 [2], while the results concerning the continuous assessment framework of the vertical will be reported in D5.4 [3]. Benchmarking activities of the demonstrators with other developments in the field will be carried out and reported in D5.4.

Finally, in terms of governance, the CAPE program demonstrates a cooperative mode of management. Several tools have the same (or very close) assessment targets. Rather than implement two times the same tool (with different techniques), we harmonized the specification of the tools so that they had complementary goals. This implemented a cooperating rather than a competing governance model, focusing on leveraging synergies and competencies between researchers to extend the coverage of our research activities. The joint design and sharing of the two verticals is also representative of the governance of CAPE, where people, competencies and platforms are collaboratively shared to elaborate advanced research platforms.

Chapter 9 List of Abbreviations

Abbreviation	Translation
ACC	Adaptive Cruise Control
ACSL	A Common Specification Language
ADAS	Advanced Driver Assistance System
AI	Artificial Intelligence
ALM	Application Lifecycle Management
API	Application Programming Interface
AST	Abstract Syntax Tree
CACC	Cooperative Adaptive Cruise Control
CI/CD	Continuous Integration / Continuous Distribution
CIE	Italian national electronic identity card
CPE	Common Platform Enumeration
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
DAST	Dynamic Analysis
DFMEA	Design FMEA
ECU	Engine Control Unit
EMF	Eclipse Modelling Framework
FARM	Faults, Activation, Readouts, Measures
FMEA	Failure Modes and Effects Analysis
FTA	Fault Tree Analysis
FTP	File Transfer Protocol
GAN	Generative Adversarial Network-based
GPG	GNU Privacy Guard
GPU	Graphics Processing Unit
GSN	Goal Structure Notation
GUI	Graphical User Interface
HARA	Hazard Analysis and Risk Assessment
HMS	Hardware Security Modules
HTML	HyperText Markup Language
HTTP(S)	Hypertext Transfer Protocol Secure
I2V	Infrastructure to Vehicle
ICC	Inter-component communication
ICFG	Inter-Procedural Control-Flow Graph

Abbreviation	Translation
IDS	Intrusion Detection System
IMAP	Internet Message Access Protocol
IP	Internet Protocol
ISO	International Organization for Standardization
JAR	Java archive
KAOS	Keep All Objectives Satisfied
KVM	Kernel-based Virtual Machine
NFC	Near Field Communication
NFV	Network Functions Virtualization
NLP	Natural Language Processing
NVD	National Vulnerability Database
OSCS	Open Source Case Studies
OSS	Open-Source Software
OWASP	Open Web Application Security Project
PFMEA	Process FMEA
PLM	Product Lifecycle Management
PP	Protection Profile
PURL	Persistent URL
RAICC	Revealing Atypical Inter-Component Communication
SAML	Security Assertion Markup Language
SARIF	Static Analysis Results Interchange Format
SAST	Static Application Security Testing
SCAP	Security Content Automation Protocol
SDLC	Software Development Lifecycle
SFC	Service Function Chains
SIEM	Security Information and Event Management
SMS	Short Message Service
SMTP	Simple Mail Transfer Protocol
SoS	System of Systems
SR	Software Requirement
SSH	Secure SHell
SW	Software
TARA	Threat Analysis and Risk Assessment
TOE	Target Of Evaluation
TRL	Technology Readiness Level

Abbreviation	Translation
UC	Use Case
UML	Unified Modelling Language
UR	User Requirements
URL	Uniform Resource Locator
V&V	Verification & Validation
V2I	Vehicle to Infrastructure
VCS	Vehicle Communication Device
VCM	Vehicle Control Module
SysML	Systems Modelling Language
VM	Virtual Machine
XML	Extensible Markup Language
YAML	Yet Another Markup Language

Chapter 10 Bibliography

[1] SPARTA CAPE D5.1 “Asessment specifications and roadmap”, 31st January 2020 https://www.sparta.eu/assets/deliverables/SPARTA-D5.1-Assessment-specifications-and-roadmap-PU-M12.pdf.

[2] SPARTA CAPE D5.3 “Demonstrator prototypes”, January 2021.

[3] SPARTA CAPE D5.4 “Demonstrators evaluation”, January 2022.

[4] European Commission - Funding & tender opportunities - Single Electronic Data Interchange Area (SEDIA) - FAQ, https://ec.europa.eu/info/funding-tenders/opportunities/portal/screen/support/faq/2890 (accessed Jan 25, 2021)

[5] European Commission - HORIZON 2020 - WORK PROGRAMME 2014-2015 - General Annexes, https://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf

[6] GSN Community Standard Version 1. 2011. Available at http://www.goalstructuringnotation.info/documents/GSN_Standard.pdf.

[7] M. Gleirscher and C. Cârlan. “Arguing from hazard analysis in safety cases: A modular argument pattern”. In HASE, 2017.

[8] L. Duan, S. Rayadurgam, M. P. E. Heimdahl, A. Ayoub, O. Sokolsky, and I. Lee. “Reasoning about confidence and uncertainty in assurance cases: A survey”. In SEHC 2017.

