SPARTA

D5.4

Demonstrators evaluation

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	Demonstrator
Deliverable reference number	SU-ICT-03-830892 / D5.4/ V1.0
Work package contributing to the deliverable	WP5
Due date	Jan 2022 – M36
Actual submission date	2nd February, 2022

Responsible organisation	LEO
Editor	Malacario Mirko
Dissemination level	PU
Revision	V1.0

Abstract	This deliverable aims to validate tools and processes developed during the CAPE programme by performing evaluability activities using Vertical 1 & Vertical 2 use cases.
Keywords	Cybersecurity, Evaluability, Certification, Standard, Process, Protection Profile, Security Requirements.

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

Editor

Mirko Malacario (LEO)

Contributors

André Maroneze, Loïc Correnson (CEA)
Philippe Massonnet, Sébastien Dupont, Guillaume Ginis (CETIC)
Yuri Gil Dantas (FTS)
Henrik Plate (SAP)
Marc Ohm (UBO)
Víctor Jiménez (EUT)
Cristina Martínez, Estibaliz Amparan, Angel López (TEC)
Paul-Henri Mignot, Gregory Blanc (IMT)
Andrea Bisegna, Roberto Carbone, Luca Verderame (CINI)
Gabriele Restuccia, Alessandro Pellegrini, Francesco Quaglia (CNIT)
Artsiom Yautsiukhin (CNR)
Claudio Porretti, Nicoletta Imperatori (LEO)
Andrius Bambalas (MRU)
Jordan Samhi, Jacques Klein (UNILU)

Reviewers

Maximilian Tschirschnitz (TUM)
Rimantas Zylius (L3CE)

Disclaimer

Executive Summary

The main objective of Task 5.4 is the validation of the tools developed in the CAPE program through a demonstration of the verticals described in D5.2 [2]. This deliverable (that includes twelve appendixes), is a supporting document for showing the results reached by the task.

For reaching this objective, this document introducesd the concept of “evaluability”. Evaluability is the process of verifing whether the set of evidences, consisting of documentation and the Target Of Evaluation (the product or the system under assessment), are sufficient to carry out an evaluation process for Cybersecurity certification.

In order to perform the evaluability task, the tools involved in the verticals have produced a set of evidences, based on Common Criteria standard.

Results from that evidences were verified as complete from an evaluability perspective by checking their completeness and their quality. In particular, they allow to understand the gap to be filled before starting a formal evaluation allowing the owner of the product to make the necessary changes timely.

Table of Content

List of Figures

List of Tables

Chapter 1 Introduction

During the last decades the context of “Cybersecurity” certification has evolved, trying to follow the fast evolution of the cyberspace in terms of growing number of devices interconnected and the complexity of softwares running on top of them.

This evolution led to the creation of a plethora of security certifications as described in the D11.3 [6] bringing a set of advantages and disadvantges.

After the end of the cold war the certification scheme developed in Europe for the certification of IT products was Common Criteria.

This deliverable is the output the last task of WP5 (also named as CAPE - Continuous Assessment in Polymorphous Environments that is one of the four programs of the SPARTA project):

- **T5.1 - Assessment procedures and tools:** during the task 5.1 tools and methods for continuous assessment and certification have been defined.
- **T5.2 - Convergence of security and safety**: during the task 5.2 have been studied techniques and specifications in order to integrate security and safety aspects
- **T5.3 - Risk discovery, assessment and management for complex systems of systems**: during the task 5.3 has been addressed security requirements on SoS using modern software engineering methods
- **T5.4 - Integration on demonstration cases and validation**: the objective of task 5.4 is to validate tools and techniques described in T5.1, T5.2 and T5.3 in the CAPE verticals.

In task 5.1 “Assessment Prodedures and Tools” and D5.1 [1], starting from the V-Model adopted for the development life cycle, Common Criteria activities was mapped to each V-model step .

The following Figure 1 shows the iterative process applied to the secure development of a system/product. In the figure, both Safety Engineering and Security Engineering processes go in parallel with the cybersecurity certification process (Common Criteria standard has been used in this specific case).

For further details about the construction of this framework please refer to the D5.1 “Assessment specifications and roadmap” [1].

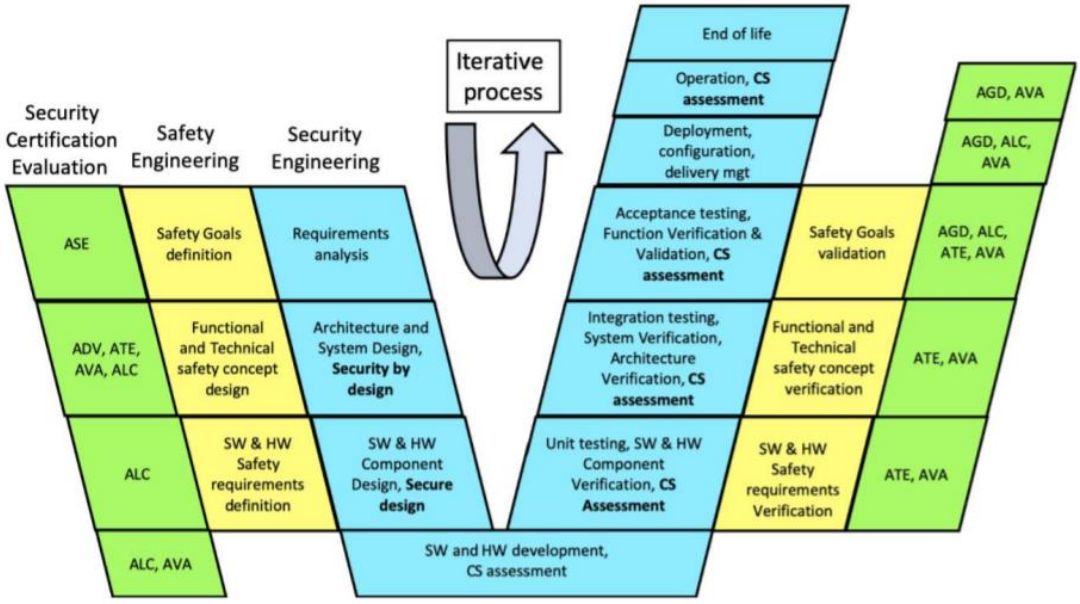

Figure 1: V-model for cybersecurity certification, safety engineering and security engineering

Activities performed in task 5.4, and summarized in this deliverable D5.4, show how the tools of the CAPE program, used in the different V-Model phases of the Verticals life-cycle, produce a set of evidences evaluable against a future cybersecurity certification scheme.

Appendix L has been created in order to provide a demonstration of standalone tools or tools not fully integrated in the two verticals, or for showing specific results not directly related to the context of the verticals.

1.1 Structure of the Document

This deliverable has been structured as following:

- Chapter 1 Introduction, presents an introduction to the activities performed in the task 5.4 and the structure of this deliverable.
- Chapter 2 Validation process definition, presents the validation process that has been defined for performing Task 5.4. In particular, the validation of CAPE tools will be made through selected activities related to Common Criteria certification. The objective is not to perform a Common Criteria evaluation but to demonstrate how the tools developed in the scenarios of verticals are able to produce evidences evaluable with respect to a security certification scheme such as Common Criteria.
- Chapter 3 Vertical 1: Demonstration of converging tools for assessing Connected and Cooperative Car Cybersecurity (CCCC) in the context of Euro NCAP, presents the result of evaluability activities performed with tools involved in Vertical 1 use case.
- Chapter 4 Vertical 2: Demonstration of a Complex System Assessment Including Large Software and Open Source Environments, Targeting e-Government Services, presents the result of evaluability activities performed with tools involved in Vertical 2 use case.
- Chapter 5 Summary and Conclusions, collects considerations resulting the activities performed during the execution of task 5.4.
- Appendixes from A to H collect the output report from the tools involved in the Vertical 1 as results of the Task 5.4.
- Appendixes from I to K collect reports from the tools involved in the Vertical 2 as results of the Task 5.4.
- Appendix L contains a set of additional tests and results performed by tools outside the two verticals scope.
- Appendix M contains a short description of the videos developed for demonstration purpose.

Chapter 2 Validation process definition

2.1 Purpose

Task 5.4 provides a demonstration that activities, techniques and tools resulting from previous tasks (T5.1, T5.2 and T5.3), applied to CAPE verticals, can provide a product evaluable with respect to a future unified Certification Scheme based on Common Criteria ([7], [8] and [9]), considering that EUCC has selected the Common Criteria as basis for the future cybersecurity certification scheme, successor of the SOG-IS.

It is important to highlight that the objectives of the task is not to certify the products/ realized with the verticals, nor the tools developed in CAPE program.The objective is to demonstrate how the tools, developed in the scenarios of verticals, are able to produce evidences evaluable with respect to a future security certification scheme.

The goal of the security process is to start from existing consolidated concepts (e.g. the V-Model) and build on them “agile” procedures and tools that allow the application of security also to complex systems and services. This does not mean less security, but a better distribution of security throughout the product / system / process life cycle. In other terms the security activities are not performed only at the end of the process, for example performing security configuration as the hardening, but including the security since the early stages by clearly define the security requirements and security objectives to be reached).

This can be achieved by evaluating that that security measures, distributed to the whole life-cycle, have been implemented.

For clarification, the term “evaluation” will not be referred in the following of this document, because this term is in general related to a formal certification. The term “Evaluability” will be used instead. (for further detail see section 2.2), meaning the effectiveness of the tools in producing evidences for a future security certification.

The objective of the task, which was to test the tools developed in CAPE program, was therefore achieved by using the tools to verify the evaluability of the two verticals, according to a future unified certification schema based on Common Criteria.

2.2 The Concept of Evaluability

As introduced in the previous section, the activities performed during the execution of T5.4 have yielded a set of evidences related to a future unified Certification Scheme. Such evidences allow the evaluator to perform “Evaluability” activities in orderto define whether the set of evidences provided, consisting of documentation and the Target Of Evaluation[1], are sufficient to carry out an evaluation process for Cybersecurity certification purposes.

In particular, evaluability activities performed on single evidences have the objective to define whether the evidences themselves have been created in a way that, in the future, can be used for evaluation purposes (in this way the tool that developed that evidence can be tested assessing its effectiveness in the context of evaluability).

Through two distinct use cases, the Task 5.4 explores and verifies that all the elements deriving from the activities and tools developed in the other tasks of the WP5, applied to the single use case, could lead to a lean and agile definition and verification of the security requirements characterizing the prototypes.

It is important to introduce two important terms that are used in the following:

[1] Product or System under assessment

- Developer: when we refer to the Developer in the context of the certification process it is intended the natural/legal person that develop the product under certification. This person is also in charge to provide the evidences for the future certification.
- Evaluator: the evaluator is the natural/legal person in charge to perform the evaluation of the evidences against the Certification Scheme. During a formal evaluation he/she is part of an accredited Cybersecurity Evaluation Facility (see also [6]).

Due to the fact that "evaluability" activities, are not considered as a formal certification, they can be performed directly by developers (although having an"evaluability" team different from the development team could help to obtain better results) in order to "internal assess" the evidences produced during the life-cycle of the product/system that will be certified.

In this way the "evaluability team" can evaluate the completeness of the evidences and the improvement area that need to be implemented to the product/system and evidences for filling the gap.

The evaluability activities represent an important shift-left [2]in the certification process because if the evaluability process finds an improvement area in a timely manner, the cost to fill the gap is minor respect of discovering it at the end of the development (with the risk of still developing a large part of the product/system).

This is the means of performing "better distribution of security throughout the life-cycle".

2.3 Evaluability approach for the CAPE program

In order to meet the goals introduced in the previous sections, the evaluability approach for the CAPE program is strictly related to the tools developed and involved in the scenarios of the following two verticals:

- Vertical 1 – Demonstration of converging tools for assessing Connected & Cooperative Car Cybersecurity (CCCC) in the context of Euro NCAP.
- Verticals 2 – Demonstration of a complex system assessment including large software and open source environments, targeting e-government services.

Further details about the two verticals can be founded in the D5.2 [2] and D5.3 [4].

The decision of performing an evaluability activitiy of the outputs of the tools in the context of the two verticals clearly defines the boundaries and the purpose of the tool testing.

As introduced in Chapter 1, in the following, only the tools fully integrated within the two verticals have been taken into account for the concept of evaluability purpose. Test results coming from the other tools outside the context of the two verticals are shown in the dedicated Appendix L.

The concept of Secure-by-design process has been taken into account for the CAPE program. In particular, it is possible to map the security engineering activities with the Common Criteria Assurance Class as depicted in Figure 2.

[2] Shift-left testing is an approach to software testing and system testing in which testing is performed earlier in the lifecycle. In this case the term has been used also for highlighting that evaluability activities allow to discover possible non-compliance to the certification scheme during the development life-cycle instead to discover them at the end of the process causing an incrementation of costs.

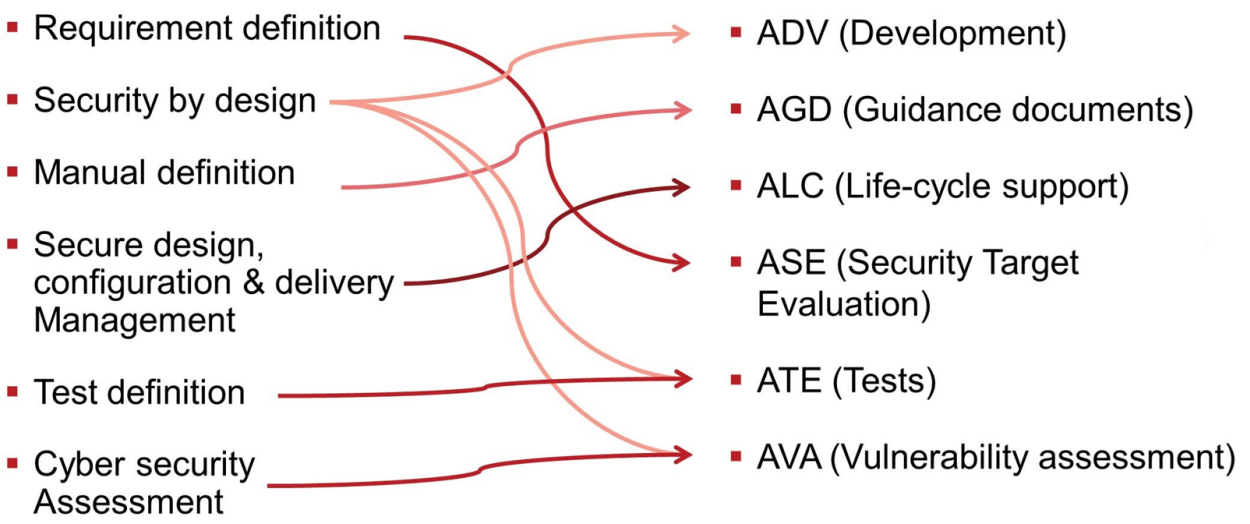

Figure 2: Security-by-design phases and Common Criteria Assurance classes

As introduced in the D5.3 [4] the Common Criteira Assurance classes are the following:

- "*ASE (Security Target Evaluation): this class deals with the evaluation of the consistency of the "Security Target" which also contains the definition of the security requirements of the TOE, therefore it is closely linked to the security requirements management phase.*
- *ADV (Development): this class deals with the evaluation of the six families of requirement for structuring and representing the security functionality realized by the target of evaluation (TOE) at various levels and varying forms of abstraction that the developer must produce during the product development phase, naturally it is linked to the features of the Secure by design processes adopted by the supplier.*
- *AGD (Guidance Documentation): this class takes care of the evaluation of the manuals that are delivered to the customer. These manuals contain both the secure configuration process of the TOE in its user environment and its safe use methods for each category of defined end-user.*
- *ALC (Life-cycle support): this is a very important class that evaluates all aspects of the management of the TOE during its life cycle: in the development phase in which it is under the responsibility of the developer, during the transitional phase of transport in its final operating environment and of course the management in the operating environment under the responsibility of the customer and the developer, in the hypothesis of maintaining the certification (security patch management).*
- *ATE (Tests): it is the class that takes into consideration all the tests that demonstrate that security functionalities operate according to its design descriptions, both the functional ones proposed by the developer and the independent ones proposed by the evaluators.*
- *AVA (Vulnerability Assessment): this class takes care of vulnerability assessment activity to analyse vulnerabilities in the development and operation of the TOE. Development vulnerabilities are those introduced during its development and these can be minimized with the adoption by the developer of "security by design" processes. Operational vulnerabilities are those that could exploit the weaknesses of non-technical countermeasures to violate the TOE security functionality. This analysis is carried out by the evaluators during TOE evaluation deliverables analysis or from the classic vulnerability analysis performed also adopting automatic tools.*".

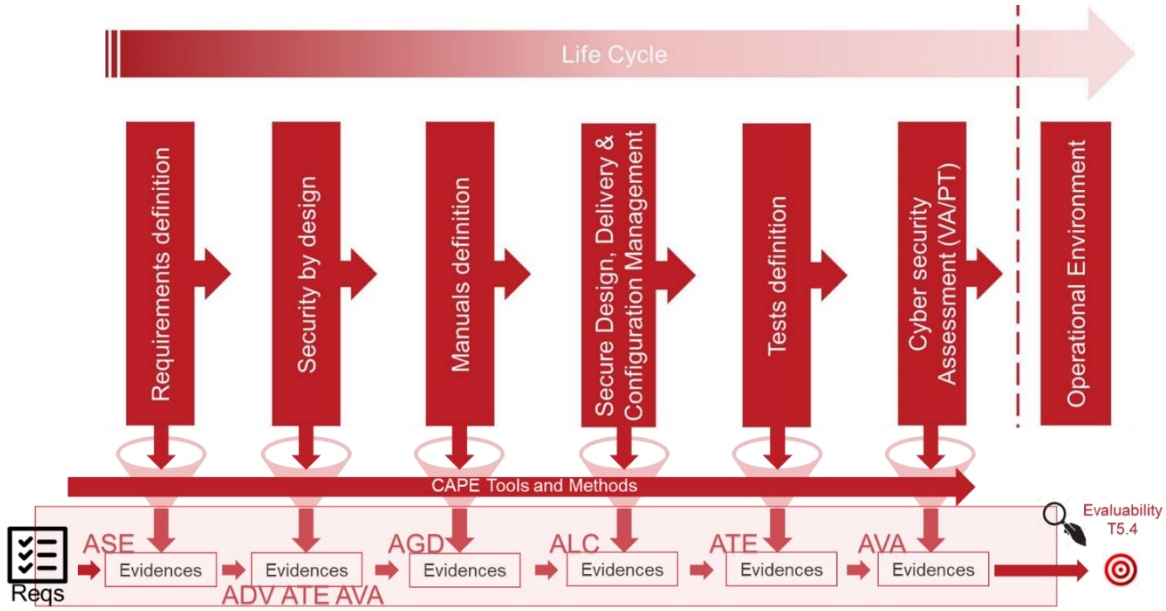

Figure 3: Evidences provided by tools in the life-cycle phases

Figure 3 shows the life-cycle of a product/system through the different phases of the Security Engineering Process. For each phase the application of CAPE tools and/or methods allow to create evidences related to the abovementioned Common Criteria Assurance Classes.

