D3.3
REWIRE Design Time Secure Operational Framework - Final Version

Project number:	101070627
Project acronym:	**REWIRE**
Project title:	REWiring the ComposItional Security VeRification and AssurancE of Systems of Systems Lifecycle
Project Start Date:	1st October, 2022
Duration:	36 months
Programme:	HORIZON-CL3-2021-CS-01
Deliverable Type:	Report
Reference Number:	HORIZON-CL3-2021-CS-01-101070627/ D3.3 / v1.0
Workpackage:	WP2
Actual Submission Date:	30th September, 2025
Responsible Organisation:	SECURA
Editor:	Sjors Van den Elzen
Dissemination Level:	Public
Revision:	v1.0
Abstract:	D3.3 presents the final versions of the components participating in the design-time phase of the REWIRE framework and the definition of the trust boundary of the system. Design and implementation details were presented for the Secure SW Update, the Compositional Verification and Validation component, the SW/FW Vulnerability Analysis, the AADL-based traceability of requirements, the MSPL-based Security Policies, the Risk Assessment, and the establishment of authenticated channels. Benchmarking results were provided, demonstrating the performance of the implemented solutions.
Keywords:	Compositional Verification, Vulnerability Analysis, Risk Assessment, Software Update, Security Policies

The The project REWIRE has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No 101070627.

Editor

Sjors Van den Elzen(SECURA)

Contributors (ordered according to beneficiary numbers)

Stylianos Basagiannis, Simone Fulvio Rollini (UTRCI)
Dimitris Karras, Nikos Fotos, Stefanos Vasileiadis, Thanassis Giannetsos (UBITECH)
Corentin Verhamme, Francois Koeune (UCL)
Sjors Van den Elzen (SECURA)
Samira Briongos, Javier de Vicente Gutierez (NEC)
Jesus Sanchez (ODINS)
Paraskevas Papanikolaou, Charoula Pechilvani, Athanasios Athanasiadis (KENOTOM)
Ilias Aliferis (UNIS)

Disclaimer

Executive Summary

One of the core offerings of REWIRE is the *secure lifecycle management of embedded systems considering the trend towards the increasing adoption of OpenHW architectures, such as RISC-V, through the provision of the necessary security and trust enablers*. In this regard, REWIRE aims to address the key challenge of ensuring the correctness of software and hardware artefacts for achieving a high level of trustworthiness guarantees in embedded systems, through a holistic security framework, whose architectural details were presented in D2.2. This framework comprises two core phases, the **Design-time** and **Runtime** phases, which aim to cover the entire operational lifecycle of the device. The present deliverable is dedicated to the detailed documentation of the **artefacts**, **schemes**, and **methods participating in the design-time phase of the framework** and their role towards the achievement of the goal of this phase, namely the *definition of the trust boundary of the system, i.e., the identification of device properties that can be formally verified during design-time and those that need to be verified dynamically during runtime*.

One of the core outcomes of the design-time phase is the deployment of mitigation measures against the identified threats and vulnerabilities of the devices through the appropriate *security and operational policies*. One such measure is the deployment and installation of software updates in order to mitigate such vulnerabilities. In this context of REWIRE, this is performed through the **Secure SW Update with Authenticated Encryption (AE)**, which is analysed in Chapter 2. As REWIRE aims to ensure the installation of such updates in a manner that fulfils the overarching security requirements of the target domain, AE is a symmetric cryptographic construction employed for securing the communication channel used for software updates. Note that REWIRE supports two modalities of the SW update process for ensuring applicability and scalability in multiple types of operational domains, namely **"1-to-1"** for the distribution of an update to a single device, and **"1-to-many"** for the distribution of an update from a single source to multiple recipients.

The backbone of the design-time phase of REWIRE is the **Compositional Verification and Validation Component**, which is described in detail in Chapter 3, and its core functionality entails the use of **Formal Verification** tools in order to verify all security protocols and schemes, before they are synthesized into the actual devices. In this deliverable, we expand on the rationale and motivation behind the use of **Verifpal** as the core Formal Verification tool of this component, as well as its implementation details in the context of REWIRE. Then, we focus on the Formal Verification of the **Configuration Integrity Verification (CIV)** scheme, which relies on local attestation using **key restriction usage policies**. Note that this is a key innovation of REWIRE, as the verification of such schemes has not yet been explored in the literature.

Chapter 4 is dedicated to the description of the **SW/FW Vulnerability Analysis** component of REWIRE, whose purpose is the analysis of the SW or FW image of a device, in order to identify any threats and vulnerabilities that need to be mitigated through the security enablers of REWIRE. This component employs a mixture of new and well-established fuzzing techniques, capable of coping with the internal interdependencies of RISC-V architectures, in order to *provide more information to the SW Service Provider that enables them to push the appropriate software update*. Then, the identified vulnerabilities are pushed to the **Risk Assessment** component, so that the appropriate security controls as mitigation measures are identified and pushed in the form of policies.

One core aspect of achieving the goal of REWIRE for enhancing the secure lifecycle managements of

devices is *the ability to trace the achievement of the security requirements associated with the technical components of REWIRE*, towards the achievement of the *overarching requirements, security goals, and specifications of the use cases of REWIRE*. This is achieved through the **Architecture Analysis & Design Language (AADL)-based System Modelling** methodology, which is employed by REWIRE in order to create a virtual representation of the assets comprising the domain infrastructure, their interconnectivity, and the associated asset properties or requirements. Then, the **Resolute** language is used for ratifying and validating the correctness and achievement of the aforementioned requirements through **architectural assurance cases**.

As aforementioned, a core outcome of the design-time phase is the identification of the device properties that need to be verified dynamically during runtime, based on the identified *trust boundary* of the device. In this regard, the **security controls** that need to be deployed for this verification is performed through the definition of **security policies**, dictating the actions (and details regarding their periodicity) that need to be performed. In the context of REWIRE, these policies are expressed using the **Medium Security Policy Language (MSPL)**, which has been identified as possessing the granularity and expressiveness requirements of REWIRE, and deployed through the **Policy Orchestrator** to the intended devices. Chapter 6 is dedicated to the provision of concrete examples of such MSPL-expressed policies in the context of the use cases, specifically the **Smart Cities**, **Automotive**, and **Smart Satellites** use cases.

The **REWIRE Risk Assessment Component**, described in Chapter 7, is a core aspect of REWIRE, whose purpose is to maintain an up-to-date **risk graph** of the entire domain ecosystem, based on the threats and vulnerabilities of the devices participating in the target domain infrastructure. One core target of the RA component is the calculation of the **Required Trust Level (RTL)** of the devices, referring to the *baseline level of trust that needs to be achieved so that they can be considered trustworthy*, based on the identified risks and available mitigation measures. Thus, the RA component can be used in order to guide the operation of a Trust Assessment methodology, which falls outside the scope of REWIRE, but REWIRE aims to provide all required tools and functionalities for guiding its operation.

Finally, Chapter 8 is dedicated to the methodology employed by REWIRE in order to establish **secure and authenticated communication channels** for achieving all overarching security and privacy requirements, considering both **communication within the same domain** and **communication between different domains**. This is achieved through the use of the **Anonymous Authenticated Credential Key Agreement (AACKA)** scheme, based on which REWIRE designs a novel BBS-based AACKA protocol (considering that the credential used in the ZTO process is a BBS signature), resulting in the establishment of secure and anonymous communication channels.

Contents

List of Figures

List of Tables

Versioning and contribution history

Version	Date	Summary of Changes	List of Contributors
v0.1	13.01.2025	Table of Contents	Thanassis Giannetsos (UBITECH), Sjors Van den Elzen (SECURA)
v0.15	26.03.2025	Definition of security properties and threat model of Compositional Verification and Validation component (Chapter 3)	Stylianos Basagiannis, Simone Fulvio Rollini (UTRCI)
v0.2	11.04.2025	Design of masked gadget and application to ASCON in REWIRE SW Update process with Authenticated Encryption (Chapter 2)	Corentin Verhamme, Francois Koeune (UCL)
v0.25	23.04.2025	Description of SW/FW Vulnerability Analysis component (Chapter 4)	Sjors Van den Elzen (SECURA)
v0.3	29.04.2025	Description of process for the establishment of an authenticated channel (Chapter 8)	Samira Briongos (NEC)
v0.35	08.05.2025	Description of the second release of the Risk Assessment component and instantiation in automotive use case (Chapter 7)	Dimitris Karras, Nikos Fotos, Thanasis Giannetsos (UBITECH)
v0.4	13.05.2025	Description of the AADL-based modelling and requirements traceability (Chapter 5)	Stylianos Basagiannis, Simone Fulvio Rollini (UTRCI)
v0.45	21.05.2025	Description of REWIRE policy orchestrator and device lifecycle management (Chapter 6)	Stylianos Basagiannis, Simone Fulvio Rollini (UTRCI), Dimitris Karras, Thanasis Giannetsos (UBITECH)
v0.5	27.05.2025	Performance evaluation of masked gadget and countermeasures (Chapter 2)	Corentin Verhamme, Francois Koeune (UCL)
v0.55	04.06.2025	Modelling and requirements traceability for Automotive use case (Chapter 5)	Stylianos Basagiannis, Simone Fulvio Rollini (UTRCI), Paraskevas Papanikolaou, Charoula Pechilvani, Athanasios Athanasiadis (KENOTOM)
v0.6	12.06.2025	Modelling and requirements traceability for Smart Cities use case (Chapter 5)	Stylianos Basagiannis, Simone Fulvio Rollini (UTRCI), Jesus Sanchez (ODINS)
v0.65	20.06.2025	Examples of MSPL policies in all use cases (Chapter 6)	Stylianos Basagiannis, Simone Fulvio Rollini (UTRCI), Dimitris Karras, Thanasis Giannetsos (UBITECH)
v0.7	01.07.2025	Benchmarking of SW/FW Vulnerability Analysis (Chapter 4)	Sjors Van den Elzen (SECURA)
v0.75	09.07.2025	Compositional Verification and Validation of CIV scheme (Chapter 3)	Stylianos Basagiannis, Simone Fulvio Rollini (UTRCI)
v0.8	15.07.2025	Liqun Chen (SURREY), Annika Wilde (RUB)	Internal review
v0.9	21.07.2025	Refinements based on comments from internal review	Dimitris Karras, Thanasis Giannetsos (UBITECH)
v1.0	29.09.2025	All results documented throughout the deliverable were ratified also against the end-to-end experiments performed in the context of the envisioned use cases (D6.2). This was the reason behind the consortium opting to submit the final version of all deliverables on M36 where all results were available	Dimitris Karras, Thanasis Giannetsos (UBITECH)
v1.0	30/09/2025	Submission of deliverable	Thanassis Charemis (UBITECH)

List of Abbreviations

Abbreviation	Translation
ABE	Attribute-Based Encryption
ABS	Attribute-Based Access
ACC	Assisted Cruise Control
AE	Authenticated Encryption
AIC	Attestation Integrity Verification
AIK	Attestation Identity Key
API	Application Programming Interface
ATL	Actual Trust Level
BBL	Berkeley Boot Loader
BIOS	Basic Input/Output System
CDI	Compound Device Identifier
CIV	Configuration Integrity Verification
CPU	Central Processing Unit
CRL	Certificate Revocation List
CRTM	Core Root of Trust for Measurement
DDA	Direct Anonymous Attestation
DICE	Device Identifier Composition Engine
DID	Decentralised Identity Documents
DMA	Direct Memory Access
EAP	Enhanced Authentication Protocol
ECU	Electronic Control Units
EK	Endorsement Key
EPC	Enclave Page Cache
FPGA	Field Programmable Gate Arrays
FSM	Finite State Machine
HSM	Hardware Security Module
HW	Hardware
IDM	Identity Management
IOMMU	Input/Output Memory Management Unit
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
KMS	Key management system
MUD	Manufacturer Usage Description
OEM	Original Equipment Manufacturer
PCR	Platform Configuration Register
PMP	Physical Memory Protection
PPI	Physical Presence Interface
PSK	Pre-Shared Key
PUF	Physically Unclonable Function

RA	Rich Application
RATS	Remote ATtestation procedureS
REE	Rich Execution Environment
RoT	Root of Trust
RT	Runtime
SBI	Supervisor Binary Interface
SC	Side-Channel
SDLC	Software Development Lifecycle
SGX	Software Guard Extensions
SM	Security Monitor
SoC	System-on-Chip
SP	Service Provider
SRK	Storage Root Key
SUIT	Software Updates for Internet of Things
SSA	Security-Sensitive Application
SSI	Self-Sovereign Identity
SW	Software
TA	Trusted Application
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TDISP	TEE Device Interface Protocol
TEE	Trusted Execution Environment
TMFS	TEE Management Framework Specification
TOCTOU	Time of Check Time of Use
TPM	Trusted Platform Module
TTP	Trusted Third Parties
UDS	Unique Device Secret
VC	Verifiable Credentials
VM	Virtual Machine
VP	Verifiable Presentations
vTPM	Virtual Trusted Platform Module
ZTO	Zero-touch onboarding

Chapter 1

Introduction

1.1 Scope and Purpose

In order to achieve the core target of REWIRE pertaining to the *secure lifecycle management of embedded systems considering the trend towards the increased adoption of OpenHW architectures through the provision of security and trust enablers*, REWIRE provides a holistic framework, which aims to capture the entire operational lifecycle of devices operating in the context of large-scale **Systems-of-Systems (SoS)**. This framework aims to address the key challenge of ensuring the *correctness of all software and hardware artefacts and achieve high security and trustworthiness profiles for embedded systems* by providing a set of **security enablers and trust extensions**. In this regard, a core innovation of REWIRE is to support the **symbiosis of design-time with runtime assurances** as part of a **hybrid methodology**, by demonstrating how these enablers can work in order to fulfil the stringent **security and privacy requirements** characterizing such SoS.

As detailed in D2.2, the operation of the REWIRE framework can be split into two core phases, namely **Design-time** and **Runtime**. This deliverable is dedicated to the *description of the final versions of all technical components participating in the Design-time phase of the REWIRE framework*, which entails the instantiation of the REWIRE framework during the initial phase of its deployment, as well as the expression of the overarching requirements of the system, as provided by the Security Administrator, the Use Case Providers, and the OEM. The end goal of this phase is the definition of the **trust boundary** of the system, i.e., *the set of device properties which can be formally verified during design-time, and for which the System Administrator can have the required guarantees on their correctness*, through the **Compositional Verification and Validation** component (Chapter 3). Note that this component employs the **Verifpal** tool for Formal Verification, which provides the required level of abstraction for verifying the **Elliptic-Curve Cryptography (ECC)**-based **Configuration Integrity Verification (CIV)** scheme, which is a core innovation of REWIRE. In this regard, Verifpal has been identified as providing a high level of flexibility in capturing the underlying complex mathematical functions, compared to other existing Formal Verification solutions. Conversely, the parts of the system that remain outside the trust boundary cannot be formally verified during design-time, and need to be dynamically verified during runtime using the security enablers of REWIRE. To this end, a core output of the Design-time phase is the definition of **security and operational policies** that dictate the actions that need to be taken in order to verify that those runtime properties are correct, based on the expected behaviour of the device. Thus, the Runtime phase receives the outcomes of the Design-time phase, and prepares the device so that it can utilize the secure lifecycle management capabilities of REWIRE, and identify any unexpected behaviours so that the appropriate mitigation measures can be applied.

At the beginning of the Design-time phase, the **Service or Application Provider** and the **Security Administrator** provide information on the assets belonging to the target operational domain and their interconnectivity. In this regard, the **Risk Assessment** component of REWIRE (Chapter 7) is responsible for performing an initial **risk quantification** based on the most prominent types of threats and vulnerabili-

ties affecting these devices, capturing all types of topologies and relationships between assets (SW-SW, HW-HW, and SW-HW relationships). Note that, in the context of REWIRE, we consider interactions and communication between both **devices belonging to the same service graph chain within a domain infrastructure**, or between **different domains**. Thus, REWIRE aims to provide the necessary cryptographic enablers for establishing secure and authenticated channels in both these types of scenarios (Chapter 8).

The REWIRE framework, as aforementioned, aims to provide operational assurance in large-scale SoS consisting of various interconnected devices. For the modelling and representation of such systems, REWIRE employs an **Architecture Analysis & Design Language (AADL)**-based methodology for specifying both SW and HW configurations. Specifically, REWIRE employs rigorous methodologies such as the **Open Source AADL Tool Environment (OSATE)**, which enables engineers to map cybersecurity requirements to system components. This approach assists in the identification of *requirements violations and security measures*, and ensures that each system component aligns with the overarching security goals of the target application domain. In addition, through the **RESOLUTE** extension, it is possible to build **assurance cases** (i.e., formal arguments linking high-level security objectives to evidence on the behaviour of low-level components) for validating and ratifying the achievement of the aforementioned requirements.

Another functionality provided by REWIRE for facilitating the software security analysis of large numbers of devices in such large-scale SoS is the **SW/FW Vulnerability Analysis Component** (Chapter 4), which enables the *partial automation of the analysis of embedded software*, through **partial emulation of the firmware image** and **fuzzing of the emulation**, where semi-random inputs are generated and mutated and the behaviour of the device is monitored. This process is also performed on any *updates or patches to the firmware image of a device* deployed through the **Secure SW/FW Update** process of REWIRE, which needs to be verified for any bugs or vulnerabilities. Note that the Secure SW/FW Update process of REWIRE, as analysed in Chapter 2, is supported by **Authenticated Encryption (AE)** capabilities for mitigating the threat of malicious software updates. Another core consideration of REWIRE in this regard is protection against **Side-Channel Attacks (SCA)**, i.e., the exploitation of unintended information leakage from physical implementations, through the use of **Ascon**, which is a lightweight NIST-approved crypto schmeme designed for providing resilince against key leakage.

Finally, a core output of the Design-time Phase is the definition of actions that need to be taken during runtime for the verification of the correctness of the properties that need to be attested during runtime. This is achieved through the compilation of **security and operational policies** (Chapter 6), which are expressed using the **Medium Security Policy Language (MSPL)** which possesses the required level of expressiveness and granularity considering the security enablers offered by REWIRE. These are deployed through the **Policy Orchestrator**, which acts as a bridge between the device and the outside world, and distributed to the **Facility Layer** of the devices so that the necessary actions can be taken. In this deliverable, we provide concrete examples of such policies deployed in the context of the REWIRE framework for each of the three envisioned use cases (Smart Cities, Automotive, Smart Satellite).

Overall, this deliverable aims to provide a detailed description and documentation of all the components participating in the Design-time phase of the REWIRE framework and their demonstration through concrete examples and evaluation results. This provides the basis for the runtime operation of the REWIRE framework, through the provision of all required information to guide the **trust assessment** of the devices. Note that, while the provision of a Trust Assessment Framework falls outside the scope of REWIRE, we aim to provide all tools and functionalities that enable calculating both the **Required Trust Level (RTL)** and the **Actual Trust Level (ATL)** of the devices, and ensuring the trustworthiness level of the devices through the available security enablers.

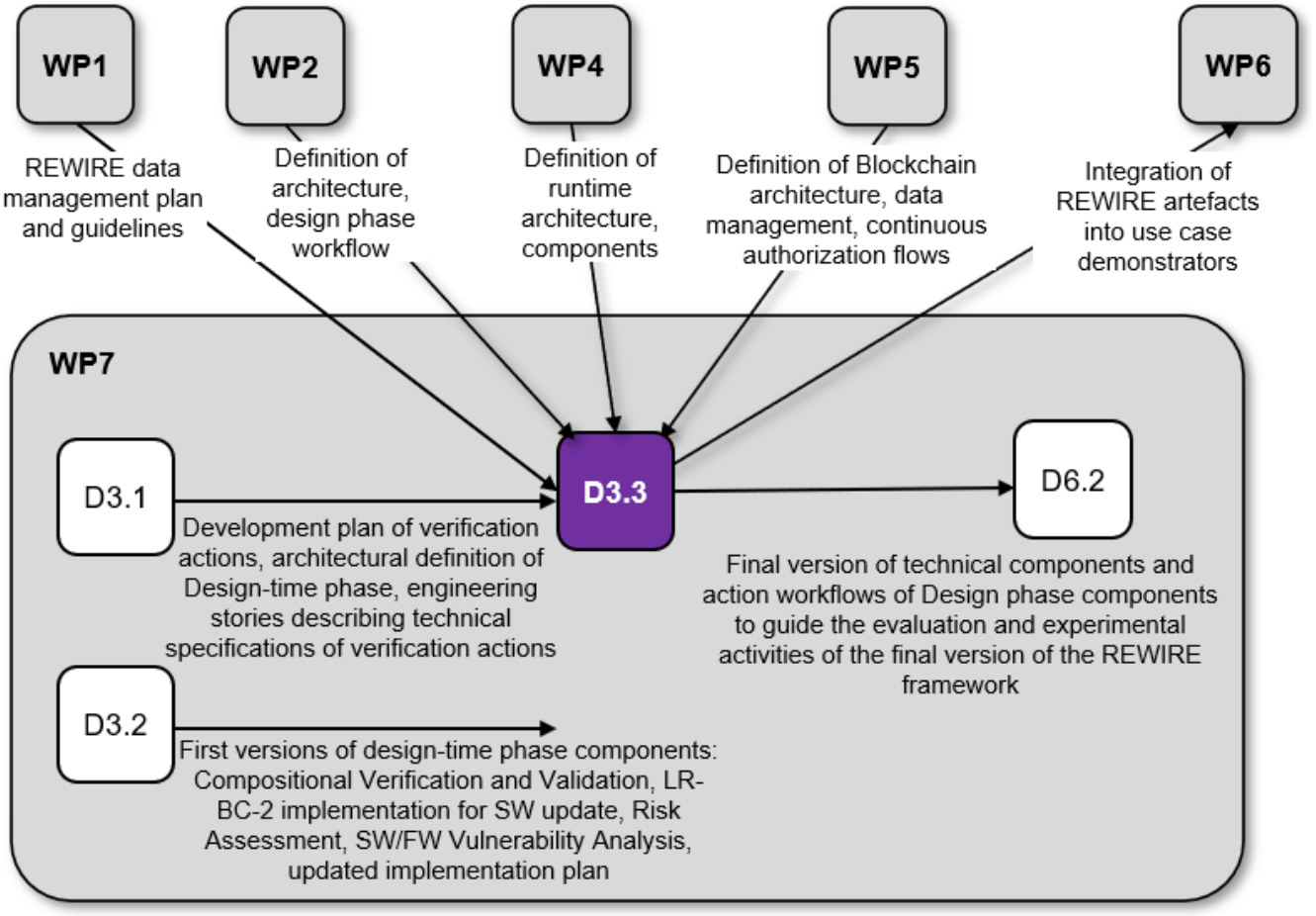


Figure 1.1: Relation of D3.3 with other WPs and Deliverables

1.2 Relation to other WPs and Deliverables

Figure 1.1 depicts the relationships of this deliverable with other Work Packages (WPs), as well as other tasks in the same WP(3). As aforementioned, the main purpose of this deliverable is to provide a detailed description and documentation of all technical components participating in the Design-time phase of the REWIRE architecture, as well as their role within the overall REWIRE workflow. Thus, this deliverable receives input from D3.1 with regards to the development plan, the architectural definition of the Design-time phase, and the engineering stories involving the concerned technical components, as well as D3.2 regarding the initial definitions of all technical components to be expanded upon in this deliverable.

In addition, D3.3 receives input from the reference architecture of REWIRE defined in the context of WP2, and specifically D2.2 which defines the final action workflow of the Design-time phase of the REWIRE framework. These also provide the basis for the operation of the components defined in the context of WP4 regarding the runtime operation of the REWIRE framework, as well as the Blockchain infrastructure defined as part of WP5. Finally, D3.3 provides input to WP6, which is responsible for the integration of all REWIRE components into the overall framework, as well as the experimental activities for the evaluation of the performance of all technical components as part of the envisioned use cases of REWIRE.

1.3 Deliverable Structure

This deliverable is structured as follows:

Chapter 2 is dedicated to the description of the Secure SW Update process with Authenticated Encryption (AE), including the design of a masked gadget against physical faults, and evaluation of primitive-

and model-level countermeasures.

Chapter 3 describes the Compositional Verification and Validation Component of REWIRE, as well as the formal verification of the Configuration Integrity Verification (CIV) process of REWIRE with local attestation.

Chapter 4 describes the SW/FW Vulnerability Analysis component, the updates performed to the Qiling framework, and benchmarking results on the component.

Chapter 5 is focused on the Architecture Analysis & Design Language (AADL)-based System Modelling methodology of REWIRE for specifying software and hardware configuration, as well as its use for the traceability of the achievement of requirements of the technical components of REWIRE.

Chapter 6 describes the definition of REWIRE policies expressed in the Medium Security Policy Language (MSPL), and provides examples in the context of all envisioned REWIRE use cases.

Chapter 7 describes the Risk Assessment component of REWIRE, with particular focus on the instantiation in the Automotive use case, as well as the process for the calculation of the Required Trust Level (RTL) of the device.

Chapter 8 describes the process for the establishment of secure authenticated channels in the context of REWIRE, inspired by the Anonymous Authenticated Credential Key Agreement (AACKA) and leveraging BBS signatures.

Chapter 9 concludes the deliverable.

Chapter 2

REWIRE Secure SW Update Process with Authenticated Encryption

In the type of modern large-scale Systems-of-Systems (SoS) and IoT environments considered in the context of REWIRE, enhancing the recovery and resilience of systems through software and firmware updates is becoming increasingly crucial, as threats of attackers aiming to substitute legitimate software and firmware with their malicious code are becoming increasingly prevalent in such ecosystems. Thus, *it cannot be assumed that the software and firmware of embedded systems will remain uncompromised*. To this end, the **Secure SW/FW Update process with Authenticated Encryption (AE)** of REWIRE aims to provide a mitigation strategy against such types of attacks. Note that, as outlined in D3.2, REWIRE offers two modalities for the SW Update process, specifically:

- **"1-to-1"** for the distribution of an update to a single device. In this case, as described in D3.2, the SW Service Provider signs and encrypts the update using an authenticated encryption (AE) algorithm, using a pre-established LRBC Key which is stored in the ASIC-based HW of the device. This has been implemented in the form of an enclave in order to ensure the protection of the key.
- **"1-to-many"** for the distribution of an update from a single source to multiple recipients. In this case, the SW Service Provider signs the update with their own key, and all concerned devices are notified about the update through the Blockchain Infrastructure.

In both cases, a Configuration Integrity Verification (CIV) process (i.e., the attestation enabler for verifying the correctness of the device configuration based on its expected state) is performed both before the installation of the update in order to verify that there are no integrity issues prior to its deployment, and after the update in order to verify that the update has been performed correctly. In the latter case, attestation reports are created for each device an update has been performed on. Then, an attestation report is created, authenticated, and encrypted with the LRBC Key, so that the SW Service Provider can verify the correctness of the update (and that the update has been installed on the intended devices, in the "1-to-many" case). Throughout this chapter, we provide a detailed description of the SW Update Process of REWIRE, where AE is used for ensuring the secure communication of SW updates, while ensuring the protection of the process against the types of attacks that may target the update process, as aforementioned.

As outlined in D3.2, the aim of REWIRE is to offer a general framework, following the relevant best practices, that is able to ensure the fulfilment of the needs and requirements of each target embedded system and environment. To this end, **REWIRE is aligned and shares a common vision with the Trusted Computing Group (TCG)** with regards to the best practices regarding the secure software and firmware updates of embedded systems [72]. Based on the guidance provided by the TCG, a set of requirements have been identified with regards to the design and implementation of a SW/FW Update system for securing the lifecycle management of the aforementioned types of environments. Here, we provide a summary of these requirements, as well as the actions taken by REWIRE for their fulfilment:

- **Secure Development:** This refers to the need to conduct a comprehensive threat analysis during the design phase of a system, in order to ensure that the available security measures are able to address the evolving threat landscape affecting the system. According to the TCG, additional practices for higher security levels are required, including automated testing (e.g., fuzzing) for the identification of bugs or security vulnerabilities, and static analysis of binary or source code to identify bugs or poor coding practices. This has been integrated into the Design-time Phase of REWIRE, where system designers and security analysts define the properties that need to be validated for fulfilling the overarching requirements. In addition, the SW/FW Validation Component of REWIRE is responsible for performing the aforementioned automated testing and static analysis.

- **Secure Update Signing:** This refers to the need to sign a software update in order to verify its origin and integrity before sending it, with a process referred to as *code signing*. The code signing process has been notably targeted by malicious parties, using attacks that aim to distribute malicious code with a valid signature, either by exploiting flaws in the validation of the signature, or by obtaining the actual signing key, resulting in the distribution of malicious code without being noticed. To this end, the TCG proposes methods such as the use of reliable, well-vetted cryptographic algorithms and tools, the use of cryptographic algorithms with high agility, and the use of Hardware Security Modules (HSMs) for cryptographic key management. These are all offered in the context of REWIRE, as well as enhancements such as the use of TEEs for performing isolated and secure execution of the signature validation, and rollback protection.