[9] C. Ponsard, G. Dallons, P. Massonet. “Goal-Oriented Co-Engineering of Security and Safety Requirements in Cyber-Physical Systems”. SAFECOMP Workshops 2016: 334-345.

[10] A. van Lamsweerde. “Systematic Requirements Engineering from System Goals to UML Models to Software Specifications”. Wiley 2009.

[11] A. Kondeva, C. Carlan, H. Ruess, and V. Nigam. “On Computer-Aided Techniques for Supporting Safety and Security Co-Engineering”. In *The 9th IEEE International Workshop on Software Certification WoSoCer*, 2019.

[12] ED 202A: Airworthiness security process specification. https://standards.globalspec.com/std/9862360/eurocae-ed-202.

[13] SAE J3061: Cybersecurity guidebook for cyber-physical vehicle systems. https://www.sae.org/standards/content/j3061/.

[14] C. Baral. “Knowledge Representation, Reasoning and Declarative Problem Solving”. In CUP 2010.

[15] C. Preschern, N. Kajtazovic, and C. Kreiner. “Security Analysis of Safety Patterns”. In PLoP 2013.

[16] GSN Community Standard Version 1, 2011. Available at http://www.goalstructuringnotation.info/documents/GSN_Standard.pdf.

[17] Common Criteria for Information Technology Security Evaluation, Version 3.1, revision 5, April 2017. Part 1: Introduction and general model.

[18] Common Criteria for Information Technology Security Evaluation, Version 3.1, revision 5, April 2017. Part 2: Functional security components.

[19] Common Criteria for Information Technology Security Evaluation, Version 3.1, revision 5, April 2017. Part 3: Assurance security components.

[20] G. Lowe. “Breaking and fixing the Needham-Schroeder public-key protocol using FDR.” In TACAS, pages 147–166, 1996.

[21] R. W. van der Heijden, T. Lukaseder, and F. Kargl. “Analyzing attacks on cooperative adaptive cruise control (CACC)”. In 2017 IEEE Vehicular Networking Conference, VNC, pages 45–52. IEEE, 2017.

[22] S. Hyun, J. Song, S. Shin, and D. Bae. “Statistical verification framework for platooning system of systems with uncertainty”. In APSEC, pages 212–219. IEEE, 2019.

[23] M. Kamali, L. A. Dennis, O. McAree, M. Fisher, and S. M. Veres. “Formal verification of autonomous vehicle platooning”. Sci. Comput. Program., 148:88–106, 2017.

[24] C. Talcott, V. Nigam, F. Arbab, and T. Kappe. “Formal specification and analysis of robust adaptive distributed cyber-physical systems”. In M. Bernardo, R. D. Nicola, and J. Hillston, editors, SFM. 2016.

[25] C. L. Talcott, F. Arbab, and M. Yadav. “Soft agents: Exploring soft constraints to model robust adaptive distributed cyber-physical agent systems”. In Software, Services, and Systems - Essays Dedicated to Martin Wirsing, pages 273–290, 2015.

[26] I. Mason, V. Nigam, C. L. Talcott, and A. V. D. Brito. “A framework for analyzing adaptive autonomous aerial vehicles”. In SEFM, pages 406–422, 2017.

[27] S. Bistarelli, U. Montanari, and F. Rossi. “Semiring-based constraint satisfaction and optimization”. J. ACM, 44(2):201–236, 1997.

[28] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. “All About Maude: A High-Performance Logical Framework”, volume 4350 of LNCS. Springer, 2007.

[29] A. Anwar, A. Abusnaina, S. Chen, F. Li and D. Mohaisen. “Cleaning the NVD: Comprehensive Quality Assessment, Improvements, and Analyses” (2020) https://arxiv.org/pdf/2006.15074.pdf.

[30] Palo Alto Networks: “The State of Exploit Development: 80% of Exploits Publish Faster than CVEs” (2020) https://unit42.paloaltonetworks.com/state-of-exploit-development/.

[31] Y. Dong, et al. “Towards the Detection of Inconsistencies in Public Security Vulnerability Reports” (2019) https://www.usenix.org/system/files/sec19-dong.pdf.

[32] A. Blum, T. Mitchell. “Combining labelled and unlabelled data with co-training”. In: Proceedings of the Eleventh Annual Conference on Computational Learning Theory, Association for Computing Machinery, New York, NY, USA, COLT’ 98, p 92–100, DOI 10.1145/279943.279962, URL https://doi.org/10.1145/279943.279962.

[33] A. Levin, P. Viola, and Y. Freund. “Unsupervised improvement of visual detectors using co-training”. IEEE, p 626, 2003

[34] M. F. Balcan, and A. Blum. “A pac-style model for learning from labeled and unlabeled data”. In: International Conference on Computational Learning Theory, Springer, pp 111–126, 2005

[35] A.D. Sawadogo, T.F. Bissyand, N. Moha, K. Allix, J. Klein, L. Li, and Y. Le Traon. “Learning to catch security patches”. arXiv-2001.09148, 2020

[36] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. “Suggesting accurate method and class names”. 38–49.