Thus, starting from the requirements up to the operational environment, the evaluability activity has the objective to check the evidences generated by the tools during the different phases in order to verifiy if the security requirements have been addressed and maintained during the whole life-cycle, up to the delivery in the operational environment of the product/system being assessed (in case of T5.4, the two verticals).

Summarizing, the general approach defined for performing the task is the usage of the tools during the different phases of the life-cycle as shown in Figure 3. This approach allows to collect all available evidences and perform an evaluability analysis.

In particular:

- **Requirement Definition** phase allow to collect evidences of the ASE class (related to Security Target/Protection Profile). The objective of this phase is to ensure that the requirements have been clearly defined.
  For the two verticals, the evaluability starts from two different perspectives: in Vertical 1 it starts from requirements clearly stated in the Protection Profile defined during the task 5.2 [3], while in Vertical 2 it starts from a tailoring of security requirements stated in common methodologies such as OWASP and CWE.
- **Security by design** phase has the objective to check that the previously defined requirements have been correctly addressed in the design phase (design is intended for both high-level/system design and low-level/component design). These activities can be grouped in the Common Criteria ADV, ATE and AVA Assurance Classes.
- **Manual Definition** phase has the objective to check that the guidance documentation for all users roles of the certified product/system is clearly defined in order to maintain the implementation of security requirements during the operational life of the product/system. This activity can be translated in the AGD CC Assurance Class.
- **Secure Design, Delivery & Configuration Management** phase has the objective to guarantee that the security requirements are continuously considered in the life-cycle of the product and that the certified product/system and the related evidences are clearly and uniquely identified. This activity can be translated in the ALC CC Assurance Class.

- **Test definition** phase has the objective to define a set of tests that ensure the coverage and verification of the requirements defined in the early stages and managed during the whole life-cycle. This activity can be translated in the ATE CC Assurance Class.
- **Cyber security assessment** phase allows to check the vulnerabilites of the operational environment and the software supporting the product/system under evaluation. This activity can be translated in the AVA CC Assurance Class.

During the execution of the task, other than the Common Criteria Assurance class, another Common Criteria process has been considered: the **Assurance Continuity** [12]. This activity is related to the Security Status Maintenance phase of the life cycle.

In fact, it is important to bear in mind that a product is certified in a "static" way. The security problem solved with a certified product is well defined and freezed. A certificate is issued in respect to a specific version of the product, configured as per certified guidance documents and running in a specific operational environment.

This implies that activities such as Patch Management and Improvement/Evolution of a certified product lead to a modification of the abovementioned boundaries modifying, potentially, the define security problem,

This lead to a set of considerations to be addressed in maintaining the certification of a product due to the "static" approach of a cybersecurity certification.

From a certain point of view, these activities (that we can call as Security Status Maintenance activities) became critical in the recent years due to the fast evolution of the cybersecurity world that requires products to be constantly updated and maintained in order to counter new threats. On the other hand it is important, for a vendor or a users, take advantage by using certified IT products.

The Assurance Continuity process defined in the Common Criteria Assurance Continuity (CCRA) addresses this challenging phase of keeping the product updated and maintaining the Certificate.

Assurance Continuity, as shown in the following Figure 4, has the objective to define a set of actions that can be performed in ordert to maintain the Common Criteria certification of a product during its evolution.

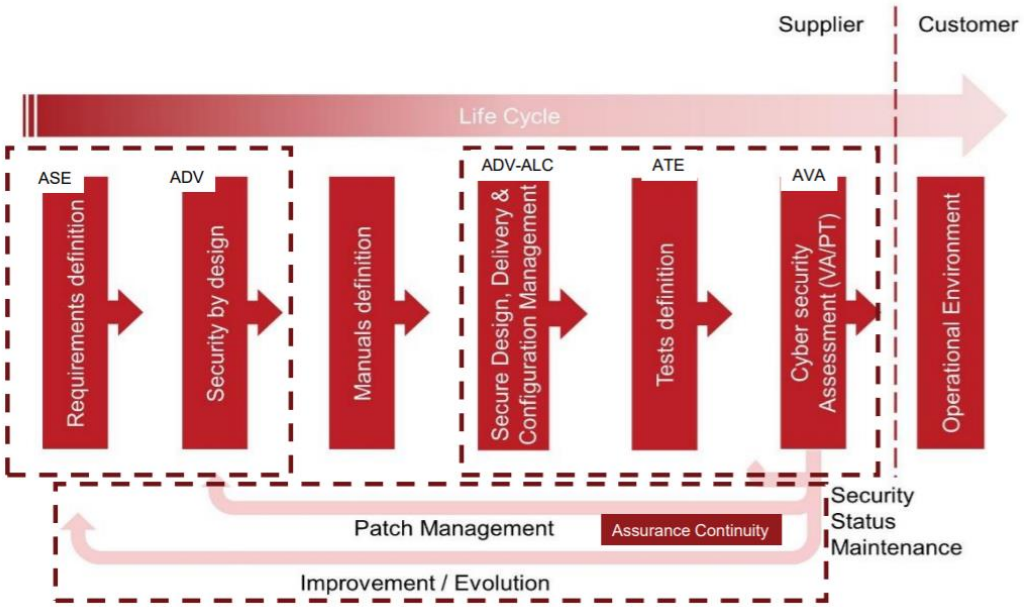

Figure 4: Assurance Continuity for Security Status Maintenance

Chapter 3 Vertical 1: Demonstration of converging tools for assessing Connected and Cooperative Car Cybersecurity (CCCC) in the context of Euro NCAP

3.1 Description

The goal of the Connected Car Vertical (Vertical 1), which has been been fully described in D5.2 [1], is to advance the cyber-security of connected vehicles driving in platoon mode. A platoon is a sequence of vehicles as depicted in Figure 5, that it is composed by a leader vehicle and a sequence of followers.

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.

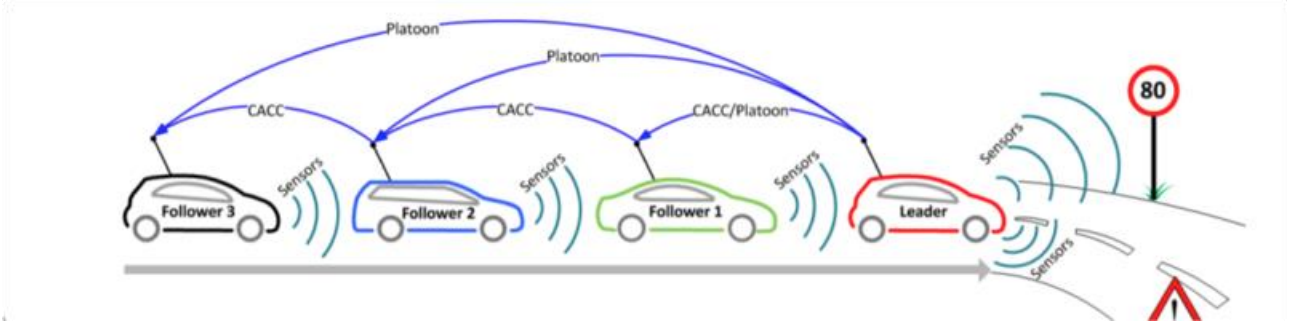

Figure 5: Platooning scenario

This Vertical deals with a platoon consisting of three members, with one leader and two followers using Cooperative Adaptive Cruise Control (CACC). All cars have the same hardware and the same platooning software but with different configurations. The platoon vehicles navigate on the circuit designed and can communicate each other thanks to a WiFi 802.11n access point.

For the purpose of T5.4 the evaluability starts from the reading of the Protection Profile developed in the context of the task 5.2 [3]. The Protection Profile allows the following:

- **Clearly defines the Security Problem** affecting the Platooning Management Module
- Security Problem Definition states **Security Objectives for the TOE** and its **Operational Environment** that **counter** the **Threats** and **address Assumptions and Security Policies**
- Security Objectives for the TOE are reached through the implementation of a set of requirements (SFRs listed in the PP).

The Evaluability task aims to check that the selected requirements (see section 3.3.8) have been correctly implemented during the entire life-cycle of the Vertical 1. This has been done by verifying the evidences of tools developed in WP5 for Vertical 1.

In T5.4 it was demonstrated how tools can support:

- the developer, in the correct implementation of security requirements during the entire life-cycle
- the developer, during the entire secure process/life-cycle (e.g. requirement definition, configuration management, etc.)
- the evaluator, to assure the correct implementation of security requirements and the management of secure process/life-cycle.

It is important to highlight that, as described above, the Security Problem defined and solved by the Protection Profile is focused to specific aspects of cyber-security (and safety).

It should be noted that considering criteria established with respect to evaluability and data to be exchanged between CACC members (speed, direction and positioning), the ethical, legal and social aspects (ELSA) of CACC platooning[3], have not been analysed, but rather outlined as possibilities for further considerations, which would be of utmost importance during the implementation phase of CACC platooning.

3.2 Validation elements and tools for Vertical 1

The following Table 1 lists the set of tools/activities involved in the Vertical 1 use case.

The table shows also the mapping with the Process Element defined in SPARTA D11.2 “Cybersecurity compliant development processes” [5]. The process elements consist of a set of activities related to the security-by-design process and their definitions aim to define a catalogue of elements that should become the founding stone of a process-centric security certification scheme.

- PE1 Organizational Security Framework
- PE2 Product/Service Risk Assessment
- PE3 SDLC Instance Definition
- PE4 Security Planning
- PE5 Software Architecture
- PE6 Threat Modelling
- PE7 Security Functional Requirements Definition
- PE8 Secure Programming Guidelines
- PE9 Code-level Security Analysis
- PE10 Security Testing
- PE11 Security Assessment of 3rd Party / Open Source Software
- PE12 Assessment of the Operational Environment
- PE13 Development Environment
- PE14 Vulnerability Analysis
- PE15 Continuous Vulnerability Checks
- PE16 Patch / Update Processes
- PE17 Secure Configuration by Default
- PE18 Secure Deployment
- PE19 Formal Modelling and Analysis
- PE20 Tools and Automation
- PE21 User Guidance.

Tool / Activity	Partner	Description	Process Element (PE) mapping
TEC demonstrator	TEC	TEC Demonstrator has been used for performing ATE test. The tool has verified a subset of PP Security Functional Requirements. The report (see Appendix A) also shows the coverage matrix for demonstrating the completeness of test execution.	PE10 PE20
Sabotage (SB)	TEC	Sabotage has been used for performing ATE test against a minimal subset of PP Security Functional Requirements (Sensor based	PE10 PE20

[3] Such as processing of personal data as well as liability for damage (due to malfunction, failure to recognise object etc.)

Tool / Activity	Partner	Description	Process Element (PE) mapping
		Plausibility check security mechanism). The report (see Appendix C) also shows the coverage matrix for demonstrating the completeness of test execution.	
Verification Tooling (including SysML usage)	EUT/IMT	EUT performs a Vulnerability Assessment activity (see Appendix D) by assessing the TECNALIA rovers. Vulnerability analysis (including penetration testing activities) have been performed on both OSS and Operational Environment.	PE11 PE12 PE14 PE20
VaCSInE (VCS)	CETIC	VaCSInE orchestrates the security response by deploying and updating a firewall on the rovers to protect them. This is based on OpenSCAP vulnerability scans, Ansible automation and GitLab-CI. The Impact Analysis Report (see Appendix G) describes the security remediation.	PE15 PE20
AutoFocus3 (AF3)	FTS	AutoFOCUS3 supports both the design of the logical architecture, including the software behaviour of logical components and design exploration of safety pattern, task allocation, etc. This is done by using formal methods and solvers (e.g., DLV and Z3). AutoFOCUS3 supports safety argumentation based on formal models (for details see D5.3 [4]). Moreover, in task 5.4 AutoFOCUS3 has been used for performing ATE test. The tool has verified a subset of PP Security Functional Requirements. The report (see Appendix B) shows also the coverage matrix for demonstrating the completeness of test execution.	PE5 PE10 PE19 PE20
Frama-C (FC)	CEA	Frama-C has been used to verify the absence of certain runtime errors in the source code that could lead to security vulnerabilities: every part of the code that is *not* indicated with an alarm is free of such vulnerabilities. Frama-C also allowed identifying some possible weaknesses in the source code and suggesting measures to mitigate them (see Appendix E).	PE9 PE19 PE20

Tool / Activity	Partner	Description	Process Element (PE) mapping
OpenCert (OC)	TEC	OpenCert has been used to manage the safety and security assessment. It has been used to store the evidences of the evaluation process, such as the ATE and AVA documents, and to generate the necessary argumentations to justify the assessment by using the previous stored evidences and explanations (see Appendix H).	PE1 PE20

Table 1: Elements and tool involved in Vertical 1

The elements and tools involved in the Vertical 1 produce a set of evidences that have been used for completing the “evaluability” task. These evidences have been organized as following:

- The OpenCert tool (Scenario 4) allows to manage the full life-cycle of evidences (part of the ALC[4] evidence)
- ASE Requirements are constituted by the Protection Profile developed in D5.2 Appendix [3] and Protection Profile developed for the Assurance Continuity purposes (Appendix F[5]).
- ADV evidences are the design of Vertical 1 as described in D5.2 [2], D5.3 [4] and evidences provided by Frama-C tool (Appendix E)
- ATE evidences have been obtained by the usage of the TEC Demonstrator, AutoFOCUS3 (both for Scenario 1) and Sabotage tool (Scenario 5)
- AVA evidences have been obtained by the activities performed with Verification Tooling, including the SysML tool (Scenario 3)
- Assurance Continuity evidences have been obtained by using Vacsine tool (Scenario 2).

[4] As described in section 3.3.5 Frama-C supports also the ALC
[5] This Protection Profile has been developed in strict relationship to the Assurance Continuity activities. Security Requirements stated in the document have been considered only insed the Impact Analysis Report.

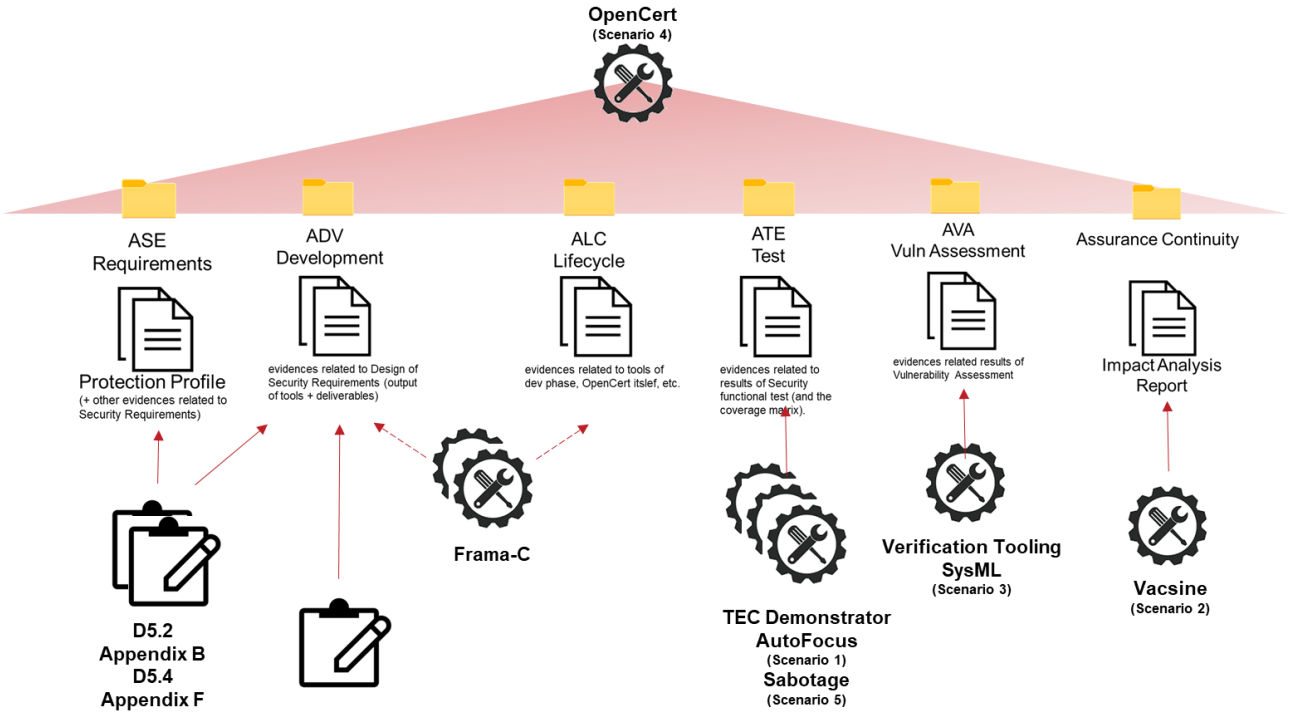

Figure 6: Evaluability evidences collection for Vertical 1

3.3 Evaluability results for Vertical 1

The following table lists the evidences produced by the tools involved in the context of Vertical 1.

Evidences	Tool / Activity	Results/Considerations paraghraph reference
ASE – Security Requirement Definition	D5.2 Appendix B [3] D.5.4 (this document) Appendix F[5]	3.4
ADV – Development	D5.2 [2], D5.3 [4] Frama-C	3.4 3.3.5
Evidences Management (ALC)	OpenCert	3.3.1 and 3.4
ATE - Test Report	TEC Demonstrator Sabotage Autofocus3	3.3.2 3.3.3 3.3.4 and 3.4
AVA – Vulnerability Assessment Report	Verification Tooling (including SysML usage)	3.3.6 and 3.4
Impact Analysis Report[6]	VaCSInE (including OpenSCAP)	3.3.7 and 3.4

Table 2: Evidences for the Vertical 1

[6] In order to prepare an Impact Analysis Report, a new Protection Profile has been developed during the T5.4

The following sections provide a brief analysis of the report generated by the use of the tools.

3.3.1 OpenCert (OC)

OpenCert is a product and process assurance/certification management tool to support the compliance assessment and certification of Cyber- Physical Systems (CPS). The tool has been used in the T5.4, inside the context of Vertical 1, to manage the life-cycle of the evaluability evidences and evidence chain.

As described in Appendix H, the tool allows the evaluator to explore the Security Evidences artefacts, moreover the evaluator can go through a set of minor goals that allow to achieve the high-level goal. In other terms the use of the OpenCert tool allows to track the requirements stated in the Protection Profile up to the high-level goal of the certification.

3.3.2 TEC Demonstrator

TEC demonstrator has allowed to check the implementation status of a subset of Security Functional Requirements (SFRs) defined in the Protection profile [3].

As described in Appendix A, tests have been described with sufficient level of detail allowing them to be repeatable. Moreover, they have been fully traced providing a clear view of the requirements/test coverage.

The information gathered from the evaluability activity is very important for the developer to understand the gap that should be filled in order to perform a future security evaluation.

The activities, as per report in Appendix A, produced 17 tests against 16 SFRs. Results show that all the tests can be considered as passed even if some of them have been passed with some deviation. This deviation can be analysed for taking a decision in view of a future security certification.

The test cases have been performed against the Security Functional Requirements stated in the Protection Profile. The PP has been produced in order to be applicable to a generic CACC. The tailoring phase during an evaluation can be performed developing the Security Target applying the Common Criteria operators (refinement, selection, assignment, iteration) to the SFRs.

In the specific case of the tested rovers, for example, there is no GPS on-board providing Geo-Position even if it is an attribute listed in the requirements, and this implies a deviation from the original requirement.