- **Robust Distribution and Authentication:** After a SW update has been signed, it needs to be distributed to the intended devices, even in cases where the distribution of large files to numerous devices in a scalable manner is needed, or in cases where the distribution network is sporadic and may lead to fragmented data transmissions. In fact, the latter is a core concern for the Smart Satellites use case of REWIRE, where satellites in orbit receives update in data chunks based on the visibility they have with ground stations while moving around the Earth. In this regard, the TCG recommends a set of best practices, including securing communications with thoroughly vetted security protocols, **establishing the identity and trustworthiness of the distribution service**, **designing SW update distribution mechanisms in a manner that avoids overloading networks or servers**, **placing the communication security keys in an HSM**, **authenticating endoints to determine which updates they are authorised to receive**, **tracking installation of updates to ensure that endpoints have been updated**, and **alerting administrators if updates cannot be installed on some endpoints**. REWIRE offers all aforementioned measures, and applies Formal Verification methods in order to guarantee the correctness of the *LR-BC-2 AE* scheme, which is used in order to secure the communication channel for the distribution of SW updates. The use of this scheme ensures authenticity, thus ensuring that the identity of the distribution service is correct, in both modes of operation (“1-to-1” and “1-to-many”). In addition, the Automotive use case of REWIRE also requires confidentiality against external observers, which is provided by the AE used in the “1-to-1” scenario. Finally, REWIRE enables devices to transmit the outcome of the update process in an authenticated manner using AE.

- **Secure Update Installation:** The installation of a SW/FW Update involves various complex operational and security considerations, such as minimising downtime, which is an especially critical concern in embedded devices used for safety-critical processes. In addition, it is possible that a SW/FW update may fail to complete successfully due to various reasons (e.g., power outages, inability to access external resources). Thus, according to the TCG, devices should have the capability to recover from such failures, and precautions need to be taken in order to protect the devices against these incidents (which may also facilitate the execution of attacks against the SW update process). In the context of REWIRE, the use of LR-BC-2 provides the capability to verify the updates before installation by verifying the source and ensuring that the update originates from a trusted update signer. In addition, through the Keystone TEE, REWIRE offers a recovery process

for detecting and recovering from failed or malicious updates, and guarantees that the recovery process cannot be used by a malicious party to roll back to a vulnerable version of the firmware.

- **Authenticated Post-Update Verification:** As stated by the TCG, SW/FW update systems must verify whether an update has been successfully performed, and if the device is operational after the update. To this end, a verification process is needed in order to test the integrity of specific functionalities. For instance, different stakeholders (e.g. administrators, manufacturers, or external entities) may need to monitor the version and integrity of the SW/FW installed on a device. In the context of REWIRE, this is offered through novel attestation schemes, which leverage the capabilities of Keystone in order to securely verify if the correct SW/FW update is running or if malicious or unknown SW/FW versions are detected. Specifically, this is performed by using *local verifiable key restriction usage policies*, and failure to confirm the correct state of the device may trigger a recovery mode. In addition, the SW/FW update process of REWIRE guarantees that a device is able to transmit the outcome of the update process back to the update distribution service, which can be used for notifying the administrator about a successful or failed update process.

A core offering of REWIRE, which is also a common denominator for the fulfilment of the aforementioned requirements, is the consideration of **Side-Channel Attacks (SCA)**, which are types of attacks that aim to exploit unintended information leakage from physical implementations of cryptosystems, rather than weaknesses in the underlying protocols themselves. These may include *timing information*, *power consumption*, *electromagnetic emissions*, or *device imperfections*.

While it falls outside the scope of REWIRE, in the past years, rapid advancements have been documented in the field of quantum computing, which has led to an increased effort towards the development and continuing transition towards **Post-Quantum (PQ) cryptography**. This is in line with the vision of the EU towards the widespread adoption of quantum-resistant infrastructures by the 2030s [70], as a response to the recent advancements in the domain of quantum computing, which may render traditional cryptographic protocols insecure. However, one core consideration in this regard is that, while PQ crypto schemes offer theoretically unbreakable security, real-world implementations can still be vulnerable to SCA, which may be employed in order to extract secret keys from such systems, thus uncovering a critical gap between theoretical and practical security. Thus, *the work performed by REWIRE can provide the basis for countermeasures against SCA whose integration into PQ schemes can be explored, and can provide the basis for the elevation of well-established crypto algorithms (e.g., AES), and elevating them to algorithms that can protect against SCA*. In this regard, REWIRE provides an important contribution to the literature which can be significantly expanded in terms of future PQ-related research.

2.1 High-level Summary of the State-of-the-Art on Side-Channel Attacks

Protecting cryptographic implementations against Side-Channel Attacks (SCAs) is a critical requirement for ensuring the overall security of modern embedded and hardware-based systems. SCAs exploit physical leakages—such as variations in power consumption, electromagnetic emissions, or timing behavior—to extract secret information during execution. Among the most effective countermeasures, masking techniques split sensitive variables into multiple randomized shares, enabling computations that maintain security even when an adversary can observe a limited number of internal states.

Overall, two complementary approaches are possible to enforce side-channel resistance: **primitive-level** and **mode-level** countermeasures. Primitive-level techniques, such as masking, target the low-level operations within cryptographic primitives (e.g., S-boxes or multipliers) and are well studied, with a strong theoretical foundation and composable security guarantees. In contrast, mode-level countermeasures, including techniques like levelling and leakage-resilient pseudorandom functions (LR-PRFs), operate at a

higher abstraction layer and aim to tolerate leakage across entire cryptographic modes or constructions. While promising, mode-level approaches still require further investigation to reach the same maturity and adoption as primitive-level methods. In a nutshell, mode-level countermeasure is at the core of the first approach we developed for REWIRE, leading to the LR-BC-2 construction detailed in D3.2 and benchmarked in D6.1. In this deliverable, we will explore a primitive-level countermeasure leveraging ASCON, which was selected by NIST as a result of the Lightweight Cryptography Standardization Process for meeting the needs of most use cases where lightweight cryptography is required. We will then provide in section 2.4 a focused discussion and comparison between these two approaches, highlighting their respective advantages, limitations, and potential for integration. This analysis gave birth to two scientific publications in the processs of REWIRE [25, 75], and the next sections are based on these papers.

2.2 Design of a Masked Gadget Robust against Physical Defaults

In the context of formal security models, such as the probing adversary model, researchers have proposed structured masking strategies to design cryptographic components—known as gadgets—that are provably resistant to a bounded number of probes. These gadgets ensure that intermediate computations do not leak secret information, even under active observation. This deliverable focuses on a family of such gadgets developed under the **Hardware Protected Circuits (HPC)** framework, emphasizing **composability**, **efficiency**, and **robustness**.

These constructions offer a scalable pathway to secure, high-order masked implementations, particularly in hardware contexts where physical leakage is unavoidable and performance constraints are stringent. Recall that, in this context, **masking** is a countermeasure against SCA based on splitting sensitive variables into multiple *shares* using random values (masks) so that intermediate values processed by the device do not directly correlate with secret data. In this regard, **first-order masking** entails splitting the sensitive value into two shares (i.e., the mask and the masked value), while **higher-order masking** generalises this idea by splitting a secret into $d+1$ shares, where d is the *order of masking*. It is desirable for the masking order to serve as a security parameter, whereby the security level increases exponentially as d grows linearly

The property of composition is of great interest for hardware components, as *it provides the capability to build hardware systems from smaller, verified subcomponents, in such a manner that individual properties (specifically, side-channel resistance) are preserved when composed*. Since in general hardware verification can be extremely resource-intensive in terms of time and computational effort, composition is an approach that enables verifying smaller components independently, and ensuring that combining them does not introduce new errors or vulnerabilities, thus allowing modular verification. In addition, composition alleviates the need for heuristics (e.g., testing, simulation, rule-of-thumb design practices) which may risk undermining critical properties, including side-channel resistance.

Another advantage of the method is the possibility to consider physical defaults. By taking into account those at the mathematical level, we can hope to tackle the issue at its root. Those advantages come with potential overheads if the model reveals itself to be overly conservative. Considering also the need to provide operational assurance to devices that may be computationally limited in the context of REWIRE, the quest of *low-latency* cryptographic primitives is recent, yet a significant amount of ciphers have been published. Among those, *Ascon* attracted our interest as the winner of the NIST lightweight cryptographic competition. We will also demonstrate an interesting application to the post-quantum world.

2.2.1 The challenge of composability

Composability (informal) refers to the approach based on the use of individual masked gadgets and their composition into a larger function. However, a key challenge in this regard is that, if the individual gadgets are secure on their own, then composing them into a secure overall implementation should not

compromise the security of the overall system. Thus, while composability offers multiple advantages, such as scalability, efficiency, and robustness, there need to be the appropriate security guarantees for ensuring that the no additional vulnerabilities are introduced when composing the individual gadgets.

In the context of protection against SCA, composability is a critical property in the design of masked cryptographic gadgets, particularly for ensuring robustness against side-channel attacks such as Differential Power Analysis (DPA). A composable masking scheme allows individual secure components (gadgets) to be combined without compromising the overall system's side-channel resistance. To achieve composability, each gadget must adhere to a set of rigorous security definitions. These conditions ensure that the leakage from combined masked components does not introduce exploitable statistical dependencies. Recent research has shown that without proper composability guarantees, even theoretically secure masked operations can leak sensitive information when integrated into a larger system. Consequently, designing composable masked gadgets is essential for constructing secure cryptographic implementations that scale correctly and maintain side-channel resistance in complex, real-world applications.

We will now present a formal approach to composition. We denote a random variable by lowercase letter : $x \in \mathbb{F}_2$ denotes a binary random variable. In the particular case of masking we will denote by x_i the i-th share of the x variable such that $\bigoplus_{\forall i} x_i = x, i \in \{0, \ldots, d-1\}$ where d denotes the number of shares. Besides, drawing a value uniformly at random from a set is denoted as $x \xleftarrow{\$} S$.

2.2.1.1 Circuit definition

The circuit (see Definition 2.2.1.1) model we use originates from [22]. Specifically, while we preserve the formalism provided therein, we restrain some of the notions to fit our needs that are limited to a single-cycle pipelined gadgets. This model first designates physical circuits as sets of gates and wires:

Definition 1 (Structural gate)**.** A structural gate is a tuple (I, P, f, lat), where

- I is the set of inputs of the gate,
- P is the set of parameters and the union of public and private parameters P^P, P^S,
- $f : (I \to \mathbb{F}_q) \times (P \to S) \to \mathbb{F}_q$ (where S is any set) is the evaluation function,
- $\mathsf{lat} : I \to \mathbb{N}$ is the latency of the inputs (i.e., one if the input is required one clock cycle before the output is produced, zero if it is required at the same cycle).

Definition 2 (Structural wire)**.** A structural wire is a pair $(g, (g', i))$ which connects a (source) gate g to the input i of a (destination) gate g', denoted as (g', i).

Merging these two notions, one can define structural circuits:

Definition 3 (Structural circuit)**.** A structural circuit is a directed graph whose nodes are structural gates and whose edges are structural wires. A wire connects its source to its destination. In a structural circuit, there must be no combinational loop (i.e., there must be no cycle for which all the wires have a destination with latency 0).

We then need to define the execution of a structural circuit:

Definition 4 (Circuit execution)**.** The execution of a structural circuit $C = (G, W)$ for the set of cycles $T = \{t_0, \ldots, t_{L-1}\}$ is a directed graph whose set of nodes is $G \times T$ (the gates) and whose set of edges is $W \times T$ (the wires). Wires connect gates according to their latency: let $w = (g, (g', i))$ be a structural wire and (g', t) a gate, the wire (w, t) connects its source $(g, t - \mathsf{lat}_{g'}(i))$ to its destination (g', t), where $\mathsf{lat}_{g'}$ is the latency function of the gate g'. If the source does not exist, then the wire is connected to a fresh "initial state" source gate (i.e., a no-input gate having as output a public parameter). Each gate may be annotated with a parametrization function, mapping the set of all (resp., public) parameters of the underlying structural gate to values, in which case the execution is full (resp., partial).

Now that objects are formally defined, we next introduce a security notion over them.

2.2.1.2 Probing model and robust probing model

We will use the following two definitions in our contributions.

Definition 5 (Probing security [47])**.** A set of probes P in a gadget execution G is a subset of its gates and input wires. In an evaluation of G with input values x, the values of the probes are denoted as $G_P(x)$. The sensitive values are defined as the sums of the values on the wires of each input sharing. A gadget execution G is secure against a set of probes P if $G_P(x)$ is independent of the sensitive values when the inputs x are uniformly distributed. G is t-probing secure if it is secure against any P such that $|P| \leq t$.

A more generic model exists that models physical defaults thanks to extended probes:

Definition 6 (Robust probing security [37])**.** A probe expansion scheme is a function that maps a gadget execution to a larger set of extended probes, which are sets of probes in the gadget execution. The probes in those sets are named expanded probes. A gadget execution is t-robust probing secure with respect to a probe expansion scheme if it is secure against all the expanded probes from any set of t extended probes.

Here, we will examine glitch- and transition-extended probes. Informally, glitch-extended probes turn any probe on combinatorial wire into all its (registered) inputs (for registered input wires, there is no extension); and transition-extended probes turn any probe on a register into a pair of values that correspond to consecutive invocations. When combining both, the set of extended probes is obtained by first computing the set transition-extended probes and then replacing every expanded probe in a transition-extended probe with all the expanded probes contained in the glitch-extended probe that correspond to it. Physically, this models the fact that the propagation of a glitch (e.g., its timing) depends on both the new and the previous values on the wire.

2.2.1.3 Circuit composition

Proving the robustness to probing becomes computationally hard when the size of the circuit to verify and the number of shares increase [6, 7, 16, 51]. A standard approach to enable the more efficient verification of complex circuits is to require the gadgets to ensure some (stronger) composability properties [**?**].
We first introduce the simulatability framework that we will use in our results:

Definition 7 (Simulatability [9])**.** A set of probes P in a gadget execution G can be simulated by a set of input shares $I = (i_1, j_1), \ldots, (i_k, j_k)$ if there exists a randomized simulator algorithm S such that the distributions $G_P(x_{*,*})$ (the values of the probes) and $S(x_{i_1,j_1}, \ldots, x_{i_k,j_k})$ are equal for any value of the inputs $x_{*,*}$.

Definition 8 (Glitch+Transition robust Simulatability [22])**.** A set of extended probes P in a gadget execution G can be simulated by a set of input shares $I = (i_1, j_1), \ldots, (i_k, j_k)$ if there exists a randomized simulator algorithm S such that the distributions $G_P(x_{*,*})$ and $S(x_{i_1,j_1}, \ldots, x_{i_k,j_k})$ are equal for any value of the inputs $x_{*,*}$.

Next, the PINI property is used to capture trivial composition:

Definition 9 (Probe-Isolating Non-Interference [21])**.** Given a gadget execution G, let I be a set of at most t_1 probes on its internal wires and O a set of probes on its output shares. Let A be the set of the share indexes of the shares in O, and $t_2 = |A|$. Let I and O be chosen such that $t_1 + t_2 \leq t$. The gadget G is t-PINI iff for all I and O there exists a set of at most t_1 share indexes B such that observations corresponding to I and O can be simulated using only the shares with indexes $A \cup B$ of each input sharing.

Eventually, the O-PINI property is used to capture trivial composition with glitches and transitions. It can be viewed as a strengthening of PINI, where the simulator must simulate the outputs with same share index as the one on which an input share index is probed:

Scheme	Security w/ glitches	Latency	Randomness
HPC2	PINI	2	$\frac{d(d-1)}{2}$
O-PINI1	O-PINI	2	$\frac{d(d-1)}{2} + d - 1$
HPC3	PINI	1	$d(d-1)$
HPC3+	O-PINI	2	$d^2 - 1$
HPC4	O-PINI	1	$\frac{5d(d-1)}{2}$

Definition 10 (Output Probe-Isolating Non-Interference [22])**.** Given a gadget execution G, let I be a set of at most t_1 probes on its internal wires and O a set of probes on its output shares. Let A be the set of the share indexes of the shares in O, and $t_2 = |A|$. Let I and O be chosen such that $t_1 + t_2 \leq t$. The gadget G is t-O-PINI iff for all I and O there exists a set of at most t_1 share indexes B such that observations corresponding to I, O and output shares with index in B can be simulated using only the shares with indexes $A \cup B$ of each input sharing. We say a gadget is O-PINI if it is t-O-PINI for any t.

It has been proven in that such gadgets can be trivially composed:

Theorem 1 (Trivial composition of O-PINI gadgets [22])**.** *Let S_i be a set of pipeline t-O-PINI structural gadgets, and let G_i be their executions. If the structural gadget S is a composition of S_i gadgets, then it is t-O-PINI.*

2.2.1.4 Hardware Private Circuit : composable and physically robust gadgets

We propose in Table **??** an overview of some selected gadgets that satisfy part of the presented composability properties. Along with the corresponding security feature we add costs metrics in the form of latency and randomness usage. We also already include our novel proposition, the HPC4 gadget that is suitable for implementations of Ascon.

2.2.2 Challenge of Physical Defaults in Composable Gadgets

Let us first recall why the HPC2 and HPC3 gadgets are not O-PINI, which will give some motivation for the design choices of HPC4.

For HPC2, and looking at Algorithm 2.2.2, a probe on $u_{ij} = (x_i \oplus 1) \otimes r_{ij}$requires knowledge of x_i and r_{ij} to be simulated.[1] Therefore, following the O-PINI definition, z_i should be simulated which, due to glitches, means that the value $x_i \otimes (y_j \oplus r_{ij})$ should be simulated. This, in turn, means that $y_j \oplus r_{ij}$ must be known to the simulator, which, combined with the simulation of r_{ij} means that y_j must be known as well. This contradicts the definition of O-PINI: it should be possible to simulate one probe using only one input share.

For HPC3, the issue is similar: if there is a probe in the computation of v_{ij}, then x_i, r_{ij} and r'_{ij} are observed. Therefore, z_i must be simulated. With glitches, it leads to the same requirement on the simulation of $y_j \oplus r_{ij}$.

From these examples, we can see that the core issue comes from (*i*) y_j being masked by a single random value, which (*ii*) appears in a computation with x_i, and that (*iii*) glitches on the output shares z_i leak a lot of information about the intermediate computations in the gadget. We may try to fix the issue by removing any of these conditions. Eliminating (*iii*) is the approach of existing O-PINI gadgets: they essentially refresh the shares z_i, but this has a latency cost. We therefore look at the other solutions: eliminating (*ii*) appears difficult while preserving the correctness of the gagdet. By contrast, for (*i*), we mask y_j using two independent random values, which appear each in a separate computation with x_i. Hence, probing one of these computations does not immediately reveal y_j.

[1] When discussing the values of wires, registers are irrelevant and therefore omitted for readability.

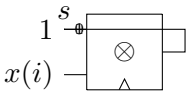


Figure 2.1: Iterative AND architecture.

Figure 2.2: Acquired traces and first-order fixed vs. random t-test over time (10M traces). Left: iterative design of Figure 2.1 instantiated with HPC3 gadgets. Middle: same design instantiated with HPC4 gadgets. Right: same as left, but keeping $s = 0$.

Let us note that, compared to HPC3, we remove the NOT gate on x_i (which can also be done for HPC3 without impacting its security [24]). The values r''_{ij} and r'''_{ij} are sampled uniformly at random from $\mathbb{F}_2$ and are symmetric: $r''_{ij} = r''_{ji}$, $r'''_{ij} = r'''_{ji}$, which is needed to ensure the correctness of the gadget. The values r_{ij} and r'_{ij} are also sampled uniformly at random from $\mathbb{F}_2$, but r_{ij} and r_{ji} are independent, while $r'_{ij} = r'_{ji}$ (these two values could be independent, but this would lead to a higher randomness usage). The motivation for having independent r_{ij} and r_{ji} comes from a probe on $u_{ji} = y_i \oplus r_{ji} \oplus r''_{ji}$. Indeed, simulating such a probe requires knowledge of y_i, which means that z_i, and therefore $\mathsf{Reg}\left[y_j \oplus r_{ij} \oplus r''_{ij}\right]$ must be simulated. If the random values in u_{ji} and u_{ij} were equal, then simulation would require knowledge of y_j, and the gadget would not be O-PINI.

[tb]
Sharings $\mathbf{x}$, $\mathbf{y}$ Sharing $\mathbf{z}$ such that $z = x \cdot y$.

$i = 0$ to $d-1$ $j = i+1$ to $d-1$ $r_{ij} = r_{ji} \xleftarrow{\$} \mathbb{F}_2$ $r'_{ij} = r'_{ji} \xleftarrow{\$} \mathbb{F}_2$ $i = 0$ to $d-1$ $j = 0$ to $d-1, j \neq i$ $u_{ij} \leftarrow \mathsf{Reg}\left[y_j \oplus r_{ij}\right]$ $v_{ij} \leftarrow \mathsf{Reg}\left[(x_i \oplus 1) \otimes r_{ij} \oplus r'_{ij}\right]$ $w_{ij} \leftarrow \mathsf{Reg}\left[x_i\right] \otimes u_{ij} \oplus v_{ij}$ $i = 0$ to $d-1$ $z_i \leftarrow \mathsf{Reg}\left[x_i \otimes y_i\right] \oplus \bigoplus_{j=0, j\neq i}^{d-1}(w_{ij})$

[tb]
Sharings $\mathbf{x}$, $\mathbf{y}$ Sharing $\mathbf{z}$ such that $z = x \cdot y$.

$i = 0$ to $d-1$ $j = i+1$ to $d-1$ $r_{ij} = r_{ji} \xleftarrow{\$} \mathbb{F}_2$ $i = 0$ to $d-1$ $j = 0$ to $d-1, j \neq i$ $u_{ij} \leftarrow (x_i \oplus 1) \otimes \mathsf{Reg}\left[r_{ij}\right]$ $v_{ij} \leftarrow y_j \oplus r_{ij}$ $i = 0$ to $d-1$ $z_i \leftarrow \mathsf{Reg}\left[x_i \otimes \mathsf{Reg}\left[y_i\right]\right] \oplus \bigoplus_{j=0, j\neq i}^{d-1}(\mathsf{Reg}\left[u_{ij}\right] \oplus \mathsf{Reg}\left[x_i \otimes \mathsf{Reg}\left[v_{ij}\right]\right])$

2.2.3 Solving the Issue even for Low-Latency

In addition to the theoretical discussion, we propose an experiment highlighting the potential issue. We make a first step towards (i) empirically confirming the practical relevance of the glitch- and transition-robust probing model, and (ii) demonstrating that HPC4 gadgets indeed offer stronger side-channel security guarantees than HPC3 ones in the context of non-pipeline circuits. For this purpose, and similarly to [55], which also analyzed the glitch- and transition-robust probing model, we implement the circuit of Figure 2.1 with HPC3 and HPC4 AND gadgets at the first security order.
We performed a Fix-versus-Random Test Vector Leakage Assessment (TVLA) [41]. The evaluation was carried out on the Spartan-6 FPGA of a SAKURA-G board, measuring the supply current with a Tektronix CT1 probe, a Picoscope 6000E with a 500MS/s sampling rate (12-bit resolution), and a target clock frequency of 6MHz.

We considered three case studies. The first one is an implementation of the AND reduction using the iterative design of Figure 2.1 instantiated with an HPC3 gadget. The second one is the same design instantiated with an HPC4 gadget. The third one uses the same circuit as the first one, but keeps $s = 0$ so that the mux is controlled by a stable value and the output of the AND gadget is never fed back to its input. This makes the circuit effectively run as a pipeline (hence removing transitions between dependent variables).

The results are shown in Figure 2.2, leading to two main conclusions. First, the integration of HPC3 gadgets in the iterative design indeed leads to a first-order flaw. Second, this flaw vanishes if either the HPC3 gadget is replaced by a HPC4 gadget or the design is run as a pipeline. The empirical evaluations are therefore in line with the theoretical results obtained within the glitch- and transition-extended robust probing model. Despite the general difficulty to interpret the origin of physical leakage with such preliminary tests, they at least do not suggest that the model is overly conservative from a qualitative viewpoint.

HPC4 is described in Alg. item 2.2.3, for a finite field $\mathbb{F}_q$. We now prove that it is O-PINI.

Proposition 1. *The gadget HPC4 is a pipeline and glitch-robust t-O-PINI for $t = d - 1$.*

Proof. First of all, we trivially observe that HPC4 is a pipeline. Next, we assume without loss of generality that the extended probes are placed at the input of registers or on output shares, since the set of wires observed by any other extended probe is always a subset of the one observed by one of these probes. For the sake of simplicity, we also ignore the presence of the register $\mathsf{Reg}\,[x_i y_i]$, which is added only for synchronization, and is not needed for security. We therefore obtain for any i and for all $j \neq i$ the following probes. The probe z_i yields $\{x_i, y_i, x_i\mathsf{Reg}\left[y_j + r_{ij} + r'_{ij}\right], x_i r_{ij} + r''_{ij}, x_i r'_{ij} + r'''_{ij}\}$; the probe u_{ij} provides $\{y_j, r_{ij}, r'_{ij}\}$; the probe v_{ij} reveals $\{x_i, r_{ij}, r''_{ij}\}$; and the probe w_{ij} gives $\{x_i, r'_{ij}, r'''_{ij}\}$.

Given a set of extended probes P on intermediates values and probed output shares A, the set of required additional input shares' indices B is computed as follows:

1. Initialize $X \leftarrow \emptyset$.
2. For each probed output z_i in A, add i to X.
3. For every pair $i < j$:
   (a) If at least two of v_{ij}, w_{ij}, u_{ij}, v_{ji}, w_{ji}, u_{ji} belong to P, add i and j to X,
   (b) Else, if i (resp., j) already belongs to X, and one of the above probes belongs to P, add j (resp., i) to X,
   (c) Else, if u_{ji}, v_{ij} or w_{ij} belongs to P, add i to X,
   (d) Else, if u_{ij}, v_{ji} or w_{ji} belongs to P, add j to X.
4. Let $B = X \setminus A$.

By construction we observe that $|B| \leq |P|$.

We now build a simulator that uses the input shares with index in $X = A \cup B$ to simulate all the required values. First, we observe that by construction of X, any probe v_{ij}, w_{ij} or u_{ij} in P yields the knowledge of the input share needed to compute it, and can therefore be perfectly simulated by following Alg. item 2.2.3. Next, we simulate all the probes z_i for all $i \in X$. For every such z_i, since x_i and y_i are known to the simulator, it remains to simulate $\mathsf{Reg}\,[u_{ij}]$ for all $j \neq i, j \notin X$ (for the other values, this is trivially done by following Alg. item 2.2.3), which we can do with fresh randomness.

Only the last step of simulating $\mathsf{Reg}\,[u_{ij}]$ for $i \in X$, $j \notin X$ is not trivially correct. By construction of X, we know that for such $\mathsf{Reg}\,[u_{ij}]$, at most one of v_{ij}, w_{ij} and u_{ji} is probed, while u_{ij}, v_{ji} and w_{ji} are not probed. If v_{ij} (resp., w_{ij}) is probed, then, thanks to the fresh random r'''_{ij} (resp., r''_{ij}), the random r'_{ij} (resp., r_{ij}) is independent of all the observations of the adversary except $\mathsf{Reg}\,[u_{ij}]$, therefore $\mathsf{Reg}\,[u_{ij}]$ appears as a fresh random. If u_{ji} if probed, then r_{ij} is not observed except through $\mathsf{Reg}\,[u_{ij}]$. □

shares $(x_i)_{0\leq i\leq d-1}$ and $(y_i)_{0\leq i\leq d-1}$, such that $\sum_i x_i = x$ and $\sum_i y_i = y$. shares $(z_i)_{0\leq i\leq d-1}$, such that $\sum_i z_i = xy$. $i = 0$ to $d-1$ $j = i+1$ to $d-1$ $r_{ij} \xleftarrow{\$} \mathbb{F}_q$ $r_{ji} \xleftarrow{\$} \mathbb{F}_q$ $r'_{ij} = r'_{ji} \xleftarrow{\$} \mathbb{F}_q$ $r''_{ij} = -r''_{ji} \xleftarrow{\$} \mathbb{F}_q$ $r'''_{ij} = -r'''_{ji} \xleftarrow{\$} \mathbb{F}_q$ $i = 0$ to $d-1$ $j = 0$ to $d-1$, $j \neq i$ $u_{ij} \leftarrow y_j + r_{ij} + r'_{ij}$ $v_{ij} \leftarrow x_i r_{ij} + r''_{ij}$ $w_{ij} \leftarrow x_i r'_{ij} + r'''_{ij}$ $i = 0$ to $d-1$ $z_i \leftarrow \mathsf{Reg}\,[x_i y_i] + \sum_{j=0, j\neq i}^{d-1} (\mathsf{Reg}\,[x_i]\,\mathsf{Reg}\,[u_{ij}] - \mathsf{Reg}\,[v_{ij}] - \mathsf{Reg}\,[w_{ij}])$

d	Primitive	Area (kGE)	Latency (in cycle to process the amount of bits)		
			576	2034	5760
2	SHA3-512	207.7	48	192	480
	Ketje-SR	50.3	39	85	201
	Ascon	45.8	78	216	564
3	SHA3-512	594	48	192	480
	Ketje-SR	118	39	85	201
	Ascon	100	78	216	564
4	SHA3-512	1072.5	48	192	480
	Ketje-SR	215.8	39	85	201
	Ascon	179	78	216	564

Table 2.1: Performance characteristic for low-latency uniformly implementations of Ascon, SHA3-512 and Ketje Sr using HPC4 gadgets: area in [kGE] and latency results for 1, 4 and 10 blocks of 576 bits (the specified block size of SHA-512).