[37] M. Allamanis, E.T. Barr, P. Devanbu, and C. Sutton. “A survey of machine learning for big code and naturalness.” ACM Computing Surveys (CSUR)51, 4 (2018), 81.

[38] M. Allamanis, H. Peng, and C. A. Sutton. “A convolutional attention network for extreme summarization of source code”. CoRR abs/1602.03001 (2016).

[39] U. Alon, S. Brody, O. Levy, and E. Yahav. “code2seq: Generating sequences from structured representations of code”. arXiv preprint arXiv: 1808.01400 (2018).

[40] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. “code2vec: Learning distributed representations of code”. Proceedings of the ACM on Programming Languages 3, POPL (Jan. 2019), 40:1–40:29.

[41] Z. Chen, and M. Monperrus. “A literature study of embeddings on source code”. arXiv preprint arXiv: 1904.03061 (2019).

[42] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. “On the naturalness of software”. In Software Engineering (ICSE), 2012 34th International Conference on (2012), IEEE, pp. 837–847.

[43] T. Mikolov, K. Chen, G. Corrado, and J. Dean. “Efficient estimation of word representations invector space”. arXiv preprint arXiv: 1301.3781 (2013).

[44] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. “Convolutional neural networks over tree structures for programming language processing”. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (2016), AAAI’16, AAAI Press, pp. 1287–1293.

[45] V. Raychev, M. Vechev, and E. Yahav. “Code completion with statistical language models”. SIGPLAN Not. 49, 6 (June 2014), 419–428.

[46] C. D. Santos, and B. Zadrozny. “Learning character-level representations for part-of-speech tagging”. 1818–1826.

[47] B. Wang, X. Yang, and G. Wang. “Detecting copy directions among programs using extreme learning machines”. Mathematical Problems in Engineering 2015(05 2015), 1–15.

[48] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt. “Learning to represent edits”. arXiv preprint arXiv:1810.13337 (2018).

[49] M. Ohm, H. Plate, A. Sykosch and M. Meier. “Backstabber’s Knife Collection: A Review of Open Source Software Supply Chain Attacks”. DIMVA 2020.

[50] H. Plate, S. E. Ponta, and A. Sabetta. “Impact assessment for vulnerabilities in open-source software libraries”. In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 411-420). IEEE. 2015.

[51] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci. “Vulnerable open source dependencies: Counting those that matter”. In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (pp. 1-10). 2018.

[52] S. E. Ponta, H. Plate, and A. Sabetta. “Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software”. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 449-460). IEEE. 2018.

[53] S. Trabelsi, H. Plate, A. Abida, M. M. B. Aoun, A. Zouaoui, C. Missaoui, and A. Ayari. “Mining social networks for software vulnerabilities monitoring”. In 2015 7th International Conference on New Technologies, Mobility and Security (NTMS) (pp. 1-7). IEEE. 2015.

[54] M. M. Casalino, M. Mangili, H. Plate, and S. E. Ponta. “Detection of configuration vulnerabilities in distributed (web) environments”. In International Conference on Security and Privacy in Communication Systems (pp. 131-148). Springer, Berlin, Heidelberg. 2012.

[55] H. Plate, S. Ponta, and A. Sabetta. U.S. Patent No. 9,792,200. Washington, DC: U.S. Patent and Trademark Office. 2017.

[56] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont. “A manually curated dataset of fixes to vulnerabilities of open-source software”. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR) (pp. 383-387). IEEE. 2019.

[57] T. Elliott. “The state of the octoverse: top programming languages of 2018”. https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/ (November 2018) (accessed Jan 22, 2021).

[58] Lutoma. “PSA: There is a fake version of this package on PyPI with malicious code”. https://github.com/dateutil/dateutil/issues/984. 2019.

[59] S. Torabi, and M. Wahde. “Fuel-Efficient Driving Strategies for Heavy-Duty Vehicles: A Platooning Approach Based on Speed Profile Optimization”. In Journal of Advanced Transportation. Volume 2018. 2018. https://doi.org/10.1155/2018/4290763.

[60] Taken from: White Paper: “Automated Driving and Platooning Issues and Opportunities”. Automated Driving and Platooning Task Force. ATA Technology and Maintenance Council. Future Truck Program. 2015.

[61] R. W. van der Heijden, T. Lukaseder, and F. Kargl. “Analyzing attacks on cooperative adaptive cruise control (CACC)”. In 2017 IEEE Vehicular Networking Conference, VNC, pages 45–52. IEEE, 2017.

[62] M. Lorio, F. Risso, R. Sisto, A. Buttiglieri, and M. Reineri. “Detecting injection attacks on cooperative adaptive cruise control”. In 2019 IEEE Vehicular Networking Conference, VNC, pages 1–8, 2019.

[63] Y. Gil Dantas, V. Nigam, and C. Talcott. “A Formal Security Assessment Framework for Cooperative Adaptive Cruise Control”. In IEEE Vehicular Networking Conference (VNC), 2020.