The evaluability activities therefore produced important information for the developer to prepare a future certification by knowing the current limits of the tested product and having the possibility to find multiple solutions in order to successfully achieve future certification.

3.3.3 AutoFocus3

As introduced in Table 1, the AutoFOCUS3 tool supports the design of the logical architecture as described in the D5.4 [4] providing a support for the ADV (Development, including design).

Moreover, in the context of the evaluability activity the tool has been successfully used for performing test activities related to the ATE class.

As described in Appendix B, AutoFOCUS3 has allowed to check the implementation status of a subset of Security Functional Requirements (SFRs) defined in the protection profile.

Tests have been described with sufficient level of detail allowing them to be repeatable. They have also been fully traced providing a clear coverage of the requirements/test coverage.

The activities, as per report in Appendix B, produced 18 tests against 16 SFRs. Results show that all the tests can be considered as passed even if some of them have been passed with some deviation.

In this specific case, the tests have been performed in a simulated scenario where some information was not available during the test. This has led to have some deviation on the information available as output of the tests.

Also in this case, the evaluability activities therefore produced important information for the developer to prepare a future certification by knowing the current limits of the tested product and having the possibility to find multiple solutions in order to successfully achieve future certification.

3.3.4 Sabotage

The Sabotage tool has been used for testing a specific requirement stated in the Protection Profile regarding the sensor plausibility check algorithm that aims to detect if the information received from the preceding vehicle is reliable or, on the contrary, the vehicle may be under attack.

As described in Appendix C, the SFR related to the sensor plausibility check has been tested through 3 different test cases. The tests are cleary stated and described in a reproducible way.

The report shows that all the 3 test cases have been passed demonstrating the correct implementation of the SFR.

In this case the evaluability activity allows to obtain an assurance that the abovementioned SFR has been fully correctly implemented. A wider consideration can be made on the usage of specific tools such as Sabotage, in the context of a certification. The tools, in fact, could be become the reference tool for testing that SFR in the context of a certification against the CACC Protection Profile [3]. The tool could be considered as supporting tool for the developer in order to verify the correct implementation of the SFR and as supporting tool for the evaluator for verifying the implementation of that requirement.

3.3.5 Frama-C

The Frama-C tool, as per Appendix E, provides a report concerning the development phase (ADV).

The analysis performed by the tool is able to exhaustively prove the absence of runtime errors. This analysis, however, may produce some false alarms that need to be inspected by the user. While the tool tries to provide as much automation as possible, two important tasks still require human intervention: finding the actual root cause of an alarm and providing a patch for it.

Some alarms, even if they do not arise in a specific test case, are possible under different circumstances, such as malicious or accidental changes. Adding extraneous checks, e.g. “is this number always positive”, or “is this pointer definitely not null”, reinforces defense in layers, providing a more convincing argument (both to evaluators and to analysis tools) about the lack of vulnerabilities in the code.

In the CACC code generated by Fortiss, Frama-C reports 15 alarms for over 11k statements, which is a very low number.

The report shows also the approach that allows easily removing 2 of them, and many of the remaining ones are low-priority (e.g. integer overflows related to counters).

From an evaluability point of view this allows to understand the gap that needs to be filled in order to improve the overall code.

Further development of the analysis, including better automation, guidance for root causes of alarms, and tool integration (e.g. support for LSP protocol to obtain better information about variables inside an IDE such as VS Code) will further help to automate test evaluation by supporting better the developer in having more clear evidence for evaluability purpose.

In addition, the tool, as depicted in Figure 6, supports the ALC phase because it is a supporting tool for developing, analysing and implementing the TOE (see also ALC_TAT described in [8]).

3.3.6 Verification Tooling (including SysML usage)

Vulnerability Assessment activities carried out in task 5.4 in Vertical 1 have been performed on Tecnalia's rovers by performing different type of attack (WiFi/Communication Channel, Ultrasound, etc.). No hardware attacks have been considered.

The activity found 7 vulnerabilities: 1 critical, 5 high and 1 medium and propose possible "lite" remediations for 3 of them.

The others could require a change in the architecture (HW/SW).

Even if during a formal Common Criteria evaluation, the AVA_VAN activies are carried out only by the evaluator, it is useful, in the context of "evaluability", for the developer to perform them in order to diminish the risk of unsuccessful certification by addressing the discovered vulnerability in time.

The AVA_VAN activity allows to focus not only on the implementation of the Target Of Evaluation but also to evaluate and check the operational environment around it.

This activity allows for a good understanding of the security problem and its resolution, and to manage better the operational environment, e.g. by avoiding misconfiguration or reducing the attack surface for a potential attacker.

3.3.7 VaCSInE

Assurance Continuity activities have been performed by means of the VaCSInE tool and other supporting tools.

The scenario extends the Target Of Evaluation (TOE) described in Protection Profile for CACC [3] by inserting a firewall as new element in the TOE. For this purpose a new Protection Profile for this extended TOE has been produced (see Appendix F).

The activity assumes that the product under assessment has been already certified and some changes occur (due to updates) to the firewall part of TOE.

This triggers the necessity to perform assurance continuity activities and generating an Impact Analysis Report (see Appendix G) with the objective to analyse and explain to the evaluator the changes occurred and how they have been categorized in major or minor impact. This is important to understand if a maintenance of the certification is easy or if there is the need to execute a subset of certification activities, or in the worst case, execute the entire certification from scratch.

By means of VaCSiNE and the DevSecOPs pipeline the developer can know more quickly the changes occurring to the TOE in terms of the part of code that has changed and the related security functional requirements impacted by these updates, and thus prepare the updates for the impacted evidences.

Moreover by using threat modelling and risk analysis tools the developer is able to understand if any implementation of the security functional requirements has been impacted or not, providing also a justification, and establish if the impact can be categorised as minor or major.

The activites have produced a set of updated evidences that can be submitted to the evaluator in order to ask the maintenance of the certificate.

THIS PAGE IS LEFT INTENTIONALLY BLANK

3.3.8 Vertical 1 Traceability Matrix

The following Table 3 provides a traceability between the SFR stated in the Protection Profile [3] and the tools involved in the life-cycle phases (in this case, the development phase of the requirement and test).

In particular, the matrix shows part of the evaluability chain: starting from the definition of a clear requirement (SFR), the tools that have supported their design and implementation until the verification.

In particular, the ADV phase has been supported also by Frama-C tool. In fact, this tool is fundamental for performing a secure development phase, but it is not focused only on the implementation of the SFRs. Frama-C, in fact, provides a check of the whole source code even if it is not part of the TOE. For this reason, no specific requirement has been addresed through this tool.

The same reasoning can be made for the OpenCert tool that support the ALC phase and the life-cycle of the “evaluabilty” evidences itself. Also, in this case the tool does not map any specific Security Functional Requirement (SFR) because it is mainly in charge to ensure the correct management of the evidences (Security Assurance Requirement).

Regarding Vulnerability Assessment, as described before, the AVA activity does not focus on specific SFR and it is mainly related to evaluate the security posture of the operational environment of the TOE and its supporting component.

Finally, the Assurance Continuity has not been tracked in this matrix. The activity has been focused on a specific Security Functional Requirement stated in the Protection Profile in Appendix F in order to evaluate the possibility to generate in an automatic way evidences regarding the impact on the certified product/system.

SFR	Description	ADV[7]	ATE
PMM_IF.1	Maintain heart-beat data (vehicle identifier, speed, direction, geo-position, timestamp) to VCS	AutoFOCUS3 / TEC DemonstratorDemonstrator	AutoFOCUS3 /TEC Demonstrator
PMM_IF.2	Maintain heart-beat data from VCS	AutoFOCUS3 / TEC Demonstrator	AutoFOCUS3 /TEC Demonstrator
PMM_IF.3	Maintain incoming emergency brake	AutoFOCUS3 / TEC Demonstrator	AutoFOCUS3 /TEC Demonstrator

[7] Evidence of design phase has been taken from the D5.3 document [4].

SFR	Description	ADV[7]	ATE
PMM_IF.4	Maintain outgoing emergency brake	AutoFOCUS3 / TEC Demonstrator	AutoFOCUS3 /TEC Demonstrator
PMM_IF.5	Maintain data from VCM	AutoFOCUS3 / TEC Demonstrator	AutoFOCUS3 /TEC Demonstrator
PMM_IF.6	Maintain data to VCM	AutoFOCUS3 / TEC Demonstrator	AutoFOCUS3 /TEC Demonstrator
PMM_PC.1	Data passes all VCS plausibility checks	AutoFOCUS3 / TEC Demonstrator	AutoFOCUS3 /TEC Demonstrator
PMM_PC.2	Data passes all VCM plausilibity checks	AutoFOCUS3	AutoFOCUS3
PMM_PC.3	Inform on Failed Plausibility Checks	TEC Demonstrator	TEC Demonstrator
PMM_VCS-HPC.1	Maintain heart-beat data history	AutoFOCUS3 (src D5.3)	AutoFOCUS3 (src D5.3)
PMM_VCS-HPC.2	Heart-beat message consistent to the history	AutoFOCUS3 (src D5.3) / TEC Demonstrator	AutoFOCUS3 (src D5.3) / TEC Demonstrator
PMM_VCS-HPC.3	Emergency brake consistent to the history	AutoFOCUS3	AutoFOCUS3
PMM_VCS-SPC.1	Maintain distances history	AutoFOCUS3	AutoFOCUS3
PMM_VCS-SPC.2	VCS message consistent to distances history	AutoFOCUS3	AutoFOCUS3
PMM_VCS-SPC.3	Emergency brake consistent to distances history	AutoFOCUS3	AutoFOCUS3
PMM_VCS-TPC.1	Consult the TOE vehicle internal clock	--	--
PMM_VCS-TPC.2	Message freshness	--	--
PMM_VCM-HPC.1	Maintain sensor data history	AutoFOCUS3	AutoFOCUS3
PMM_VCM-HPC.2	Sensor message consistent to the history	AutoFOCUS3	AutoFOCUS3 / Sabotage

SFR	Description	ADV[7]	ATE
PMM_VCM-TPC.1	Consult the TOE vehicle internal clock	--	--
PMM_VCM-TPC.2	Message freshness	--	--
FPT_ITA.1	Inter-TSF availability within a defined availability metric	--	--
FPT_ITC.1	Inter-TSF confidentiality during transmission	--	--
FPT_ITI.1.1	Inter-TSF detection of modification	--	--
FPT_ITI.1.2	Inter-TSF verify integrity	--	--
FPT_FLS.1	Failure with preservation of secure state	--	--
FCO_NRO.1	Selective proof of origin	--	--
FIA_UAU.2	User authentication before any action	--	--
FIA_UAU.3	Unforgeable authentication	--	--
FIA_UAU.6	Re-authenticating	--	--
FIA_UID.1	Timing of identification	--	--
FRU_FLT.1	Degraded fault tolerance	--	--
FAU_GEN.1	Audit data generation	--	--
FAU_GEN.2	User identity association	--	--

Table 3: Traceability matrix for Vertical 1 (Scenario 1 and 5)

THIS PAGE IS LEFT BLANK INTENTIONALLY

3.4 Considerations

The activities carried out in Vertical 1 allowed to collect a fair number of evidences which almost completely cover the entire life-cycle process of a product.

In particular, the collected evidences made it possible to conclude the "evaluability" task in a positive way, allowing to verify that the selected requirements (see 3.3.8), as defined within the Protection Profile, have actually been implemented in Vertical 1.

This assures that the life cycle, that goes from the definition of the requirements to their implementation and verification (see the matrix in paragraph 3.3.8), was carried out efficiently using the secure-by-design approach.

The activity carried out in the context of Vertical 1 was therefore able to confirm what it was established in the approach defined in the previous paragraphs.

At the end of the task, the evaluator has a sufficient set of evidences (also in terms of their completeness) which allow him to express regarding the evaluability.

The evaluability has not only the objective of verifying the presence of evidence and its completeness in the perspective of a future certification, but also to identify improvements, indicating to the developer the way to go to fill the gaps in order to carry out with success a possible future certification

For what concerns Vertical 1, the definition of the Security Functional Requirements was clearly carried out within the Proteciton Profile. The design and development phase has been described (although not in a formal way) within the deliverable D5.3 [4], allowing to have sufficient level of functional specification.

The development phase (ADV) was also supported by the use of the Frama-C tool. The alarms raised by the tool allow the developer to have clear indications on how to fill the highlighted gaps, in order to assure an adequate level of security.

The presence of alarms does not compromise the success of an evaluability analysis.

In Common Criteria the life-cycle phase also includes the management of the evidence itself. The OpenCert tool, through the Assurance Project Lifecycle Management and Evidence Management functionality, has allowed the management of the evidence stored in SPARTA SVN.

The use of the AutoFocus3, TEC Demonstrator and Sabotage tools, allowed to generate evidences relating to the Security Testing (ATE) phase by providing a reproducible set of exhaustive tests and a demonstration of the coverage of the security requirements analysed.

It means that the traceability matrices confirm that each requirement has been tested by at least one procedure and each procedure tests at least one requirement and that tests have been considered exhaustive in testing the selected security requirement (e.g. there are a sufficient number of parameters used for testing the requirement).

The evidences generated by these three tools allow in some cases to ensure (assurance) the implementation of the security requirements defined in the Protection Profile, and in other cases to highlight the gaps to be filled.

In fact, the presence of some remarks marked with the "passed with deviation" note, allow, in the perspective of future certification, to be aware of which requirements, even correctly implemented, contain fewer attributes than those required by the Protection Profile.

This knowledge allows the developer to have two choices in case of a future certification:

- Proceed, during the development of the Security Target, with a customization of the Protection Profile requirements (using means provided by the Common Criteria such as Refinement, Selection and Assignment) so as to adapt the product under certification, or
- Fill the gap related to the specific requirement by making necessary architectural / implementation changes.

Moreover, the Sabotage tool has been tested and validated for verifying the implementation of a specific Security Functional Requirement (SFR). In the view of a future certification of a product against the CACC Protection Profile [3], this tool could be used as "standard" tool for supporting the developer in checking the correct implementation of the requirement and to the evaluator to carry out the evaluation activities.

Vulnerability Assessment activities allowed to identify some general security improvements that can be pursued in order to make the product "certifiable".

The report attached in Appendix D details the vulnerabilities founded (explaining the execution of the attack) and it propose remediation actions to be put in place in order to mitigate them.

It is important to note that the vulnerability assessment activities do not provide information directly about the correct implementation of the security functional requirements functions (this verification has been made through the ATE activities) but to check the robustness of the operating environment where the TOE runs (e.g. supporting Operating Systems and other softwares). This allows to improve the security of the component around the Target Of Evaluation in order to perform its functionalities (Operational Environment).

A further general consideration can be made on the AVA activities. As anticipated in section 3.3.6, during a formal Common Criteria evaluation the developer is not required to provide formal evidences on AVA activities because they are fully carried out by the evaluator. However, by using a shift-left approach in order to anticipate possible problems the developer could perform these activities and, by using a reproducible report, perform periodically tests also for checking no-regression in security functionality.

Furthermore, for the maintenance of security the activity has explored a flexible approach that in the context of a DevSecOPs pipeline could be considered as "incremental" in order to integrate a DevSecOPs pipeline with the Assurance Continuity activity. This integration makes it possible to automate the collection of the evidences necessary to update the certification, and to quickly assess the impact on the certified product.

Finally, as introduced in section 3.1, there were some considerations related to CACC platoon operation that go beyond SPARTA's T5.4 task, but are directly related with its ethical, legal and social aspects (ELSA). The Security Problem defined for the CACC platooning model has a limited range on variety of data input, which was employed in model of T5.4 (speed, direction and positioning), did not provide a fertile ground for in-depth study on ethical, legal and social aspects of CACC platooning. Nevertheless, even the limited range of parameters that were used in CACC platooning allowed to identify the following ELSA aspects, that are important for further considerations:

- Data which is shared/exchanged when there are changes in composition of CACC platoon might have data protection implications under the GDPR[8] and will depend on the following:
    - The 'business model' of CACC platoons – how information is being transferred within CACC platoon and between CACC platoons;
    - How members of platoon identify each other, i.e. what information is processed and stored in each CACC platoon unit?
    - What information is shared/exchanged when platoon leader changes (how to determine credentials/identity of new leader; what information is stored/processed only by the leader;

[8] Art. 4(1) of the Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC prescribe that personal data is any information relating to an identified or identifiable natural person, i.e. the one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person. Moreover, Art. 5 of the Regulation indicate that processing of persona data should satisfy the following requirements (principles): lawfulness, fairness and transparency; purpose limitation; data minimisation; accuracy; storage limitation; integrity and confidentiality.

  - how the transfer/exchange of information between new leader and platoon is conducted (direct vehicle-vehicle, cloud, internet)?
  - How and what information is shared/exchanged when follower joins or leaves?
  - With whom such information is exchanged/shared;
  - What is the business model for management (collection, processing, storage) of such information?
- Information gathered during actual operation of CACC platoon related to determination of safe speed might have either liability or data protection considerations, in particular depending on the following:
  - What and from what direction do sensors of CACC platoon pick information – only information that is ahead of them or also to the sides of the road?
  - How do sensors identify the object that appears between CACC platoon and make judgement on further actions (i.e. when decision is not made by the manually operated leader)? In particular how does the car identify and differentiate leaf, bag or other “safe” object from fatal one (e.g. pedestrian, stone etc.)) that might appear between CACC platoon especially in urban environments with low speed?
  - If such information is processed internally (within vehicle), who is liable for incorrect assessment and accompanying damages under the applicable ‘business model’: manufacturer, software developer?
- If information is processed in the cloud, what information is shared, processed, stored and otherwise managed, who is liable for incorrect assessment and accompanying damages.

Chapter 4 Vertical 2: Demonstration of a Complex System Assessment Including Large Software and Open Source Environments, Targeting e-Government Services

4.1 Description

The e-Government services vertical (Vertical 2), fully described in D5.2 [1], has the goal to improve the cyber-security of the innovative authentication solutions based on the usage of the Italian national electronic identity card (CIE). This vertical leverages the collaboration between Fondazione Bruno Kessler (FBK, one of the institutions part of the SPARTA partner CINI) and Instituto Poligrafico e Zecca dello Stato (IPZS), the Italian National Mint and Printing House, which handles the production of the identity cards in Italy.

We identified two main scenarios: the CIE ID mobile app (CIE ID APP) and the SAML-based Identity provider server (SAML IdP Server). The SAML IdP Server, based on Shibboleth, is responsible for the authentication of the users willing to access the services offered by the Italian Public Administrations. To authenticate the users, the SAML IdP Server interacts with the CIE ID APP. The CIE ID APP is installed on the smartphone of the users, and leverages the CIE (through the NFC interface) as an authentication tool.

The demonstration scenarios of the vertical 2 involve the development and testing environments managed by FBK, where the preliminary versions of the CIE ID APP and SAML IdP Server were developed and tested, before being migrated on the Italian Ministry of the Interior servers.

In the context of evaluability, Vertical 2 follows a different approach with regards to Vertical 1.

As mentioned above, during the Task 5.2 a Protection Profile has been defined in the context of vertical 1; in Vertical 2 the activities of Task 5.4 start from the identification of a set of security requirements. These requirements have been selected starting from international standards such as OWASP Mobile Application Security [14], OWASP Top Ten [15] and MITRE ATT&CK tactics and techniques [16].