2.2.4 Other potential Applications

As aforementioned, in the context of REWIRE, we leverage Ascon as a NIST-selected lightweight crypto scheme in order to provide SCA-resistant implementations. In this regard, Ascon is a timely target for implementations with HPC4 gadgets, as it has good features for side-channel security, both in terms of latency and levelling opportunities. In this section, we briefly discuss additional primitives that can benefit from HPC4 gadgets. Specifically, we start with other low-latency ciphers in Section 2.2.4.1 and follow with iterative AND computations that can be used by authenticated encryption schemes or even post-quantum (public-key) cryptographic algorithms.

2.2.4.1 Low-latency ciphers

Among the ciphers using quadratic substitution layers, one illustration is Keccak and the SHA-3 standard, the permutation of which shares the low-latency features of Ascon. We give results for the (most standard) 1600-bit version SHA3-512. Another one is the authenticated encryption mode Ketje which is derived from Keccak (see `https://keccak.team/ketje.html`), which is more comparable to Ascon, but does not benefit from its strong model-level security guarantees. We give results for the 400-bit version Ketje Sr (which has a size close to Ascon). Our area (in kilo-gate equivalent) and latency results for uniformly protected implementations of these algorithms are in Table 2.1. Other examples can be found in the literature, like the Gimli permutation [14].

2.2.4.2 Iterative AND reduction

The computation of the conjuction of many bits, also known as the “AND reduction” of a vector of bits, appears in many cryptographic algorithms, for example when verifying a tag of an AE or when verifying component-wise properties of vectors in lattice-based cryptography (e.g., this happens for ciphertext equality checks in Kyber [3] or for norm checks in Dilithium [35]). In both cases, leakage on the AND reduction can lead to security issues requiring masking [5, 18, 74].[2] Therefore, these algorithms require verifying that all the bits of two η-bit vectors v and v' are identical, without leaking about these bits. For this purpose, a standard option is to compute the difference $x = \overline{v \oplus v'}$ and to output the conjunction of this difference to test equality.

Figure 2.1 depicts a simple architecture to perform this computation at a rate of one bit of x per cycle. The MUX selects the sharing of 1 to reset ($s = 1$) for one clock cycle, then activates the loop ($s = 0$)

[2] In the AE case, leakage-resilient tag verification can serve as alternative [15, 32].

to iteratively process the AND with x's bits afterward. The throughput of this circuit can be increased by adding a pre-processing pipeline circuit connected to the input x that computes the AND of a chunk of the input bit vector. Due to its single-cycle iterative architecture, this implementation may exhibit composition issues when the AND gadget is not O-PINI, leading to lower-order leakage on the value of the accumulator bit, that is, on the conjunction of a subset of the bits of x.

2.3 Application of the Masked Gadget to ASCON

This section concludes the bottom-up approach by showcasing how the protected primitive through masking can be leveraged in a levelled implementation. We tackle first this style of implementation because it is the most naturally following Ascon's specification. Taking the use-case of Ascon, we develop on the performance, technique of implementation and comparison with the state of the art.

2.3.1 Mode-level security against leakage

Authenticated Encryption (AE) aims at ensuring both message integrity and confidentiality. This can be formalized both with an "all-in-one" definition as proposed by Rogaway and Shrimpton [63] or with the composite definitional framework of Guo et al. [44]. We use the second option which allows leveraging the fact that integrity with leakage and confidentiality with leakage impose quite different security requirements, which is the basis of leveled implementations [12]. It relies on the following notions:

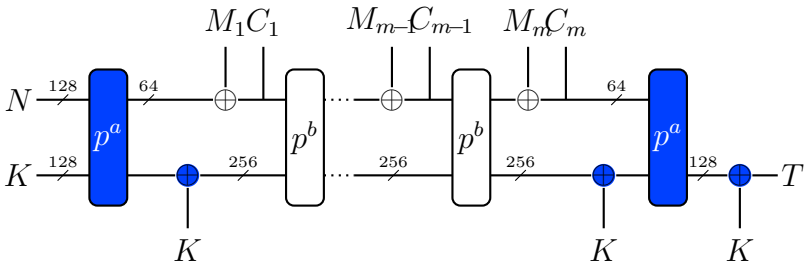
Definition 11 (Ciphertext Integrity with Leakage (informal))**.** In the Ciphertext Integrity with Leakage (CIL) security game, the adversary can perform a number of queries to encryption and decryption oracles enhanced with leakage functions, that capture the implementation of an AE scheme. Her goal is to produce a valid fresh ciphertext. The implementation is secure if the adversary can only succeed with negligible probability.

Definition 12 (Confidentiality with leakage (informal))**.** In the Chosen Ciphertext Attack with Leakage (CCAL) security game, the adversary can perform a number of queries to encryption and decryption oracles enhanced with leakage functions, that capture the implementation of an AE scheme. During a so-called "challenge query", she picks up two fresh messages X_0 and X_1 and receives a ciphertext Y_b encrypting X_b for $b \in 0, 1$, with the corresponding leakage. Her goal is to guess the bit b. The implementation is secure if the adversary can only succeed with negligible advantage over a random guess.

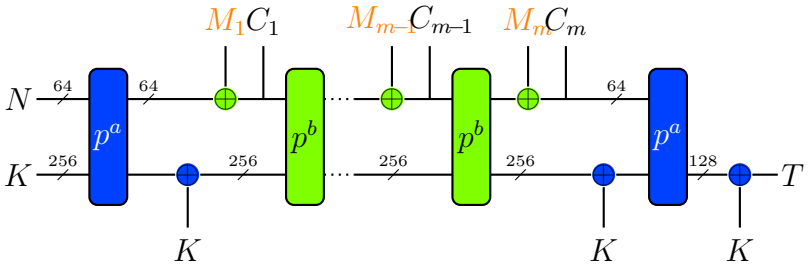
These two games can be played in different variants, corresponding to more or less powerful adversaries. A first axis of variants specifies whether leakage can be observed in encryption only (which is reflected by a "1" in the notations) or in encryption and decryption (which is reflected by a "2" in the notations). For example, CIL1 and CCAL1 stand for integrity and confidentiality with leakage in encryption only, CIL2 and CCAL2 stand for their counterpart with additional leakage in decryption. Another axis of variants specifies whether the implementation is assumed to be nonce-respecting or if some kind of nonce control is allowed. We consider the possibility of nonce misuse-resistance [63] (which is reflected by a capital M in the notations) when targeting integrity with leakage, and the possibility of nonce misuse-resilience [2] (which is reflected by a small m in the notations) when targeting confidentiality with leakage. In the first case, the nonce can be misused even in the challenge query. In the second case, misuse is excluded for the challenge query, which aims to ensure that although confidentiality may be lost when the nonce is under adversarial control, it is restored as soon as fresh nonces are used. For example, CIML1 and CIML2 stand for integrity with misuse-resistance and leakage, CCAmL1 and CCAmL2 stand for confidentiality with misuse-resilience and leakage.[3]

[3] As discussed in [44], this restriction is needed because confidentiality with leakage and nonce misuse-resistance can only be obtained with unrealistic (leak-free) requirements for the implementers.

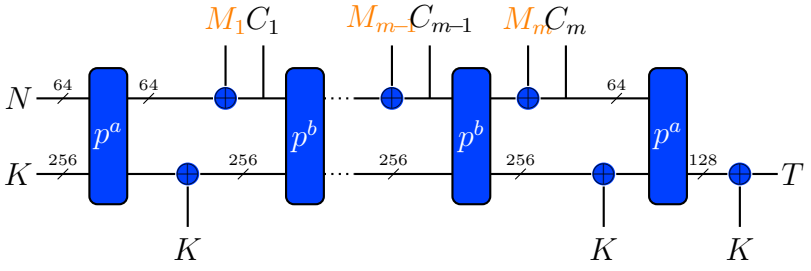
Thanks to this definitional framework, it is possible to analyze the security of a mode of operation with leakage under different physical assumptions for its different parts, in turn leading to leveled implementations. Ascon's leveling opportunities have been formally investigated in [45] and are summarized in Figure 2.3. Informally, CIML2 security only requires to protect the key derivation and tag generation against Differential Power Analysis (DPA), leaving the message processing part unprotected; CCAmL1 security additionally requires that the message processing part is protected against Simple Power Analysis (SPA); CCAmL2 security finally requires a uniform protection against DPA. By DPA (resp., SPA), we mean side-channel attacks where the adversary can observe the encryption of many plaintexts (resp., a few plaintexts) with the same key. We can already observe that taking advantage of Ascon's leveling opportunities requires knowing the security target (e.g., for confidentiality) in advance. This raises the question whether an implementation with a security target selected at execution time can be built more efficiently than with the naive solution where multiple designs are just implemented independently.



(a) CIML2-secure implementation.



(b) CCAmL1-secure implementation.



(c) CCAmL2-secure implementation.

Figure 2.3: Leveled implementation of Ascon for different security targets. The blue blocks have to be protected against DPA, the green ones have to be protected against SPA and the white ones do not require protection against side-channel leakage.

2.3.2 Implementations of the Ascon permutation

The Ascon AEAD is made of two permutations p^a and p^b that iteratively apply an SPN-based round transformation p that consists of three steps : p_C (a constant addition), p_S (a non-linear S-box layer) and p_L (a linear diffusion layer). The mentioned permutations p^a and p^b differ by the number of rounds and the initial round constant.

Scheme	d	Latency [cycle]	Randomness [bit]	Area [GE]
	2	2	1	86.6
HPC2	3	2	3	216
	4	2	6	402.6
[0.8ex]	2	1	2	74.6
HPC3	3	1	6	177
	4	1	12	330.6
[0.8ex]	2	2	3	89
HPC3+	3	2	8	200
	4	2	15	362.3
[0.8ex]	2	1	5	102.3
HPC4	3	1	15	260
	4	1	30	487.3

Table 2.2: Area comparison of HPC AND gadgets. Post-synthesis results for the NanGate 45 Open Cell Library obtained with Cadence Design Vision design toolchain.

2.3.2.1 Unprotected implementations

We denote here an unprotected implementation as one that is not embedding any primitive-level countermeasures against Side-Channel attacks. There it follows the classical guidelines of designing hardware cryptographic implementations and we will use the following unprotected implementations:

- **Round-based**: a full transformation p is computed, then stored in a register to be fed back to the transformation. This implementation computes one round per cycle.
- **Unrolled**: instead of computing a single transformation, one can use more physical resources to implement u times the transformation p and perform u rounds per cycle.

We will next denote a round-based implementation as rb and an unrolled implementation as unr-u where u is the number of round processed between two registers layers. These unprotected implementations will be used in the message processing parts of the mode.

2.3.2.2 Protected implementations

We will use the following protected implementations:

- **Round-based**: as depicted in Figure 2.4, a protected round-based implementation computes the same layers but takes advantage of the register layer in the p_S step. Hence the substitution operation is split between the variables that are input to the AND operation (p_S^1) and the ones that are computed with its output (p_S^2).
- **Serialized**: in order to reduce the high cost of masked gadget (shown in Table 2.2) we can leverage serialized implementations that trade latency for area by implementing only a fraction of the required logic gates and looping over them.

2.3.2.3 Insecurity of HPC3 in a round-based Ascon permutation

In subsection 2.2.1, we raised the theoretical concern that transitions can harm the security of masked implementations. The HPC4 gadget offers strong security guarantees even in this case, whereas the HPC3 gadget does not. As discussed in [22], this is not always a problem and there are examples of (e.g., serial) architectures where transitions do not compromise security. In this section, we nevertheless

[flipflop AB, scale=.5] (ff1) at (0,0) ;
[draw, rectangle, thick, minimum height=35, left=.15 of ff1] (p$_s$)p$_S^1$; [draw, rectangle, thick, minimum height=35, left=.15 of p$_s$](p_c)p$_C$; [draw, rectangle, thick, minimum height=35, right=.15 of ff1] (p$_s 2$)p$_S^2$;
[draw, rectangle, thick, minimum height=35, right=.15 of p$_s 2$](p_l)p$_L$;
[draw, rectangle, thick, minimum height=50, minimum width = 125] (pi) at (0, 0) ; [above = .15 of pi] (pi$_t ext$)p;
[left = .75 of p$_c$](in)S$_{in}$; [right = .75 of p$_l$](out)p(S$_{in}$); [-¿] (in) – (p$_c$); $[- >](p_l) - -(out)$;

Figure 2.4: Masked Ascon primitive implementation: p_c denotes the round constant addition, p_s denotes the substitution layer that is split between the operations prior to the AND gate and the operations that use its output. For the operands that are not used in the AND gate, it require a synchronization register. The linear layer p_l is applied over the state and $p(S_{in})$ designates a round of the Ascon permutation made over the state S_{in}.

show that in the specific case of single-cycle round-based architectures that is motivated by our low-latency goal, such transitions lead to concrete weaknesses. For this purpose, we study the integration of the HPC3 gadget in a round-based masked implementation of the Ascon permutation.

Informally, in such a round-based implementation of Ascon-p, the inputs and outputs of the p_S layer are not isolated by registers (see Figure 2.4). Therefore, when using gadgets that are not robust against transitions, the extended probes can recombine over the same gadget and lead to reduce the security order of the implementation. The HPC3 gadget, which is only PINI and not O-PINI, is typically vulnerable to such leakages. We confirmed this issue by using the PROLEAD tool [54] and analyzed the security of a toy implementation where a single Ascon S-box is looping on itself. This provides a subset of the leakage that a full Ascon implementation leads to and is already sufficient to identify critical leakage with the HPC3 gadget. (We note that running the tool on the full Ascon is not possible since the memory cost of the PROLEAD verification rapidly explodes in the presence of extended probes that “touch” many bits). We tested the same round-based architecture with HPC3 and HPC4 gadgets in the fixed vs. random setup with up to 2,500,000 simulations. Significant leakage was consistently flagged (during the second cycle of the S-box computation) when using HPC3 gadgets, while the detection threshold (of 5) was never reached when using HPC4 gadgets. We refer to [53] for in-depth discussions of how such robust probing security issues translate into concrete attacks.

2.3.3 CIML2+CCAmL2 with uniform masking

This first implementation achieves the highest security level (i.e., a combination of CIML2 and CCAmL2, as defined in [12]) at the cost of having all its operations being uniformly protected with masking. It is typically relevant when confidentiality is required for a leaking decryption (e.g., which may be the case with software updates, movies or video games). This architecture is described in Figure 2.5. We keep the same color code as in Figure 2.3: the blue color, which reflects leaking operations that must be secure against DPA, is used everywhere. We then instantiate the masked Ascon permutation with either a round-based (parallel) HPC4-based implementation, since this is the architecture that best benefits from the gadget’s low-latency features, or an 80-bit serialized HPC2-based implementation, since the higher latency of this gadget is amortized in this serial architecture.

The mode was implemented carefully in order to avoid the introduction of architectural-level leakages. In particular, the sensitive key additions are performed by muxing the key with a vector made of zeros of the same size. This mux is controlled by a flag that passes through a register in order to provide a clean edge and remove glitches. Another relevant mechanism that is included in our designs is the possibility to stall the design. Such a feature is useful for applications where the data feed may not be synchronized with the cryptographic core. This stalling module is located just before the data processing, so that one can stall and directly unfreeze the design in order to process the data.

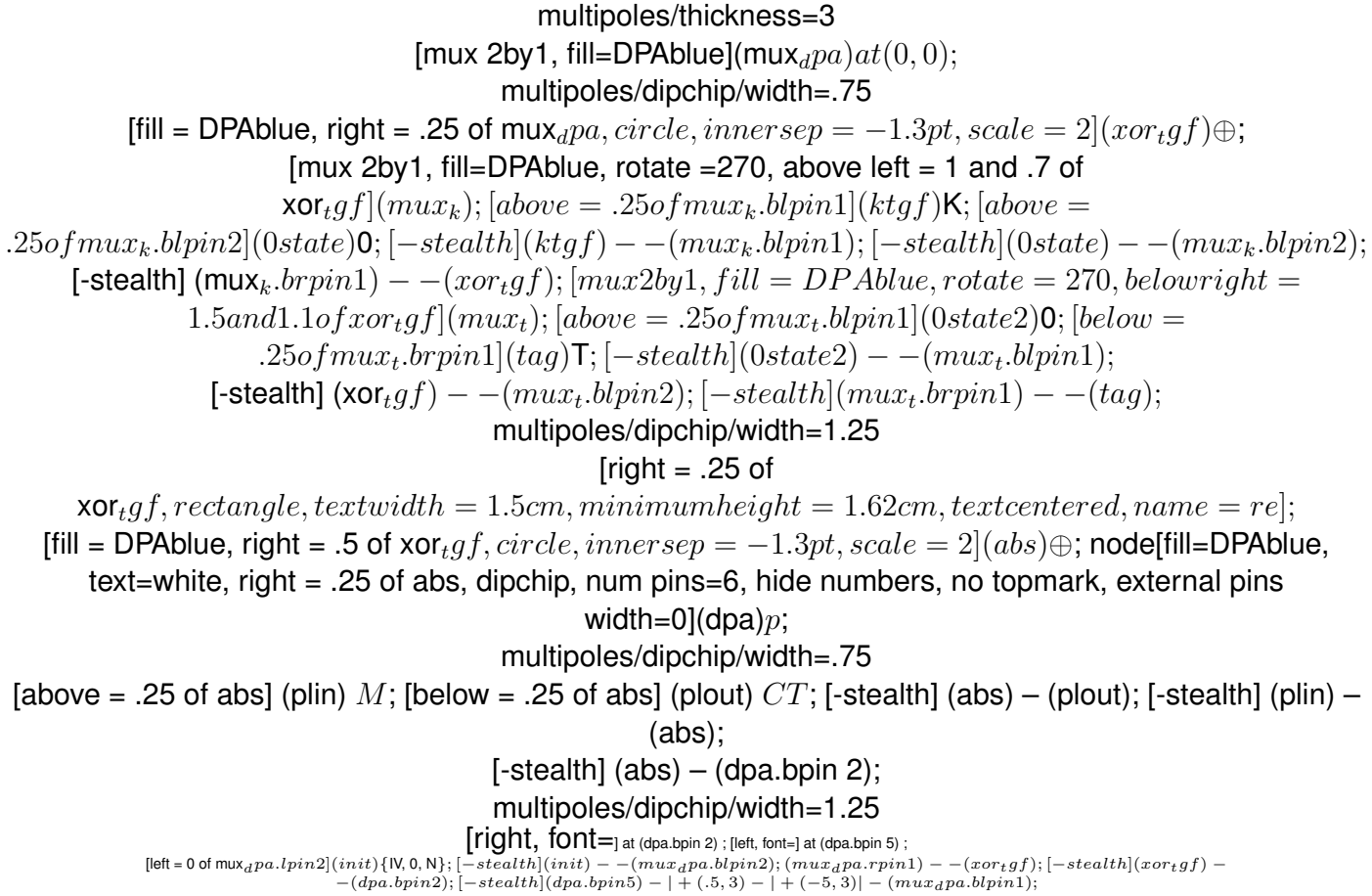


Figure 2.5: Uniformly protected implementation ensuring CIML2 and CCAmL2.

2.3.4 CIML2+CCAmL1 with leveling

This second implementation drops the CCAmL2 security requirement and only achieves confidentiality with leakage in encryption (CCAmL1), while preserving the highest integrity guarantees (CIML2). It corresponds to the leveled design of Figure 2.6 which, as reflected by the green color, leverages weaker (SPA) protections for the permutations used during the message processing (the key derivation and tag generation still require DPA protections).

Informally, the protected permutation processes the initial state to produce a fresh ephemeral key which is forwarded to the unprotected permutation. The unprotected permutation then loops over until the data is fully absorbed, and the final state is fed to the protected primitive again, to generate the tag. The masked permutation is the same as previously described (instantiated with HPC2 or HPC4 gadgets). The unprotected core achieves SPA security through a parallel hardware implementation. As a result, this part of the implementation can be unrolled in order to achieve a lower latency.

2.3.5 Performance evaluation

We implemented the different single-target Ascon architectures in the NanGate 45 Open Cell Library, for designs synthetized using the Cadence Design Vision toolchain. Area comparisons are depicted in Figure 2.7. As for the latencies of the modes, they are depicted on Figure 2.8 and we also provide the explicit equations.
First, when using a uniformly protected implementation, we have:

$$L_u(m) = 24c + \left\lceil \frac{m+1}{64} \right\rceil * 6c, \tag{2.1}$$

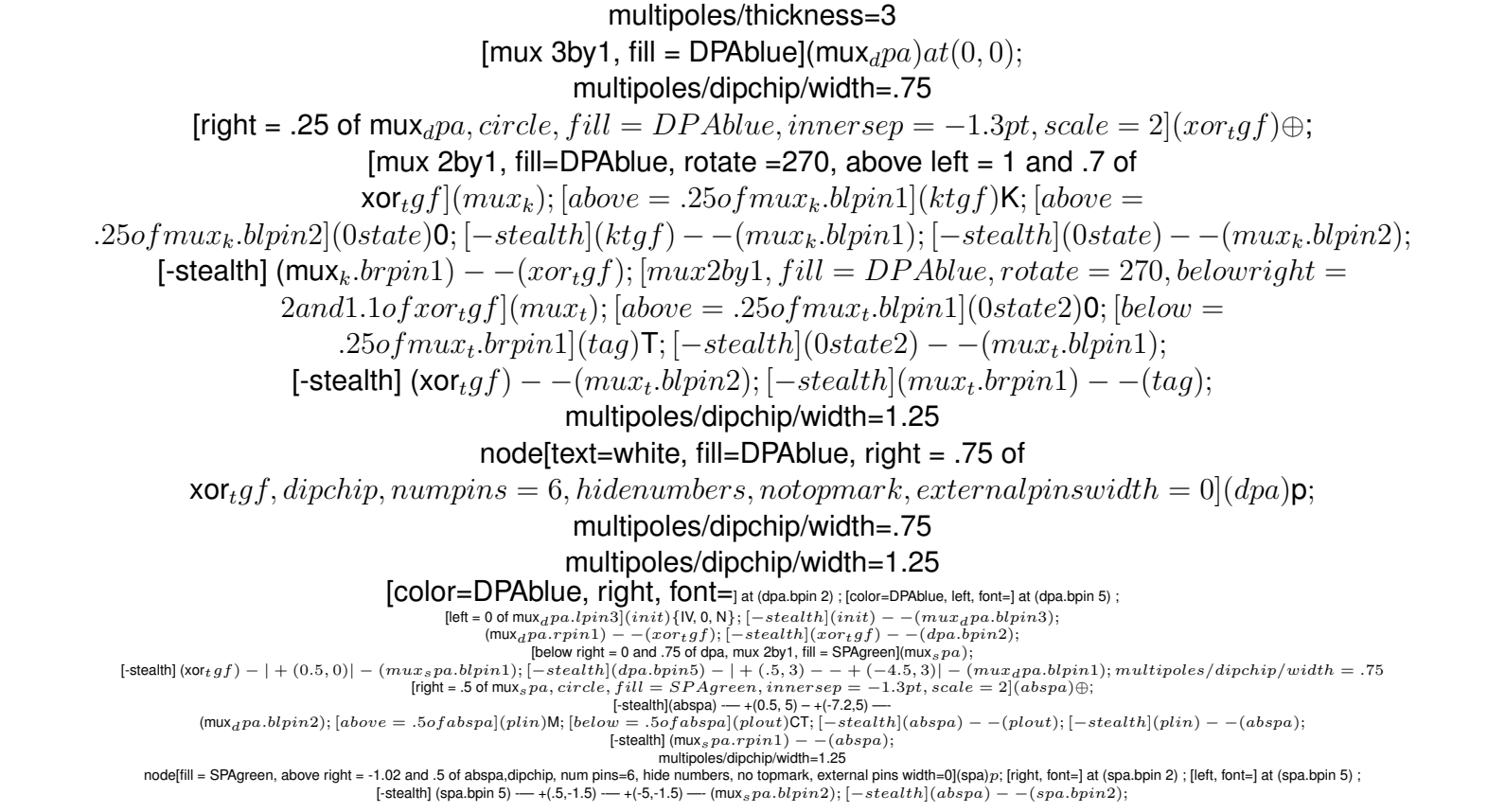


Figure 2.6: Leveled implementation ensuring CIML2 and CCAmL1.

which gives the latency of the mode of operation when encrypting an m-bit message, where $c = 6$ cycles if using HPC2-ser gadgets and $c = 1$ cycle if using HPC4-r.b. gadgets.

Next, when using a leveled implementation, we have:

$$L_l(m) = 24c + \left\lceil \frac{m+1}{64} \right\rceil * \frac{6}{u}, \tag{2.2}$$

where u is the number of rounds computed in one clock cycle for the unprotected implementation used in the message processing part of the mode. Similarly to the uniform case, $c = 6$ cycles (resp., $c = 1$ cycle) when using HPC2-ser (resp., HPC4-r.b.) gadgets for the key derivation and tag generation parts of the authenticated encryption mode.

In the uniformly protected implementation case, HPC2 is less expensive than HPC4 but implies significant latency overheads. In the leveled case, unrolling the unprotected implementation provides significant latency gains while the choice between HPC2 and HPC4 provides a cost vs. latency tradeoff. Overall, leveling comes at limited area cost and is therefore relevant in case only CCAmL1 security is needed.

Comparison with state of the art implementations. The work in [43] reports an area of 42.75kGE and consumes 2048 bits of randomness per cycle for a uniformly protected implementation. Our uniformly protected implementation with HPC4 requires an area of 46.66kGE and consumes 1600 bits of randomness per cycle. In term of latency, they both achieve one round per cycle (see the continuous blue line on Figure 2.8). We therefore achieve similar performances while provably avoiding the flaws put forward in [53]. The work in [76] proposes a leveled implementation where the key derivation and tag generation function are instantiated with 80-bit serialized implementations using HPC2 gadgets, while the message is processed by a round-based unprotected implementation. Their reported area is 43kGE while we achieve 31.5kGE for a similar architecture using similar latencies (see the dashed red line on Figure 2.8). We additionally offer the possibility to unrol the unprotected implementation. These comparisons are summarized in Table 2.3.

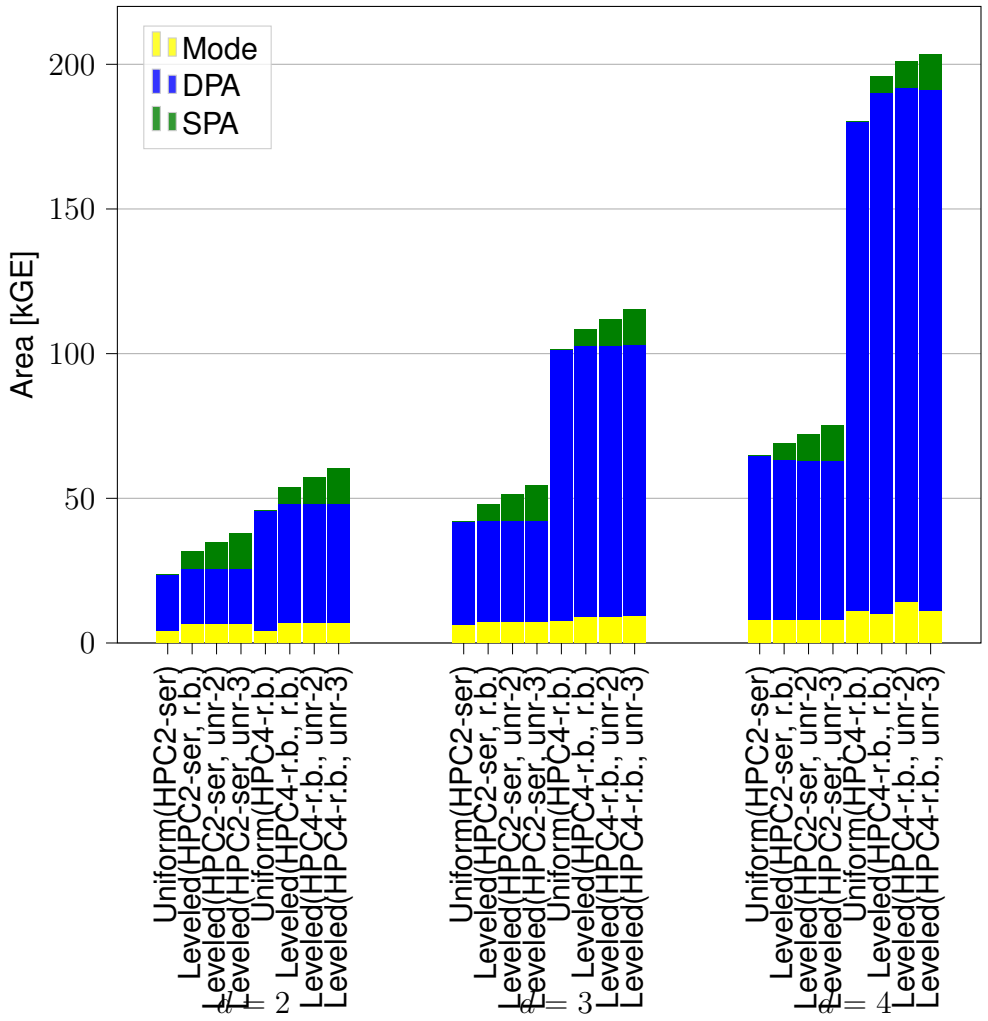


Figure 2.7: Single-target implementations: area comparisons.