[64] R. W. van der Heijden, T. Lukaseder, and F. Kargl. “Analyzing attacks on cooperative adaptive cruise control (CACC)”. In 2017 IEEE Vehicular Networking Conference, VNC, pages 45–52. IEEE, 2017.

[65] M. Iorio, F. Risso, R. Sisto, A. Buttiglieri, and M. Reineri. “Detecting injection attacks on cooperative adaptive cruise control”. In 2019 IEEE Vehicular Networking Conference, VNC, pages 1–8, 2019.

[66] John Vissers et al, 2018. “V1 Platooning use-cases, scenario definition and Platooning Levels”. D2.2 of H2020 project ENSEMBLE (platooningensemble.eu).

[67] M. Iorio, F. Risso, R. Sisto, A. Buttiglieri, and M. Reineri. “Detecting Injection Attacks on Cooperative Adaptive Cruise Control”. VNC 2019: 1-8

[68] R. Wouter van der Heijden, T. Lukaseder, and F. Kargl. “Analyzing attacks on cooperative adaptive cruise control (CACC)”. VNC 2017: 45-52.

[69] A. Morgagni, P. Massonet, S. Dupont, and J. Grandclaudon. “Towards Incremental Safety and Security Requirements Co-Certification”. EuroS&P Workshops 2020: 79-84.

[70] C. Baral. “Knowledge Representation, Reasoning and Declarative Problem Solving”. In CUP. 2010.

[71] T. Eiter, G. Gottlob, and H. Mannila. “Disjunctive Datalog”. ACM Trans. Database Syst. 1997.

[72] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. “The DLV system for knowledge representation and reasoning”. ACM Trans. Comput. Logic 7, 499–562. 2006.

[73] H. Martin, Z. Ma, Ch. Schmittner, B. Winkler, M. Krammer, D. Schneider, T. Amorim, G. Macher, Ch. Kreiner, Combined automotive safety and security pattern engineering approach, Reliability Engineering & System Safety, Volume 198, 2020, 106773, ISSN 0951-8320, https://doi.org/10.1016/j.ress.2019.106773.

[74] Y. G. Dantas, A. Kondeva, and V. Nigam. “Less Manual Work for Safety Engineers: Towards an Automated Safety Reasoning with Safety Patterns”. In International Conference on Logic Programming (ICLP), 2020.

[75] Y. G. Dantas, A. Kondeva, and V. Nigam. “Towards Automating Safety and Security Co-Analysis with Patterns” (Position Paper). In Safecomp, 2020.

[76] C. Talcott, V. Nigam, F. Arbab, and T. Kappe. “Formal specification and analysis of robust adaptive distributed cyber-physical systems”. In M. Bernardo, R. D. Nicola, and J. Hillston, editors, SFM. 2016.

[77] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. “All About Maude: A High-Performance Logical Framework”, volume 4350 of LNCS. Springer, 2007.

[78] Y. G. Dantas, V. Nigam, and C. Talcott. SoftAgents-Platoon. https://github.com/ygdantas/SoftAgents-Platoon.git. 2020.

[79] I. Mason, V. Nigam, C. L. Talcott, and A. V. D. Brito. A framework for analyzing adaptive autonomous aerial vehicles. In SEFM, pages 406–422, 2017.

[80] V. Nigam and C. L. Talcott. “Formal security verification of industry 4.0 applications”. In ETFA, pages 1043–1050, 2019.

[81] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. “Scandroid: Automated security certification of android applications”. Manuscript, Univ. of Maryland, https://www.cs.umd.edu/~avik/projects/scandroidascaa/ 2(3) (2009).

[82] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. “Chex: statically vetting android apps for component hijacking vulnerabilities”. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 229–240 (2012).

[83] P. P. Chan, L. C. Hui, S. M. Yiu. “Droidchecker: analyzing android applications for capability leak”. In: Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks. pp. 125–136 (2012)

[84] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M.C. Rinard. “Information flow analysis of android applications in droidsafe”. In: NDSS. vol. 15, p. 110 (2015)

[85] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. “Effective real-time android application auditing”. In: 2015 IEEE Symposium on Security and Privacy. pp. 899–914. IEEE (2015)

[86] A. Nirumand, B. Zamani, and B. Tork Ladani. "VAnDroid: A framework for vulnerability analysis of Android applications using a model-driven reverse engineering technique." Software: Practice and Experience 49.1 (2019): 70-99.

[87] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. “Profiledroid: multi-layer profiling of android applications”. In: Proceedings of the 18th annual international conference on Mobile computing and networking. pp. 137–148 (2012)

[88] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. “Copperdroid: Automatic reconstruction of android malware behaviors”. In: Ndss (2015)

[89] M. Y. Wong, and D. Lie. "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware." NDSS. Vol. 16. 2016.

[90] V. Aravantinos, S. Voss, S. Teufl, F. Hölzl, and B. Schätz. “AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems”. ACES-MB&WUCOR@MoDELS 2015: 19-26

[91] L. De Moura and N. Bjørner, “Z3: An efficient SMT solver,” Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340, 2008.