Starting from these standards a set of Security Requirements has been tailored (the list of requirements has been listed in section 4.3.4) in order to solve the Security Problem of Vertical 2

Also in this case, evaluability aims to test the security requirements in order to provide a set of evidences useful for understanding the status of the product under assessment in the prespective of a future certification.

4.2 Validation elements and tools for Vertical 2

The following Table 4 lists the set of tools/activities involved in the Vertical 2 use case.

The table shows also the mapping with the Process Element defined in SPARTA D11.2 “Cybersecurity compliant development processes” [5]. The process elements consist of a set of activities related to the security-by-design process and their definitions aims to define a catalogue of elements that should become the founding stone of a process-centric security certification scheme.

- PE1 Organizational Security Framework
- PE2 Product/Service Risk Assessment
- PE3 SDLC Instance Definition
- PE4 Security Planning
- PE5 Software Architecture
- PE6 Threat Modelling
- PE7 Security Functional Requirements Definition
- PE8 Secure Programming Guidelines
- PE9 Code-level Security Analysis
- PE10 Security Testing

- PE11 Security Assessment of 3rd Party / Open Source Software
- PE12 Assessment of the Operational Environment
- PE13 Development Environment
- PE14 Vulnerability Analysis
- PE15 Continuous Vulnerability Checks
- PE16 Patch / Update Processes
- PE17 Secure Configuration by Default
- PE18 Secure Deployment
- PE19 Formal Modelling and Analysis
- PE20 Tools and Automation
- PE21 User Guidance.

Tool / Activity	Partner	Description	Process Element (PE) mapping
Approver (RAA)	CINI	Approver supports the automatic evaluation of security vulnerabilities in Android apps by exploiting static and dynamic analysis techniques. The tool has been used for performing ATE tests. The tool has verified a set of potential Security Functional Requirements. The report (see Appendix I) also shows the coverage matrix for demonstrating the completeness of test execution.	PE9 PE10 PE20
Logic Bomb Detection (TSOpen)	UniLu	TSOpen is a static analysis tool that is able to detect logic bombs in Android applications. Logic bombs are mechanisms used by malicious apps to evade detection techniques. Typically, an attacker uses a 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 tool can be run by security analysts or directly on the marketplace side. It can be run over existing applications that could be removed from the marketplace if a logic bomb happens to be found. In the context of Vertical 2, the tool has been used for performing ATE tests (see Appendix I). It demonstrated its ability to detect a time-related logic bomb found in the app (when the logic bomb has been artificially added).	PE9 PE10 PE20
SafeCommit	UniLu	SafeCommit aims at automatically detecting commits that introduce vulnerabilities in Continuous Integration Ecosystem. SafeCommit is built on top of AI techniques relying on innovative features and advanced patch representation learning.	PE9 PE10 PE15 PE20

Tool / Activity	Partner	Description	Process Element (PE) mapping
		SafeCommit has been integrated into the repository of vertical 2 for verifying new commits of source code for the SAML IdP Server scenario (see Appendix K).	
ProjectKB (KB)	SAP	Supports the creation, management and aggregation of a distributed, collaborative knowledge base of vulnerabilities that affect open-source sofòtware. This repository contains both a tool and vulnerability data. Moreover, in the Vertical 2 evaluability context, KB has been used for verifying a subset of Security Requirements and producing the test report ATE document ([reference to ATE document]) also showing the coverage matrix for demonstrating the completeness of test execution	PE10 PE11 PE15 PE20
Steady (VA)	SAP	Analyses Java and Python applications for open-source dependencies with known vulnerabilities, using both static analysis and testing to determine code context and usage for greater accuracy. Moreover in the Vertical 2 evaluability context, VA has been used for verifying a subset of Security Requirements and producing the test report ATE document ([reference to ATE document]) also showing the coverage matrix for demonstrating the completeness of test execution	PE10 PE11 PE15 PE20
Visual investigation of security information (VI)	UKON	The Visual Investigation (VI) enables to audit entire software development organizations. The VulnEx (Vulnerability Explorer) demonstrator allows investigating exposure to open-source software vulnerabilities on an organization-wide level. The demonstrator allows examining problematic projects and applications (repositories), third-party libraries, and vulnerabilities across a software organization. Moreover in the Vertical 2 evaluability context, VI has been used for verifying a subset of Security Requirements and producing the test report ATE document ([reference to ATE document]) also showing the coverage matrix for demonstrating the completeness of test execution	PE10 PE11 PE14 PE20

Table 4: Elements and tool involved in Vertical 2

Tools involved in Vertical 2 produced a set of evidences used for performing the “evaluability” task for the Vertical 2.

In this second use case the evidences are the following:

- ASE requirements are the output of the first task performed during the T5.4. Starting from standards (such as MITRE and OWASP) a set of security requirements has been tailored for the vertical 2 (see section 4.3.4)
- ADV evidences have been provided by SafeCommit tool; this tool supports also the ALC phase (in particular as supporting tool for developing, analysing and implementing the product under evaluability task. See also Common Criteria ALC_TAT in [8]).
- ATE evidences have been obtained by running the following tools on the two scenarios of Vertical 2: Approver, TSOpen, ProjectKB, Steady and VI.

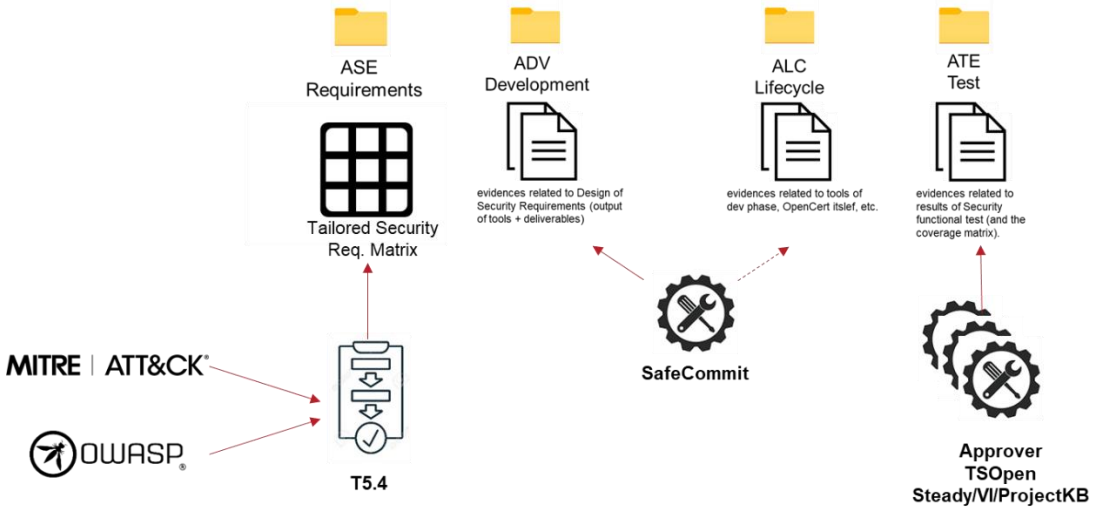

Figure 7: Evaluability evidences collection for Vertical 2

The evaluability task for Vertical 2 has been focused more in detail to secure development and test phase.

4.3 Evaluability results for Vertical 2

The following Table 5 lists the evidences produced by tools involved in the context of Vertical 2.

Evidences	Tool	Results / Consideration
ATE - Test	Approver TSOpen ProjectKB Steady VI	4.3.1 and 4.4 4.3.2 and 4.4
ADV – Development	SafeCommit	4.3.3 and 4.4

Table 5: Evidences for the Vertical 2

In the following sections is presented a brief analysis of the report generated by using that tools.

4.3.1 Approver and TSOpen

Approver and TSOpen have allowed to check the implementation status of a set of Security Functional tailored starting from OWASP standard (see details in section 4.3.1).

Tests have been described with a sufficient level of detail in order to be repeatable.

Test summary coverage provides a demonstration that each test cover at least one requirement and that each requirement has been verified at least by one test. The condition of “Necessary and Sufficient” has been proved through the test coverage matrix that allow to demonstrate the completeness of the test phase.

The activity, as per report in Appendix I, produced 37 tests in order to cover 37 Security Requirements.

In this case some test results have been FAILED, however the developer provides a note with mitigation instructions in order to make changes at the product under test and allowing the product to pass the tests in a second iteration.

The report shows a “first” iteration of “evaluability”; the process should be repeated until the security posture of the product under assessment reach an acceptable level.

The output of these tests should be presented to the developer in order to implement the corrective countermeasures and the test should be repeated.

In particular in this case the target of vertical 2 is a SW product (for test scope) developed by an “external” entity not part of the SPARTA program (Istituto Poligrafico Zecca dello Stato) and for this reason only a single iteration has been performed.

This result, from an “evaluability” point of view, is very important because allows the developer to know the improvement areas and the remediation to be implemented for addressing the failed tests.

Moreover the structure of the report allows the test to be repeatable and the developer can re-asses the test in order to check if the suggested implementation meet the requirement.

This activity, in an iterative way, allow to meet well defined security requirements and ensure an adequate security posture of the product under test.

4.3.2 Project KB / Steady / VI

This set of tools allowed to check the implementation status of Security Requirements tailored starting from the MITRE ATT&CK (see 4.3.4).

Also in this case test coverage proofs the condition of “Necessary and Sufficient” for confirming that all tests verify the implementation of at least one requirement and each requirement has been checked by at least one test.

The activity, as per report Appendix J, produced 3 tests against 3 Security Requirements.

Also in this case some test results have been FAILED and a remediation has been suggested to the developer in order to proceed to the mitigation phase for updating the product under test.

Also in this case the “evaluability” allows to anticipate possible non-compliance to the future cybersecurity certification allowing the developer to address timely the changes before the end of the project or the end of the certification.

Even if we discuss later this aspect, it is interesting to highlight that the selected requirements for this phase are related to the vulnerability assessment phase.

This set of tool, in fact, provide results part of the ATE family but on the other hand starts to assess the security posture of the product under testing from a vulnerability point of view.

These requirements could be translated, in case of future certification, in Security Functional Requirement[9] in the Security Target document by using the Extended Component SFR as per Common Criteria part 1 [7] and 3 [9]. On the other hand they could be translated in Security Assurance Requirement (SAR)[10] [9] and define specific activities related to the cybersecurity evaluation of the product (or this family of product).

4.3.3 SafeCommit

The SafeCommit tool has the task of detecting if new commits introduce vulnerabilities

This tool has also been integrated into the DevSecOPs pipeline for Vertical 2 (see Appendix K).

The tool has not found vulnerable commit performed in the repository, and for this reason the appendix shows how the tool could detect them during further development of the product by committing a safe code and a potential vulnerable code.

SafeCommit product shows its potential in supporting the secure development phase of the Vertical 2 product. SafeCommit concurs to support the developer to understand timely potential vulnerability introduced in the development phase and during the maintenance phase (in the context of Vertical 2 this phase has not been analysed but for further works could be an interesting area of analysis and improvement).

4.3.4 Vertical 2 Traceability Matrix

The following Table 6 summarizes the set of security requirements taken in consideration for performing the activities of Task 5.4 for the vertical 2 (Mobile Scenario).

The table has been organized as following:

- ID column: contains the unique ID for the requirement
- Detailed Verification Requirement: contains the descripton of requirement
- ATE column: provide a mapping between the requirement and the tool that has been used for providing evidences about its implementation

ID	Detailed Verification Requirement	ATE
MSTG-STORAGE-2	No sensitive data should be stored outside of the CieID-App container or system credential storage facilities.	Approver
MSTG-STORAGE-3	No sensitive data is written to CieID-App logs.	Approver
MSTG-STORAGE-4	No sensitive data is shared with third parties unless it is a necessary part of the architecture.	Approver
MSTG-STORAGE-8	No sensitive data is included in backups generated by the mobile operating system.	Approver
MSTG-STORAGE-9	The CieID-App removes sensitive data from views when moved to the background.	Approver
MSTG-CRYPTO-1	The CieID-App does not rely on symmetric cryptography with hardcoded keys as a sole method of encryption.	Approver
MSTG-CRYPTO-2	The CieID-App uses proven implementations of cryptographic primitives.	Approver

[9] Security Functional Requirements (SFR) are a translation of security requirements in the standardised (Common Criteria) language (see CC part 2 [8])
[10] Security Assurance Requirements describe how the Target Of Evaluation is to be evaluated. In oderr words they describe "*of how assurance is to be gained that the TOE meets the SFRs*" [7].

ID	Detailed Verification Requirement	ATE
MSTG-CRYPTO-3	The CieID-App uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.	Approver
MSTG-CRYPTO-4	Taking into account the constraints posed by the cryptographic algorithms supported by the card (CIE), the CieID-App does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes.	Approver
MSTG-CRYPTO-6	All random values used in the CieID-App are generated using a sufficiently secure random number generator.	Approver
MSTG-NETWORK-1	Data is encrypted on the network using TLS. The secure channel is used consistently throughout the CieID-App.	Approver
MSTG-NETWORK-2	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.	Approver
MSTG-NETWORK-3	The CieID-App verifies the X.509 certificate of the CieID-Server when the secure channel is established. Only certificates signed by a trusted CA are accepted.	Approver
MSTG-NETWORK-4	The CieID-App pins the CieID-Server 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.	Approver
MSTG-NETWORK-6	The CieID-App only depends on up-to-date connectivity and security libraries.	Approver
MSTG-PLATFORM-1	The CieID-App only requests the minimum set of permissions necessary.	Approver
MSTG-PLATFORM-4	The CieID-App does not export sensitive functionality through IPC facilities, unless these mechanisms are properly protected.	Approver
MSTG-PLATFORM-5	JavaScript is disabled in WebViews unless explicitly required.	Approver
MSTG-PLATFORM-6	WebViews are configured to allow only the minimum set of protocol handlers required (ideally, only https is supported). Potentially dangerous handlers, such as file, tel and app-id, are disabled.	Approver
MSTG-PLATFORM-10	A WebView's cache, storage, and loaded resources (JavaScript, etc.) should be cleared before the WebView is destroyed.	Approver
MSTG-CODE-1	The CieID-App is signed and provisioned with a valid certificate, of which the private key is properly protected.	Approver
MSTG-CODE-2	The CieID-App has been built in release mode, with settings appropriate for a release build (e.g. non-debuggable).	Approver
MSTG-CODE-3	Debugging symbols have been removed from native binaries.	Approver
MSTG-CODE-4	Debugging code and developer assistance code (e.g. test code, backdoors, hidden settings) have been removed. The CieID-App does not log verbose errors or debugging messages.	Approver

ID	Detailed Verification Requirement	ATE
MSTG-RESILIENCE-1	The CieID-App detects, and responds to, the presence of a rooted or jailbroken device either by alerting the user or terminating the app.	Approver
MSTG-RESILIENCE-9	Obfuscation is applied to programmatic defenses, which in turn impede de-obfuscation via dynamic analysis.	Approver
TSOpen-1	The CieID-App's specific values are symbolically executed	TSOpen
TSOpen-2	The CieID-App's suspicious checks are identified	TSOpen
TSOpen-3	Suspicious checks' behavior is scanned to check for malicious behavior	TSOpen
TSOpen-4	Based on control dependency, logic bombs are identified	TSOpen
TSOpen-5	Apps are checked against existing logic bombs that trigger malicious code under specific circumstances, bypassing detection techniques.	TSOpen

Table 6: Traceability matrix for Vertical 2 Mobile Scenario

The following Table 7 summarizes the set of security requirements taken in consideration for performing the activities of Task 5.4 for the vertical 2 (Mobile Scenario).

The table has been organized as following:

- ID column: contains the unique ID for the requirement
- Detailed Verification Requirement: contains the descripton of requirement
- ATE column: provide a mapping between the requirement and the tool that has been used for providing evidences about its implementation

ID	Detailed Verification Requirement	ATE
T1195.001-S1	The SAML IdP Server must not depend on components with known vulnerabilities.	Steady/VI
T1195.001-S2	The SAML IdP Server must not depend on components that are unmaintained or do not produce security patches anymore.	Steady/VI
T1195.001-S3	The SAML IdP Server must not include un-used components.	Steady/VI

Table 7: Traceability matrix for Vertical 2 SAML IdP Scenario

Security Requirments stated in both Table 6 and Table 7 could be “translated” in Security Functional Requirement as per Common Criteria during a future cybersecurity evaluation activity and tested as shown in the reports of Appendix I and Appendix J.

It is important to highlight that the implementation of these requirements could imply also the creation of of development (ADV) evidences, because some of this tools work for verifying source code implementation (in particular for what concerning security requirements stated in Table 6).

Concerning Table 7 these requirements could support the creation of security evidences related to Vulnerability Assessment activities (AVA) by checking vulnerabilities and component obsolescence. In particular this second set of requirement could be translate in Security Assurance Requirement

allowing to perform specific assurance activities during the future evaluation (e.g. about the management of software components obsolescence).

4.4 Considerations

As introduced in D5.3 [4] and mentioned in section 4.1, the approach of CAPE tool testing in Vertical 2 is different with respect to the Vertical 1.

In Vertical 2, in fact, Common Criteria Protection Profile tailored for this use case has not been developed.

However, as introduced before, the Security Problem defined and solved with a Protection Profile (and after with a Security Target for the specific TOE) is a formalization in a standardized language of a security problem.

The security problem, in the first instance, can be developed in “natural language” then translated in Common Criteria “language” starting from the Security Requirements (as per Table 6 and Table 7) and translating them by using the Common Criteria catalogue (CC part 2 [8] and part 3 [9]) or the Extended Component Definition (CC part 1 [7] and part 3 [9]).

Even if the Security Requirements have not been stated with the formalism required by the Common Criteria, the evaluability activities allow to verify the completeness of requirement traceability against their implementation through the testing phase.

In Vertical 2 starting for requirements tailored from standard such as OWASP ([14] and [15]) and MITRE ATT&CK ([16]) it is possible to evaluate the overall security posture of the product under test.

In particular, tools have produced results useful for the “evaluability” of the product under assessment by founding different improvement areas and providing information for filling the gap and improving the robustness of the future TOE.

In particular, the Vertical 2 use case highlights a potential improvement in view of a security certification.

In Vertical 2 the tools have been integrated in a DevSecOPs pipeline. This could allow to stress the concept of security-by-design and the related creation of evidence during the whole life-cycle of the product that should be vertified.

The tools, in fact, provide a set of evidences useful for the evaluability purpose. This allows to understand that possible future works could led to an evolution where the tools in the DevSecOPs pipeline could support, in a more automated way, the creation of formal evidences for an evaluation against a defined certification scheme.

In view of evaluability task, the tools of Vertical 2 have been used focusing to the verification of the implementation of security requirements by providing results in Security Test category (ATE).

However the tools demonstrate to be useful for supporting the production of evidences concercing the development phase (ADV in particular for the development of secure code) and the vulnerability assessment activities.

Chapter 5 Summary and Conclusions

This deliverable collects the output of activities performed during the execution of Task 5.4 by applying the “evaluability” concept on the use cases of Vertical 1 and Vertical 2.

The evaluability process, applied to the use case of the two verticals, allows to test the CAPE tools involved in both Verticals by verifying the evidences produced for a future certification involving the use cases.

In addition, a set of validation tests on the tools outside the scope of the two verticals (standalone tools) has been performed, providing evidences about the results reached by these tools (collected in Appendix L).