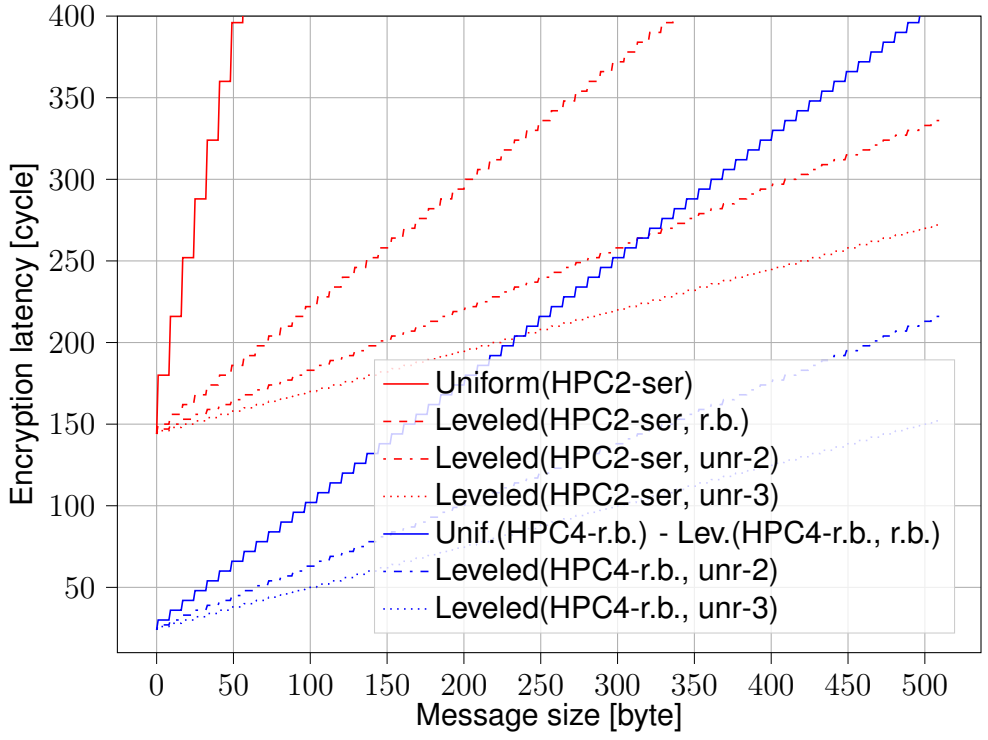


Figure 2.8: Single-target implementations: latency comparisons.

Reference	Architectures	Area (kGE)	Latency (cycle for specified blocks)		
			1	4	10
[76]	(HPC2-ser, r.b.)	43	150	168	204
This		31.5	150	168	204
[43]	GLM	42.75	30	48	84
This	HPC4-r.b.	46.66	30	48	84

Table 2.3: Comparison between the state of the art and our proposed solutions.

2.4 Primitive-Level and Model-Level Countermeasures Evaluation

Differential Power Analysis (DPA) is a very powerful type of side-channel attack [52]. As for example reflected by the linear bound in [29], it allows reducing the security of a block cipher key exponentially in the number of plaintexts for which the adversary can combine her physical measurements. There exists two main approaches in order to mitigate such differential attacks. One is to reduce the leakage at the implementation level. The most popular solution for this purpose is masking (aka secret sharing) [26, 42]. Under some (noise and independence) assumptions, it allows reducing the information leaked by an implementation exponentially in the number of shares used by the countermeasure, which in turn leads to an exponential security amplification [8, 34, 47, 48, 57]. The approach of REWIRE with LR-BC2 is to design primitives that limit the exponential security decrease that DPA causes, by limiting the number of plaintexts for which the adversary can combine her measurements, thanks to re-keying. One popular solution for this purpose is to use a leakage-resilient PRF [11, 33, 38, 69, 79].

It is already known that masking and leakage-resilient PRFs may not blend gracefully [10, 73]. The reason for this poor combination is intuitive and informative. On the one hand, Boolean masking, which is prominently used in order to protect ciphers operating in binary fields like the AES [28] or Ascon [31], essentially increases security against DPA via noise amplification. That is, it aims to force the adversary to estimate higher-order statistical moments of the leakage distribution, a task of which the complexity grows exponentially in the number of shares if the measurements are sufficiently noisy. On the other hand, leakage-resilient PRFs limit the number of plaintexts for which the adversary can observe the leakage, but they do not limit the number of times each plaintext can be observed. That is, they aim to limit the amount of side-channel signal that remains (for example) after averaging the measurement noise, thanks to repetition. Since Boolean masking amplifies the noise and leakage-resilient PRFs allow repetition to get rid of the noise, it implies that the complexity to attack their combination adds up, rather than multiplies, as expected for a graceful combination of countermeasures [62].

Given that combining Boolean masking and leakage-resilient PRFs is not desirable, the important next question becomes: *which one to privilege in which context?* Quite naturally, the fact that primitive-level and implementation-level countermeasures leverage very different security mechanisms makes their general comparison difficult and, to the best of our knowledge, there are no works in this direction so far. In this section, we therefore relax this problem by comparing these two types of countermeasures in a specific yet practically-relevant context. Namely, we consider the security that can be obtained by using commercial MCUs as could be exploited in lightweight (e.g., IoT) applications. We focus in particular on the security that can be obtained by leveraging the AES hardware implementations which are popular coprocessors in the embedded security market.

Unsurprisingly, AES unprotected coprocessors remain the most widespread and the landscape of MCUs protected with implementation-level countermeasures is quite scarce, especially when focusing on devices that are accessible for academic research. One of the few (if not the only) option for this purpose is the STM32U5 family of chips, which is based on the ARM Cortex M33 architecture. Of particular interest for our investigations, these devices embed an AES coprocessor protected against physical attacks which is PSA-certified Level-3.[4] As a natural competitor, we integrate the unprotected AES coprocessor of the

[4] https://newsroom.st.com/media-center/press-item.html/n4377.html.

(ARM Cortex M4 based) STM32F4 family of chips into a leakage-resilient PRF.

Due to the closed-source nature of these implementations, our study is performed without an accurate description of the targets' hardware architecture nor, for the STM32U5 device, of the countermeasures it implements. As a result, we start our investigations with some side-channel reverse engineering. For the unprotected AES implementation of the STM32F4, Signal-to-Noise Ratio (SNR) analyzes confirm that it corresponds to a 128-bit loop architecture (matching the specification that the AES is performed in 11 cycles). For the protected AES implementation of the STM32U5, a similar analysis rather suggests a 32-bit architecture, with noise/jitter engines (mentioned in the specifications), first-order masking for the rounds and no masking for the key scheduling algorithm.[5]

Based on these preliminary efforts, we then try to compare the STM32U5 AES protected with implementation-level countermeasures, which we from now assume to leverage Boolean masking for simplicity, and the STM32F4 AES integrated in a leakage-resilient PRF, on sound empirical bases. For this purpose, we perform an in-depth investigation of state-of-the-art profiled attacks, trying to optimize their main parameters. Due to the very different nature of the challenge to evaluate the STM32U5 and STM32F4 implementations, we borrow different strategies for both targets. Namely, we follow a (more) certification-style/qualitative approach for the protected AES, where we use one month of measurements in order to exhibit a first baseline attack, and we follow a (more) worst-case/quantitative approach, trying to bound the best attacks, for the unprotected AES integrated in a leakage-resilient PRF [68]. In both cases, we discuss the risk of security overstatements. For the STM32F4, we additionally discuss the possible gap between our close-to-worst-case evaluations and more practical ones, as advertized by the backwards security evaluations put forward in [4]. By combining these "best-effort" estimations, we observe that the cycle count vs. security tradeoff of the leakage-resilient PRF using an unprotected coprocessor is similar to slightly better than the one of the protected AES core of the STM32U5 chip. Given the more worst-case analysis of the leakage-resilient PRF, we deem this conclusion more likely to be amplified than weakened with more evaluation efforts.

We conclude this section by discussing the generalization and the impact of our findings. From the generalization viewpoint, we highlight that our comparisons are highly technology-dependent and therefore hard to capture with theoretical models. From the impact viewpoint, we nevertheless argue that considering the standardization ofleakage-resilient PRFs able to leverage the (e.g., AES) coprocessors that are readily available on a vast range of products could be highly desirable: not because of a better cost vs. security tradeoff but because of the easier risk assessment (i.e., security evaluation) that they enable, their more direct link between functional correctness and physical security and, mostly, their currently wider availability than protected implementations. This makes them a solution of choice for integration in low-cost applications where secure coprocessors and/or the expertise required to implement masking securely may be too expensive.

Responsible disclosure and open designs. Our analyzes put forward a first-order flaw in the (presumably masked) AES coprocessor of the STM32U5 chip. This observation has been communicated to the STMicroelectronics Product Security Incident Response Team before publishing, so they can provide guidance to end users so that this flaw cannot be exploited in practical applications. We further note that this flaw is only detectable with a large number of measurements (in the tenths of millions range) and may not contradict the PSA certificate claims. It could be tempting, based on these results, to conclude that preventing cryptographic implementations from being scrutinized by the academic community could be beneficial to security. We believe the exact opposite conclusion should prevail. Namely, these results mostly question (once again) the limited quantitative value of current certification schemes, and the importance of a continuous and open approach to gain confidence in implementation security.[6,7,8] Positively, enabling the public investigation of cryptographic coprocessors can only help identifying tracks for

[5] We also observed redundancy, presumably for fault detection/correction purposes.

[6] https://ninjalab.io/ledger-challenge-2018/

[7] https://ninjalab.io/a-side-journey-to-titan/

[8] https://ninjalab.io/eucleak/

improving these products in the long-term. Such public investigations would become even more effective if they could leverage more open designs (avoiding the loss of time devoted to reverse engineering). Negatively, a time-limited and closed-source evaluation process inherently comes with an increased risk of overstated security claims, potentially leaving (sometimes important) attack vectors hidden to end users, despite exploitable by determined adversaries.

Terminology. The leakage-resilient PRF is usually denoted as a mode-level countermeasure in the literature. We denote it as a primitive-level countermeasure since such a PRF is only a building block that still has to be integrated in a mode of operation for authentication, encryption or authenticated encryption. Most of our observations are nevertheless valid for re-keying, whether used in a primitive or a mode of operation.

2.4.1 Experimental setup

In this subsection we describe the experimental setup and some initial results on the performed experiments.

2.4.1.1 Targets and setup description

Our first target is the STM32F415: a microcontroller from STMicroelectronics' STM32F4 series, which is well-known for delivering high performance along with a wide range of integrated features. The STM32F415 is built around an ARM Cortex-M4 core, combining advanced processing capabilities with Digital Signal Processing (DSP) features and a Floating Point Unit (FPU). In addition to its computational performance, the STM32F415 MCU stands out with its fully parallel (unprotected) AES cryptographic accelerator.

Our second target is the STM32U585: a microcontroller from STMicroelectronics' STM32U5 series, which is designed with a strong emphasis on ultra-low power consumption. Built around the ARM Cortex-M33 core, the STM32U585 benefits from the integrated TrustZone technology for hardware-based security. One of the key features of the STM32U585 is its combination of security tools, such as a True Random Number Generator (TRNG) and a secure AES encryption hardware engine, enhancing its suitability for secure and connected applications. In order to perform a secure AES encryption with the protected hardware accelerator, one must ensure that the TNRG is feeding fresh randomness (which we did) and select a way to load the key. Among the possibilities to load the key in the coprocessor, we chose to use the software loading mechanism, since it ensured a better acquisition throughput than the others. We got confirmation through STM32's forum that the SCA countermeasure(s) remain(s) active in that scenario.

Both MCUs are packaged in an LQFP-64 format.

In terms of measurement capabilities, we replicate our setup for both targets in order to process them in parallel, ensuring similar conditions and measurement capacities for each. The targets are soldered on a CW308T-STM32F: a circuit board supporting several STM32F devices in the LQFP-64 package. These boards are then plugged in a CW308 UFO board that allowed us to drive the MCU clock with an 8MHz external crystal and to supply power through a potentiometer, which we set at 1.8V in our experiments.

We run the unprotected AES core at 8MHz to obtain a low operating clock as indicated in the guidelines from [13]. In the case of the protected AES core, we have to operate at 48MHz through a secure clock. The signal is collected with a CT1 current probe on the CW308 UFO shunt jumpers and sampled with a PicoScope 6424E, operating at 5[GSamples/s] with a 500 MHz analog bandwidth, using 10-bits of vertical resolution.

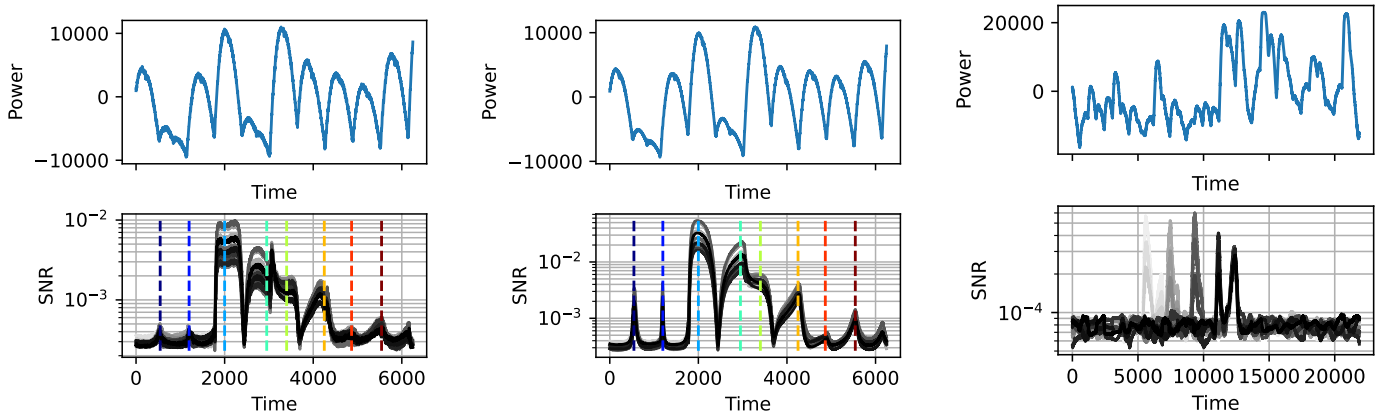
For each MCU, we finally apply the methodology of [65] to speed up measurements. Namely, we use the AES coprocessor as a Pseudo-Random Number Generator (PRNG) to generate plaintexts and keys, thereby significantly reducing the communication overheads with a desktop system. This involves encrypting a zero vector in CBC mode under a chosen initialization vector (IV). We compute the SNRs in

real time to quickly adjust acquisition parameters, such as signal windowing and acquisition range. The configuration that yields the highest SNR estimation is selected. SNR calculations, the LDA modeling and the rank estimation are performed using the open-source SCALib library [23].

2.4.1.2 Architectural assumptions

The main information that specifications provide about the target's respective architectures is a cycle count of 11 for the STM32F4 and 528 for the STMU5 (suggesting a more parallel architecture for the former one). As a preliminary before evaluating the physical security of these targets, we therefore performed some side-channel reverse engineering in order to infer/confirm architectural assumptions. We use SNR estimations for the 16 S-boxes of the first AES round for this purpose. They are illustrated in Figure 2.9.

Starting with the unprotected implementation, we can confirm a loop architecture processing a 128-bit round per cycle from the (left and middle) plots with the traces on top, where the 16 SNR peaks overlap, suggesting a parallel execution of the S-boxes.



(a) STM32F415, without repetition ($N_r = 1$) for one million observations ($N_p = 10^6$).

(b) STM32F415, with repetitions ($N_r = 1024$) for one million observations ($N_p = 10^6$).

(c) STM32U585, with $N_r = 1024$ repetitions for 3 million observations ($N_p = 3 \times 10^6$).

Figure 2.9: Comparison of the SNR for both targets and impact of the repetition factor N_r. The 16 SNR curves with different shades of gray correspond to different S-boxes.

2.4.2 Leakage-resilient PRF (STM32F4)

In this section, we detail the methodology followed for evaluating the physical security of the STM32F4 target integrated in the leakage-resilient PRF of Figure 2.10. We begin with the profiling phase and explain how we optimized our LDA models in subsubsection 2.4.2.2. Next, we present the performances achieved by our models and assess the security of the LR-PRF in subsubsection 2.4.2.3. Finally, we discuss the limitations of this analysis and its potential improvements in subsubsection 2.4.2.4. We also put our results in the perspective of a backwards security evaluation by discussing model robustness issues in subsubsection 2.4.3.4.

Before describing these results, we recall that the LR-PRF bounds the number of plaintexts N_x^s that can be observed by the side-channel adversary, for each of its stage keys. This bound is the mos important parameter to select since it directly determines the cost vs. security tradeoff of the construction. In order to estimate it, the main difficulty is that an adversary against the LR-PRF can actually repeat the observation of each plaintext N_r times and, for example, average the leakage traces in order to make them less noisy (or even noise-free). So a critical step of our evaluations is to gain confidence that the

impact of increasing the number of repetitions N_r at some point "saturates", indicating that most of the (bounded) side-channel signal has indeed been captured by the evaluation.

2.4.2.1 The construction

The tree-based PRF we consider in this deliverable was initially proposed by Goldreich, Goldwasser and Micali [40]. It has then been shown in a sequence of works that it has good properties for improving security against leakage [11, 33, 38, 69, 79]. As illustrated in Figure 2.10, the function processes a 128-bit input x with a 128-bit key k in order to create a 128-bit output y, by sequentially applying $128/n_x$ AES-based stages.

In each of those stages, n_x bits of the input x are used to select an AES plaintext among $N_x^s = 2^{n_x}$ fixed (public) ones. At the i-th stage, the selected plaintext is AES-encrypted by the stage key k^i (with $k^0 = k$), which corresponds to the output of the previous stage. Therefore, a complete LR-PRF execution requires $128/n_x$ AES executions.

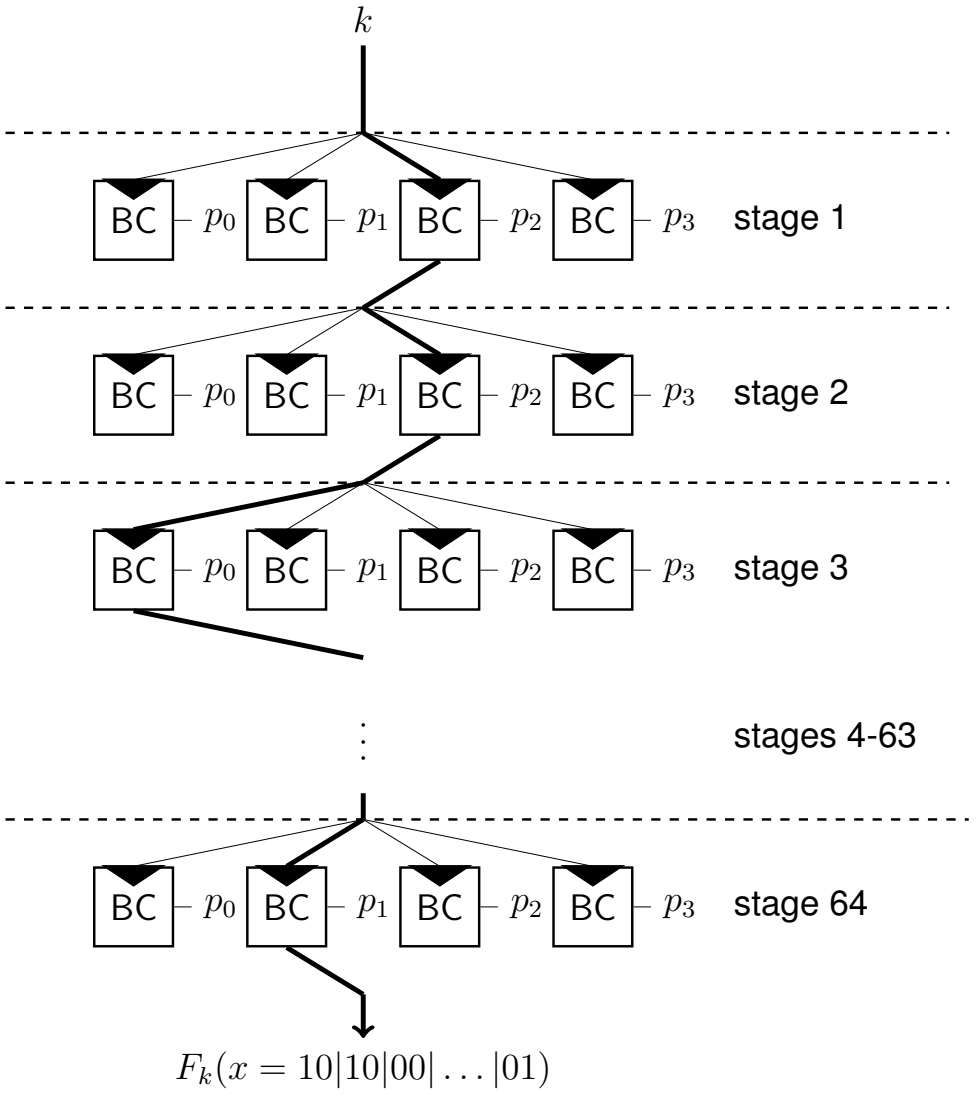


Figure 2.10: Leakage-resilient PRF, $N_x^s = 4$ ($n_x = 2$).

Such a PRF is interesting from the leakage viewpoint since it bounds the amount of AES plaintexts that an adversary can observe in order to perform a DPA against each stage key to N_x^s. By contrast, it does not bound the number of times each of these N_x^s plaintexts can be observed. We will next use the notation N_r (where r stands for repetition) to denote the number of times each AES plaintext is observed

by the adversary. The expectation is that such a LR-PRF can be secure against DPA without expensive implementation-level countermeasures, for example by leveraging parallel hardware implementations which are assumed to be difficult to break with a few plaintexts, even after many repetitions.

2.4.2.2 Model profiling

As for the STM32U5, the POIs' selection is based on the SNR results. We observed significant SNR levels for each (first-round) key byte, spanning over 2500 time samples, when no repetitions are considered (i.e., $N_r = 1$). This situation does not change much when a high amount of repetitions is considered (e.g., $N_r = 1024$), apart from higher SNR values. So, again, we decided to estimate Gaussian templates in a linear subspace optimized thanks to LDA, for the informative POIs.

As in the previous section, we conduct an exhaustive exploration of the parameter space considered, in order to identify the parameters achieving a model that maximizes the PI for each byte. Nevertheless, this time we do it in function of the N_r parameter. The results obtained are summarized in Figure 2.11 for exemplary values $N_r = 1$. In both cases, we observe that all models lead to positive PI values that are higher than for the protected coprocessor in the previous section. Given the (presumably) more parallel nature of the unprotected coprocessor, this again confirms that implementation-level countermeasures were active for the STM32U5 target.

The most interesting observation of these figures appears when comparing the impact of an increased averaging. Namely, a notable effect of moving from Figure 2.11 to Figure 2.12 is that the increased N_r of the second figure enables the characterization of higher-dimensional models, reflected by larger PI values for larger number of dimensions after projection N_d. This suggests that, as the traces used for profiling (and, later on, attacking) become less noisy, the LDA models are able to capture smaller details of the leakage distribution.

As a side observation, we note that in both cases, there are key bytes that stand up due to an increased PI. Yet, the key bytes that stand up differ for the $N_r = 1$ and $N_r = 1024$ cases. This also suggests that different parts of the traces can be exploited by the LDA models depending on the level of noise/averaging. For example, one possible explanation is that the leakage of bytes 3 and 15 is concentrated on less POIs with higher SNR values, explaining that they lead to the most informative models when $N_r = 1$. By contrast, the leakage of bytes 6 and 8 could be more spread in time, including POIs with lower SNR values that more significantly benefit from the increased averaging in Figure 2.12.

2.4.2.3 Security evaluation

In order to embrace the more worst-case goal of our security evaluation for the LR-PRF, we first ran attacks on both the training set and the test set. Running attacks on the training set (i.e., with overfitting) is reminiscent of the TI metric and is aimed to provide a conservative bound on the adversary's success. Running attacks on the test set is rather connected to the PI metric and is therefore reflective of the models' generalization. We will additionally run attacks against an independent attack set (measured without interleaving) as part of the backwards security evaluation discussion of subsubsection 2.4.3.4.

In order to assess the impact of the N_r parameter, we ran attacks considering three levels of repetition (i.e., $N_r \in [1, 32, 1024]$), using the models with the highest PI for all the key bytes, identified thanks to Figure 2.11 and Figure 2.12 for $N_r \in [1, 1024]$ and following a similar strategy for $N_r = 32$. The key ranks obtained on the training dataset and on the test dataset are given in Figures 2.13 and 2.14, respectively. The median and quartiles are estimated based on 100 independent attacks for each case (i.e., $N_r = 1, 32, 1024$).

The more realistic results of Figure 2.14 confirm attack complexities in the range of what is predicted by the inverse of the PI values in subsubsection 2.4.2.2.

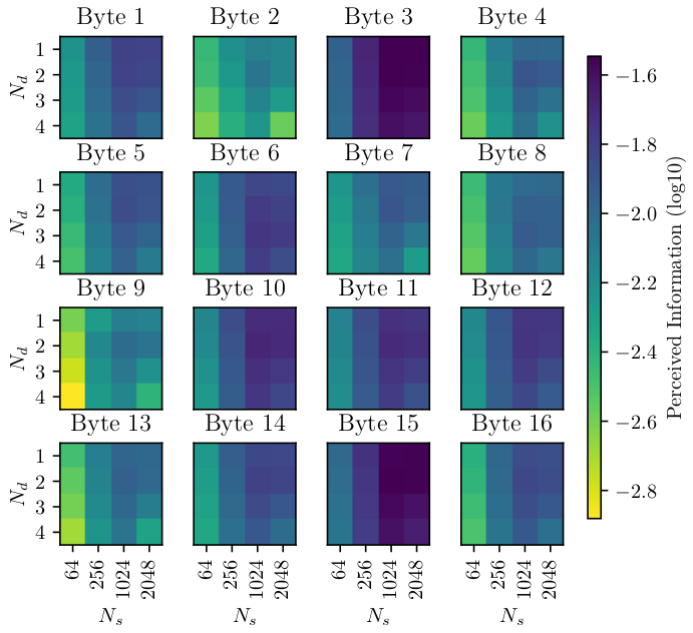


Figure 2.11: PI of the STM32F4 target for all the bytes and the whole parameter space with $N_r = 1$, displayed in log scale. As X-axis we display the number of selected Points-of-Interest (N_s) and as Y-axis the number of output dimensions (N_d) for the LDA. The PI is computed for one million traces corresponding to different plaintexts/keys.

Most importantly, these figures highlight that for small number of observable plaintexts (i.e., n_a values), high key ranks are maintained despite increasing N_r. Assuming that we target a 96-bit security level, and that the gap between the measurement campaign of the two targets (i.e., one week vs. one month in our experiments) is no sufficient to perform 2^{96} AES encryptions (which seems reasonable with current computing power), it means that the LR-PRF maintains a key rank similar to the one of the protected STM32U5 target with n_a up to 4. Given that the protected AES core runs in 528 cycles and the LR-PRF runs in $11 \cdot \frac{128}{n_x}$ cycles, the cost vs. security tradeoff of the LR-PRF becomes better starting from $n_x = 3$ and is therefore similar to slightly better in our study.[9]

2.4.2.4 Risk assessment

Starting with the risks of improved measurement setups, they are similar. If one becomes capable of very efficient measurements on the implementation, it may lead to higher signal, more informative leakage and better attacks. So the LR-PRF does not bring improvements on this side. By contrast, we next argue that it comes with lower risks of improved statistical processing for two main reasons.

First, a look at the PI/TI convergence plots of Figure 2.15 highlights that the models used to evaluate the unprotected coprocessor of the STM32F4 are closer to profiling saturation than the models used to evaluate the protected coprocessor of the STM32U5, in Figure 2.20. We also confirm in the left part of Figure 2.16 that for relevant POIs used in the models of the STM32F4, the SNR approaches saturation with respect to repetition/averaging. The same holds for the PI metric and sets of 128 POIs around the SNR peaks in the right part of the figure. So our LDA-based attacks against this unprotected target could only be marginally improved with more evaluation efforts (i.e., larger N_p or N_r values). Second, this target is unlikely to exhibit fancy leakage distributions that would strongly benefit from advanced statistical

[9] A bit more precisely, this comparison ignores the fact that the implementation of the STM32U5 includes some redundancy which we assume being due to countermeasures against fault attacks [49]. On the other hand, a leakage-resilient PRF integrated in a mode of operation may also offer security against certain fault attacks, as for example discussed by the ISAP designers [30]. Overall, our comparisons should anyway not be taken as strict indications of better or worse, but as evidence that the LR-PRF can be a competitive option when security against side-channel attacks is necessary.