[92] S. Gérard, C. Dumoulin, P. Tessier, and B. Selic. “Papyrus: A UML2 tool for domain-specific language modeling”. In Model-Based Engineering of Embedded Real-Time Systems - International Dagstuhl Workshop, Dagstuhl Castle, Germany, November 4-9, 2007. Revised Selected Papers, ser. Lecture Notes in Computer Science, H. Giese, G. Karsai, E. Lee, B. Rumpe, and B. Schätz, Eds., vol. 6100. Springer, 2007, pp. 361–368.

[93] T. O. M. Group, “Semantics of a Foundational Subset for Executable UML Models (FUML)”. Pearson Higher Education, 2013. [Online]. Available: http://www.omg.org/spec/FUML/1.1

[94] C. Cârlan, V. Nigam, A. Tsalidis, and S. Voss. “ExplicitCase: Tool-support for creating and maintaining assurance arguments integrated with system models”. In IEEE International Workshop on Software Certification, WoSoCer, 2019

[95] Stitching Cuckoo Foundation. “Cuckoo Sandbox - Automated Malware Analysis”. https://cuckoosandbox.org, 2019 (accessed Jan 25, 2021).

[96] Trey Darley and Ivan Kirillov. “STIXTM Version 2.0. Part 3: Cyber Observable Core Concepts”. OASIS Open, committee specification 01 edition, 2017. http://docs.oasis-open.org/cti/stix/v2.0/stix-v2.0-part3-cyber-observable-core.pdf.

[97] J. Woodcok, P. G. Larsen, J. Bicarregui and J. Fitzgerald. “Formal methods: Practice and experience”. In ACM computing surveys (CSUR), Volume 41, nº 4, pages 1-36. ACM New York, NY, USA, 2009.

[98] J. C. Knight, C. L. Colleen, S. Matthew and L.G. Nakano. “Why are formal methods not used more widely?”. In Fourth NASA formal methods workshop, 1997.

[99] A. Milenkovski, M. Vieira, S. Kounev, A. Avritzer and B. D. Payne. “Evaluating Computer Intrusion Detection Systems: A Survey of Common Practices”, ACM Computing Survey 48(1), pp. 1-41, 2015.

[100] M. Ring, S. Wunderlinch, D. Scheuring, D. Landes and A. Hotho. “A Survey of network-based intrusion detection data sets”. Computers & Security 86, pp. 147-167, 2019.

[101] A. Botta, A. Dainotti, and A. Pescapé. “A tool for the generation of realistic network workload for emerging networking scenarios”. Computer Networks 56(15), pp. 3531-3547, 2012.

[102] J. Sommers, H. Kim, and P. Barford. “Harpoon: a flow-level traffic generator for router and network tests”. ACM SIGMETRICS Performance Evaluation Review 32(1), 2004.

[103] M.R. Shahid, G. Blanc, H. Jmila, Z. Zhang and H. Debar. “Generative deep learning for internet of things network traffic generation”. 25th IEEE Pacific Rim International Symposium on Dependable Computing, 2020.

[104] M. Ring, D. Schlör, D. Landes, and A. Hotho. “Flow-based network traffic generation using generative adversarial networks”. Computers & Security 82, pp. 156-172, 2019.

[105] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, “Triggerscope: Towards detecting logic bombs in android applications”, in 2016 IEEE symposium on security and privacy (SP). IEEE, 2016, pp. 377–396.

[106] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, “Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps.” in NDSS, 2017.

[107] Q. Zeng, L. Luo, Z. Qian, X. Du, Z. Li, C.-T. Huang, and C. Farkas. “Resilient user-side android application repackaging and tampering detection using cryptographically obfuscated logic bombs”. IEEE Transactions on Dependable and Secure Computing, 2019.

[108] V. Nigam and C. Talcott. “Formal Security Verification of Industry 4.0 Applications”. In *ETFA*, 2019.

[109] M. Clavel, F. Durán, S. Escobar, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, P. C. Ölveczky, R. Rubio, and C. L. Talcott. “The Maude System”.
Available: http://maude.cs.illinois.edu/w/index.php/The_Maude_System.

[110] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. “Maude: specification and programming in rewriting logic”. Theor. Comput. Sci. 285(2): 187-243 (2002)

[111] P. Borovansky, C. Kirchner, H. Kirchner, P. E. Moreau, and C. Ringeissen. “An overview of ELAN”. In: C. Kirchner, H. Kirchner (Eds.), Proc. 2nd Int. Workshop on Rewriting Logic and its Applications, WRLA’98, Pont-Ya-Mousson, France, September 1– 4, 1998, Electronic Notes in Theoretical Computer Science, Vol. 15, Elsevier, Amsterdam, 1998, pp. 329 –34.

[112] R. Diaconescu, K. Futatsugi, and S. Iida. “Component-based algebraic specification and verification in CafeOBJ”. In: J.M. Wing, J. Woodcock, J. Davies (Eds.), Proc. FM’99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20 –24, Volume II, Lecture Notes in Computer Science, Vol. 1709, Springer, Berlin, 1999, pp. 1644–1663.