In summary, the evaluation activities involved 13 tools that performed 78 security tests in order to check 73 security requirements (other than security test, the tools provide Life-Cycle management support, security development support ADV and Vulnerability Assessment activities support). And additional tests have been performed for 5 tools outside the vertical context.

In both Vertical 1 and Vertical 2, the tools have reached the objectives. As introduced in the previous sections, the evaluability task is not a formal certification of the Verticals and aims to check the completeness and the quality of the security evidences produced, by the CAPE tools, for a future security certification.

Evaluability has also the objective to understand any gap to be filled before starting a formal evaluation. In particular, the tools producing evidences for “evaluability”, since the early stages of the life-cycle allow to have the advantages of a shift-left approach. Any issues related to the future security certification could be addressed timely, instead of discovering it during the certification phase (where typically the product is finished and modifications could have an important impact on costs and time-to-market).

Even if the entire life-cycle has been managed during the execution of the task, the activities focused particularly on the output provided by the CAPE tools for the ATE Phase. This phase is the most critical phase in an evaluation because ensures that the Security Problem[11] countered by the Target Of Evaluation has been correctly solved. In other terms, the testing phase and the demonstration of the coverage of all security requirements allow the evaluator to have a confidence that the requirements have been correctly implemented.

The CAPE tools have provided complete evidences in order to complete the ATE phase of evaluability task of both two verticals.

It is important to highlight that the security requirements of Vertical 1 have been defined by performing a risk analysis, and the validation of them through the test performed by using the CAPE tools is a proof of the correct resolution of the Security Problem (see chapter 3 “Security Problem Definition” in Appendix B of D5.2 [3]). Security Requirements of Vertical 2 have been derived from standard/best-practices for solving the general Security Problem of a mobile app scenario and server scenario.

Also, another analysis has been performed during the task in order to map the CAPE tools involved in the two verticals with the Process Element defined in D11.2 [5]. This mapping allows to perform the following consideration: tools created for specific purpose (such as modelling and design, see e.g. Sabotage, TEC Demonstrator, AutoFOCUS3, etc.) can provide Security Test evidences for a future cybersecurity certification scheme.

PE10 “Security Testing” in particular, is described as following: “*Perform security testing against the product/service. This includes traditional* ***functional testing for the implementations of the security functional requirements*** *as well as the application of dedicated dynamic security analysis*

[11] That is how the threats are countered by the product/system under evaluation

and test methods and tools for the full code base in order to find known vulnerabilities. ***A test coverage analysis*** *should be included in the element.*"

All the tools providing ATE evidences, provide a demonstration of security functional requirements, providing testing and coverage analysis.

An evolution of such tools could allow to make more "automatic" the generation of security test against functional security requirements, allowing the developer to collect certification evidences quickly during the whole life-cycle of the product under certification.

Concerning specific results for Vertical 1, the activities performed in this use case allow to make some considerations about two main themes. The first one is related to Skill and Tools adopted by the Cybersecurity Evaluation Facility during the certification, and the second one is related to the maintenance of security certification.

About the first topic, the following considerations apply: Protection Profiles typically are accompanied by supporting documents that describe the actions to be performed by the evaluator when checking that specific requirements have been met and defining more in detail the security assurance requirement (this is done in addition to what is defined in the Common Criteria Evaluation Methodology [10]). The supporting documents assist the Cybersecurity Evaluation Facilities (CSEFs) in understanding the skills and tools required for the evaluator to perform the specific evaluation. Starting from a consideration stated in D11.3 about the CSEFs "*Skills and tools, moreover, could benefit also by involving in certification process R&D and Research Institutes in a more structured way, in order to take advantage from competences of Subject Matter Experts and by using, for example, complementary software tools for performing security tests, vulnerability assessment and penetration test activities*", the Sabotage tool, in the context of the Vertical 1, is a candidate for performing specific tests against specific security requirements.

In this case a potential supporting document, developed as annex to the CACC Protection Profile [3], could suggest to the evaluator the usage of such tools for verifying the specific requirement.

Sabotage, for example, addresses exactly what is specified in the previous statement. In the same view the OpenCert tool could be used, in general, for supporting both developer and evaluator to maintain a clear view about the traceability between security functional requirements, their implementation and their tests (see Figure 19 Appendix H).

The second main topic is related to the certification maintenance (automation in production of an Impact Analysis Report, in this case produced by the usage of the VaCSIne tool). This activity generates potential added value:

- From a developer's point of view, it allows the developer to make a better trade-off analysis in defining where to make changes by having a DevSecOPs pipeline as support in understanding the changes that occur to the evaluated product. In other words, the developer could be aware if a change that he makes has a major impact on certification and requires to perform a new certification (impact on time and costs, or minor impact that implies a certification maintenance with a subset of activities)
- From an evaluator's point of view, having a standardized impact analysis report defined and implemented by apriori defined DevSecOPs pipeline, allows to perform checks quickly (in particular for minor changes) making faster the process of certificate maintenance.

In a longer term vision, as stated in [22], making faster and cheaper the maintenance of a security certification could lead to benefits in having a growing number of updated certified product improving the global security posture.

Considerations for results coming from Vertical 2 are similar to the points described above. The results obtained in this second Vertical highlight that possible improvement and future evolution of the tools could allow to define and integrate a complete DevSecOPs pipeline that supports the developer in providing cybersecurity certification evidences timely. This could allow the developer to address timely potential no-compliance or to fill the gap to meet specific security requirements. Possible evolution could allow to provide evidences in a more automated way.

Is important to note that from an evaluability point of view it's not necessary to provide evidences in a standardized language, but it is important to ensure, during the whole life-cycle, that the security requirements have been correctly defined, implemented and tested by proving the complete coverage between tests and security requirements. This provides the developer a sufficient level of assurance that the Security Problem has been solved.

Chapter 6 List of Abbreviations

Abbreviation	Translation
AGD	Guidance documents Assurance Class
ADV	Development Assurance Class
AF3	AutoFOCUS3
APP	Application
ASE	Security Target Evaluation Class
ATE	Tests Assurance Class
AVA	Vulnerability Assessment Assurance Class
BSI	Bundesamt für Sicherheit in der Informationstechnik
CACC	Cooperative Adaptive Cruise Control
CAN	Controller Area Network
CAPE	Continuous assessment in polymorphous environments
CC	Common Criteria
CCCC	Connected and Cooperative Car Cybersecurity
CCRA	Common Criteria Recognition Arrangement
CD	Continuous Development
CEM	Common Criteria Evaluation Methodology
CI	Continuous Integration
CIE	Carta d'Identità Elettronica
CPS	Cyber – Physical Systems
CSEF	Cybersecurity Evaluation Facility
CWE	Common Weakness Enumeration
DevSecOps	Development Security Operations
ELSA	Ethical Legal and Social Aspects
EUCC	Common Criteria based European candidate cybersecurity certification scheme

Abbreviation	Translation
Euro NCAP	European New Car Assessment Programme
FBK	Fondazione Bruno Kessler
FC	Frama-C
GDPR	General Data Protection Regulation
GSN	Goal Structuring Notation
GUI	Graphical User Interface
HW/SW	Hardware/Software
IDE	Integrated Development Environment
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
JSON	JavaScript Object Notation
LSP	Language Server Protocol
LSP	Language Server Protocol
MITRE ATT&CK	MITRE Adversarial Tactics, Techniques, and Common Knowledge
MSTG	Mobile Security Testing Guide
OC	OpenCert
OSS	Open Source Software
OWASP	Open Web Application Security Project
PP	Protection Profile
SAE	Society of Automobile Engineers
SafSecPMM	Safety and Security Platooning Management Module
SAML IdP	Security Assertion Markup Language Identity Provider
SAR	Security Assurance Requirement
SARIF	Static Analysis Results Interchange Format
SB	Sabotage
SFR	Security Functional Requirement

Abbreviation	Translation
SOG-IS MRA	Senior Officials Group Information Systems Security Mutual Recognition Agreement
SVN	Subversion
SysML	Systems Modeling Language
TARA	Threat Analysis and Risk
TOE	Target of Evaluation
VA	Vulnerability Assessment
VCS	VaCSInE
VI	Visual Investigation
VS Code	Visual Studio Code

Chapter 7 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 D5.2 “Demonstrators specifications”. January 2021.

[3] SPARTA CAPE D5.2 “Appendix B Protection Profile for a Safety and Security Platooning Management Module”, January 2021

[4] SPARTA CAPE D5.3 “Demonstrator prototypes”. January 2021.

[5] SPARTA D11.2 “Cybersecurity compliant development processes”. 31st July 2020 https://www.sparta.eu/assets/deliverables/SPARTA-D11.2-Cybersecurity-compliant-development-processes-PU-M18.pdf

[6] SPARTA D11.3 “Cybersecurity evaluation facilities”. August 2021.

[7] Common Criteria for Information Technology Security Evaluation, Version 3.1, revision 5, April 2017. Part 1: Introduction and general model. CCMB-2017-04-001

[8] Common Criteria for Information Technology Security Evaluation, Version 3.1, revision 5, April 2017. Part 2: Functional security components. CCMB-2017-04-002

[9] Common Criteria for Information Technology Security Evaluation, Version 3.1, revision 5, April 2017. Part 3: Assurance security components. CCMB-2017-04-003

[10] Common Criteria for Information Technology Security Evaluation, Version 3.1, revision 5, April 2017: Evaluation methodology. CCMB-2017-04-004

[11] Bundesamt für Sicherheit in der Informationstechnik (BSI) Guidelines for Developer Documentation according to Common Criteria Version 3.1 Version 1.0

[12] Common Criteria Assurance Continuity: CCRA Requirements version 2.1 June 2012

[13] EUCC, a candidate cybersecurity certification scheme to serve as a successor to the existing SOG-IS, V1.0 | 01/07/2020

[14] OWASP Application Security Verification Standard https://owasp.org/www-project-application-security-verification-standard/

[15] OWASP Top Ten https://owasp.org/www-project-top-ten/

[16] MITRE ATT&CK tactics and techniques https://attack.mitre.org/versions/v8/

[17] *SARIF Home*: links to specification, tools, libraries and viewers related to the Static Analysis Results Interchange Format. Microsoft. https://sarifweb.azurewebsites.net

[18] SARIF *v2.1.0 specification*, an OASIS standard. https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html

[19] Introducing *JSON*: brief overview of the JSON format, with railroad diagrams. https://www.json.org

[20] *Frama-C* User *Manual*: presents the Frama-C platform, including a chapter on its Graphical User Interface. https://frama-c.com/download/frama-c-user-manual.pdf

[21] *SARIF Viewer*: extension, developed by Microsoft, for Visual Studio Code, to visualize SARIF files. https://marketplace.visualstudio.com/items?itemName=MS-SarifVSCode.sarif-viewer

[22] S. Dupont, G. Ginis, M. Malacario, C. Porretti, N. Maunero, C. Ponsard and P. Massonet "Incremental Common Criteria Certification Processes using DevSecOps Practices," *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2021, pp. 12-23, doi: 10.1109/EuroSPW54576.2021.00009.

[23] ISO26262. Iso 26262-1:2018 - funtional safety road vehicles. URL https://www.iso.org/standard/68383.html

[24] SAEJ3061. Sae j3061 - cybersecurity guidebook for cyber-physical vehicle systems. URL https://www.sae.org/standards/content/j3061_201601/

[25] GSN Community Standard Version 1. 2011. Available at http://www.goalstructuringnotation.info/documents/GSN_Standard.pdf

Appendix A
ATE Tests – Vertical 1 Scenario 1 (TEC Demonstrator)

Tests results obtained by the usage of the TEC Demonstrator tool for verifying the correct implementation of Security Functionalities of Vertical 1 Scenario 1 are described in detail in SPARTA-D5.4-M36_AppendixA.

Appendix B
ATE Tests – Vertical 1 Scenario 1 (AutoFOCUS3)

Tests results obtained by the usage of the AutoFOCUS3 tool for verifying the correct implementation of Security Functionalities of Vertical 1 Scenario 1 are described in detail in SPARTA-D5.4-M36_AppendixB.

Appendix C
ATE Tests – Vertical 1 Scenario 5 (Sabotage tool)

Tests results obtained by the usage of the Sabotage tool for verifying the correct implementation of Security Functionalities of Vertical 1 Scenario 5 are described in detail in SPARTA-D5.4-M36_AppendixC.

Appendix D
AVA Vulnerability Assessment – Vertical 1 Scenario 3 (Verification Tooling)

Vulnerability Assessment results on TECNALIA Rovers (Vertical 1 Scenario 3) are described in detail in SPARTA-D5.4-M36_AppendixD.

Appendix E
ADV Development – Vertical 1 (Frama-C)

Presentation of the repository under test

The code under test comes from Fortiss' Cooperative Adaptive Cruise Control (CACC) code. The C code is generated by AutoFocus from a model, for running on a rover using a Raspberry Pi-based board. The code includes:

- 150 C source files, along with 150 C headers;
- Several sources correspond to hardware devices, such as accelerometers, back lights, and other inputs;
- Other sources correspond to "higher-level" functions related to cruise control, such as "lane keeping", "acceleration/deceleration", "target velocity", etc.
- The code also includes some extra components, but without drivers nor source code, other than function signatures in header files: a camera server and a rumblepad.
- The code contains about 11k statements, with 2k decision points (very few loops; mostly if statements), 1.6k global variables, and 1.1k functions.

Overall, the code structure is relatively simple, with a main function performing general initialization of several components, then a main loop periodically calling a worker function, which realizes the behavior: reading sensors, computing new states of the automaton, and emitting control signals.

The code being generated from a model which incorporates security protections, it is supposed isolated from external tampering and correct by construction, i.e., no undefined behaviors should be produced by the code generator.

For the above reasons, the code does not incorporate redundant defensive programming measures. Still, in order to help static analysis, and to provide an extra layer of defense, such measures can be adopted.
Besides the existing CACC code, the following files were added to the analysis:

- Several Linux headers (for components such as the CAN network layer);
- Dozens of short specifications for function prototypes without code; these are needed by Frama-C when the source is not available, or only available in assembly format.

Running the tool on the selected part of the code

We ran the exhaustive runtime error analysis performed by the Eva plug-in of the Frama-C platform. The source files are parsed, then the program flow is statically evaluated, starting at the main function, and following all possible execution paths. If, at any moment, Frama-C/Eva is not certain that a runtime error is impossible, it emits an alarm, and continues the analysis. Alarms are either true or unknown: true means there is definitely an error in the code (which may or may not trigger an actual runtime exception), while unknown means that there may be an error.

Thus, the absence of alarms guarantees the code is safe, but their presence does not mean it is unsafe.

The analysis has been prepared using Frama-C scripts, which populate a template with some interactive prompts.

They produce a Makefile which contains:

- the list of sources to be parsed;
- a set of default command-line options;
- placeholders for adding or changing command-line options;
- a set of targets for the most useful operations: parse, eva, sarif.

Running ‘make parse’ will parse the sources; ‘make eva’ will run the analysis; and ‘make sarif’ will produce a SARIF report, a file in a standardized JSON format which can be opened by different tools, such as VS Code.

Showing results

The standard output format for Frama-C is a detailed textual log, which includes alarms, warnings, and several sorts of feedback. It also outputs a summary of the analysis:

```
------------------------------------------------------------------------------
791 functions analyzed (out of 1051): 75% coverage.
In these functions, 10746 statements reached (out of 11224): 95% coverage.
------------------------------------------------------------------------------
Some errors and warnings have been raised during the analysis:
  by the Eva analyzer:      0 errors    3 warnings
  by the Frama-C kernel:    0 errors    1 warning
------------------------------------------------------------------------------
15 alarms generated by the analysis:
     2 accesses out of bounds index
    10 integer overflows
     1 nan or infinite floating-point value
     2 illegal conversions from floating-point to integer
------------------------------------------------------------------------------
Evaluation of the logical properties reached by the analysis:
  Assertions         0 valid     0 unknown     0 invalid      0 total
  Preconditions     75 valid     8 unknown     0 invalid     83 total
90% of the logical properties reached have been proven.
------------------------------------------------------------------------------
```

The most important data are the *warnings* and *alarms*. As mentioned before, such warnings and alarms indicate *potential* issues in the code, which arise due to the inherent imprecision of the analysis[12].

Frama-C also provides a graphical user interface, where alarms can be inspected for further details, including complete information about variables related to the alarms, as illustrated in the screenshot below.

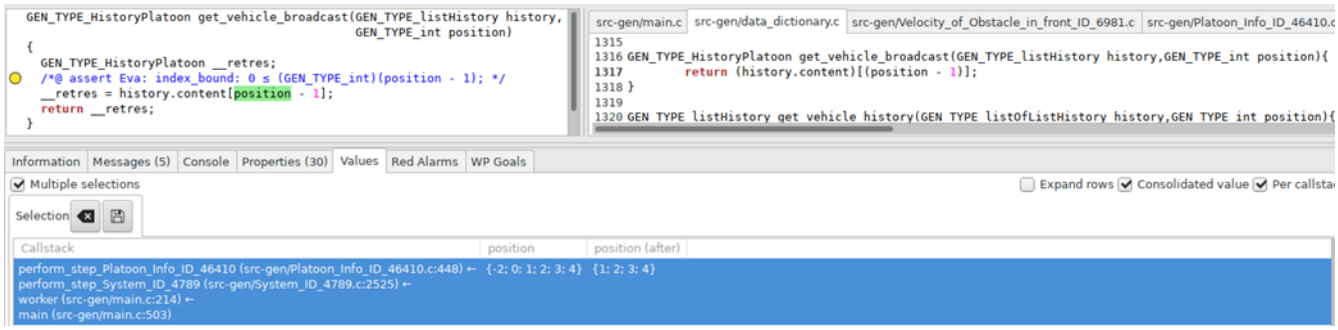

Figure 8: Frama-C GUI screenshot showing an alarm ('position' must be positive), with the original source on the top right corner, the normalized Frama-C code on the top left, and the current callstack and possible values for the selected expression, 'position', in the bottom of the image).

For better integration with external tools (including CI/CD pipeline outputs), Frama-C implemented, during the SPARTA project, an output based on the SARIF[13]. When viewed on an IDE such as VS

[12]Static analyses cannot be both *complete* (exhaustive) and *precise* (exact) for all programs; Frama-C opts for completeness at the cost of some loss of precision (false alarms).
[13] Static Analysis Results Interchange Format; see References at the end of this document for more details.

Code, which includes a free SARIF Viewer extension, we obtain results such as the one displayed in the screenshot below.