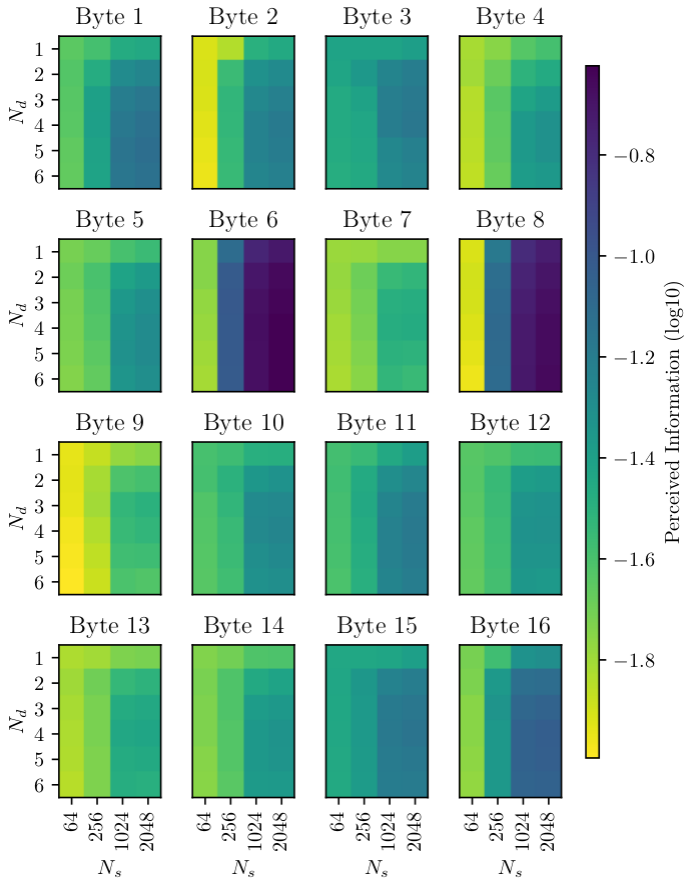


Figure 2.12: PI of the STM32F4 target for all the bytes and the whole parameter space with $N_r = 1024$, displayed in log scale. As X-axis we display the number of selected Points-of-Interest (N_s) and as Y-axis the number of output dimensions (N_d) for the LDA. The PI is computed for one million traces corresponding to different plaintexts/keys.

methods as surveyed in [56]. In other words, it is likely that LDA models are well suited to estimate the side-channel security of an unprotected implementation and we do not expect significant gaps on this side either.

We therefore conclude that our evaluation of the STM32F4 AES coprocessor integrated in a LR-PRF is overall more confident than the one of the STM32U5 target.

2.4.3 Protected AES Core (STM32U5)

In this section, we detail the methodology followed for evaluating the physical security of the STM32U5 target, together with experimental results. We begin with the profiling phase and explain how we optimized our LDA models in subsubsection 2.4.3.1. Next, we present the performances achieved by our models, and demonstrate that they lead to successful attack against an independent validation set in subsubsection 2.4.3.2. Finally, we discuss the limitations of this analysis and its potential improvements in subsubsection 2.4.3.3.

2.4.3.1 Model profiling

The protected core is based on undisclosed countermeasures possibly including masking. Yet, and as already pointed out in Figure 2.9, a first-order leakage is present in the traces. We therefore decided to estimate Gaussian templates in a linear subspace optimized thanks to LDA, for the informative time samples (located around the 5 peaks of the figure).

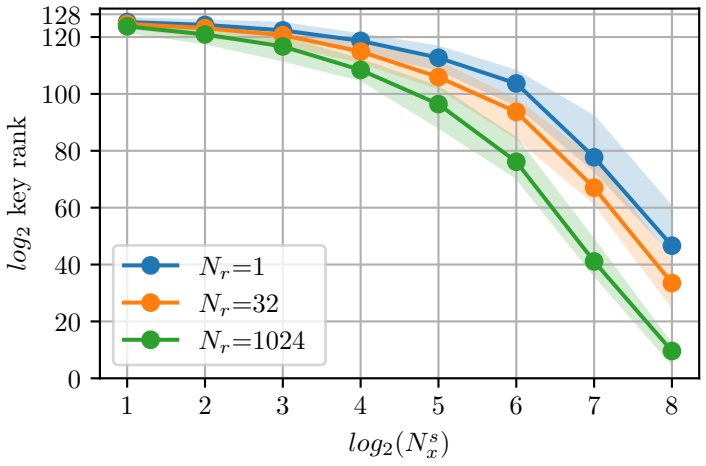


Figure 2.13: Median and quartiles of the log key rank estimated on the training set for the unprotected AES coprocessor of the STM32F4 (estimated from 100 independent attacks).

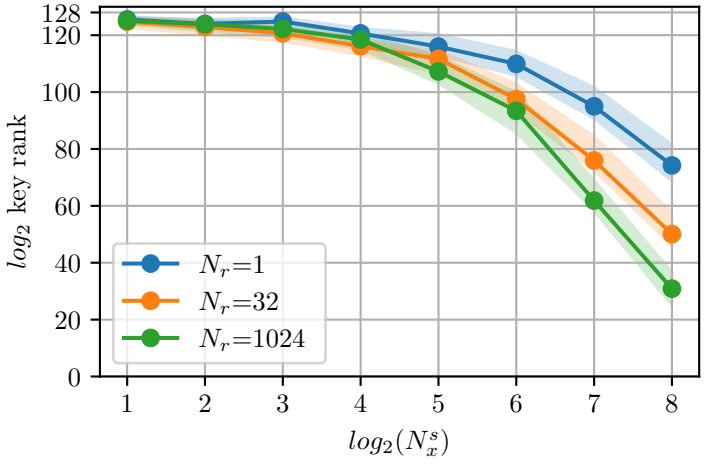
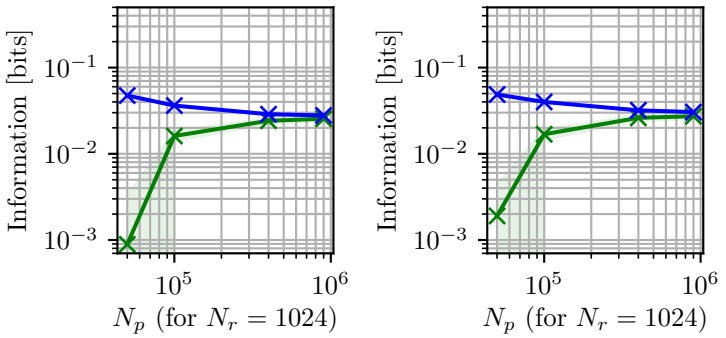
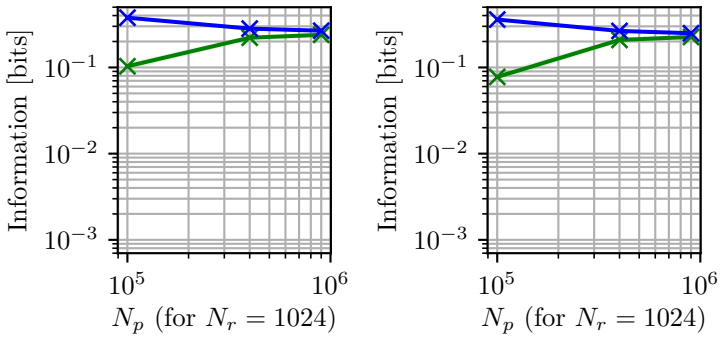


Figure 2.14: Median and quartiles of the log key rank estimated on the test set for the unprotected AES coprocessor of the STM32F4 (estimated from 100 independent attacks).

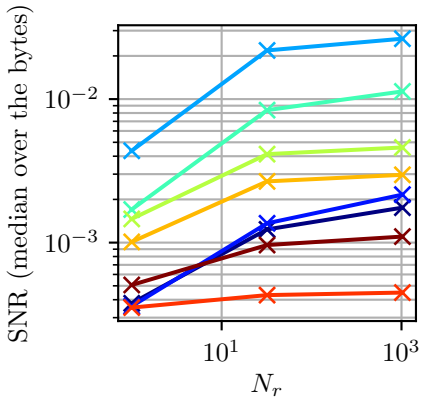


(a) STMF415, $N_r = 1$.

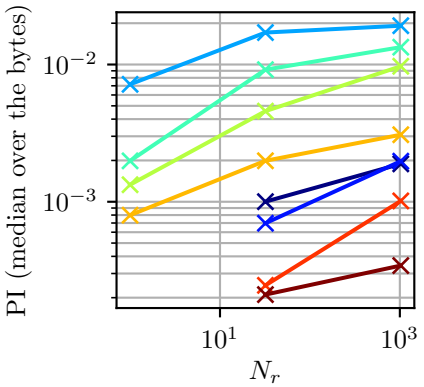


(b) STMF415, $N_r = 1024$.

Figure 2.15: PI (green) and TI (blue) for the most informative bytes on the STM32F4.



(a) Impact of N_r on the SNR.



(b) Impact of N_r on the PI.

Figure 2.16: Saturation of the median SNR (resp., PI) metrics when increasing N_r, for relevant POIs (resp., sets of POIs) of the STM32F4. Colors are the same as in Figure 2.9.

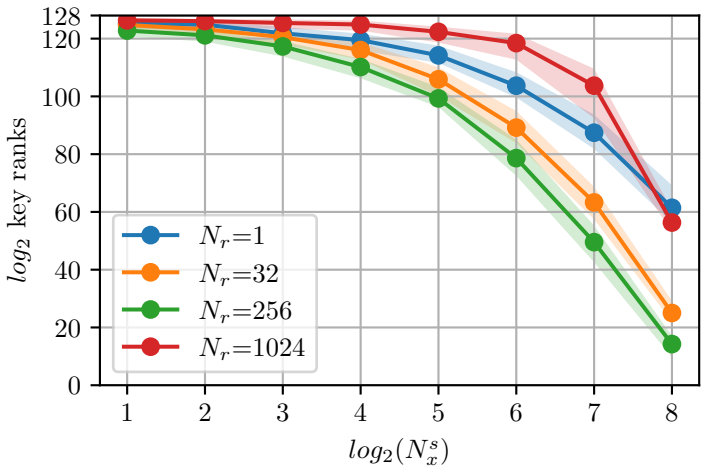


Figure 2.17: Median and quartiles of the log key rank estimated on the attack set for the unprotected AES coprocessor of the STM32F4 (estimated from 100 independent attacks).

As usually observed with the estimation of multidimensional models, the choice of good parameters is particularly important. The standard way to find them is to explore the parameters' space in a somewhat exhaustive manner, to evaluate the resulting models, and to retain the best ones. In our case study, we evaluate the quality of a model based on its PI (i.e., the amount of information that can be extracted with it) computed thanks to k-fold cross-validation, so that the traces used for profiling a model and the ones used for testing this model are always independent (to prevent overfitting). The information values obtained for all the models constructed with the considered parameter space are summarized in Figure 2.18, where the white boxes denote negative PI values (i.e., models that are not informative). This figure leads to the following main observations.

First, and most importantly, it highlights that we are able to estimate informative models for all the key bytes, confirming the presence of exploitable leakage hinted at by the previous SNR-based investigations. Such models should therefore enable successful attacks, which we will confirm in the next section. Second, it is noticeable that the best models are in quite low-dimensional subspaces (i.e., $N_d \in [1, 2]$) and use a lot of POIs (i.e., large N_s values). This may suggest that the month of measurements used for profiling is not sufficient to fully characterize the information leakage of the target, which we will discuss in Section subsubsection 2.4.2.4. Third, Figure Figure 2.18 shows some column-based patterns (e.g., a more informative second column), which is consistent with the previous SNR plots, potentially suggesting some architectural feature making these bytes (slightly) more leaky.

Remark. The PI plots of Figure 2.18 are for average traces with $N_r = 1024$. The motivation of this averaging is only to reduce the time and memory complexity of our (already expensive) attacks. However, it may be that averaging is not an optimal strategy to exploit the leakage of the STM32U5 target (e.g., if there is an informative second-order leakage).

2.4.3.2 Security evaluation

We confirmed the previous observations by running concrete attacks using the models with the highest PI for all the key bytes, identified thanks to Fig. Figure 2.18. To this end, we captured an attack set corresponding to random plaintexts and a fixed key, with a repetition factor of 1024. In order to emulate more realistic attack conditions, the collection of the attack set was not interleaved with the one of the training and test sets. As illustrated in Fig. Figure 2.19, we can nevertheless perform successful key recoveries in this context. Namely, we observe that the median rank drops below 80 bits of side-channel security after 10,000 average traces. This number is further reduced to 5000 traces if we target 96 bits of security.

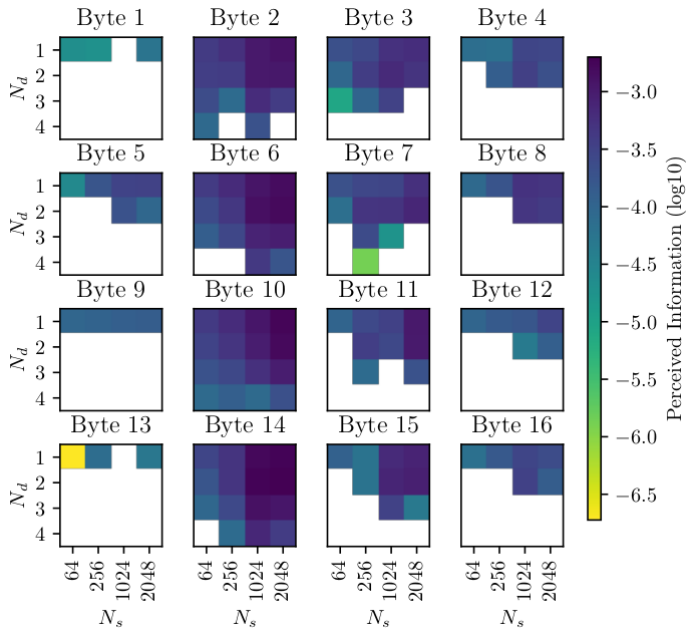


Figure 2.18: PI of the STM32U5 target for all the bytes and the whole parameter space with $N_r = 1024$, displayed in log scale. As X-axis we display the number of selected Points-of-Interest (N_s) and as Y-axis the number of output dimensions (N_d) for the LDA. The PI is computed for three million traces corresponding to different plaintexts/keys.

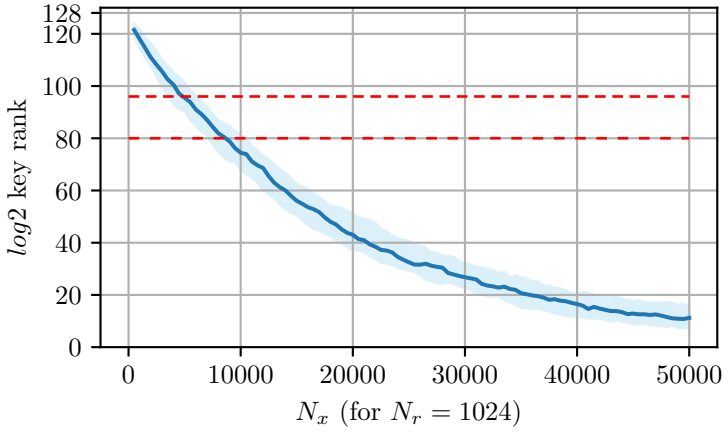


Figure 2.19: Median and quartiles of the key ranks for 100 independent attacks on the STM32U5 target (attack dataset computed after the interleaved training/test sets). Red horizontal lines correspond to target security levels / key ranks of 2^{80} and 2^{96}.

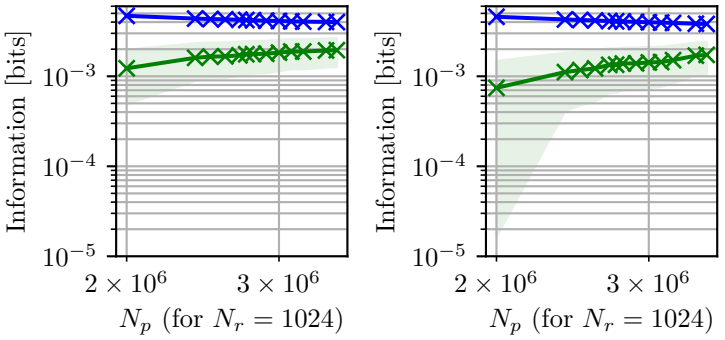


Figure 2.20: PI (green) and TI (blue) for the most informative bytes on the STM32U5.

2.4.3.3 Risk assessment

Given the time constraints of evaluators, an important part of any security evaluation is to assess the risk of security overstatements. In the present case, this risk is admittedly high as we only demonstrate a first baseline attack, and the identification of better attacks is delayed by the closed-source target and non-worst-case profiling conditions.

More precisely, one generic source of risk naturally relates to the measurement setups. Improving the setups may lead to higher SNRs, more informative leakages and better attacks. A more specific source of risk relates to the statistical processing of the measurements, which leads to two main questions. First, can we improve the models of Fig. Figure 2.18 with more profiling efforts (in data and time)? In order to answer this question, Fig Figure 2.20 illustrates the convergence of the PI and the TI bound of our LDA models, for two exemplary bytes. Clearly, these models have not converged yet (especially the byte in the left plot) and more profiling should therefore lead to better attacks. Second, can more advanced statistical tools such as surveyed in [56] lead to stronger attacks? While a formal/definitive answer to this question is out of reach, the fact that the LDA models we leverage only exploit first-order signal for a presumably masked implementation at least leaves room for advanced models that may or may not improve the attack results.

2.4.3.4 Backwards analysis

We conclude our investigations by analyzing a difference in the way we estimated security for our two targets. Namely, the investigations of subsubsection 2.4.3.2 were based on more realistic attack conditions, in which the collection of the attack set was not interleaved with the one of the training and test sets By contrast, the attacks in this section were performed on the training set (as a bound) and on a test set collected in an interleaved manner. Yet, due to the strong incentive to increase the averaging factor N_r, one can wonder whether the very accurate models estimated on strongly averaged traces could lead to robustness issues. That is, as put forward in [59], it may happen that in case of discrepancies between a profiled model and the attack traces, keeping the traces (slightly) noisy can make them more robust to variations of the measurements' conditions. Informally, this is because the gap between the profiling sets and the test sets that could cause model overfitting could then be covered by a larger noise in the model.

We provide a preliminary investigation of this question with Figure 2.17, where the attacks against the STM32U5 target are performed on an independent attack dataset, of which the collection was not interleaved with the one of the training and test sets. It clearly illustrates that moving from $N_r = 256$ to $N_r = 1024$ affects the attack negatively in this more realistic scenario. Such an observation essentially backs up the backwards evaluation approach and shows that worst-case evaluations can come with mild additional security margins due to the powerful adversarial conditions they consider. Yet, the results of Figure 2.17 do not fundamentally affect our conclusions, neither qualitatively nor quantitatively. Hence, they essentially confirm that worst-case security evaluations as a useful shortcut to reach more confident

conclusions on the side-channel security of a cryptographic implementation. It is an interesting open problem to investigate whether and how much robust profiling can cope with this model variability issue for larger N_r values [27, 78].

2.5 Lessons Learned towards Sustainable Security

In this section, we take a step back to reflect on the numerous security assessments we have conducted. We emphasize that security is a critical concern—one that can sustain irreversible damage if not properly addressed.

First, we revisit the key challenges we encountered in addressing physical security. For each of these, we outline the lessons learned to help build a more sustainable and resilient security approach.

In the second part, we take the point of view of the REWIRE project, which requires, not only a block cipher, but a full cryptographic primitive (i.e. authenticated encryption), achieved through a mode of operation. We show that both of the considered solutions are suitable for software update, with some pros and cons that are embedded in a qualitative discussion.

2.5.1 The challenges in achieving physical security

Since physical security can be achieved through two main concepts, namely *masking* and *mode-level countermeasure*, we will talk through the two of those.

Masking with physical defects remains a significant challenge, particularly for commercially viable products. The previously detailed work of subsection 2.4.3 illustrates this point through a real-world analysis of a commercially available protected core. Masking places considerable demands on the implementation side, requiring a high level of expertise and meticulous attention from the engineering team. This expertise translates into higher non-recurring engineering (NRE) costs and increases the risk of security flaws if not properly managed.

From a more theoretical perspective, several studies have shown that inherent physical effects can severely undermine physical security. For example, glitches—natural phenomena in digital circuits—have been demonstrated to compromise the independence assumptions that masking relies on [53]. As a result, the security benefits of masking can be completely negated. Similarly, our observations reveal that signal transitions, which occur when one value is replaced by another, can leak information about the unmasked value [25].

Those elements paint a challenging picture for secure implementation of cryptographic algorithm when using masking. However it also comes with a rich and growing field, being made of verification tools, mathematical proven gadget, etc.

Physical security evaluation of the LR-PRF can effectively be reduced to evaluating the physical security of an unprotected AES core. This stems from the fact that the pseudo-random function (PRF) constrains the number of observable leakages to those of a classical AES implementation. While this simplification might initially be perceived as a weakness, we demonstrated in [75] that it actually enables a tighter and more meaningful security evaluation. This leads to a more refined risk assessment:

- Profiling saturation is easier to achieve as the leakages follow relatively simple statistical distributions. Estimating means and variances (or, in the multivariate case, multiple means and covariances) is a natural fit for such data. As a result, we can expect more accurate and reliable security evaluations, since the statistical models used in attacks better reflect the actual leakage behavior.
- High-resolution measurements could expose severe leakages, potentially compromising the implementation with only a few traces. While this risk should not be underestimated, it is important to

	Primitive-level countermeasure (e.g. Ascon)	Mode-level countermeasure (e.g. LRBC2)
Implementation	High	Low
Verification	High	Low
Security tightness	Hard to assess	Easy to asses

Table 2.4: Comparison of the two approaches for protected implementations.

note that such advanced measurement setups would similarly threaten protected cores. Furthermore we highlighted that the absence of noise does not necessarily imply a lack of security—our results show that highly averaged models can suffer from over-fitting, leading to reduced attack efficiency.

- Potential impact of averaging must also be considered. On the one hand, averaging can suppress noise and help extract meaningful leakage; on the other, it can blur subtle but critical signal characteristics, making it harder for attackers to build robust profiling models. Our findings suggest that in the LR-PRF setting, excessive averaging can in fact hinder attack performance by amplifying model mismatch and reducing generalization.

Overall those trends reflect on Table 2.4 : since the LR-PRF construction has a low-level of complexity, its implementation and verification remains straightforward. For the same reason, few points of failure are present and the security level is tight with respect to the theoretical expectations.

2.5.2 Sustainable security for REWIRE

Leveraging a cryptographic primitive in a leveled implementation is a promising strategy, exemplified during the recent NIST Lightweight Cryptography competition through the selection of Ascon. This mode of operation, known as levelling, combines a strongly protected primitive with unprotected components to enable lightweight message processing. Both countermeasures discussed earlier—masking and LR-PRF—can be effectively employed in such a leveled design. The comparative discussion we propose here is summed up in Table 2.4.

In the context of REWIRE, the following implementation strategies may be considered:

An LR-BC-2 implementation, using the LRPRF. Based on innovative and convincing ideas [19] and supported by heuristic validation [75], the LR-BC-2 algorithm is a well-suited mode of operation for secure software updates. Its key advantages include a logical flow embedding the security countermeasures in the algorithm specifications without additional constraints.
Therefore, it requires minimal implementation expertise, reducing the likelihood of introducing points of failure. At the same time, its simplicity makes the system easier to verify, which facilitates the identification of any potential weaknesses. Ultimately, this results in a low-cost and reliable solution.
However, this approach does not come without drawbacks. It remains relatively underexplored in the literature, and its resilience against other types of attacks—such as fault injection—still needs to be thoroughly investigated.
A notable and inherent limitation is its “always-on” security countermeasure, which leads to a constant overhead in latency. While embedding security directly into the design specification enhances robustness, it also means that protection mechanisms are continuously active, regardless of context.
An Ascon implementation, using the HPC4 masked gadget. Alternatively, one may opt for a solution that relies more heavily on implementation-level countermeasures. In this case, our analysis clearly shows that overlooking physical defects is not a viable option. Therefore, we strongly recommend using a masking scheme—such as the HPC4 gadget—specifically designed to withstand the impact of these hardware-level effects.

However the link with Table 2.4 is here also really straightforward. Since it requires high expertise on the implementation-level, many points of failure exist and may compromise the software update process. In a logical continuity, a system that is difficult to verify may show some undetected flaws and merging a difficult to implement and verify solutions makes it hard to ensure security tightness over it.
As a final and cautious remark, we note that this consideration equally applies to our proposed solution—the HPC4 masked gadget. Being relatively new, it will need to withstand the test of time to validate its properties, and even more time will be necessary before real-world implementations can instill full confidence in its reliability and effectiveness.

Chapter 3

Compositional Verification and Validation of REWIRE Attestation Protocol

As detailed in D2.2, the **Compositional Verification and Validation** component is a core aspect of the REWIRE framework, whose functionality entails the use a variety of **formal verification tools** in order to verify the protocols and schemes employed by the devices, before they are synthesized and deployed to the actual devices prior to their execution. The purpose and key motivation behind this process is *the identification of the **trust boundary** of the system, i.e., the categorisation of device properties into those that can be formally verified during design-time, and those that cannot be verified during design-time and need to be abstracted in order to be verified during runtime using the available security enablers (e.g., attestation)*. In this regard, a core innovation of REWIRE is the minimisation of the set of properties that need to be verified during runtime. This approach serves to both facilitate the convergence of the formal verification processes during design-time, and to increase the efficiency of the attestation process during runtime.

With regards to the verification of the protocols employed by REWIRE, note that this has already been performed for the LR-BC-2 protocol and documented in D3.2, and the functional verification of the REWIRE protocols using the AADL modelling approach has been documented in Chapter 5. In this Chapter, we focus on the **Formal Verification of the attestation protocols of REWIRE**, which are responsible for verifying the configuration correctness of devices. Note that, while runtime trust assessment falls outside the scope of REWIRE, the attestation capabilities of REWIRE can be considered as a trust extension that can provide input to a Trust Assessment framework for assessing the trustworthiness level of a device during runtime. As detailed in D2.2, *REWIRE is one of the first projects of its kind to perform Formal Verification of Configuration Integrity Verification (CIV) with local (implicit) attestation*, which relies on **key restriction usage policies** and does not require an external Verifier. Specifically, this refers to the type of attestation where the Prover device does not need to share any actual configuration information or attestation evidence with the Verifier in order to prove its correctness, but its ability to sign the attestation challenge is predicated on the fulfilment of the key restriction usage policy if and only if the device is in a correct state. Thus, the act of signing the challenge is in and of itself proof of correctness (*attestation-by-proof*). In addition, as the security enablers of REWIRE are **agnostic to the underlying RoT**, the Formal Verification methodology of REWIRE entails the consideration of **perfectly secure crypto** provided by the RoT. This approach enables REWIRE to leverage this property and translate it to the overall application security based on the cryptographic capabilities provided by such RoTs.

Thus, in the context of the Compositional Verification and Validation activities of REWIRE, *cryptographic protocols can be viewed as concurrent programs that communicate over public channels in a system to achieve specific security goals*. **These channels are assumed to be under the control of a so-called "attacker" with "Dolev-Yao" capabilities** (also implicitly representing the behaviour of dishonest participants), that, in the most general scenario, has full access to the communication channels and can intercept, modify, delete, or inject messages at will. In the standard "symbolic" model adopted by REWIRE, cryptography is considered perfect as aforementioned, meaning that primitives are treated as mathe-

matical functions that perform according to their specifications, without implementation weaknesses or vulnerabilities; this implies that the attacker is limited to only applying the primitives explicitly provided by the system, for example, manipulating data and decrypting messages, having acquired or derived the necessary keys.

The adoption of formal methods in the specification and verification of security cryptographic protocols yields multiple benefits. First, **by encoding the protocol in a formal language, a precise documentation of the protocol behaviour is obtained**, that prevents ambiguities, contributes to detecting logical issues, and serves as implementation guidance. Second, **applying formal verification techniques ensures mathematical rigour in proving security properties**, such as *authenticity* and *confidentiality*, identifying potential flaws, already in the specification phase, that might not be apparent through conventional testing during development.

3.1 Tools and Languages

In the field of formal verification of security protocols, the two most appreciated and renown frameworks are ProVerif[1] and Tamarin[2], that, in the course of the last twenty years, have been successfully applied to a number of protocols, including TLS, ARPKI, Needham-Schroeder [66]. They specify available primitives (i.e., fundamental actions that a programming/modelling language directly supports) via equational theories, and rely on different deductive systems, namely Horn clause-based reasoning in first-order logic, and multiset rewriting rules; the ability to address reachability, correspondence assertions, observational equivalence problems, allows to encode and verify security properties like confidentiality and authenticity, possibly in presence of multiple sessions of the protocol.