[113] C. Talcott, V. Nigam, F. Arbab, and T. Kappe. “Formal specification and analysis of robust adaptive distributed cyber-physical systems”. In M. Bernardo, R. D. Nicola, and J. Hillston, editors, Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems. 2016.

[114] S. Bistarelli, U. Montanari, and F. Rossi. “Semiring-based constraint satisfaction and optimization”. J. ACM, 44(2):201–236, 1997.

[115] I. I. Mason, V. Nigam, C. L. Talcott, and A. Vasconcelos De Brito. “A Framework for Analyzing Adaptive Autonomous Aerial Vehicles”. In Software Engineering and Formal Methods - SEFM 2017 Collocated Workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA, Trento, Italy, September 4-5, 2017, Revised Selected Papers, pp. 406–422, 2017.

[116] M. S. Lund, B. Solhaug, and K. Stolen. “Model-Driven Risk Analysis. The CORAS Approach”. Springer-Verlag, 2011.

[117] B. Karabacak and I. Sogukpinar. “ISRAM: information security risk analysis method”. Computers & Security, 24(2):147-159, 2005.

[118] CLUSIF. Mehari 2010. Risk analysis and treatment guide. Club De La Securite De L'Information Français, August 2010.

[119] M. A. Amutio and J. Candau. “MAGERIT- Methodology for Information Systems Risk Analysis and Management. Book I – The Method”. Ministerio de Hacienda y Administraciones Publicas, 3.0 edition, 2014.

[120] R. A. Caralli, J. F. Stevens, L. R. Young, and W. R. Wilson. “Introduction Octave Allegro: Improving the information security risks assessment process”. Technical Report CMU/SEI-2007-TR-012, Software Engineering Institute, May 2007.

[121] D. Firesmith. “Common Concepts Underlying Safety, Security and Survivability Engineering”. Software Engineering Institute, Carnegie-Mellon University, report CMU/SEI-2003-TN-033,

December 2003. Available for download at http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=655

[122] S. Lautieri, D. Cooper, and D. Jackson. “SafSec: Commonalities Between Safety and Security Assurance”. Thirteenth Safety Critical Systems Symposium, Southampton, 2005.

[123] AMASS H2020 Project, “D2.4 AMASS reference architecture (c)”, 2018, https://amass-ecsel.eu/sites/amass.drupal.pulsartecnalia.com/files/documents/D2.4_AMASS-reference-architecture-%28c%29_AMASS_Final.pdf (accessed Jan 22, 2021)

[124] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi and C. Dangremont, "A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software," 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada, 2019, pp. 383-387, doi: 10.1109/MSR.2019.00064.

[125] NIST National Vulnerability Database (NVD), https://nvd.nist.gov/

[126] https://github.com/google/vulncode-db

[127] U.S. DHS, "Critical Infrastructure Sectors," U.S. Department of Homeland Security, 2015, online https://www.dhs.gov/topics (accessed 2015-2019).

[128] "Electric sector failure scenarios and impact analyses," Electric Power Research Institute, June 2014.

[129] CEN-CENELEC-ETSI Coordination Group on Smart Energy Grids (CG-SEG), "SEGCG/M490/G Smart Grid Set of Standards 22," European Standards Organizations, 2017.

[130] Expert Group on the security and resilience of communication networks and information systems for smart grids, "Cyber Security of the Smart Grids," European Commision

[131] J. Arlat, A. Costes, Y. Crouzet, J. Laprie, and D. Powell. “Fault injection and dependability evaluation of fault-tolerant systems”. IEEE Trans. Comput., 42(8), 913-923. (1993)

[132] L. Yan, X. Li, Y. Yu. “Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes”. In: GLOBECOM 2017 - 2017 IEEE Global Communications Conference, pp 1-7, DOI 10.1109/ GLOCOM.2017.8254428 (2017).

[133] A. Sabetta, M. Bezzi. “A practical approach to the automatic classification of security-relevant commits”. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 579-582, DOI 10.1109/ICSME.2018.00058 (2018)

[134] H. Perl, S. Dechand, M. Smit, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. “VccFinder: Finding potential vulnerabilities in open-source projects to assist code audits”. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA, CCS '15, p 426{437, DOI 10.1145/2810103.2813604, URL https://doi.org/10.1145/2810103. 2813604 (2015)

[135] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. “Spectre attacks: Exploiting speculative execution”. In Proceedings of the 2019 IEEE Symposiumon Security and Privacy. 2019.

[136] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. “Meltdown: Reading kernel memory from user space”. In Proceedings of the 27th USENIX Security Symposium. 2018.

[137] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. “FORESHADOW: Extracting the keys to the intel SGX kingdom with transient out-of-order execution”. In Proceedings of the 27th USENIX Security Symposium. 2018.

[138] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. “Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution”. Proceedings of the 27th USENIX Security Symposium (2018).