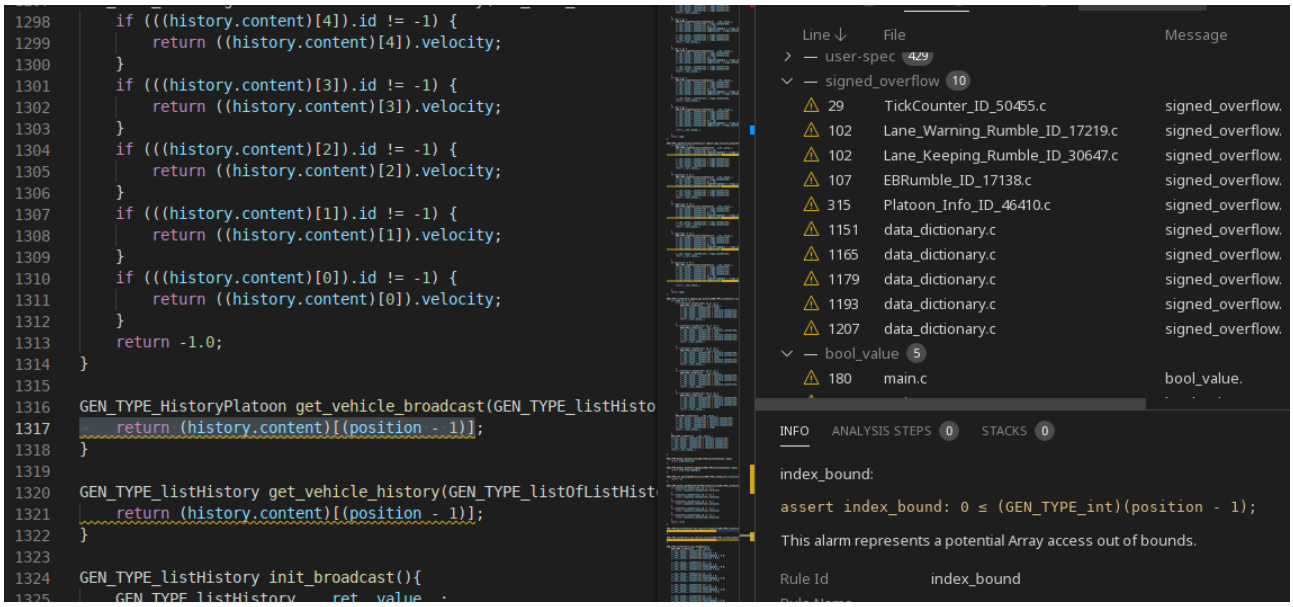

Figure 9: VS Code with SARIF Viewer extension, showing the same alarm as displayed in Figure 1, but with information provided by the SARIF report.

In the top right corner, we have the SARIF Viewer extension panel, which includes a table with the alarms, their location, and their kind (*signed_overflow*, *bool_value*, etc). The bottom right corner displays the full details of the alarm, including the assertion generated by Frama-C. The left half of the screen displays the original source code, with warning squiggles for the lines mentioned in the report. Note that, unlike the Frama-C GUI, VS Code does not display the variable values for each expression at each program point, so we cannot know, for instance, which values caused the *index_bound* alarm.

Note that an equivalent output can also be obtained via other interfaces; e.g. Github supports SARIF integration with code scanning[14].

Finding the root cause of an alarm

As illustrated in the above figures, Frama-C/Eva can report all occurrences of possible runtime errors in the code, but to identify their origin and fix them, some manual work and expertise remains necessary.

Using the Frama-C GUI, a few clicks allow us to inspect the variables, navigate to the caller, quickly grasp the code pattern, and identify a possible fix.

Here's the initial point, at one of the alarms identified in the code:

[14]https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/sarif-support-for-code-scanning

Figure 10: Alarm identified by Frama-C/Eva: possible index out of bounds, due to values -2 and 0 in position.

The variable position is a parameter coming from the caller. The Frama-C GUI allows quickly navigating to the caller:

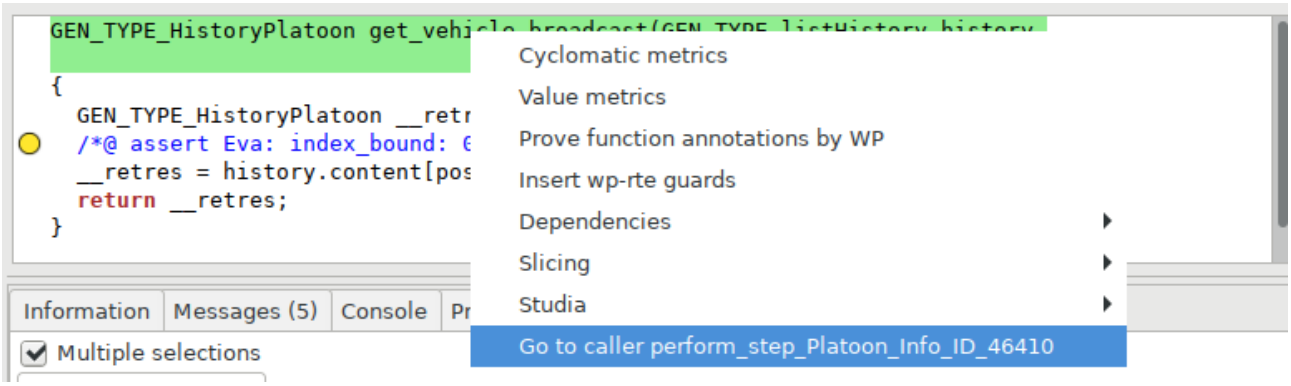

Figure 11: Frama-C GUI provides navigation via contextual menus.

In the caller, we inspect the expression, which is used as parameter, and via its occurrences we identify where it comes from, a few lines above.

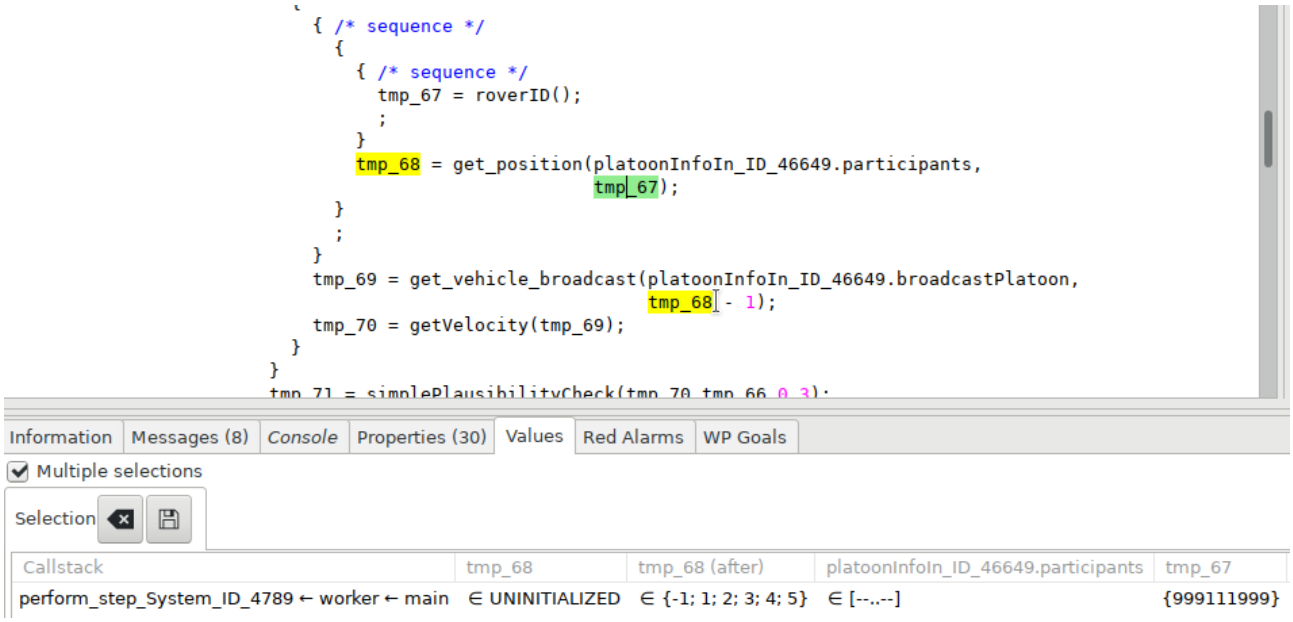

Figure 12: Identifying the link between the position parameter in get_vehicle_broadcast and the return value of get_position, which is temporary variable tmp_68.

Note that the normalized code used by Frama-C expands some expressions, adding temporary variables such as tmp_67 and tmp_68. These variables can be inspected and their values provide useful information about where lies the problem.

In this example, the “spurious” values -1 and 1 come from get_position. Again, using the GUI we navigate to the function, and see its code.

Figure 13: Code of get_position: it returns a value between 1 and 5 when the participant with specified id is found, or -1 otherwise.

Now, we turn our attention to the list of participants, which is shown as [--..--] in the Frama-C GUI. This value, an unbounded interval, means “any value”. Such values often arrive due to loss of precision, but that is not necessarily bad for the analysis: here, it means the function is assuming it must handle *all* arrays of 5 integers, that is, with any IDs for any of the participants.

While this is overly general, one could imagine a scenario where an attacker might want to disrupt the vehicle platoon by arbitrarily modifying the IDs of the participants. In this specific case, such situation should never happen: the upper layers developed by Fortiss already provide defenses against this; however, from the point of view of the code, it could be interesting to add supplementary protection against such cases; even hardware defects could lead to such cases (although, to be fair, there are *many* other parts of the code where such protection would be needed, if we assume this kind of errors is possible).

In any case, adding defensive programming to this part of the code might, if nothing else, help placate static analysis tools (and, possibly, inquisitive evaluators) by convincing them very clearly that values leading to undefined behaviors cannot happen in practice.

Therefore, we now focus on what happens *after* get_position is called. In particular, we have the following *if* condition, where the return value of get_position is used three times[15]:

```
if (hasHistory(get_vehicle_history(historyOld_ID_72073,
  get_position((platoonInfoIn_ID_46649.participants), roverID()))) == 1 &&
  simplePlausibilityCheck(getVelocity(get_vehicle_broadcast(
  (platoonInfoIn_ID_46649.broadcastPlatoon),
  (get_position((platoonInfoIn_ID_46649.participants), roverID()) - 1))),
  computeMeanVelocityHistory(get_vehicle_history(historyOld_ID_72073,
  get_position((platoonInfoIn_ID_46649.participants), roverID())), roverID()), 0.3) == 0)
{
```

In the interest of readability, let us refactor the above code to an equivalent form, by introducing some temporaries to compute intermediate values; we also rename the `GEN_TYPE_*` types to their equivalent ones, e.g. `GEN_TYPE_int` is actually defined to `int`.

```
int roverId = roverID();
int position = get_position(platoonInfoIn_ID_46649.participants, roverId);
HistoryPlatoon broadcast =
  get_vehicle_broadcast(platoonInfoIn_ID_46649.broadcastPlatoon, position - 1);
if (hasHistory(get_vehicle_history(historyOld_ID_72073, position)) == 1 &&
    simplePlausibilityCheck(
      getVelocity(broadcast),
      computeMeanVelocityHistory(
        get_vehicle_history(historyOld_ID_72073, position), roverId
      ), 0.3) == 0) {
```

It is still very hard to read, but the overall idea is: the position is given to `get_vehicle_broadcast` and `get_vehicle_history`, where it will be used as index in an array, to obtain other information.
Therefore, if `position` is a negative value (or 0, since it will be subtracted), the code will try to access a negative array index, leading to undefined behavior: a runtime error (crash) at best, or a silent error that may lead to catastrophe later.
We identified the cause of the alarm emitted by Frama-C, which points to a solution based on defensive programming: before using the result of get_position, we check that it is valid (> 1), and if that is not the case, then we abort the execution: raise an exception, emit an error message, fallback to a “safe mode”, etc. This will not change the behavior of the code in case nothing goes wrong; except for an extra test (with minimal impact to execution time), the code will do exactly as it did

15 The attentive reader will notice that get_position is always called with the same arguments; modern compilers should be able to optimize the calls, so this should not incur any performance issues.

before. If, for any reason, the index should become invalid (attacker, hardware fault, undiagnosed bug), the check will detect it and fail quickly and cleanly. This may also help future-proof the code: should some of the components change, breaking the invariant, it will be detected by the check.

Patching the code

Assuming the refactored code above, we simply insert a call to an appropriate error handler; for simplicity, here we print an error message and call `exit()`:

```
int roverId = roverID();
int position = get_position(platoonInfoIn_ID_46649.participants, roverId);
if (position <= 1) {
  fprintf(stderr, "invalid position: %d", position);
  exit(1);
}
HistoryPlatoon broadcast =
  get_vehicle_broadcast(platoonInfoIn_ID_46649.broadcastPlatoon, position - 1);
if (hasHistory(get_vehicle_history(historyOld_ID_72073, position)) == 1 &&
    simplePlausibilityCheck(
      getVelocity(broadcast),
      computeMeanVelocityHistory(
        get_vehicle_history(historyOld_ID_72073, position), roverId
      ), 0.3) == 0) {
```

Note that, since we are dealing with generated code, this patch is not intended to be permanent, but a proof of concept. For a persistent solution, the *code generator* needs to be modified to incorporate the change.

Re-running the analysis

In order to ensure that the code change does remove the possible undefined behavior, we re-run the analysis on the patched code and observe the results. Indeed, we had 15 alarms, and now we have 14:

```
  ------------------------------------------------------------------------
  14 alarms generated by the analysis:
       1 access out of bounds index
      10 integer overflows
       1 nan or infinite floating-point value
       2 illegal conversions from floating-point to integer
  ------------------------------------------------------------------------
```

This verification also ensures we did not inadvertently *add* a different alarm. This can happen due to loss of precision, even if the patch is correct.

Using an IDE, we can also see that the warning disappeared, as in the figure below.

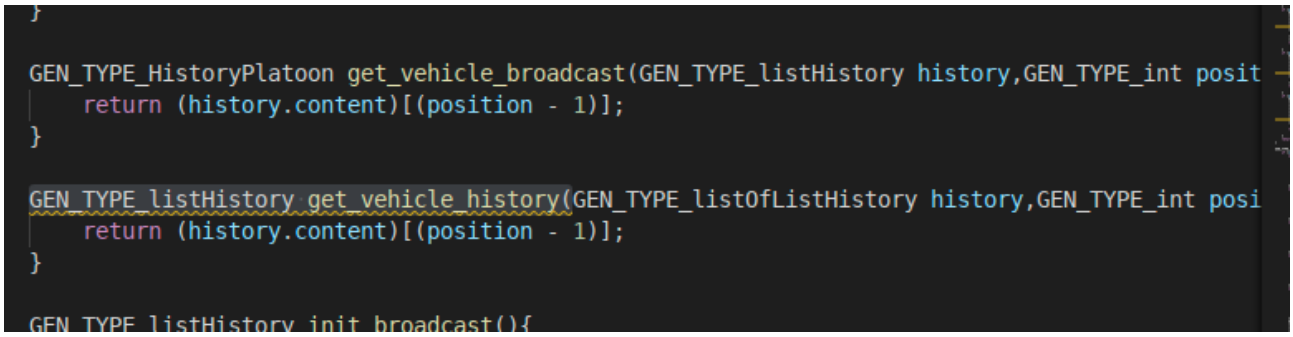

Figure 14: VS Code screenshot after the patch adding a check for variable 'position'.

Appendix F
Protection Profile for a Safety and Security Platooning Management Module including a firewall

A base Protection Profile (PP) for a Safety and Security Platooning Management Module (SafSecPMM) including a firewall is described in detail in the document SPARTA-D5.4-M36_AppendixF.

Appendix G
Impact Analysis Report - Vertical 1 - Scenario 2

An Impact Analysis Report for the Vertical 1 Scenario 2 is described in detail in the document SPARTA-D5.4-M36_AppendixG.

Appendix H
ALC Life-Cycle – Vertical 1 (OpenCert)

OpenCert (https://www.eclipse.org/opencert/) is an open source 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. For more details about OpenCert, we refer the interested reader to D5.1 [1] and D5.2 [2].

At high-level, OpenCert is divided into 8 functional groups, as shown in Figure 15. The functional groups that are involved in the Connected Car Platooning scenario (Vertical 1) are marked with the SPARTA's project logo:

- Prescriptive Knowledge Management: This feature supports management of knowledge about standards.
- Assurance Project Lifecycle Management: This functionality factorizes aspects such as the creation of safety assurance projects and any project baseline information related with the product to certify (the baseline includes only those parts of the standards that are applicable to the asset to be certified, in our case, the platooning system).
- Safety Argumentation Management: This feature manages argumentation information in a modular fashion.
- Evidence Management. This functionality allows to manage the full life-cycle of evidences and evidence chains.

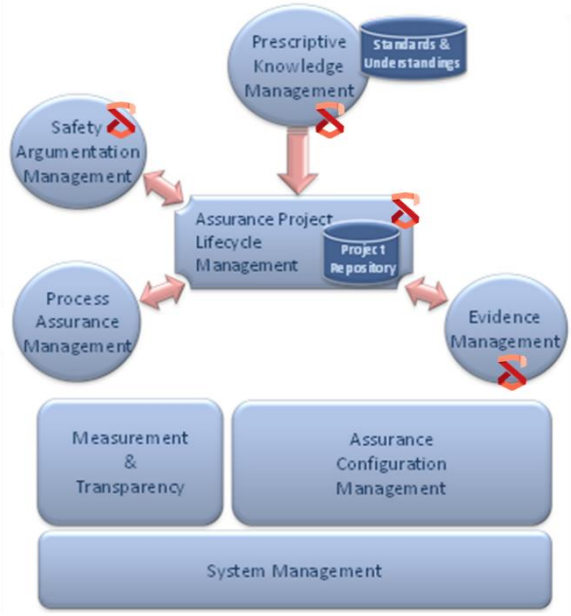

Figure 15: Functional decomposition for the OpenCert platform

In D5.3 [4] we described the activities that were developed in the first iteration of the scenario 4 ("Safety and Security compliance assessment and certification") of Vertical 1. These activities include: (i) the digitalization of a safety standard (ISO 26262 "Functional Safety Road Vehicles" [23]) and a security standard (SAE J3061 "Cybersecurity Guidebook for Cyber-Physical Vehicle System" [24]); (ii) the creation of an Assurance project, named "Platooning Assurance", with two baselines, one per each standard that the Platooning Management System must comply with; (iii) the addition

of some of the evidences into the Assurance project; and (iv) the justification of the compliance of the Assurance project with the argumentations.

In the second iteration of the Scenario 4, we have added the security evidences to the Assurance project and have made the necessary argumentations to specify convincing justification that our platoon system is adequately safe and secure according the ISO 26262 and SAE J3061 standards. Co-assessment between safety and security has also been taken into account in our safety cases.

The security evidences have been added to the Assurance project by generating a new evidence model in the OpenCert "Evidence Management" module. The evidences are obtained by analysing, testing, simulating and estimating the properties of a system. The new evidence model added to the "Platooning Assurance" project follows the structure of the Common Criteria standard, as is shown the Figure 16.