With regards to **ProVerif**, its operation is based on specifying the behaviour of the participants in applied **pi-calculus** where the exchanged messages are algebraic terms. The most commonly supported properties that can be checked in this regard are **reachability** (used to check secrecy), **correspondence** (used to check authenticity), and to a lesser extent **equivalence** on different protocol runs. Internally, ProVerif functions similarly to a theorem prover, where pi-calculus processes are translated into Horn clauses, and the tool attempts to prove the properties assuming an unbounded number of sessions (i.e., executions of the protocol to be verified). An attempt to prove a given property may produce three outcomes: (i) **property is proven**, (ii) **property does not hold**, and (iii) **no result** (outcome is unknown and the check does not terminate). In the next phase, ProVerif produces a trace with a protocol run demonstrating identified attacks. *Lack of termination is a possible outcome due to the complexity of the protocol and the underlying proof procedure, which is generally undecidable on an unbounded number of sessions and inputs*.

Regarding **Tamarin**, the protocol to be evaluated is modelled in terms of **multi-set rewriting rules** and, as such, Tamarin belongs to a larger family of tools based on rewriting logic. The properties are specified as formulas in a fragment of **First-Order Logic (FOL)** and are checked on the traces derived from protocol runs. Similarly to ProVerif, cryptographic protocols in Tamarin are specified in equational theories. Internally, Tamarin uses a **constraint-solving algorithm** for falsification and verification of properties, assuming an unbounded number of sessions. Also, as in ProVerif, non-termination is a possible outcome of the verification process.

Considering the above for both ProVerif and Tamarin, since demonstrating the **correctness of a security protocol** (intended as *the validity of properties w.r.t. its encoding in first-order logic*) **is in general undecidable**, the tools may not terminate on a given verification task. In addition to that, the use of **abstraction mechanisms** may yield a “don’t know” answer, due to the approximations performed during the analysis. Despite this, the tools are sound: if a property is claimed to be valid or invalid, then the answer is reliable; in the first case, a **proof of correctness** is possibly returned, while, in the second case,

[1] https://bblanche.gitlabpages.inria.fr/proverif/
[2] https://tamarin-prover.com/

an **execution trace** is provided, to represent an *attack that violates the target property*. However, **both ProVerif and Tamarin also present additional challenges with regards to their usability**, specifically learning and mastering their specification languages. Specifically, security protocol engineers tend to think in terms of message sequences exchanged among the protocol participants (e.g., the classic Alice-Bob representation for cryptographic operation) and to specify the content of messages based on a set of well-known cryptographic functions (e.g., hash and key derivation functions, symmetric encryption), but **the translation of such concepts into ProVerif and Tamarin representations is not straightforward and may be quite challenging**.

In order to address the aforementioned usability issues, **VerifPal**[3] was introduced, whose inner workings resemble ProVerif, and functions with symbolic models. **VerifPal aims to provide a simpler syntax that closely resembles the Alice-Bob representation**, and aims to **provide the most used crypto primitives as first-class language constructs**. Thus, the rationale behind the development of Verifpal was the need for **usability** above other characteristics. ProVerif and Tamarin, although being extremely sophisticated tools, are difficult to approach by those without a background in formal methods, both in developing protocols, and in analysing counterexample execution traces, looking for the root cause of a property violation. However, in addition to that, VerifPal possesses a variety of other benefits that make it more appropriate for adoption in REWIRE compared to ProVerif and Tamarin, which we expand upon in the following.

While not as expressive as that of ProVerif or Tamarin, the Verifpal language offers a **straightforward syntax**, immediately **supporting a range of common cryptographic operations**, including **symmetric and asymmetric encryption**, **hashing and signature primitives**, **means for composing and decomposing messages**, without the need for additional encoding. A protocol is formalized as an interleaving of local computation actions by the participants and of message exchange across a public communication channel. As for primitives, a collection of **predefined security properties** is made available, associated with the data either locally derived or shared (details in the following sections). *When a property is found invalid, a trace is returned that exemplifies the violation and the actions carried out by the attacker and the participants*.

Overall, at the cost of a less powerful reasoning system than ProVerif or Tamarin, this approach facilitates the modelling and understanding of cryptographic protocols by engineers and security practitioners, without sacrificing the rigour of analysis. In addition, considering that the REWIRE CIV scheme intentionally uses **Elliptic Curve Cryptography (ECC)**-based schemes in order to implement more efficient attestation that needs to run continuously in the devices, the level of abstraction provided by Verifpal is appropriate for these crypto operations, thus allowing us to focus on the attestation-related aspects of the scheme outside the crypto itself. Thus, **Verifpal not only provides a higher level of usability in the representation of crypto operations**, but also offers a higher level of **abstraction** and **flexibility** for capturing complex cryptographic functions compared to ProVerif and Tamarin, at the cost of some granularity in the modelling process (as, for instance, they may be able to better represent states capturing a complex leakage condition). In addition, as described in D3.2, we assume that the underlying Root-of-Trust (RoT) of the device provides **perfectly secure crypto**, thus enabling us to leverage the abstractions provided by tools adopting the symbolic approach, such as VerifPal, reducing the complexity of the model even further.

3.1.1 Security Properties

Verifpal provides the user with a collection of predefined standard security properties, which will be used throughout this Chapter in order to perform the Formal Verification of the REWIRE CIV scheme. These properties are the following:

- Confidentiality: applies to a piece of data m; the property is that the attacker cannot obtain m.

[3] https://Verifpal.com/

- Authentication: applies to a message m sent from a sender S to a recipient R; the property is that R is able to use m in an operation, such as for signature verification or authenticated decryption, if and only if R received m from S.
- Freshness: applies to a piece of data m, assuming the possibility of multiple protocol sessions; the property is that m cannot be reused in multiple executions by the attacker, since m depends on information that is freshly generated at each execution.

3.1.2 Threat Model

Here, we provide some details on the elements of the threat model supported by VerifPal, and will provide the basis for the analysis scenarios to be defined in 3.3.2. Specifically, as will be demonstrated, the REWIRE methodology entails following a layered approach, based upon the consideration of an attacker with increasing capabilities. In this regard, we leverage the two attacker modes envisioned by the VerifPal framework:

- A *passive* attacker is a malicious observer of the communication channel, that cannot inject or modify messages, but can derive information based on messages passing through the channel.
- An *active* attacker cannot directly affect data locally computed by a protocol participant (e.g., the trace generated by the Tracer in Fig. 3.4), but can arbitrarily modify messages sent through the channel, and inject their own crafted messages; it can start an unbounded number of protocol executions, and reuse knowledge gained across executions (except for data freshly generated).

Although an attacker cannot directly access a participant's local data, it is possible to explicitly introduce leakage; constraints can also be placed to an active attacker's capabilities, by relying on the use of "guarded" messages, i.e., messages that can be read, but not modified, thus allowing for a number of different scenarios.

3.2 High-level Overview of the CIV Protocol

As described in D4.2, the REWIRE **Configuration Integrity Verification (CIV)** scheme aims to ensure the configuration correctness of any device participating in the service graph chain. A core consideration behind the design of the REWIRE CIV scheme is the ability to operate in a **zero-knowledge manner**, i.e., to enable a device to respond to an attestation challenge without disclosing any information regarding its internal structure or the traces themselves. This is achieved through the use of **Key Restriction Usage Policies**, which predicate the use of the **Attestation Key (AK)** on the correctness of the device state. Thus, the device is only able to create a valid signature on the attestation challenge using the AK if and only if it is in a correct configuration state. Note that one key innovation of REWIRE is the use of a **Verifiable Policy Enforcer (VPE)**, which is responsible for verifying that the correct key restriction usage policy is enforced, and that a malicious entity is not attempting to use an invalid policy (e.g., a previous outdated policy which has been replaced with a new one following a software update in the device).

In the context of REWIRE, the key restriction usage policies are dictated by the **Domain Manager** and enforced by the **Security Monitor**, acting as a trusted entity that governs the overarching process, thus ensuring that a signature can only be created if the device configuration is correct, referring to both the **configuration of the hosted application** and the **configuration of the enclave handling the attestation mechanism**. Thus, the **Prover** can be verified using a simple challenge-response protocol that neither requires, nor reveals any configuration information to the **Verifier**.

As aforementioned, the REWIRE CIV scheme has been described in detail in D4.2, where we also provided detailed descriptions of all underlying mathematical processes. Here, we provide a brief high-level overview of the scheme, focused on the actions performed in the two core phases, namely the **Join phase** and the **Runtime Attestation phase**:

- **Join phase:** We first consider a TEE-capable edge device that has successfully completed the secure onboarding protocol of REWIRE, and obtained the Verifiable Credential containing the necessary attributes for granting it access to the Domain Manager. Then, the Domain Manager fetches the reference values of the application corresponding to the correct and expected configuration state of the device. These consist of both reference values of the safety-critical applications, but also the enclave applications of the Prover. Afterwards, with the assistance of the Domain Manager, the AK of the device is created, along with the Key Restriction Usage Policy it is bound to (based on the aforementioned reference values). Finally, the VPE requests from the Domain Manager the creation of the VPE Key pair, bound to the correctness of the configuration of the **Attestation Agent (AA)**, the **Tracer**, and other safety-critical artefacts hosted in the edge device.
- **Runtime Attestation phase:** A Verifier who wishes to verify the correctness of a device participating in the REWIRE infrastructure initiates the attestation protocol by issuing an attestation challenge to the Prover device. Then, the AA and the VPE each create a random nonce, which is then signed by the Tracer. Then, the Tracer extracts the required traces as attestation evidence and signs the measurement bound both with the nonce of the AA, and the nonce of the VPE. Both signatures are sent back to the VPE, and if the Key Restriction Usage Policy is verified, it signs the accepted software version of the AA, bound with the nonce issued by the AA. Finally, the AA also verifies the policy, and if successful, the AK is used in order to sign the challenge, thus completing the attestation process.

In the following, we focus on the modelling and verification approach for the REWIRE CIV scheme as described in this Section, using the Verifpal tool which has been selected for the verification process.

3.3 CIV Modeling and Verification Approach

The protocol involves three key actors: the Prover, the device that is having its state attested; the Verifier, that challenges the Prover to remotely check the correctness of its configuration state; the Domain Manager, responsible for setting the security policies associated with the attestation key, managing the secure launch of the trusted computing base, comprising the AA and the VPE, and checking that the correct policy is enforced during runtime for predicating the use of the attestation key. In turn, the Prover includes three trusted components: the Key Manager, that manages creation and usage of the device attestation key; the Tracer, that monitors and reports runtime measurements of chosen device applications; the VPE, in charge of enforcing the security policies.

3.3.1 Modelling in Verifpal

In Verifpal, protocol participants are referred to as **principals**, which have the ability to locally create data by means of primitive operations, and to send data in the form of messages to target recipients through the public communication channel. For example, a principal in the context of the REWIRE CIV protocol may be the **AA** or the **VPE**, which have the ability to create keys and signatures, and exchange messages as part of the attestation process.

A Verifpal model appears as an interleaving of local computation blocks, associated with principals, and message exchanges; for example, Fig. 3.1 shows the Tracer generating a public key from its private part, and sharing it with the Domain Manager in the Join phase.

```
principal Tracer[
        knows private Tracer_priv
        Tracer_pub = G^Tracer_priv
]
Tracer -> DomainManager: Tracer_pub
```

Figure 3.1: Tracer Keys for Asymmetric Encryption

Thanks to the standardized cryptographic structures used by the CIV and the decision to keep the design lightweight — hence the adoption of ECC-based cryptography — modeling can rely on Verifpal's built-in operators. The CIV protocol, expressed in a structured natural language notation, employs in fact a collection of cryptographic operations, which is actually a subset of the primitives made available by Verifpal[4]; this is indeed an advantage, for the modeler, for the original protocol designer, not necessarily an expert of formal methods, and for any reviewer, since the protocol specification very closely resembles its formal encoding.
Table 3.1 reports the operations used by CIV, together with their correspondent primitives in Verifpal.

Operation	**Primitive**	**Description**
$K_{pub} = K_{priv} * G$	$K_{pub} = G \hat{} K_{priv}$	Generation of a private-public key pair for asymmetric encryption
$Y = H(X)$	$Y = hash(X)$	Hashing
$X \| \| Y$	$concat(X, Y)$	Data concatenation
not explicitly represented	$X, Y = split(Z)$	Data split, where $Z = concat(X, Y)$
$Y = random()$	$generates\, Y$	Generation of fresh data
$Y = KDF(K, S)$	$Y = hkdf(S, K, nil)$	Hash-based key derivation function, with salt and key arguments
$Y = ENC(X, K)$	$Y = enc(K, X)$	Symmetric encryption
$X = DEC(Y, K)$	$X = dec(K, Y)$	Symmetric decryption. Successful if $Y = enc(K, X)$
$Y = HMAC(K, X)$	$Y = mac(K, X)$	Keyed hash function
$Y = RSA\_ENC(X, K_1)$	$Y = pke\_enc(K_1, X)$	Asymmetric encryption
$X = RSA\_DEC(Y, K_2)$	$X = pke\_dec(K_2, Y)$	Asymmetric decryption. Successful if $Y = pke\_enc(K_1, X)$ and $K_1 = G \hat{} K_2$
$X? = Y$	$assert(X, Y)?$	Checked equality: execution aborted if not satisfied
$Y = SIGN(X, K_1)$	$Y = sign(K_1, X)$	Digital signature
$VERIFY(Y, X, K_2)$	$signverif(K_2, X, Y)$	Digital signature verification, checked if followed by $?$. Successful if $Y = sign(K_1, X)$ and $K_2 = G \hat{} K_1$

Table 3.1: CIV operations and corresponding Verifpal primitives.

Prior knowledge, which does not change across executions, is introduced by the "knows" keyword, and by "private" or "public": the former denotes data known only by the participants where it is declared, while the latter denotes data known by all participants, attacker included. The declaration of local data "leaks" specifies that a participant discloses those data to the attacker, at the point of leakage.
The following data was assumed to be public:

- CC: restriction policy command code;
- mr_Tracer, mr_AA, mr_VPE: respectively enclave measurements of Tracer, Attestation Agent, and VPE;
- mr_Signer: hash digest of the public key of the Security Administrator;
- $mr\_Enclave$: reference measurements of the Attestation Agent and the VPE.

The following data was assumed to be private to one or more participants:

- DM_priv: Domain Manager key, known by the Domain Manager;
- $root\_id\_priv$: device identity key, known by the Key Manager;
- $Tracer\_priv$, Tracer key, known by the Tracer;
- $runtime\_policy$: initial security policy, known by Domain Manager, VPE, Attestation Agent;

[4] A user, willing to define an operation not present in Verifpal, could take advantage of the tool capability to automatically translate a Verifpal specification into a ProVerif or Coq model, and proceed from there.

- $reference\_values$ reference measurements of the untrusted application to be attested, known by Domain Manager and Attestation Agent.

Verifpal assumes a single, public communication channel where an attacker can at least observe the exchanged messages. In contrast, the CIV framework operates under different security assumptions for the channels between various participant pairs:

- Communication between the Key Manager and the VPE, as well as between the Key Manager and the Attestation Agent, is fully secure - an attacker cannot read or alter these messages.
- Messages exchanged between the VPE and the Attestation Agent can be intercepted, but not tampered with.

To meet the second assumption, guarded messages were used between the VPE and Attestation Agent. However, to ensure confidentiality as well, an extra layer of encryption was introduced. Three symmetric sealing keys were pre-shared as private data among Key Manager, Attestation Agent, and VPE, based on the functionalities offered by the Keystone platform. The Key Manager would have access to all three keys, while each of the other two participants knew their own key and that of the Key Manager (Fig. 3.2); each message exchange between these three participants was then protected by an encryption step before transmission and a decryption step upon receipt (Fig. 3.3).

```
principal KeyManager [
        knows private aa_sealing_key
        knows private vpe_sealing_key
        knows private km_sealing_key
]

principal AttestationAgent[
        knows private aa_sealing_key
        knows private km_sealing_key
]

principal VPE[
        knows private vpe_sealing_key
        knows private km_sealing_key
]
```

Figure 3.2: Symmetric Sealing Keys

```
principal VPE[
        rand1_enc_sealed = enc(vpe_sealing_key,rand1_enc)
]
VPE -> KeyManager: [rand1_enc_sealed]
principal KeyManager[
        rand1_enc_ = dec(vpe_sealing_key,rand1_enc_sealed)
]
```

Figure 3.3: Additional Enc-Dec Step in VPE to Key Manager Communication

Despite CIV comprising two phases, it was modeled as a single protocol specification in Verifpal, since the second phase relies heavily on the information exchanged during the first.

3.3.2 Analysis Scenarios

The CIV scheme is a complex protocol, making it difficult to verify due to the large number of operations involved and the complex data interactions between participants; the range of possible attacker's actions is vast, especially when allowing for an unbounded number of protocol executions.

To manage this complexity, an incremental verification strategy was chosen, referred to as a **layered verification approach**, consisting of gradually exploring scenarios of increasing difficulty. Specifically, it begins with a "passive" scenario, where the attacker's capabilities are minimal, and only has the ability to read messages and not to modify messages or inject malicious information to benign messages. Then, the layered approach advances to an "active" attacker, who has the capability to modify or perform injection in transmitted messages, and whose actions are restricted by the use of guarded messages. The rationale is to evaluate the protocol resilience against progressively more sophisticated security threats,

starting with the disclosure of messages content, as they are transmitted, to then allow message tampering and knowledge reuse across sessions. This process continues until the point where the verification becomes too resource-intensive, and Verifpal is unable to complete the analysis within a reasonable time limit.

3.3.3 Properties of Interest

In this work, we focused on checking a number of security properties, that the protocol developers had highlighted as important:

- **Confidentiality** of $rand1$ and $rand2$, the nonces generated by the Domain Manager and used to encrypt the two secrets, respectively sent to the VPE and the Attestation Agent.
- **Freshness** of $trace\_hash$, runtime measurements of the untrusted application to be attested, taken by the Tracer.
- **Authenticity** of the *Ticket* issued by the Domain Manager and used by the Attestation Agent. This property needed to be expressed in a different form. On one side, in the protocol, the Ticket is combined with other data by the Domain Manager to create $secret2$, which is sent encrypted to the VPE, and in turn forwarded to the Attestation Agent as a guarded message; on the other side, in Verifpal (see Section 3.1.1) authenticity properties are associated with a single message exchange from a sender to a recipient, that uses the message in an operation such as decryption or signature verification. For these reasons, the target property was encoded as authenticity of $secret2\_enc$, upon reception by the Attestation Agent from the VPE.
- **Authenticity** of $s1$ ($\sigma 1$ in the original protocol), created and signed by the Tracer, sent to the VPE and forwarded to the Attestation Agent, that performs signature verification. $s1$ is obtained by combining the $trace\_hash$, the $nonce\_AA$ (freshly generated by the Attestation Agent and sent to the VPE and in turn to the Tracer, see Fig. 3.5), and the CC, and then applying a digital signature with the Tracer's private key.

Properties were verified both in passive and active scenarios, also introducing leakage of the trace generated by the Tracer in the Attestation phase, to evaluate its impact on the protocol security (Fig. 3.4).

```
principal Tracer[
        generates trace
        trace_hash = hash(trace)
        s1 = sign(Tracer_priv,hash(concat(CC,nonce_AA,trace_hash)))
        s2 = sign(Tracer_priv,hash(concat(CC,nonce_VPE,mr_Tracer)))
        leaks trace
]
```

Figure 3.4: Trace Leakage by the Tracer

```
principal AttestationAgent[
        generates nonce_AA
]

AttestationAgent -> VPE: [nonce_AA]

principal VPE[
        generates nonce_VPE
]

VPE -> Tracer: nonce_AA,nonce_VPE

principal Tracer[
        generates trace
        trace_hash = hash(trace)
        s1 = sign(Tracer_priv,hash(concat(CC,nonce_AA,trace_hash)))
        s2 = sign(Tracer_priv,hash(concat(CC,nonce_VPE,mr_Tracer)))
]
```

Figure 3.5: Creation of $s1$ by the Tracer

3.4 CIV Verification Results

The verification of the CIV scheme followed the layered approach outlined in Section 3.3.2, which aims to address the complex and vast range of potential attacker's actions that may target the attestation process. This approach incrementally considers scenarios of increasing difficulty, starting from an attacker with limited capabilities (passive) and building up to a stronger attacker with increased capabilities.

Given the above, the CIV scheme was analysed in the following scenarios, where each of the security properties outlined in Section 3.3.3 was subject to verification:

1. **Passive**, with and without leakage. Specifically, the case without leakage refers to an attacker not having access to any data local to protocol participants, while the case with leakage refers to an attacker being able to read local data marked as leaked - in this setting, the trace generated by the Tracer.
2. **Fully active**, which refers to an attacker that is able to intercept, modify, and inject malicious information to a transmitted message.

The experiments were run on a Linux server, equipped with four 2GHz Intel Xeon CPUs, and 251 GB of RAM.

Passive. In the passive scenario, characterized by a passive attacker, all the properties were proved valid in a fraction of a second.

Property	Time	Result
$rand1$ confidentiality	$< 1s$	Valid
$rand2$ confidentiality	$< 1s$	Valid
$trace\_hash$ freshness	$< 1s$	Valid
$secret2\_enc$ authenticity	$< 1s$	Valid
$s1$ authenticity	$< 1s$	Valid

Table 3.2: Passive Scenario Results

The same results were obtained in the extended passive scenario, where the attacker has the same capabilities as in the passive scenario, and the additional knowledge of leaked data; as expected, the introduction of leakage did not affect the freshness property, and it was not sufficient for a passive attacker to compromise confidentiality or authenticity.

Property	Time	Result
$rand1$ confidentiality	$< 1s$	Valid
$rand2$ confidentiality	$< 1s$	Valid
$trace\_hash$ freshness	$< 1s$	Valid
$secret2\_enc$ authenticity	$< 1s$	Valid
$s1$ authenticity	$< 1s$	Valid

Table 3.3: Passive Scenario With Leakage Results

Active. Remember that, in the active scenario, the attacker has significant freedom of action: it can read, alter, and inject messages transmitted over the channel. By applying primitive operations, it can create new data and infer knowledge, to be exploited while executing multiple protocol sessions.
The attestation protocol exhibits a high level of complexity, both in terms of participant communication and inter-phase dependencies. This complexity is mirrored by the wide range and complexity of actions

available to an attacker during protocol execution. As a result, the verification task proved too demanding for the Verifpal tool, which failed to complete 4 out of 5 property checks within a 30-day time limit.[5]

Property	Time	Result
$rand1$ confidentiality	$43200m$	Timeout
$rand2$ confidentiality	$43200m$	Timeout
$trace\_hash$ freshness	$43200m$	Timeout
$secret2\_enc$ authenticity	$43200m$	Timeout
$s1$ authenticity	$< 1s$	Invalid

Table 3.4: Active Scenario Results

Nevertheless, the analysis of the fifth property — namely, authenticity of $s1$ upon its reception by the Attestation Agent — revealed a critical aspect of the protocol. Verifpal declared the property invalid and produced a counterexample, illustrating a specific sequence of attacker's actions that leads to its violation.

Fig. 3.6 presents a simplified version of the protocol, including only the local computations and message exchanges relevant to this property, while Fig. 3.7 displays the resulting counterexample.

```
principal Tracer[
        knows public CC
        knows private Tracer_priv
        Tracer_pub = G^Tracer_priv
]

principal AttestationAgent[
        generates nonce_AA
]

AttestationAgent -> VPE: [nonce_AA]

VPE -> Tracer: nonce_AA

principal Tracer[
        generates trace
        trace_hash = hash(trace)
        s1 = sign(Tracer_priv,hash(concat(CC,nonce_AA,trace_hash)))
]

Tracer -> VPE: s1,trace_hash

VPE -> AttestationAgent: [s1],[trace_hash]
Tracer -> AttestationAgent: Tracer_pub

principal AttestationAgent[
        signed_digest1 = hash(concat(CC,nonce_AA,Trace_hash))
        _ = signverif(Tracer_pub,signed_digest1,s1)?
]
```

Figure 3.6: Simplified Protocol Encoding

In this context, the simplification process can be viewed as an abstraction technique: filtering out local computations irrelevant to the target property (e.g. hashes, signatures, encryptions) does not compromise protocol security; moreover, reducing the amount of exchanged data limits what the attacker can intercept and exploit. Provided the filtered protocol remains executable, any counterexample found in the simplified model implies the existence of a corresponding counterexample in the full protocol.

The counterexample returned by Verifpal describes the following attacker's behavior:

1. The attacker computes its own private-public key pair $k, G\hat{}k$, as a substitute for $Tracer\_priv$ and $Tracer\_pub$;

2. The attacker computes its own trace hash $hash(t)$, where t could be, e.g., a previously leaked trace, here replayed, or a counterfeited one;

[5] A possible direction for future work is to experiment with abstraction techniques. Unlike the filtering method discussed later, which limits the attacker's knowledge and actions so that any counterexample in the filtered model also appears in the full one, abstraction requires the opposite guarantee: if a property is proven in the abstract model, it must also hold in the original one. This is challenging because the abstraction must preserve every conceivable attacker's behaviour, particularly across multiple sessions. One approach is to show that, from the attacker's perspective, the concrete and abstract models are indistinguishable.

Figure 3.7: Counterexample to $s1$ Authenticity

3. The attacker intercepts $nonce\_AA$, when sent by the VPE to the Tracer;
4. The attacker replaces $trace\_hash$ with $hash(t)$, when sent by the Tracer to the VPE;
5. The attacker replaces $Tracer\_pub$ with $G\hat{}k$, when sent by the Tracer to the Attestation Agent;
6. The attacker computes its own $s1' = sign(k, hash(concat(CC, nonce\_AA, hash(t))))$ (CC is publicly known);
7. The attacker replaces the original $s1$ with the forged $s1'$, when sent by the Tracer to the VPE.

The outcome of the attacker's actions is a successful signature verification of $s1'$ by the Attestation Agent, that accepts any trace hash injected by the attacker as legitimate.

The counterexample was reported to the protocol developers and analysed collaboratively. The underlying issue was traced back to an aspect not adequately accounted for during the protocol encoding, that is, the unprotected transmission of $Tracer\_pub$ from the Tracer to the Attestation Agent, which enabled the attacker to substitute it with their own public key. Based on this observation, the protocol encoding in Verifpal was revised: $Tracer\_pub$ was encrypted with a pre-shared symmetric key, prior to transmission (Fig. 3.8).
This update proved sufficient to prevent the attack, and, in a subsequent verification run, the property was successfully validated (Fig. 3.9). In our view, this represents a successful example of collaboration between specialists in cybersecurity and formal verification.
The solution presented so far revealed a discrepancy between the protocol encoding and the specification, due to an implicit assumption: the need to encrypt the Tracer's public key when sent over an unsecured channel. This was not explicitly stated in the specification, as it is typically regarded as common knowledge within the security domain.
Other feasible approaches, not relying on the Tracer's key encryption, would require a refinement of both the protocol specification and the encoding:

1. Pre-sharing the Tracer's public key with the Domain Manager, the Attestation Agent, and the VPE, in a similar fashion to what done with the sealing keys. This method avoids key transmission entirely.
2. Pre-sharing the Tracer's public key with the Domain Manager, and the VPE. The Attestation Agent receives two copies of the Tracer's public key: one directly from the Tracer, and one indirectly from the Domain Manager, embedded in the authorization ticket signed, encrypted, and issued by the Domain Manager itself. Upon successful decryption and signature verification of the ticket, the Attestation Agent could perform a validity check of the first key, by equating it with the second one.

```
principal Tracer[
        knows private t_sealing_key
]

principal AttestationAgent[
        knows private t_sealing_key
]

principal Tracer[
        knows public CC
        knows private Tracer_priv
        Tracer_pub = G^Tracer_priv
]

principal AttestationAgent[
        generates nonce_AA
]

AttestationAgent -> VPE: [nonce_AA]

VPE -> Tracer: nonce_AA

principal Tracer[
        generates trace
        trace_hash = hash(trace)
        s1 = sign(Tracer_priv,hash(concat(CC,nonce_AA,trace_hash)))
]

Tracer -> VPE: s1,trace_hash

VPE -> AttestationAgent: [s1],[trace_hash]

principal Tracer[
        Tracer_pub_sealed = enc(t_sealing_key,Tracer_pub)
]
Tracer -> AttestationAgent: Tracer_pub_sealed
principal AttestationAgent[
        Tracer_pub_= dec(t_sealing_key,Tracer_pub_sealed)
]

principal AttestationAgent[
        signed_digest1 = hash(concat(CC,nonce_AA,Trace_hash))
        _ = signverif(Tracer_pub_,signed_digest1,s1)?
]
```

Figure 3.8: Simplified Protocol Encoding Revised

```
Deduction • Output of SIGN(nonce_aa, HASH(CONCAT(nonce_aa, nonce_aa, HASH(nil)))) obtained by reconstructing with nonce_aa, HASH(CONCAT(nonce_aa, nonce_aa, HASH(nil))). (Analysis 7627)
Deduction • Output of SIGN(nonce_aa, HASH(CONCAT(nonce_aa, nonce_aa, HASH(cc)))) obtained by reconstructing with nonce_aa, HASH(CONCAT(nonce_aa, nonce_aa, HASH(cc))). (Analysis 7645)
 Stage 6, Analysis 17000...

 Verifpal • Verification completed for 'civ_scheme_all_v3_p5_simpl_rev_act.vp' at 12:42:38 PM.
 Verifpal • All queries pass.

 Verifpal • Thank you for using Verifpal.
```

Figure 3.9: Successful Verification of $s1$ Authenticity

In Section 3.4.1, we provide further insight on this finding, as well as how it assists us in the definition of protocol specifications in the context of Formal Verification, as well as the process followed for addressing identified violations.