[139] “Owasp dependency check”, https://owasp.org/www-project-dependency-check/ (accessed Jan 22, 2021)

[140] "Whitesource", https://www.whitesourcesoftware.com/open-source-security/ (accessed Jan 22, 2021)

[141] S. S Alqahtani, E. E. Eghan, and J. Rilling. "Tracing known security vulnerabilities in software repositories – a semantic web enabled modelling approach". Science of Computer Programming, vol. 121, pp. 153–175,2016.

[142] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries". In 2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015

[143] "Snyk", https://snyk.io/ (accessed Jan 22, 2021).

[144] "BlackDuck", https://www.blackducksoftware.com/technology/vulnerability-reporting (accessed Jan 22, 2021).

[145] SourceClear, "The busy managers guide to open source security", https://www.sourceclear.com/resources/TheBusyManagersGuideToOpenSourceSecurity.pdf, 2017.

[146] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen. "Tracking known security vulnerabilities in proprietary software systems". In Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE22nd International Conference on. IEEE, 2015, pp. 516–519

[147] F. Balarin et al. "Hardware-Software Co-Design of Embedded Systems, The POLIS Approach", 5th ed. KLUWER ACADEMIC PUBLISHERS, 2003.

[148] R. Rosales, M. Glass, J. Teich, B. Wang, Y. Xu, and R. Hasholzner. "MAESTRO - Holistic Actor-Oriented Modeling of Non-functional Properties and Firmware Behavior for MPSoCs". ACM Trans. Des. Autom. Electron. Syst., vol. 19, no. 3, pp. 23:1–23:26, Jun. 2014. http://doi.acm.org/10.1145/2594481.

[149] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna. "UML-based Multiprocessor SoC Design Framework". ACM Trans. Embed. Comput. Syst., vol. 5, no. 2, pp. 281–320, May 2006.

[150] M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. "mbeddr: instantiating a language workbench in the embedded software domain". Automated Software Engineering, vol. 20, no. 3, pp. 339–390, Sep 2013.

[151] P. H. Feiler, B. A. Lewis, and S. Vestal, "The SAE architecture analysis & design language (AADL) a standard for engineering performance critical systems". In Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE, pp. 1206–1211.

[152] P. H. Feiler, B. A. Lewis, S. Vestal, and E. Colbert, "An overview of the SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering". In IFIP-WADL, ser. IFIP, vol. 176. Springer, 2004, pp. 3–15.

[153] T. Lodderstedt, D. A. Basin, and J. Doser. "SecureUML: A UML-Based Modeling Language for Model-Driven Security". In Proceedings of the 5th International Conference on The Unified Modeling Language, ser. UML'02. London, UK, UK: Springer-Verlag, 2002, pp. 426–441

[154] J. Jürjens, "UMLsec: Extending UML for Secure Systems Development", in Proceedings of the 5th International Conference on The Unified Modeling Language, ser. UML '02. London, UK, UK: Springer-Verlag, 2002, pp. 412–425.

[155] Y. Xiao, Y. Jia, C. Liu, X. Cheng, J. Yu and W. Lv. "Edge Computing Security: State of the Art and Challenges", in Proceedings of the IEEE, vol. 107, no. 8, pp. 1608-1631, Aug. 2019, doi: 10.1109/JPROC.2019.2918437.

[156] R. Roman, J. Lopez, M. Mambo. "Mobile edge computing, Fog et al.: A survey and analysis of security threats and challenges", Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.11.009

[157] S. N. Shirazi, A. Gouglidis, A. Farshad and D. Hutchison. "The Extended Cloud: Review and Analysis of Mobile Edge Computing and Fog from a Security and Resilience Perspective", in

IEEE Journal on Selected Areas in Communications, vol. 35, no. 11, pp. 2586-2595, Nov. 2017, doi: 10.1109/JSAC.2017.2760478.

[158] Chadni Islam, Muhammad Ali Babar, and Surya Nepal. “A Multi-Vocal Review of Security Orchestration”. ACM Comput. Surv. 52, 2, Article 37 (May 2019), 45 pages. 2019. DOI: https://doi.org/10.1145/3305268

[159] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang and W. Shi, "Edge Computing for Autonomous Driving: Opportunities and Challenges". In Proceedings of the IEEE, vol. 107, no. 8, pp. 1697-1716, Aug. 2019, doi: 10.1109/JPROC.2019.2915983.

[160] [2] D. L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta. Poster: “Towards Using Source Code Repositories to Identify Software Supply Chain Attacks”. In CCS, 2020.

[161] [3] D. L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta. “Typosquatting and Combosquatting Attacks on the Python Ecosystem”. In IEEE European Symposium on Security and Privacy Workshops, 2020.

[162] [4] M. Ohm, A. Sykosch, and M. Meier. “Towards detection of software supply chain attacks by forensic artifacts”. In ARES, 2020.