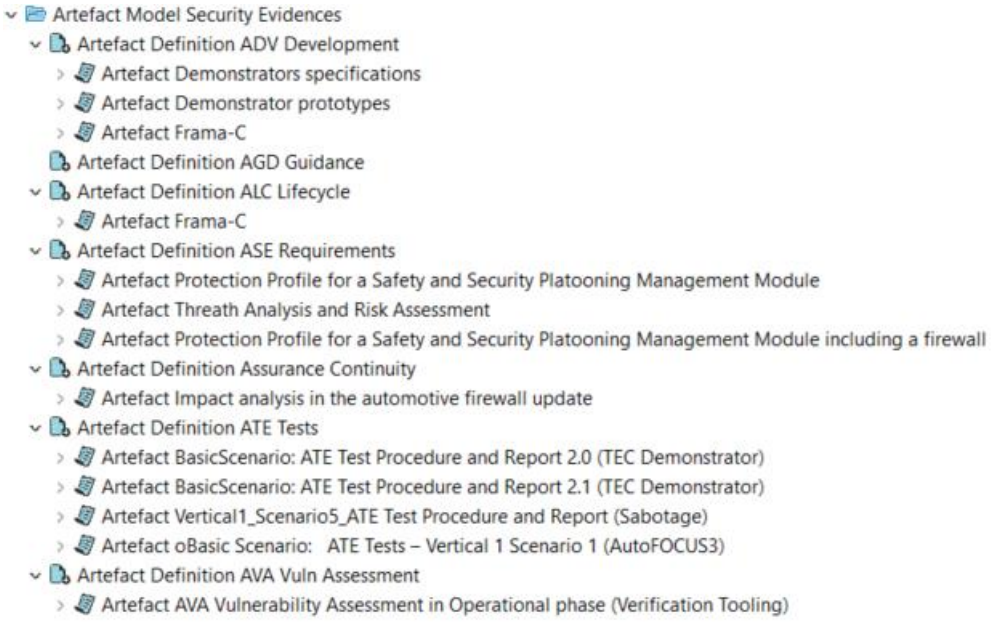

Figure 16: Security evidence model structure

- ADV Development: This class deals with the evaluation of the six families of requirements for structuring and representing the security functionality realized by the target of evaluation (TOE). In the "Platooning Assurance" project, the D5.2 [2] and D5.3 [4] deliverables have been added, as well as the Frama-C output results.
- AGD Guidance: This class takes care of the evaluation of the manuals that are delivered to the customer. These manuals contain both the secure configuration process of the TOE in its user environment and its safe use methods for each category of defined end-user. In our case, no manual has been generated.
- ASE Requirements: This class takes care of all the security requirements of the Security target of evaluation. In the "Platooning Assurance" project, the Protection Profile (PP) developed in D5.2 Appendix B [3], the Protection Profile developed for the Assurance Continuity purposes (Appendix F) and the Threat Analysis and Risk (TARA), which was created during the first iteration of the scenario 4, have been added.
- ATE Tests: It is the class that takes into consideration all the tests that demonstrate that security functionalities operate according to its design descriptions, both the functional ones proposed by the developer and the independent ones proposed by the evaluators. In the "Platooning Assurance" project, three test procedure documents have been added to demonstrate the security of the "Platooning Assurance" project:
    - ATE Tests – Vertical 1 Scenario 1 (TEC Demonstrator) (see Appendix A).
    - ATE Tests – Vertical 1 Scenario 1 (AutoFOCUS3) (see Appendix B).
    - ATE Tests – Vertical 1 Scenario 5 (Sabotage tool) (see Appendix C).

- AVA Vulnerability Assessment: This class takes care of the vulnerability assessment activity to analyse vulnerabilities in the development and operation of the TOE. Development vulnerabilities are those introduced during its development that can be minimized through the adoption of “security by design” processes by the developer. Operational vulnerabilities are those that could exploit the weaknesses of non-technical countermeasures to violate the TOE security functionality. The vulnerabilities of the Platooning system have been analysed during the operational phase. An exhaustive penetration testing has been performed by EUT using different Verification tools and SysML tool. To obtain more information about Vulnerability Analysis, please see Appendix D.
- Assurance Continuity: This activity is related to maintain the certificate of an evaluated product during its phases of patch management and improvement/evolution. The document reports the changes affected by the evaluated TOE and provide a classification of them in minor (in this case a subset of evaluation activities should be performed for updating the certification) or major (a re-certification is needed for the updated TOE). To see the impact of the second iteration, an Impact analysis Report has been developed (see Appendix G).
-

All the evidences have been stored in the SPARTA SVN repository. Also, we have modelled Artefacts inside the Evidence model in OpenCert that have been linked to the real documents in the SVN folder. In this way the evidences can be opened and edited directly from OpenCert and the history of changes is also shown in the tool. As an example, Figure 17 shows how the vulnerability assessment document developed by EUT has been modelled in the “AVA Vulnerability assessment” artefact in the operational phase, the evolution of different versions of the resource and the contents of the document.

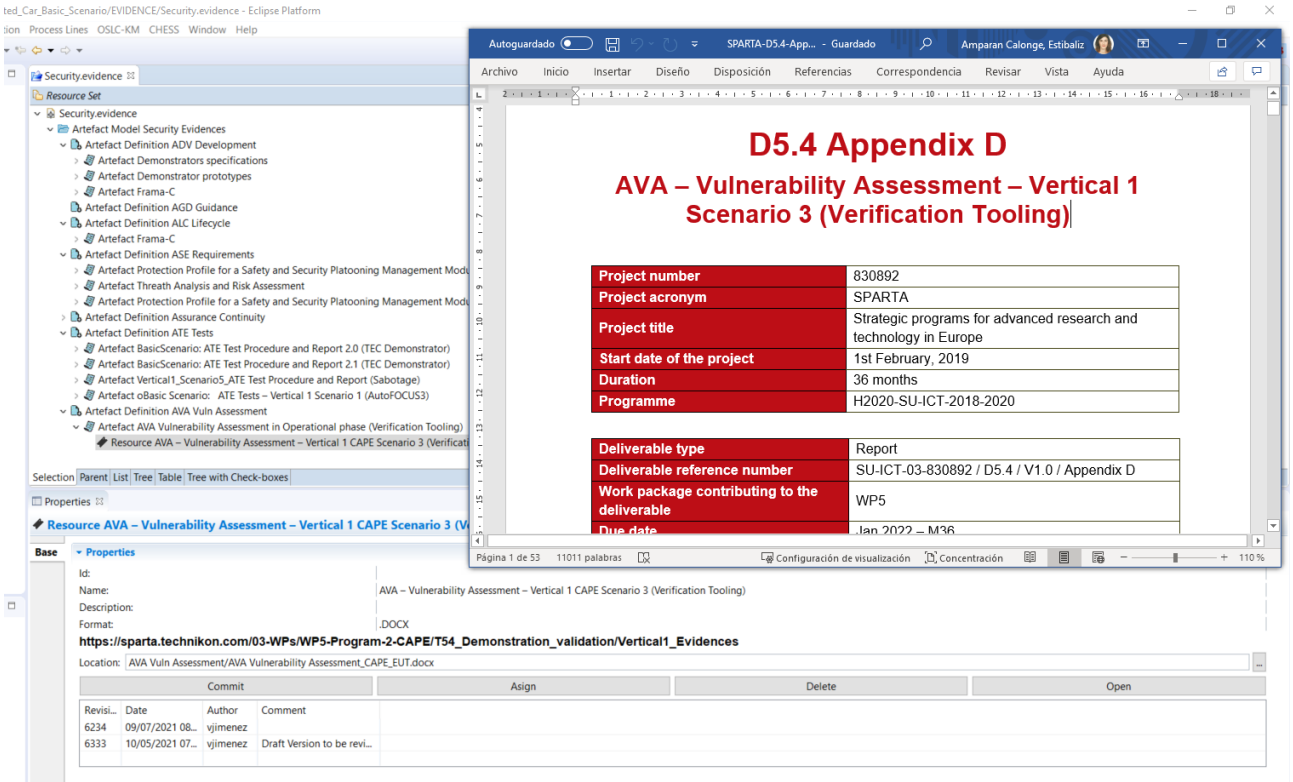

Figure 17: Example of an evidence stored in the OpenCert tool

Besides, we have created several argumentation models to confirm that our platooning is adequately safe and/or secure for the platooning system in a highway environment. In addition, these evidences should come with reasonable explanations indicating the acceptable security and safety, usually by demonstrating compliance with requirements, avoidance of hazards and threats, etc. The assurance case is created in a graphical way by adding argumentations using the Goal Structuring Notation

(GSN) [25] for presenting the goals/requirements, the evidences, the strategy adopted, the justification and the context in which goals are stated. All the goals, subgoals and requirements are defined, together with a context where the system is set, and the strategy to explain the approach adopted.

Figure 18 shows how the main goal of our system is to obtain a safe and secure Platooning management module. To achieve this high-level goal, some other minor goals have been defined. The strategy used has been to use the Protection Profile document of the Common Criteria standard as a guide. As mentioned before, the Platooning system has been settled in a specific environment where the platooning is formed, the vehicles are on highway and it is assumed that the system is HW tampered resistant. These subgoals are also divided in some requirements that are implemented in the Platooning system. To have more information about the subgoals and requirements defined in the argumentation, the reader can refer to the protection profile document [3] where are deeply described.

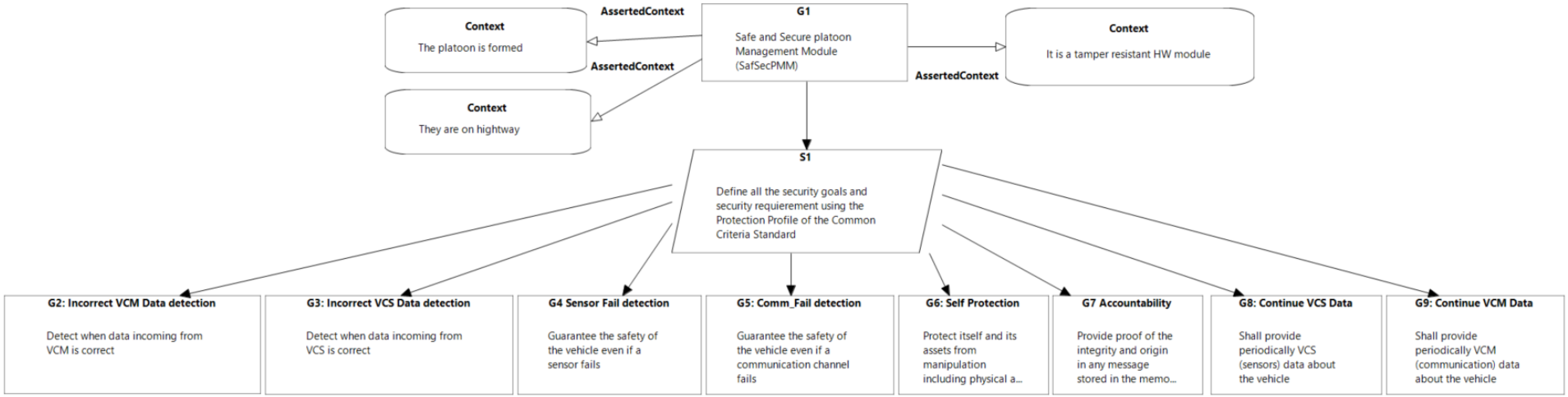

Figure 18: Main assurance case for the Platooning system

Figure 19 shows how the requirements of each subgoal are linked to one or more evidences justifying the acceptable assurance level. The evidences added in the Assurance case are those that have already been added previously in the "Evidence Management" module and it is possible to open them by selecting the evidence, for example in this case, the "ATE Tests – Vertical 1 Scenario 1 (TEC Demonstrator)" document (see Appendix A).

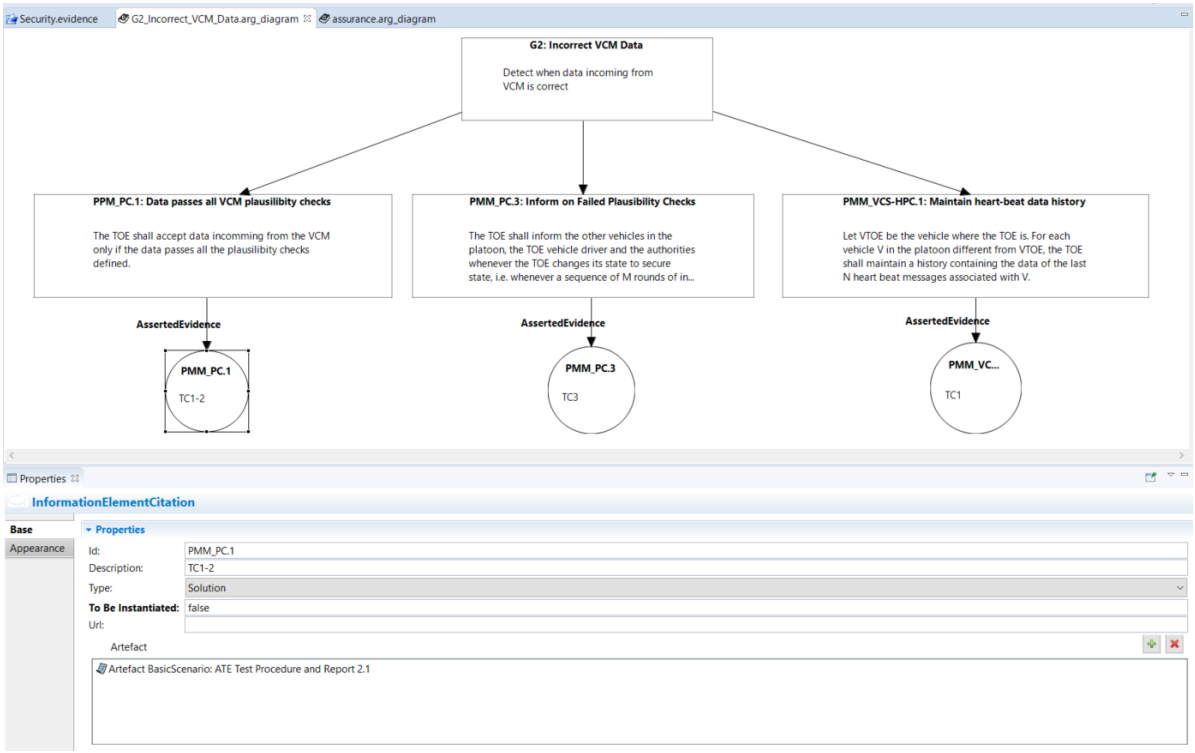

Figure 19: Argumentation with evidences

Appendix I
ATE Test Procedure and Report - Vertical 2 - Mobile Scenario

Tests results obtained by the usage of the APPROVER and TSOpen tools for verifying the implementation of Security Functionalities of Vertical 2 Mobile Scenario are described in detail in SPARTA-D5.4-M36_AppendixI.

Appendix J
ATE Test Procedure and Report - Vertical 2 - SAML IdP Server Scenario

Tests results obtained by the usage of ProjetKB, Steady and VI for verifying the implementation of Security Functionalities of Vertical 2 SAML IdP Server Scenario are described in detail in SPARTA-D5.4-M36_AppendixJ.

Appendix K
ADV Development – Vertical 2 (SafeCommit)

SafeCommit Overall Goal:

SafeCommit aims at automatically detecting commits that introduce vulnerabilities (we will also refer to commits as patches for the 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.

In the Wild Assessment: The Vertical 2 Case.

Introduction

SafeCommit has been integrated into the continuous integration system of Vertical 2 (i.e., into GitLab). In particular, SafeCommit has been fully integrated into the repository of vertical 2 for verifying new commits of source code for the SAML IdP Server scenario. Each time a patch is committed into the code base of the SAML IdP Server repository, SafeCommit analyzes the patch, and a report is generated indicating whether the patch introduces a vulnerability or not.

To assess SafeCommit in a such setting, we commit three different patches into the repository:

1- safe.patch, a patch that does not introduce any vulnerability.
2- unsafe.patch, a patch that does introduce a vulnerability.
3- safeCommit_skip, a patch that only adds text into a text file. So, this patch is not related to code, and does not introduce any vulnerability.

We expect that SafeCommit will yield a security warning for unsafe.patch, and no warning for both safe.patch and safeCommit_skip.

Test Case #1: safe.patch

For the first test case, we commit into the repository a patch that does not introduce any vulnerability. The commit is identified in Figure 20, and the patch is detailed in Figure 21.

Figure 20: Commit identification

In Figure 21, we can see that the patch creates and instantiates a set (variable “cars”), and then populates the set by adding car brands. Finally, each element of the set is “printed”.

0001-safe.patch | 0001-unsafe.patch | 0001-safecommit_skip.patch

```
From b72abc280b893ad3adcdbf4d826f2c1d863c6c79 Mon Sep 17 00:00:00 2001
From: Kevin Allix <kevin.allix@uni.lu>
Date: Mon, 8 Nov 2021 20:35:17 +0100
Subject: [PATCH] LoopHashSet.java

---
 LoopHashSet.java | 17 +++++++++++++++++
 1 file changed, 17 insertions(+)
 create mode 100644 LoopHashSet.java

diff --git a/LoopHashSet.java b/LoopHashSet.java
new file mode 100644
index 0000000..428b7b8
--- /dev/null
+++ b/LoopHashSet.java
@@ -0,0 +1,17 @@
+// Import the HashSet class
+import java.util.HashSet;
+
+public class Main {
+   public static void main(String[] args) {
+       HashSet<String> cars = new HashSet<String>();
+       cars.add("Volvo");
+       cars.add("BMW");
+       cars.add("Ford");
+       cars.add("BMW");
+       cars.add("Mazda");
+       for (String i : cars) {
+           System.out.println(i);
+       }
+   }
+}
+
--
2.30.2
```

Figure 21: safe.patch, a patch that does not introduce any vulnerability.

Once the patch is committed, SafeCommit automatically analyzes and “test” the patch. If the patch does not contain any vulnerability, the test passes. This can be seen in Figure 22. By putting the mouse on the green checkbox, a popup appears that indicates the test is passed.

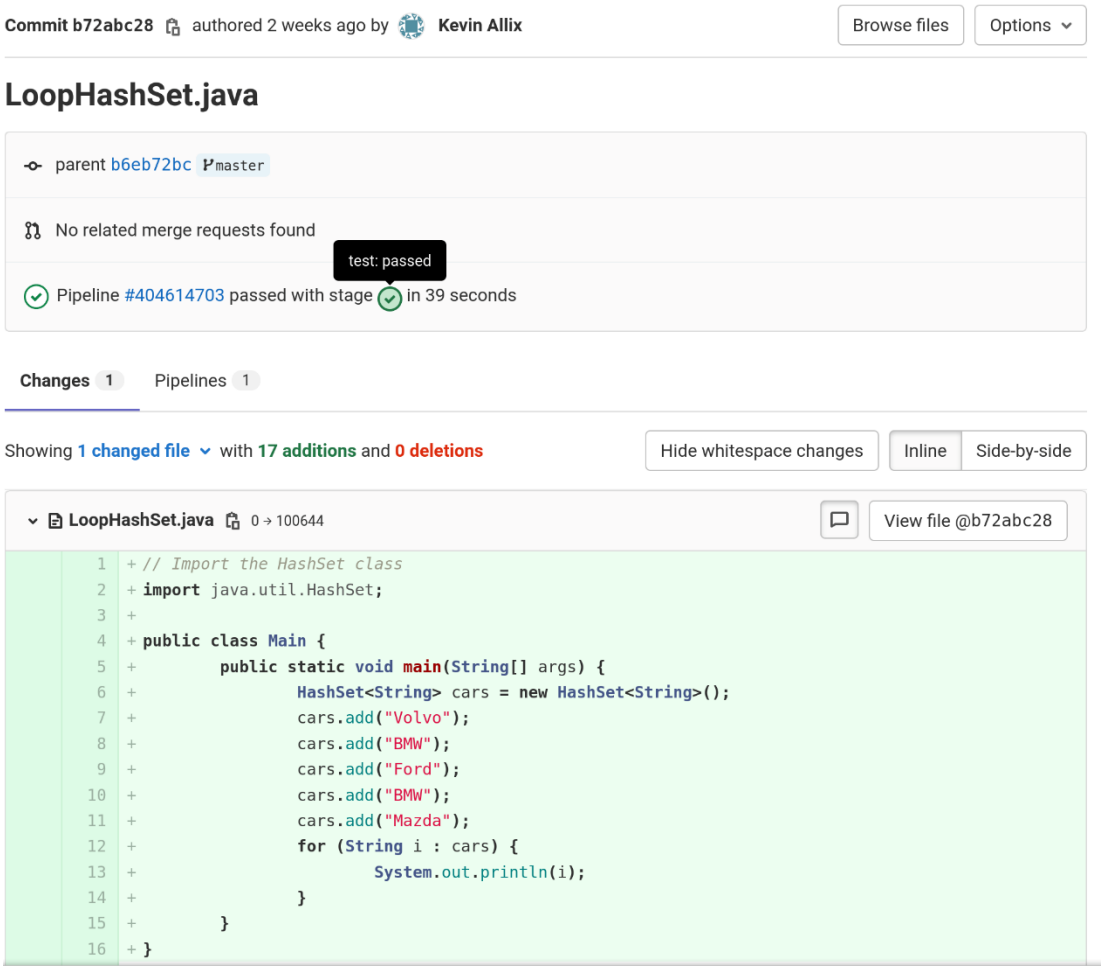

Figure 22: The test is passed meaning that SafeCommit does not detect any vulnerability in this patch.