3.4.1 Interpreting the Violation

This section is dedicated to the counterexample presented in the previous section, which was identified by the Verifpal-based verification as an exploit which can be leveraged by an attacker in order to *forge the signature of the Attestation Agent and make any trace collected as attestation evidence appear legitimate*. This can, in turn, *make the device appear to be in a correct configuration state, even though it has been compromised by the attacker*. While this may appear as a vulnerability in the REWIRE CIV scheme, here we demonstrate that this is not the case, as it is related to the **definition of requirements and specifications in the CIV protocol**.

In order to illustrate this point, we first provide some technical details on the mathematical underpinnings of the REWIRE CIV scheme, which were originally presented in D4.2. First, it is important to note that the Tracer possesses a **private-public Tracer Key pair** ($Tracer_{priv}$, $Tracer_{pub}$) which can be used in order to sign traces collected as attestation evidence. As demonstrated in D4.2, during the **Runtime Attestation** phase of the CIV scheme, the Tracer is responsible for collecting measurements as attestation evidence from the **runtime execution of the untrusted binary to be attested**, as well as the **secure enclave**. Then, the Tracer receives two random nonces from the **Attestation Agent (AA)** ($nonce_{AA}$) and the **VPE** ($nonce_{VPE}$), and creates two signatures using the **private part of the Tracer Key**: (i) $\sigma 1$ on $nonce_{AA}$ and the traces from the untrusted binary, and (ii) $\sigma 2$ on $nonce_{VPE}$ and the traces from the secure enclave. Thus, by binding the two signatures to two distinct nonces, verification of those signatures grants both

the AA and the VPE proof of liveliness of the Tracer.

Then, both signatures are sent back to the VPE, which in turn verifies $\sigma 2$ using $Tracer_{pub}$, and is then able to decrypt its own Private Key VPE_{priv}, which had previously been issued by the Domain Manager. Using this key, the VPE then creates a new signature $\sigma 3$ on the nonce issued by the AA, and both $\sigma 1$ and $\sigma 3$ are sent to the AA for verification. Upon reception, the AA verifies $\sigma 1$ using the public part of the Tracer Key $Tracer_{pub}$, and $\sigma 3$ using the public part of the VPE Key VPE_{pub}. If both are successful, the AA is able to verify the **Authorization Ticket** which had been issued during the Join phase, recompute the private part of the Attestation Key AK_{priv}, and create its response to the attestation challenge issued by the Verifier by creating a signature on the challenge.

Thus, we are now able to better explain the violation identified in Section 3.4. Specifically, in this example, an attacker is able to target the step of the aforementioned process pertaining to the creation of signature $\sigma 1$. In this regard, the attacker is able to create their own private/public key pair, and then uses the private part to sign a fake set of traces. These are then sent to the AA, who assumes that they were signed with the private key of the legitimate tracer, and verifies the signature using the public key of the attacker (which the AA believes is the public part of the Tracer key). Thus, *the attacker is able to present a fake set of traces to the AA as legitimate*.

While this may appear as a vulnerability, it is actually caused by a **mistaken specification in the CIV protocol** provided as input to the Formal Verification process. Specifically, in the REWIRE CIV scheme, we assume that **the public part of the Tracer Key is known a priori to all trusted entities** (e.g., the AA and the Domain Manager), which is a common assumption in such attestation schemes (even in zero-trust attestation). Thus, the attacker will not be able to exploit this violation, since the AA will know the correct Tracer key and, as such, will not verify the forged signature. However, this assumption/hypothesis was not correctly highlighted and integrated into the Formal Verification process, thus leading to the identification of the counterexample presented in Section 3.4. This result serves as evidence highlighting both **the importance of the assumption of a trusted Tracer in the context of attestation** and the **correct specification of all hypotheses and assumptions when performing the Formal Verification process**.

Overall, this outcome also serves as a representative demonstration of how the Formal Verification process is an essential part of the REWIRE framework, which is important for the identification of potential issues in the protocols to be verified. Consolidating all the above, and using the identified violation as an example, the following two approaches can be used for addressing issues identified during Formal Verification:

- **Better definition of the hypotheses and assumptions:** In this example, including the assumption of a trusted Tracer into the Formal Verification process will lead to a successful verification, as the attacker will not be able to forge the Tracer Key. Thus, the Formal Verification process is a valuable tool that may assist developers, manufacturers, and system administrators towards the better, concrete, and complete definition of the requirements and assumptions behind the evaluated protocol.
- **Modification of the scheme to be verified:** Another possible approach is the modification of the scheme in order to address the identified violation, in case it is not possible to introduce assumptions as aforementioned. For instance, in our example, we may wish to weaken the assumption of the trusted Tracer rather than introducing it as a hypothesis. In this case, as also described in Section 3.4, it is possible to extend the set of Key Restriction Usage policies in order to prevent the verification of such forged signatures. Thus, if an attacker attempts to forge the signature of the Tracer using their own key pair, then the AA will observe that the two signatures $\sigma 1$ and $\sigma 3$ are bound to two different Tracer Keys. Thus, we can introduce a superseding policy dictating that *if the AA receives signatures bound to two different Tracer keys, it will not be able to use its AK in order to sign the attestation challenge*.

Chapter 4

SW/FW Vulnerability Analysis

As detailed in Chapter 2, one of the core functionalities of REWIRE is the enforcement of software and firmware updates in a secure and verifiable manner. In this regard, one core capability of REWIRE is the provision of components such as the **AI-Based Misbehaviour Detection Engine (AIMDE)** and the **attestation enablers** of REWIRE, which act as **Trust Sources (TSs)** that are able to to perform analysis on the software installed on embedded devices, in order to determine whether any unexpected or potentially malicious behaviours have been detected. The detection of such behaviours constitutes an **Indication of Risk**, and signifies the need for further analysis in order to determine and mitigate the cause of the identified behaviour.

In this context, analysing the security of devices when an event of interest has been detected is a challenging task, especially considering not only the **heterogeneity of the SW and FW stack running on embedded devices**, but also the **ever-increasing scale of the domain infrastructures** employing such devices. Specifically, analysing the software security of embedded devices becomes increasingly difficult as the number of devices increases and, although the scale on which Internet of Things (IoT) devices are used increases rapidly, the pool of experts with the capabilities to perform adequate analysis grows much more slowly. *Thus, the endmost goal of REWIRE in this regard is to provide a partially automated process which can support the aforementioned vulnerability analysis pipeline, with particular focus on performing such vulnerability analysis against RISC-V architectures*. This chapter is dedicated to the description of this process, which allows anyone with access to the software stack or firmware image of a device to perform an analysis on the software security of the device. Note that *any update on the software or firmware image deployed through the Secure SW/FW Update component will also be analysed using this process in order to check whether its installation induces any unexpected behaviours*.

In order to facilitate the categorisation of devices in the context of the SW/FW Vulnerability Analysis process, in D3.2 [60], we defined three types of embedded devices:

- **Type 1: Devices with a general-purpose operating system (OS).** Devices that are built on a general-purpose OS need more computational resources to run the operating system. Embedded devices by definition have more limited resources than most devices running a general-purpose OS, such as laptops or backend servers running a general-purpose OS (e.g., Windows- or UNIX-based). This type of devices is therefore the devices with most power and other resources available, such as routers, cameras, and televisions. In the context of REWIRE, these may be Zonal Controllers, Main Vehicle Computers or Onboard Units in the context of the Automotive use case.
- **Type 2: Devices with a Real-time Operating System (RTOS).** Some embedded devices operate in a context where actions reliably need to be completed on time. These systems run a real-time operating system, which guarantees that actions are completed on time. The operating systems on these devices is typically built for specific hardware. Examples are Electronic Control Units (ECUs) in the automotive use case, or Programmable Logic Controllers (PLCs), telecommunication control units, and digital signal processors in the context of the Smart Cities use case.

- **Type 3: Devices with monolithic software.** Devices with the heaviest restrictions on their resources do not make use of any operating system, and instead run code written and built specifically for that device. This is called monolithic software, as all actions of the device are contained in one large program. Typically these are built on a library and are compiled into one binary blob. Examples if these devices are network cards, GPS trackers, and Bluetooth modules, or Exterior Air Temperature and Soil Humidity sensors in the context of the Smart Cities use case.

Considering this categorisation, the SW/FW Vulnerability Analysis in REWIRE targets **a mix of Type 1 and Type 2 devices**. Specifically, we targeted *SW and FW that can be instantiated in devices running a generic purpose OS, whether they belong to the Type 1 or Type 2 category*. Type 2 devices that typically run an RTOS are instantiated with a general-purpose OS, for instance the NVIDIA **Advanced Driver Assistance System (ADAS)** in the context of the Automotive use case, which was integrated without the use of an RTOS. This was a choice made for simplicity purposes, in order to *enable the provision of a full-fledged vulnerability analysis pipeline for the devices of interest in the envisioned use cases, which can be used as a stepping stone for the consideration of additional types of devices*.

At a high level, the SW/FW vulnerability analysis consists roughly of two phases, as depicted in the sequence diagram of Figure 4.1. The steps of **Phase 1** can be summarised as follows:

1. Consider an attestation process executed as part of the enforcement of an MSPL security policy. In case of a failed attestation, the **SW Service Provider** is notified about the resulting Indication of Risk, obtains the device traces collected as attestation evidence following the process outlined in D4.3 [61], and verifies the failed attestation.

2. Then, the the SW/FW image to be analysed in order to pinpoint the cause of the attestation failure is sent to the **SW/FW Vulnerability Analysis** component and is immediately unpacked by the **Unpacker**.

3. Since the device is assumed to be built with a general-purpose operating system, this image contains a file system. To this end, the **Preliminary Analysis** aims to find this file system, identify interesting programs to analyse in Phase 2, and make a list of hardcoded passwords and typically vulnerable functions used in the firmware. It also gathers general information about the firmware for step 4 of this phase.

4. Along with the static analysis of the preliminary analysis, it is attempted to analyse the firmware dynamically. To this end a **Full-system emulator** is used to attempt to run parts of the firmware image as if it were running on an actual device. This emulation uses information gathered during the preliminary analysis to guess the correct parameters to do so. If this succeeds, it allows gathering information which is difficult to get statically, such as a list of externally exposed services.

5. Specific software in the SW/FW is selected based on several factors:

   - *Does the software communicate over a network?* – This makes it an interesting target for an attacker, as they might be able to execute their own code on the device if they can find and exploit a vulnerability in this software.
   - *Does the software deal with privileges?* – Exploiting a vulnerability in such software could allow an attacker to escalate their privileges.
   - *Does the software make use of typically vulnerable patterns*? – For example, memory access through functions that do not check the boundaries of the buffers they operate on are indicators of vulnerabilities.

6. The resulting software is sent to the **Scanner** as input to Phase 2 along with the other gathered information, which analyses the software for vulnerabilities.

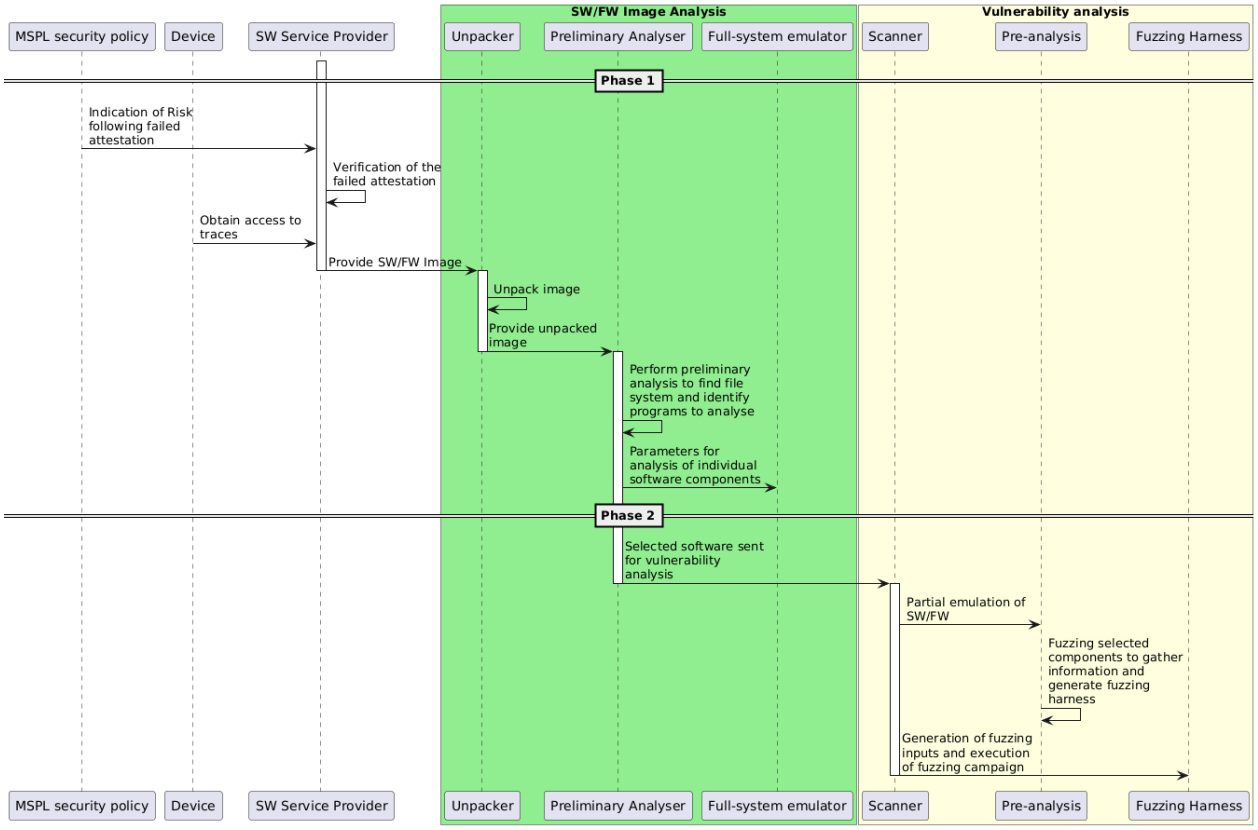


Figure 4.1: A high-level overview of the SW/FW vulnerability analysis

After the completion of Phase 1, the purpose of **Phase 2** is the analysis of the SW/FW for the actual identification of vulnerabilities through the execution of a **fuzzing campaign**, i.e., the construction of the appropriate set of inputs which are provided to the targeted software components in order to trigger unexpected behaviours, and the identification of vulnerabilities based on which inputs triggered such behaviours. Specifically, Phase 2 consists of the following steps:

1. A **Pre-analysis** is performed, which uses gathered information to set up a partial emulation of the software/firmware. *This process, which allows analysing RISC-V binaries on a different architecture than the firmware was built for, is a core innovation of REWIRE and will be further analysed later in this Chapter*. For example, in the REWIRE project the RISC-V architecture is used, but most software analysis is performed on the X86_64 architecture. By only partially emulating the firmware it is possible to forego device-specific properties that can be difficult to emulate, and to target specific software components.

2. The analysis itself consists of fuzzing the selected software components with AFL++ using a generated **fuzzing harness**. Note that *a fuzzing harness is a program that interacts with the target software and ensures that fuzzing input is stored at the right location for the target software to process it.* In the process of fuzzing, semi-random inputs are generated and mutated for the software on the device, and the device's behaviour when provided with these inputs is monitored. The aim of the fuzzing process is to find as many bugs (indicated by e.g. crashes) as possible. To achieve this, heuristics can be used. In this case program coverage is used as a heuristic to guide fuzzing efforts. This means that the fuzzer aims to execute as much of the target program's code as possible. Intuitively, as more parts of the target program are executed, more of the bugs and vulnerabilities present in the program under test are found.

3. The fuzzing process starts with one or more starting inputs, which are called seeds, and determine the sequence of actions performed by the software and behaviours that are triggered through the

provided inputs. During the **fuzzing campaign** these seeds are then mutated randomly and used as program input. Test cases that improve target program coverage, i.e. reach new parts of the code, are favoured for further mutation.

4. If the input causes interesting behaviour, such as when it causes the program under test to crash, it is saved for further analysis as part of the fuzzer output. Seeds can be empty, or chosen such that they consist of or resemble valid input. This can allow the fuzzer to reach more parts of the program under test more quickly.

Here, it is important to note one basic challenge that had to be overcome in this regard, which resulted in one of the core innovations of REWIRE. Specifically, as aforementioned, REWIRE targets devices instantiated on **RISC-V architectures**, and the Vulnerability Analysis process requires the interpreting how systems running on such architectures manage the internal operation of internal software processes (e.g., system call tracing, multithreading, debugging, internal memory management). However, *while this approach has been previously applied on various types of software and architectures, embedded systems were typically not supported, and such mechanisms that can interpret the internal operations of RISC-V architectures did not exist in the literature*. Therefore, **REWIRE constructed an interpreter in order to perform the internal analysis of such operations in embedded systems based on RISC-V architectures**. As will be demonstrate in the following, the question answered by REWIRE in this regard is *how to analyse the software or firmware running in embedded RISC-V devices in a manner that enables its analysis to be performed on different architectures than the one it has been built for?*

Another issue that needs to be addressed in the process outlined above is the construction of the **fuzzing campaign**. Specifically, the seed (which essentially refers to the code sequence that is tested through the fuzzing process) should be selected so that we are able to quickly capture a wide array of behaviours of the software to be tested, including edge cases which may be difficult to capture if the seed is restricted. Thus, a core challenge is the construction of a fuzzing campaign, which consists of seeds which are configured in a manner that enables the widest possible coverage of the code. In this regard, a core question that REWIRE aims to answer is *how to construct and configure a wide set of seeds that can achieve wide coverage of the software, including edge cases, in a somewhat automated manner?*

4.1 The REWIRE Solution

In the previous Section, we provided a high-level overview of the entire SW/FW Verification process of REWIRE. As part of this description, we posed two questions corresponding to the two core challenges identified during the design and implementation of this process. Here, we expand upon these, and we provide a detailed description of the process followed for addressing the underlying challenges, which are essentially two core innovations of REWIRE.

Figure 4.2 depicts a more detailed view of the preliminary phase pertaining to the analysis of the internal operation of RISC-V architectures, and figure 4.3 a more detailed view of the software analysis phase related to the construction and execution of the fuzzing campaign. In the following Sections, we expand upon each of these phases, detailing the core innovations of REWIRE in each.

4.1.1 Analysis of RISC-V Architecture through Updating Qiling

As aforementioned, in the preliminary analysis of the SW/FW image on which the Vulnerability Analysis process should be performed, a core innovation of REWIRE was the design and implementation of a novel interpreter of RISC-V architectures, which provides the basis for the application of this analysis to different types of architectures. In Figure 4.2, we provide a more detailed overview of this process, which was previously depicted at a high level in Figure 4.1. Specifically, after the SW/FW image is received

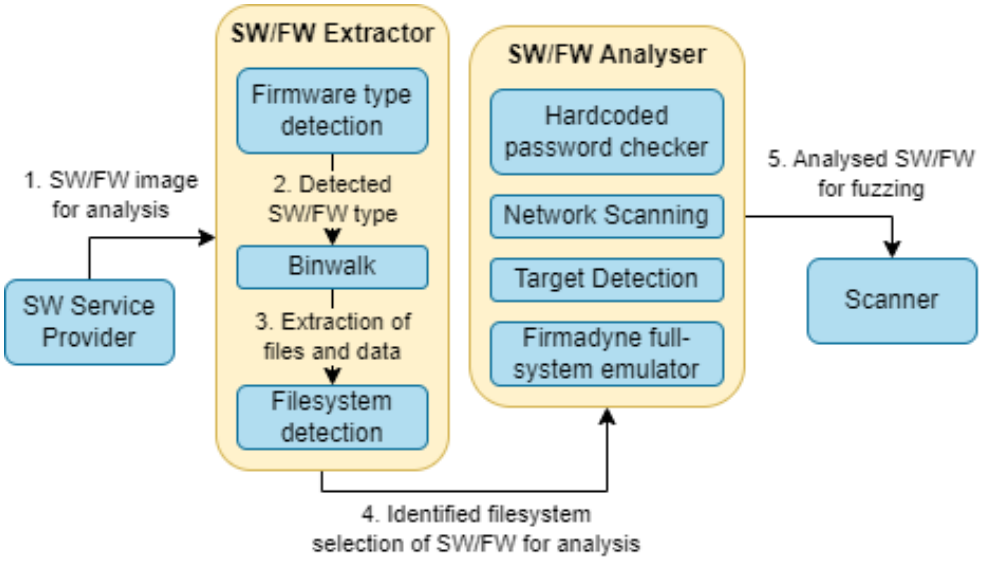


Figure 4.2: The preliminary analysis of the SW/FW vulnerability analysis

by the SW/FW Extractor for analysis, three internal steps are performed: (i) the detection of the **type of SW/FW** to be analysed, (ii) the use of **Binwalk** for the identification and extraction of files and data that have been embedded inside of other files in the image, and (ii) the detection of the **file system** used by the OS. Then, the processes parameters required for the fuzzing process are determined, and the image is sent to the Scanner for the initiation of the fuzzing campaign.

As previously described in D3.2 [60], the aforementioned interpreter was designed through the **expansion of the Qiling framework**, which is an emulation and analysis framework, offering flexibility and control over the analysed software. It is based on a fork of QEMU, from which it inherits several concepts used to emulate OS functions. Over the years some new features from QEMU were backported to Qiling as well, but not all of them.

REWIRE Innovation

One feature that is notable in the context of the REWIRE project is support for the RISC-V architecture. While QEMU implemented this support already, it was never included in Qiling. Therefore, a fork of Qiling was created and this support was added.

4.1.1.1 Dependencies

Some of the dependencies of Qiling already supported RISC-V in their newest version. However, for the sake of stability, Qiling did not always use these versions. Additionally, for those dependencies that did not yet have support for RISC-V it was necessary to make Qiling work with the forks of those dependencies built by REWIRE.

REWIRE Innovation

Modified Qiling in order to work with the forks of the dependencies built by REWIRE in order to introduce support for RISC-V.

4.1.1.2 Assembling

Using Qiling it is possible to instrument the code that is emulated. This is possible since all code is run under Unicorn, a stripped-down fork of QEMU that only emulates the CPU but adds various bindings so the CPU state can be controlled by the user. The execution can be paused at any time, and the values of

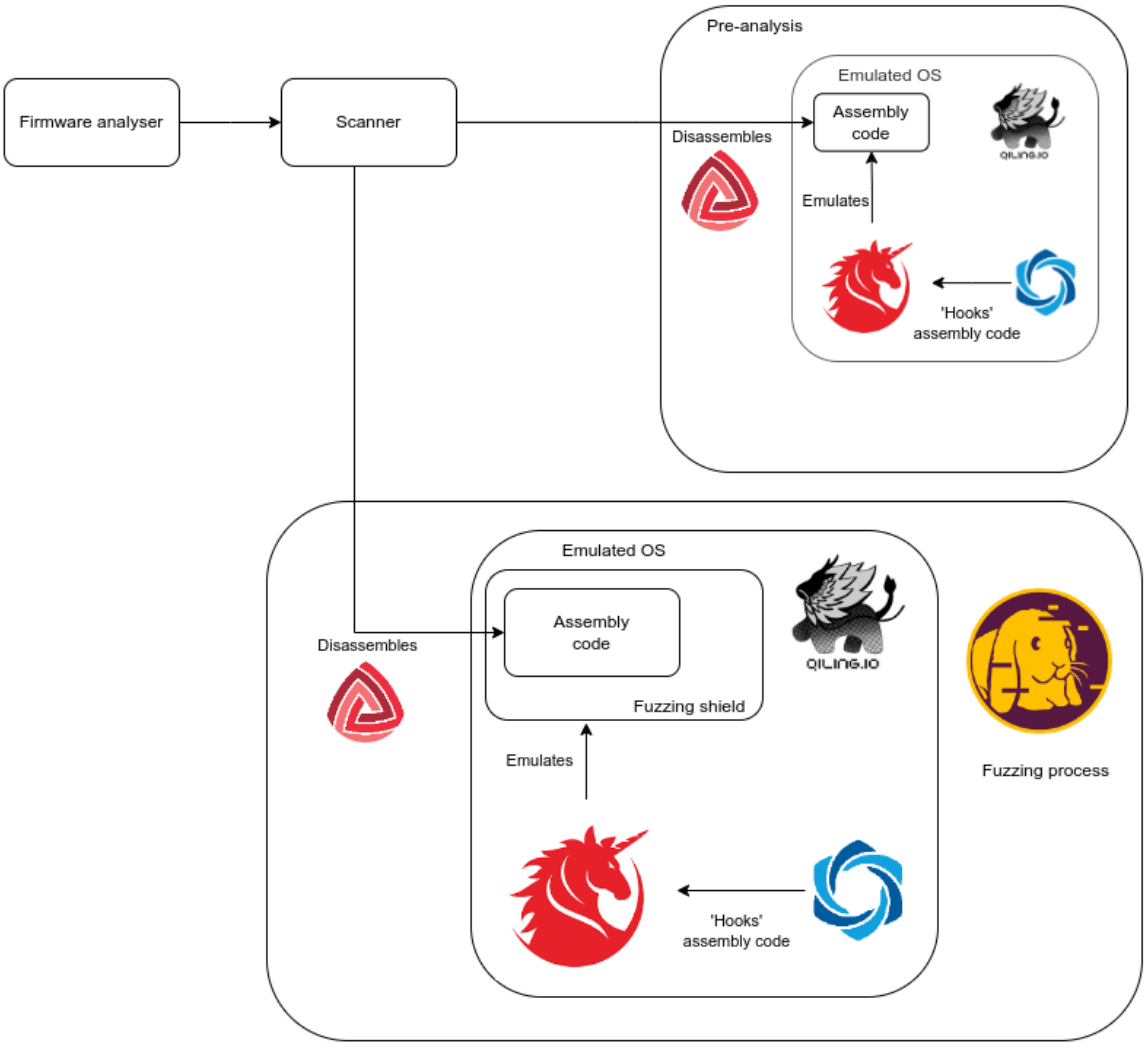


Figure 4.3: The main phase of the SW/FW vulnerability analysis

registers and contents of memory can then be altered. This can be done using for example Python code. In some cases, however, it is necessary to include assembly code of the ISA that is being emulated. This is done using the Keystone assembler. At the start of the project, there was no support for RISC-V however. The Keystone assembler is a fork of LLVM, taking its definitions for each assembly language. While LLVM supports various RISC-V extensions, the fork was not updated to reflect this.

REWIRE Innovation

The support for RISC-V added in newer versions of LLVM was backported by REWIRE to the Keystone assembler. With this support it is possible to add a snippet of RISC-V assembly and translate it to its corresponding opcode.

4.1.1.3 OS support

Qiling itself provides emulation for OS-level functions such as system calls. This support is necessary since binaries that are analysed are generally built for a different platform than they are analysed on. While both systems might be Unix-like, the analysing system might be a Linux distro whereas the target could be a custom BSD build. Qiling recognises which platform the binary was built for, and defines different properties for each platform. While a custom build can of course differ in some aspects, this generally leads to improved accuracy and stability in its emulation. If a binary makes a system call this is

then first handled by Qiling, and can optionally be passed on to the host operating system. However, if Qiling does not implement a certain system call then this causes an emulation to crash. For the binaries under test in the demo several of these system call definitions were missing and added.

In addition, for every Instruction Set Architecture (ISA), a mapping must be defined of system call numbers to the system calls themselves. For the REWIRE project a mapping was added for system calls in RISC-V based on the mapping in QEMU, which uses a similar mapping.

REWIRE Innovation

Added several missing system call definitions to Qiling. Also, mapping added for system calls in RISC-V based on QEMU mapping.

- **Threading:** For threading there are architecture-specific definitions required as well. Again this is related to the target binary and analysis system not necessarily being built for the same architecture. Multiple threads can run in the same process and share part of the memory they have access to. They also make use of thread-specific data though, which is stored in thread-local storage. This is a memory region reserved for data used only by one thread. In RISC architectures such as ARM and RISC-V a register is reserved to store a pointer to this memory region, the thread pointer. This must therefore also be defined in Qiling, which keeps a record of which registers exist per architecture, and what these registers are used for. For every architecture there is a definition of how a thread refers to its thread-local storage, which in the case of RISC-V is the dedicated register. On top of this definition an architecture-specific thread object has to be defined in Qiling. This is used when other parts of the framework refer to the current thread, and makes sure the correct definitions are used.
- **Sockets:** Many programs communicate over some type of socket. This is a software structure over which communication is sent or received, either over a network or within a system. Various types of sockets are defined in standards, and depending on how the socket is used a specific type can be selected. For example, to send information over an encrypted channel to another system one would generally use a TCP/IP socket. To communicate with another process on the same system one could use a UNIX socket. The definitions for these different types of sockets can differ between platforms, and as such are also defined in the Qiling framework. This exists as a similar mapping to the system call mapping. Each socket family, e.g. AF_UNIX, and socket type, e.g. SOCK_STREAM, has a specific number per target architecture. Support for sockets in RISC-V was added by copying the definition for each from the buildroot project.