Chapter 11 Appendix A: FMEA of the Platooning System

After applying the recommended actions

Item /Function	Requirement	Potential Failure Mode	Potential Effect(s) of Failure	Severity	Potential Cause of Failure	Current Design				RPN	Recommended Action	Action Results				
						Controls Prevention	Occurrence	Controls Detection	Detection			Actions Taken	Severity	Occurrence	Detection	RPN
Image Streamer	Each vehicle should get an image each 10 milliseconds	No image captured	Vehicle not able to follow the lane and get out of road	10	HW failure	-	3	Initialisation test	5	150	When the initialization test detects it, do not run.					
				10	Camera communication failure	-	3		5	150						
				10	Camera driver failure	-	3		5	150						
				10	An attacker points to the cameras with a powerful light causing an inoperative camera during seconds.	-	3	-	10	300	Check incoming values and restart the function					
Lane Detection	Each vehicle should detect white lines on the road and calculate a optimums trajectory of 10 points	No trajectory defined	Vehicle not able to follow the lanes and get out of road	10	Physical attack: Modification of the environment (road) with the aim of confusing the vehicles	-	3		10	300	-					
				10	Lines are not detected, or no correct lines	Adjust the camera parameters	4	Camera remote visualization	10	400	Set a sensor to auto-adjust the Gain and					

Item /Function	Requirement	Potential Failure Mode	Potential Effect(s) of Failure	Severity	Potential Cause of Failure	Current Design				RPN	Recommended Action	Action Results				
						Controls Prevention	Occurrence	Controls Detection	Detection			Actions Taken	Severity	Occurrence	Detection	RPN
		Wrong trajectory defined		10	detected because of de quality of the image (bad Gain, Exposure and/or Focus) or not well-defined function parameter (gradient, intensity...)	and function parameters before running the vehicle	4	-	10	400	Exposure parameters or if a trajectory is not detected the vehicle ignore wrong images					
				10	Physical attack: Modification of the environment (road) with the aim of confusing the vehicles	-					-					
		Trajectory is not defined in the middle of the lane	The vehicle will not follow the middle of lane	7	Lines are not detected, or no correct lines detected because of de quality of the image (bad Gain, Exposure and/or Focus) or not well-defined function parameter (gradient, intensity...)	Adjust the camera parameters and function parameters before running the vehicle	4	-	10	280	Set a sensor to auto-adjust the Gain and Exposure parameters or if a trajectory is not detected the vehicle ignore wrong images					
		Less than 10 points are detected	The vehicle will not detect the trajectory completely correct	7			4	-	10	280						
Lateral control	Each vehicle should be kept	The position of the	The vehicle gets out of the lane	10	HW failure or mechanical failure	-	3	Impossible to detect	10	300	-					

Item /Function	Requirement	Potential Failure Mode	Potential Effect(s) of Failure	Severity	Potential Cause of Failure	Current Design				RPN	Recommended Action	Action Results				
						Controls Prevention	Occurrence	Controls Detection	Detection			Actions Taken	Severity	Occurrence	Detection	RPN
	between the two lines	servomotor is not correct	Insufficient or Excessive lateral control adjustment, vehicle is not on the centre of the lane	10		-	3		10	300	-					
		The servomotor does not move	The vehicle gets out of the lane	10		-	3		10	300	-					
Longitudinal control	The vehicle shall drive to the established speed	Loss of longitudinal control	The vehicle does not respond to the establish speed, possible crash with other vehicles	10	Hardware failure	-	3	Impossible to detect	10	300	Modify the speed depending on distance sensor to maintain the most appropriate velocity and to avoid a collision					
		Unintended acceleration or brake		10		-	3		10	300						
Communication	Each follower vehicle should be connected with the Leader via Wi-Fi and the Leader vehicle should send its speed each 10 millisecond	Connection lost	No speed communication between vehicles, possible crash	10	Denied of Service attack (DoS)	-	5	-	10	500	Check there is connection all time, if not restart the communication					
				10	• Poor Wi-Fi signal • Hardware and physical infrastructure not optimal for data transfer, or corrupted or buggy	-	4	-	10	400						
		Delay	Followers do not set the current speed of the Leader	7		-	4	-	10	280	Check there is a delay and restart the communication					
		Wrong value	The follower vehicles do not have the leader speed and could crash	10		-	4	-	10	400	Plausibility checks. Evaluate last data values and get the consistent value					
				10	A malicious attack. The vehicle in front is sending a malicious value	-	4	-	10	400						

Item /Function	Requirement	Potential Failure Mode	Potential Effect(s) of Failure	Severity	Potential Cause of Failure	Current Design				RPN	Recommended Action	Action Results				
						Controls Prevention	Occurrence	Controls Detection	Detection			Actions Taken	Severity	Occurrence	Detection	RPN
		Interferences	Short period of loss of data	8	Some wireless devices are using the same frequency	-	4	-	10	320	Check if there are interferences and restart if it is needed					

Chapter 12 Appendix B: Protection Profile for a Safety and Security Platooning Management Module

A base Protection Profile (PP) for a Safety and Security Platooning Management Module (**SafSecPMM**) is described in detail in the document SPARTA-D5.2-TEC-R-M24_AppendixB.