Eventually, SafeCommit also generates a report which can be seen in Figure 23. This report is an xml file (Junit format). A message is generated in the report only if SafeCommit detects a vulnerability. This is not the case for this test case.

```
−<testsuite>
    <testcase name="SafeCommit" classname="SafeCommit.SafeCommit"> </testcase>
  </testsuite>
```

Figure 23: SafeCommit Report

Test Case #2: unsafe.patch

For the second test case, we commit into the repository a patch that introduces a vulnerability. The commit is identified in Figure 5, and the patch is detailed in Figure 24.

Figure 24: Commit Identification of the unsafe.patch patch

In Figure 25, we can see that the patch contains a buffer overflow vulnerability. Indeed, `name` is a char array of size 64. The problem is that `name` is "controlled" by the user that can feed `name` with no constraint because of `scanf`.

0001-safe.patch | 0001-unsafe.patch | 0001-safecommit_skip.patch

```
From 892a0054de6c39cdf0adf443eaa07c96103060c0 Mon Sep 17 00:00:00 2001
From: Kevin Allix <kevin.allix@uni.lu>
Date: Mon, 8 Nov 2021 20:25:07 +0100
Subject: [PATCH] example1.c

user input
---
 example1.c | 9 +++++++++
 1 file changed, 9 insertions(+)
 create mode 100644 example1.c

diff --git a/example1.c b/example1.c
new file mode 100644
index 0000000..e116a83
--- /dev/null
+++ b/example1.c
@@ -0,0 +1,9 @@
+int _tmain(int argc, _TCHAR* argv[])
+{
+    char name[64];
+    printf("Enter your name: ");
+    scanf("%s", name);
+    Sanitize(name);
+    printf("Welcome, %s!", name);
+    return 0;
+}
--
2.30.2
```

Figure 25: unsafe patch that introduces a vulnerability

Once the patch is committed, SafeCommit automatically analyzes and “test” the patch. If the patch contains a vulnerability, the test is passed but with warnings. This process can be seen in Figure 26. By putting the mouse on the red exclamation mark, a popup appears that indicates the test passed but with warnings.

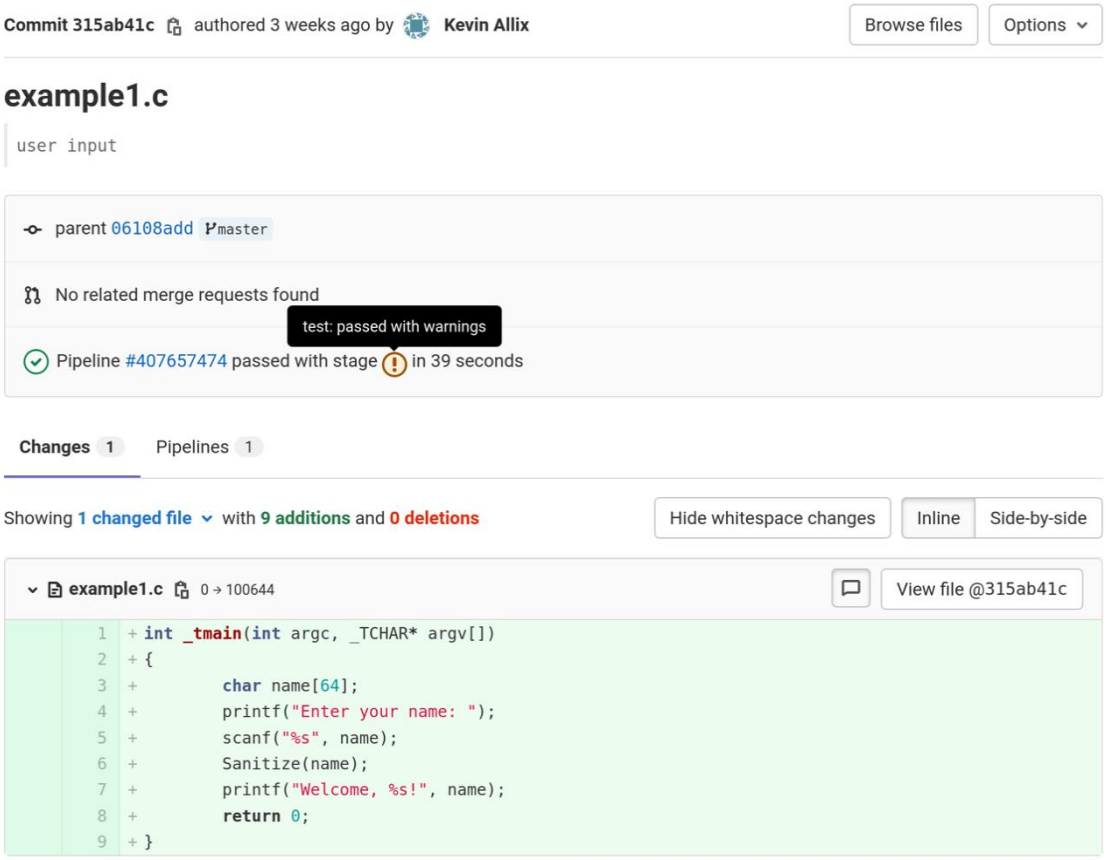

Figure 26: The test contains a warning meaning that SafeCommit has potentially detected a vulnerability in this patch.

Eventually, SafeCommit also generates a report which can be seen in Figure 27. This report is an xml file (Junit format). A message is generated in the report because SafeCommit detects a vulnerability.

```
−<testsuite>
  −<testcase name="SafeCommit" classname="SafeCommit.SafeCommit">
    −<failure message="SafeCommit thinks commit 23268419b92f12e8e1a42c1c8a76619fbc6c5ef3 introduces a vulnerability">
       Potential vulnearbility in commit 23268419b92f12e8e1a42c1c8a76619fbc6c5ef3
     </failure>
  </testcase>
</testsuite>
```

Figure 27: Safe Commit Report when it detects a vulnerability

Test Case #3: safecommit_skip.patch

For the third test case, we commit into the repository a patch that simply adds text into a text file. Consequently, this patch does not introduce any vulnerability. The commit is identified in Figure 9, and the patch is detailed in Figure 28.

Figure 28: Commit Identification of the safecommit_skip patch

In Figure 29, we can see that the patch adds three lines in a text file.

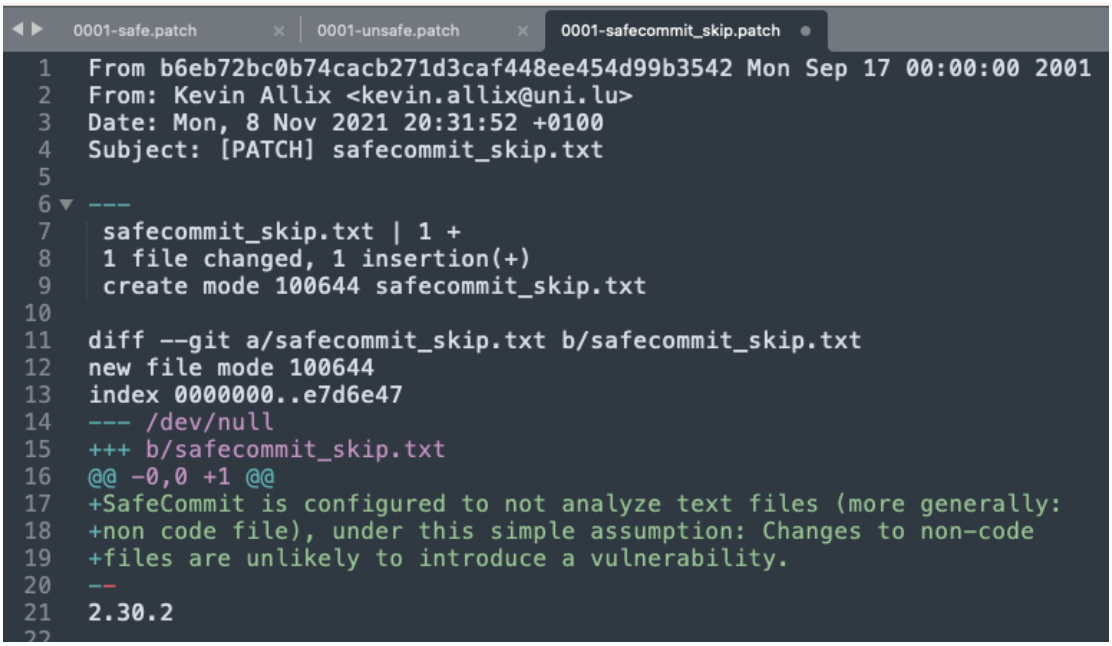

Figure 29: patch that simply adds texts into a text file

Once the patch is committed, SafeCommit automatically analyzes and “test” the patch. However, since the patch targets a text file, the analyze stop immediately, and the test is marked as passed. This can be seen in Figure 30. By putting the mouse on the green checkbox, a popup appears that indicates the test is passed.

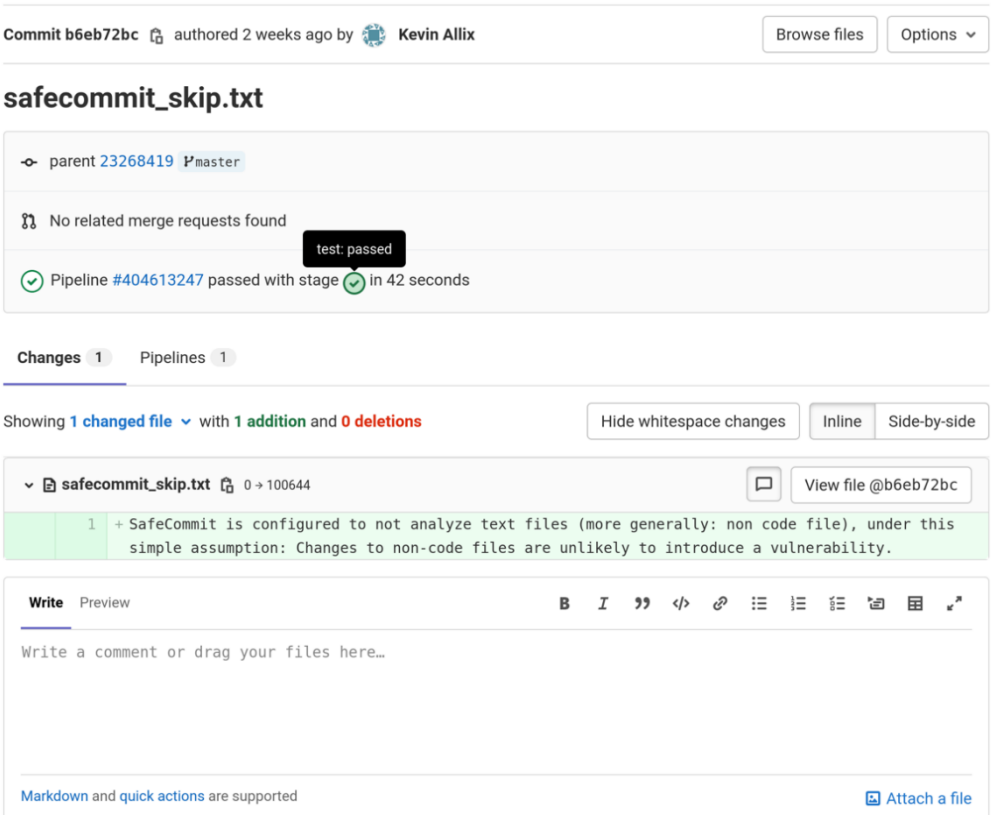

Figure 30: The test is passed meaning that SafeCommit does not detect any vulnerability in this patch.

Eventually, SafeCommit also generates a report which can be seen in Figure 31. This report is an xml file (Junit format). In this report, we can see that no vulnerability has been reported (no message has been generated), but also that the analysis has been skipped.

```
▼<testsuite>
  ▼<testcase name="SafeCommit" classname="SafeCommit.SafeCommit">
     <skipped/>
   </testcase>
 </testsuite>
```

Figure 31: SafeCommit Report

Conclusion

Three test cases have been executed to assess SafeCommit on Vertical 2. The test cases are:

1- safe.patch => SafeCommit expected result: no vulnerability detected
2- unsafe.patch => SafeCommit expected result: a vulnerability detected
3- safeCommit_skip => SafeCommit expected result: no vulnerability detected

The three test cases passed meaning that the SafeCommit outputs were as expected.

Appendix L
Additional test and results reported outside the Verticals scope

Tests results obtained by the usage of:

- SATRA (ex NeSSoS tool)
- Buildwatch
- SideChannelDefuse
- Steady/ProjectKB (tests outside the scope of the Vertical 2)
- Legitimate Traffic Generation System (LTGen)

are described in detail in SPARTA-D5.4-M36_AppendixL

Appendix M
Short videos description

In the following is presented a short description of the videos of tools developed during the Task 5.4 for demonstration purpose.

The following Table 8 provides information about the videos of tools involved in the Vertical 1.

Tool	File Video	Short Description
(TECNALIA) TEC demonstrator	ATE_Vertical1_Scenario1_TEC_demo.mp4	The TEC demonstrator for Vertical 1 Scenario 1 implements a subset of requirements described in the Protection Profile for a Safety and Security Platooning Management Module (PMM).
(TECNALIA) Sabotage	ATE_Vertical1_Scenario5_Sabotage_demo.mp4	The video shows the procedure to validate the correctness of the implementations for those requirements, using as an example the test case to validate the generation and the composition of Emergency Break messages.
(Eurecat) Verification Tooling (including SysML usage)	AVA_Vertical1_Scenario3_demo.mp4	The video describes the different steps performed in the Verification Tooling Scenario, from the analysis of the protection profile and elaboration of an attack tree, preparation of the Verification Tools, pre-assessment in an internal laboratory and AVA Vulnerability assessment. Some vulnerabilities are illustrated with real videos. 7 vulnerabilities have been found and remediations actions have been recommended
(CETIC) Vacsine	AVA_Vertical1_Scenario2_Vacsine_demo.mp4	Securing platoon firewall updates - Improving security of the continuous integration and deployment pipeline of connected cars. We demonstrate a supply chain attack on a connected car firewall and how to protect it with a DevSecOps approach: how to detect the attack with vulnerability scanning and how to orchestrate the remediation using the Vacsine tool.
(FORTISS) AutoFocus	Vertical1_Scenario1_Autofocus_demo.mkv	This demo shows how one can validate the requirement "reception of emergency brake messages (EB) generated by the platoon leader and received by a follower" from the Protection Profile.
(CEA) Frama-C	Video_Frama-C_Vertical1_demo.mp4	The video shows how to setup Frama-C on a code base and how to run an analysis to identify security vulnerabilities, with an example of a fix.

Tool	File Video	Short Description
TECNALIA (OpenCert)	ALC_Vertical1_Scenario4_OpenCert_demo.mp4	The video shows the use of the OpenCert tool for the management of the safety (ISO 26262) and security assessment (SAE J3061) in the context of Vertical 1. It shows how all the evidences of the evaluation process for Vertical 1 are stored, and the generation of argumentations to justify the safety and security assessment by using the previous stored evidences and explanations.

Table 8: Short videos description for tools involved in Vertical 1

The following Table 10 provides information about the video of tools involved in the Vertical 2.

Tool	File Video	Short Description
- CINI (Approver) - (UniLu) TSOpen - (UniLu) Safe Commit - (SAP/UKON) ProjectKB/Steady/VI	SPARTA_APP_withComment.mp4	The video shows how the CAPE tools SATRA, APPROVER and TSOpen have been integrated in DevSecOps and contribute to secure the CIE ID APP scenario of Vertical 2.
	SPARTA_IdP_withComment.mp4	The video shows how the CAPE tools SATRA, Steady, Vulnex and SafeCommit have been integrated in DevSecOps and contribute to secure the SAML IdP scenario of Vertical 2

Table 9: Short videos description for tools involved in Vertical 2

The following Table 10 provides information about the videos of standalone tools.

Tool	File Video	Short Description
(CNR) SATRA	SATRA demo.mp4	The video provides a quick overview of the SATRA tool, a simple and fast approach for quick risk assessment of a software product based on the analysis of security practices applied during the SDLC. The video shows the key steps for operating the tool and provides brief explanations for these steps. It demonstrates the complete process (omitting repeating steps like filling in the questionnaire) and makes the viewer familiar with the required inputs and expected outcomes of the tool.

Tool	File Video	Short Description
(UBO) Buildwatch	UBO_buildwatch-demo.mp4	This demo video shows how Buildwatch may be integrated into a CI pipeline in order to detect suspicious forensic artifacts (e.g. files, processes, or network connects) that occur during the build/install/test phase of a software developed. In this example a small projects updates one of its dependencies two times. The first time only benign forensic artifacts are found. On the second update there occurs a new process that would create a reverse shell to the attackers server. As the code was tested in an isolated sandbox in the background no harm to the rest of the pipeline would happen. A developer working on the project would notice the use of a malicious dependency and hence could stop the distribution of the software
(CNIT) SideChannelDefuse	CNIT_D64_SideChannelDefuse_DEMO.mp4	The video represents a demonstration of the tool's capabilities of continuous detection and mitigation at Kernel level of Side Channel attacks. We first show a regular usage of a Linux system. After that, we launch a simulated Side Channel attack. We show that the module immediately detects and mitigates the attack without impacting performances on the host.
(IMT) LTGen	IMT_LTGen_demo.mp4	The video demonstrates the workflow of legitime traffic generation on a couple of packet-level features (packet length, inter-arrival time). It describes how from a set of features to reproduce (min, max, mean, std) of each feature, we are able to generate a \beta-distribution that aims at reproducing an existing legitimate flow behaviour. The distribution is then transformed into concrete packets using scapy and replayed into an emulated network environment using ContainerNet. This demonstrates the ability to build a testing environment with legitimate traffic generation against security measures, notably Network IDS. Ultimately, the video shows how well the generation performs with respect to the initial features provided

Table 10: Short videos description for standalone tools