REWIRE Innovation

REWIRE added RISC-V architecture-specific definitions for threading to Qiling, as well as definitions for different types of sockets used for communication between different platforms.

4.1.1.4 Debugging

Qiling supports debugging the emulation using GDB. To do so, it pauses execution on the first emulated instruction and exposes a GDB server. Any GDB instance with a definition of RISC-V can connect to this server, after which it will communicate back and forth between the server and client using the GDB serial protocol. By loading the emulated binary in the client it can have symbol information available as well. The GDB server in Qiling itself must have the right definition of RISC-V for the binary, which at the start of the REWIRE project it did not. This was added by copying the structure of existing definitions in Qiling. To add support for the serial protocol, basic commands were copied from the Qiling GDB definition for ARM. The registers were added based on the reference manual for RISC-V itself [77]. More complex commands from the GDB server were implemented based on the GDB documentation [67].

REWIRE Innovation

Added RISC-V definition to the GDB server in Qiling itself by copying the structure of existing definitions in Qiling. Support for the serial protocol was added based on Qiling GDB definition for ARM, and more complex commands were implemented based on the GDB documentation.

4.1.2 Selection of Seeds for Construction of Fuzzing Campaign

Here, we provide a detailed analysis on the approach followed by REWIRE with regards to the second question mentioned above, specifically regarding the *selection of the appropriate seeds construction of a fuzzing campaign with a wide set of seeds, in order to achieve the widest possible coverage of the software to be analysed (including edge cases) in a somewhat automated manner*. This is performed after the initial firmware analysis has been performed, and the internal structure of the SW/FW to be analysed has been identified, as detailed in the previous section.

In order to process the output of the initial firmware analysis, and use it as input for the fuzzing process, several scripts were written. These allow the SW/FW vulnerability analysis to work in a more automated manner than other solutions. In most cases, the fuzzing process requires someone with technical knowledge of the targeted software to write a **fuzzing harness** for this target. *This harness contains knowledge about the way in which the software accepts user input, and in some cases about its memory layout*. **The solution offered by REWIRE does not need this expert input, as it is automated by the scripts of the SW/FW vulnerability analysis**. The most important steps of these scripts are the following:

- **Pre-analysis firmware:** In an initial analysis, the SW/FW image is unpacked and analysed. This checks whether the image contains a file system following the POSIX standard. In this file system it attempts to select individual software that is interesting from a security perspective. This is scored based on heuristics such as the likelihood that something is custom software, and which software communicates over a network. Lastly, in this phase some properties of the interesting software are analysed. These include for example the (incorrect) use of typically vulnerable system calls such as 'strcpy' and the use of hardcoded passwords.
- **Pre-analysis software:** In the second part of the pre-analysis more information about the software itself is gathered. Primarily, this attempts to detect the way in which the software accepts input. The first step in doing so is finding the input vector where the software accepts in main input. The supported input vectors are STDIN, command line arguments, and TCP network socket input. After this input vector has been located the script attempts to discover what kind of input the program is expecting over this vector. This could for example be simple strings, as generally is the case for command line arguments, or HTTP, which can be the case for TCP network socket input.
- **Fuzzing harness template:** Using the output of the previous two steps a fuzzing setup is created for the selected software. This setup is built from a fuzzing harness template. This makes sure that the fuzzer knows where to direct its test cases such that the software takes them as input, and that this input is in an appropriate format. There is also support for directing the fuzzer towards typically vulnerable functions, and for including hardcoded passwords in the test cases.

4.2 REWIRE SW/FW Vulnerability Analysis Benchmarking

4.2.1 Test cases

To verify whether the additions to the FW/SW Vulnerability Analysis functioned as intended, four unit test cases were defined in D6.1. The additions to the Keystone assembler could be tested separately from the FW/SW Vulnerability Analysis, and thus have their own test case.

The other three test cases cover the fuzzing process. The first test case is about the code coverage reached in a fuzzing campaign. The analysis is run for a fixed period of time. The code coverage a fuzzer is able to find with its test cases indicates how much of the code in a program was actually used when the program is run with the various inputs found by the fuzzer. If the coverage is larger, this means the fuzzer was able to test more of the code, generally resulting in more bugs found.

The second test case measures the time to the first finding. If a fuzzer is supplied with a good seed input it is generally able to find more of the code of the program in a short timespan. This generally results in finding a bug more quickly as well.

The last test case measures the total findings after a fuzzing campaign. The fuzzer is run for a fixed period of time, after which the results are first analysed in order to deduplicate them. The inputs leading to unique code execution paths are then further analysed to determine what bug they identified. The number of unique bugs identified in this manner is the result to the test case.

UT_FWSW_1	RISC-V assembling Test
Description	Test RISC-V assembling in Keystone
Ref. Code	UT_FVC_1
Component	FW/SW Validation, Keystone assembler
Input	Specification of RISC-V instructions mnemonics
Output	A pass/fail per instruction whether the produced instructions align with the output from well-known assemblers
Status & Results	**[PERFORMED]** - Most instructions passed. The assembler was verified to work correctly by translating various RISC-V instructions using both the Keystone assembler and a well-known assembler such as GCC. In some cases however, the resulting opcodes differed. It was not clear whether this happened due to faults in GCC or the Keystone assembler.

Table 4.1: Unit Test UT_FWSW_1 for FW/SW Validation

UT_FWSW_2	Code coverage Test
Description	Measure code coverage in fuzzing RISC-V
Ref. Code	UT_FVC_2
Component	FW/SW Validation, Fuzzer and emulator
Input	Demo firmware OdinS
Output	A coverage percentage averaged over multiple fuzzing campaigns.
Status & Results	**[PARTIALLY PERFORMED]** - Microbenchmarking has been performed in the demo test program, and we are currently performing this test with SW binaries from the Smart Cities use case. The results will be documented in D6.2.

Table 4.2: Unit Test UT_FWSW_2 for FW/SW Validation

UT_FWSW_3	Duration Test
Description	Measure time to first finding in fuzzer
Ref. Code	UT_FVC_3
Component	FW/SW Validation, fuzzer
Input	Demo firmware OdinS
Output	The time to the first finding of the fuzzer, averaged over multiple fuzzing campaigns
Status & Results	**[PARTIALLY PERFORMED]** - Microbenchmarking has been performed in the demo test program, and we are currently performing this test with SW binaries from the Smart Cities use case. The results will be documented in D6.2.

Table 4.3: Unit Test UT_FWSW_3 for FW/SW Validation

UT_FWSW_4	Fuzzer findings Test
Description	Measure fuzzer findings after set time
Ref. Code	UT_FVC_4
Component	FW/SW Validation, fuzzer
Input	Demo firmware OdinS
Output	The number of findings from the fuzzer, averaged over multiple fuzzing campaigns
Status & Results	**[PARTIALLY PERFORMED]** - Microbenchmarking has been performed in the demo test program, and we are currently performing this test with SW binaries from the Smart Cities use case. The results will be documented in D6.2.

Table 4.4: Unit Test UT_FWSW_4 for FW/SW Validation

All tests of the SW/FW vulnerability analysis were yet to be performed. The results of the tests are listed below.

4.2.2 Setup

The unit tests described above were performed on a **demo program written in C**. This program loaded a library with functions that each contained a different and deliberately introduced vulnerability or bug. *The motivation behind their selection was in order to emulate the most impactful types of vulnerabilities identified for the types of devices considered in REWIRE.* Specifically, the included vulnerabilities are as follows:

- Stack buffer overflow
- Heap buffer overflow
- Format string injection
- Command injection
- Integer overflow
- Use after free
- Logic error

The first characters of the input determined which library function was executed. As each function contained one of the aforementioned vulnerabilities, each fuzzer finding can be easily mapped to a specific function. This can be used as a ground truth, and to evaluate the performance of a fuzzer: findings of a fuzzer can be mapped to these known vulnerabilities, as is illustrated in the analysis below. *This setup was designed to aid in the development of the benchmarking software and to illustrate the workings of the fuzzer.* The tested program was compiled for a **64-bit RISC-V architecture**.

As of now, tests have only been performed on the aforementioned demo program. ZTO and its facility layer will be tested as part of D6.2. The tests were run on a **Thinkpad T14 Gen 2** with a **16 core AMD Ryzen 7 pro 5850u CPU** and **16 gigabytes of DDR4 RAM**. The tests themselves were run in a **docker container**, to which **6 gigabytes of RAM** was made available. For analysis of crashes found during the fuzzing campaign GDB, **QEMU and Qiling's qltool** were used.

4.2.3 Results

During a single 20 hour fuzzing campaign of this demo program 780 crashes with unique execution paths were found. During a subsequent 9,5 hour run another 451 crashes with unique execution paths were found. The next section describes some of these crashes to illustrate how fuzzer output can be used to find bugs and exploitable vulnerabilities in code.

4.2.4 Fuzzer output analysis

Below some manual analysis of four crashes found during aforementioned fuzzing campaigns is described in order to illustrate how fuzzer findings can be mapped to software vulnerabilities.

4.2.4.1 Stack buffer overflow

This crash occurred as a result of an input with a first line that consisted of more than 20 characters, as is shown in figure 4.4.

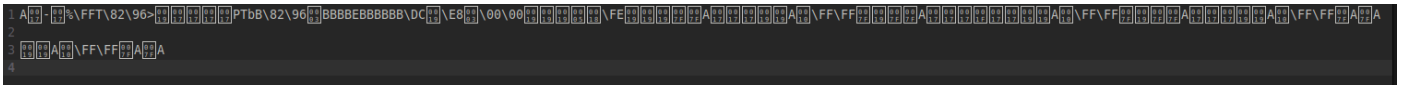


Figure 4.4: The input that triggered a stack buffer overflow, as rendered by the Gnome text editor.

The first two characters in this input are read, which causes the execution of `stack_buffer_overflow`. The source code of this function is shown in Listing 4.1.

Listing 4.1: Source code of the `stack_buffer_overflow` function

```
1 void stack_buffer_overflow() {
2   printf("User input buffer:\n");
3   char user_input[20];
4   fgets(user_input, 200, stdin);
5   printf("Your input was %s\n", user_input);
6 }
```

In this function, the rest of the first line of input is read and stored in a buffer with the size of 20 characters. This occurs on line 4 of Listing 4.1. As the input consists of more than 20 characters, and the program will read up to 200 characters (as is defined on line 4 of Listing 4.1), the buffer overflows. When the `printf` function attempts to read the buffer contents it reaches invalid memory and the program crashes. This vulnerability could be exploited by an attacker to manipulate stack values and influence the program flow, which could lead to arbitrary code execution.

4.2.4.2 Format string injection

This crash occurred as a result of the following input:

```
Cq%n
```

In the target program, the first characters caused the execution of `format_string_injection`, the source code of which is included in Listing 4.2.

Listing 4.2: Source code of the `format_string_injection` function

```
1 void format_string_injection() {
2   printf("User input buffer:\n");
3   char user_input[100];
4   fgets(user_input, 100, stdin);
5   printf("Your input was:\n");
6   printf(user_input);
7   printf("\n");
8 }
```

The target program was run with the specified input using qltool, and the output of this tool was observed. The beginning of this output is shown in figure 4.5.

```
[+]     Profile: default
[+]     Mapped 0x10000-0x7f000
[+]     Mapped 0x80000-0x85000
[+]     mem_start : 0x10000
[+]     mem_end   : 0x85000
[+]     mmap_address is : 0x7fffb7dd6000
[+]     Received interrupt: 0x8
[+]     0x000000000004aeb6: brk(inp = 0x0) = 0x87000
[+]     Received interrupt: 0x8
[+]     brk: increasing program break from 0x87000 to 0x88000
[+]     0x000000000004aeb6: brk(inp = 0x87b30) = 0x88000
[+]     Received interrupt: 0x8
[+]     0x000000000004a96c: uname(buf = 0x80000000db48) = 0x0
[+]     Received interrupt: 0x8
[+]     readlinkat(18446744073709551516, "/proc/self/exe", 0x80000000cc00, 0x1000) = 5
[+]     0x0000000000050652: readlinkat(dirfd = 0xffffffffffffff9c, pathname = 0x782c8, buf = 0x80000000cc00, bufsize = 0x1000) = 0x5
[+]     Received interrupt: 0x8
[+]     brk: increasing program break from 0x88000 to 0xa9000
[+]     0x000000000004aeb6: brk(inp = 0xa9000) = 0xa9000
[+]     Received interrupt: 0x8
[+]     0x000000000002ce24: mprotect(start = 0x80000, mlen = 0x1000, prot = 0x1) = 0x0
[+]     Received interrupt: 0x8
[+]     0x000000000002c204: newfstatat(dirfd = 0x1, path = 0x771e0, buf_ptr = 0x80000000d540, flags = 0x1000) = -0x1 (EPERM)
[+]     Received interrupt: 0x8
[+]     0x000000000002ccee: pselect6(nfds = 0x1, readfds = 0x80000000dc40, writefds = 0x0, exceptfds = 0x0, timeout = 0x80000000dc00,
[+]     Received interrupt: 0x8
[+]     read() CONTENT: b'Cq'
[+]     0x000000000002c2fc: read(fd = 0x0, buf = 0x80000000dcc0, length = 0x2) = 0x2
[+]     Received interrupt: 0x8
[+]     0x000000000002c204: newfstatat(dirfd = 0x0, path = 0x771e0, buf_ptr = 0x80000000d9f0, flags = 0x1000) = -0x1 (EPERM)
[+]     Received interrupt: 0x8
[+]     read() CONTENT: b'%n'
[+]     0x000000000002c2fc: read(fd = 0x0, buf = 0x8a3a0, length = 0x2000) = 0x2
[+]     Received interrupt: 0x8
[+]     read() CONTENT: b''
[+]     0x000000000002c2fc: read(fd = 0x0, buf = 0x8a3a0, length = 0x2000) = 0x0
[x]     CPU Context:
```

Figure 4.5: The beginning of the qltool output. At the end of this snippet it can be observed that, shortly after the input is read, the program crashes.

From this information can be deduced that the first two characters are read, and the function `format_string_inje` is selected. Subsequently, the characters `%n` are read. This corresponds to line 4 of the source code in Listing 4.2. This input is then subsequently printed on line 6 of the same listing. In this way a format string vulnerability is triggered[1]. The characters `%n` form a format parameter, which causes the format function `printf` to write the number of characters printed so far to an integer variable. This leads to undefined behaviour, as no such variable is provided in our program. In this case, the undefined behaviour constitutes a crash. When exploited, such a vulnerability could allow an attacker to read and write to arbitrary memory locations.

[1] https://owasp.org/www-community/attacks/Format_string_attack

4.2.4.3 Integer overflow

This crash was caused by the input shown in figure 4.6.

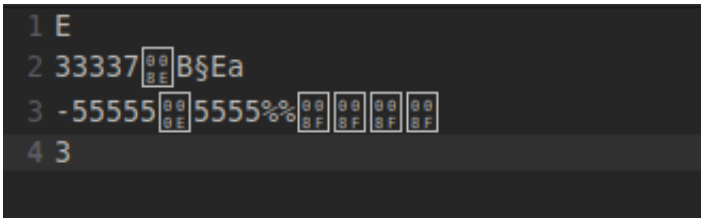


Figure 4.6: The input that triggered a crash in the integer overflow function, as rendered by the Gnome text editor.

In the target program the first character of the input caused the execution of `integer_overflow`. The source code of this function is shown in Listing 4.3.

Listing 4.3: Source code of the `integer_overflow` function

```
1  void integer_overflow() {
2    printf("User input buffer:\n");
3    char user_input[100];
4    char user_output[100];
5    int input_len, output_end;
6
7    fgets(user_input, 100, stdin);
8    printf("How many characters would you like to leave out of the output?\n");
9    int result = scanf(" %d", &output_end);
10   if (result == EOF || result == 0) {
11     printf("Invalid input!\n");
12     return;
13   }
14
15   input_len = strlen(user_input);
16   strncpy(user_output, user_input, (unsigned) input_len - output_end - 1);
17   user_output[input_len - output_end - 1] = '\0';
18   printf("The selected part of your input was %s\n", user_output);
19 }
```

In this function the second line of the input shown in figure 4.6 is read and stored into the buffer `user_input`, which corresponds to line 7 in the source code shown in Listing 4.3.

After that, the third line of the input in figure 4.6 is stored in the integer `output_end`, this corresponds to line 9 in the source code shown in Listing 4.3. This value is then used in the part of the program that corresponds with line 16 of the same listing: `output_end` is subtracted from `input_len` as part of the `strncpy` statement.

The execution of this subtraction is analysed with GDB attached to an instance of the program running in QEMU. The crashing input is then provided to this program provided. The register values and assembly code as presented by GDB at the point of this subtraction are shown in figure 4.7.
From figure 4.7 we see that the value in register a5 is subtracted from the value in register a4. This means that the second line of the input shown in figure 4.6 was stored as 11 characters. This is the case because the undefined character between the 7 and B shown in the figure represents two bytes, and each character is stored as a byte in C. This also shows that (the start of) the third line was stored as the integer value -55555.

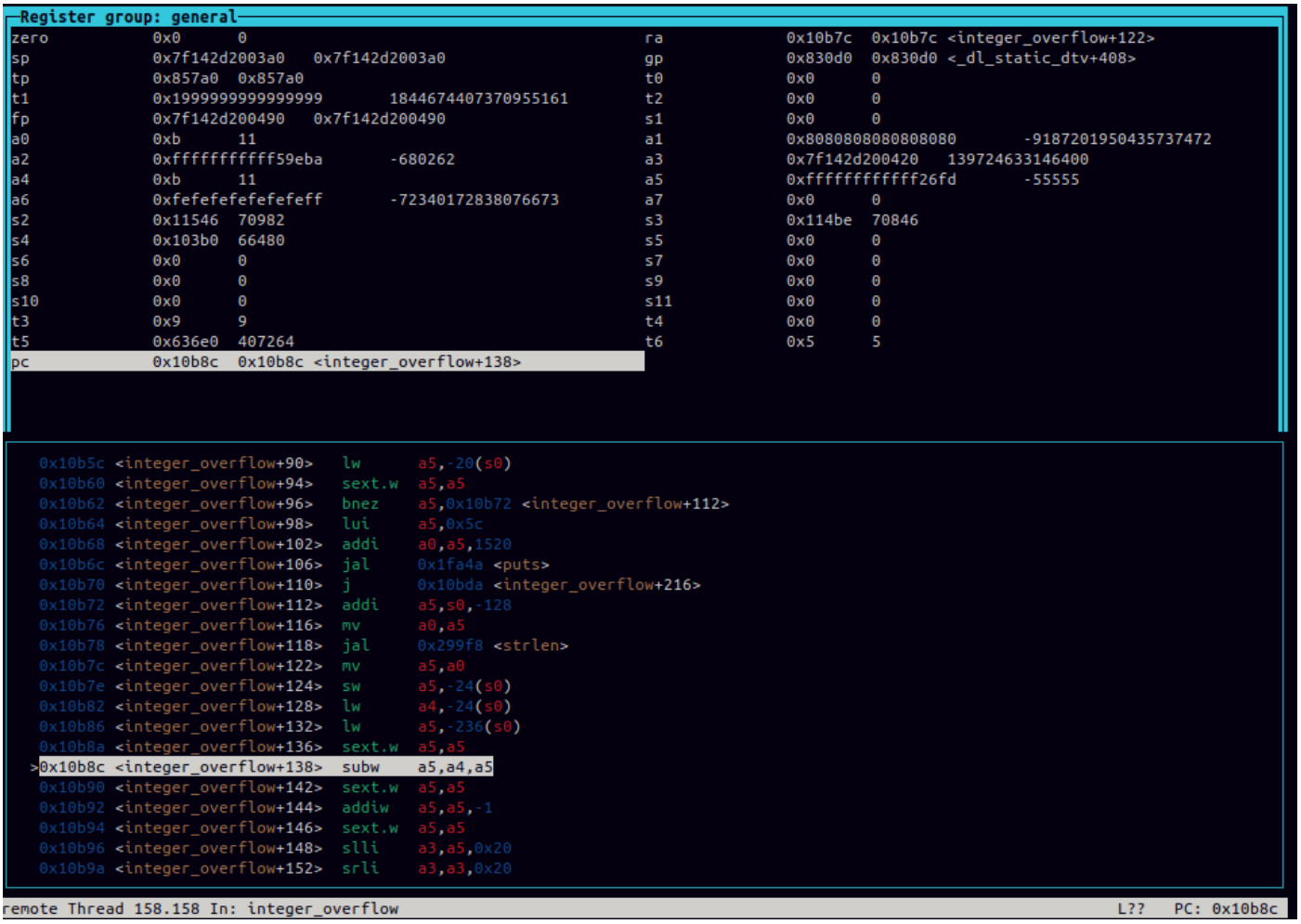


Figure 4.7: The values of the registers and executed assembly code when `output_end` is subtracted from `input_len`.

Predictably, when stepping the program one step further the value in register a5 becomes 55566. This means that, after 1 is subtracted from this value, 55565 characters from the buffer `user_input` are copied to `user_output`, as the instructions corresponding to line 16 of the source code in Listing 4.3 are executed. However, as both of these buffers only have a size of 100 characters, and only 11 characters are actually present in the buffer `user_input`, both out-of-bounds reads and out-of-bounds writes occur, causing undefined behaviour, which in this case constitutes a program crash.

4.2.4.4 Subroutine bug

This crash did not correspond to one of the deliberately introduced vulnerabilities, but was the result of a bug in a subroutine present in the library. The crash was caused by the following input:

```
H\ADHH
```

In the target program the first characters of the input caused the execution of a function called `logic_error`. In this function up to 10 characters of user input are read and subsequently used as input of the function `eraseNl`, the source code of which is shown in Listing 4.4.

Listing 4.4: Source code of the `eraseNl` function

```
1 int eraseNl(char* line){
2     for(;*line != '\n'; line++);
3     *line = 0;
4     return 0;
```

```
5 }
```

This function iterates over the input until it encounters a newline character. However, when the input does not contain a newline character, as is the case with this crashing input, the function keeps iterating past the provided input, causing undefined behaviour. In this case, the function continues until it hits invalid memory, which causes a crash.

Chapter 5

Modelling and Requirements Traceability of Use Cases through AADL

Towards achieving the main goal of REWIRE for *enhancing the secure lifecycle management and guiding the trustworthiness level of embedded systems operating in the context of large-scale Systems-of-Systems (SoS)*, a set of trust requirements has been defined in D2.1 and D2.2 for both the Design-time phase and Runtime phase of the REWIRE framework, encompassing the entire operational lifecycle of the devices. These requirements have been distilled and mapped into a set of **trust requirements and functional specifications** that each technical component of REWIRE needs to fulfil, so that the required level of assurance in the operation of the device is achieved.

Up until now, this set of technical and architectural deliverables of REWIRE has provided the entire design space of security requirements from a **bottom-up perspective**, *starting from the security requirements per component and moving towards how these reflect the security in trust requirements in various application domains in the context of the envisioned use cases*. In addition, recall that one of the core innovations of REWIRE, as documented in D2.2, is the provision of a **Compositional Verification and Validation** pipeline for defining the envisioned **assurance cases** that enable us to capture and formally verify the **trust boundary** of the system. Then, in D3.2, starting from these assurance cases, we demonstrated how to formally verify the aforementioned requirements of the underlying security processes of REWIRE (e.g., Authenticated Encryption, the various crypto schemes used in the context of attestation). However, in order to complete this verification pipeline, it is important to be able to *trace not only the fulfilment of the initially defined assurance, cases, but also whether these formally verified components and building blocks can still achieve the security requirements, considering the interconnectivity between components*.

Towards this direction, the core aspect of the secure lifecycle management that we need to consider is the **Secure SW Update Process**. In D3.2, we demonstrated the Formal Verification of the **establishment of authenticated and encrypted communication channels** through the use of **LR-BC-2-enhanced crypto structures**. Then, based on this, we traced the evaluation of the assurance cases provided by **Authenticated Encryption (AE)** in the context of the Smart Satellites use case in D6.2. The last step in order to close the SW update control loop, which is the core focus of this chapter, is to be able to *trace the assurance cases leveraging AE for safeguarding the SW update process, in order to ensure that the device has correctly re-established its trust level after the enforcement of SW Updates*. In this regard, we are considering both modalities of the SW update capability of REWIRE as outlined in D3.2, specifically (i) the **"1-to-1"** scenario where an update is distributed to a single device (e.g., in the Automotive use case), and (i) the **"1-to-many"** scenario a single source distributes the same update to multiple recipients (e.g., in the Smart Cities use case).

The approach described above is supported by the **Architecture Analysis & Design Language (AADL)-based System Modelling** methodology which is used for specifying both software and hardware configurations, and building a representation of the system to be validated, capturing the interactions between

components to be verified as part of the REWIRE use cases. Note that *AADL is a standardized architectural description language for hierarchical structures and connections between software and hardware components*, and is employed by REWIRE in order to create a virtual representation of the **assets comprising a target domain infrastructure**, their **interconnectivity**, and any associated **asset properties or requirements defined by the Security Administrator or Use Case provider**. Afterwards, based on the AADL-defined architectural model, the **RESOLUTE** language is used for ratifying and validating the correctness and achievement of the aforementioned requirements through **architectural assurance cases**. Considering all the above, the end goal of this approach is to **evaluate whether the design-time assurances offered by the individual components can be cascaded for validating the overall correctness of the interactions between components**.

Considering all the above, this Chapter is dedicated to the description of the aforementioned methodology, and how it can be used to achieve the traceability, validation, and verification of the security requirements, especially considering cases where embedded devices need to isolate critical components or enforce zero-trust models for untrusted applications. This approach is applied in the context of two REWIRE use cases, namely the **Adaptive In-Vehicle SW & FW Patch Management & Software Functions Migration** and **Smart Cities for Empowering Public Safety** cases, where we provide a detailed breakdown and description of the methodology employed.

5.1 Introduction

The REWIRE project operates in a highly interconnected digital ecosystem, where securing complex cyber-physical systems presents significant validation challenges. Specifically, due to the complexity of the types of environments considered in the context of REWIRE, there are multiple **trust-related requirements**, which may also be interconnected between them. Those interdependable requirements need at the design time a fine-grained formal representation that leads to infeasible verification results due to computational resources exhaustion. This not only increases the complexity of their validation, but also the complexity of the **traceability regarding their correct achievement and fulfilment**. Thus, it is important to consider the traceability, validation, and verification of security requirements, especially in embedded devices operating in a **zero-trust manner**. This adds a degree of complexity to the aforementioned process, as no device is considered trusted by default, thus increasing the complexity and design space of the requirements considering all components that need to be captured in the overall model.

As previously outlined, in order to address this, REWIRE employs an approach aiming to **evaluate whether the design-time assurances offered by individual components can be cascaded for validating the overall correctness of the interactions between components**. As part of this approach, REWIRE uses rigorous methods and tools like the **Open Source AADL Tool Environment (OSATE)**, enabling engineers to map cybersecurity requirements to system components using Architecture Analysis and Design Language (AADL). This structured modelling helps not only to identify requirements violations and verify security measures, but also to build upon this in order to trace the fulfilment of these requirements considering the modelled system comprising multiple interconnected components. Thus, OSATE, with its formal modelling capabilities, ensures that each system component aligns with the overarching security goals in the context of specific application domains.

Through OSATE, and its RESOLUTE extension, engineers can build **assurance cases**, i.e., *formal arguments linking high-level security objectives to evidence concerning the behaviour of low-level components*. To this end, RESOLUTE generates verification evidence directly from AADL models, improving transparency and trust in the system. Each REWIRE use-case follows this architecture-driven method, linking requirements to verification results from formal tools, with the overarching goal of assessing how much assurance REWIRE truly provides compared to what was initially envisioned during the modelling phase, following the previously outlined bottom-up approach. Consequently, this process also serves to validate the security properties claimed to be provided by REWIRE-enabled devices, considering their

interconnectivity within the overall domain architecture. The REWIRE process can then be expanded to any type of application domain comprising mixed-criticality services and devices, considering that it requires a distinct **security profile** to be met.

As aforementioned, this process is performed during the **design-time Phase of the REWIRE framework**, which has been described in detail in D2.2, and is essential for establishing a secure and trustworthy foundation before implementation and deployment. This process follows a **compositional approach**, where rather than verifying the entire system monolithically, the trust assessment architecture is broken down into smaller, more easily verifiable modules. This is primarily motivated by the need for manageable verification and is particularly suited to focusing on properties that enable a device to restore its trust level, such as securely performing a software update with strong guarantees that the process cannot be circumvented. At the beginning of this process, stakeholders first define the **system's functional and security requirements** (e.g., cryptographic protocols, binary instrumentation). These are then **modelled in AADL and formalized using RESOLUTE**. **Assurance cases** are developed to **break down high-level requirements into sub-requirements**, linked to software/hardware components, and supported by evidence from formal verification tools like model checkers (nuXmv, CBMC), and protocol verifiers (Verifpal, Tamarin, ProVerif).

In the following, this methodology is exemplified by application to two use cases, respectively presented in Section 5.3 and Section 5.4: *Adaptive In-Vehicle SW & FW Patch Management & Software Functions Migration, and Smart Cities for Empowering Public Safety.* The next two sections, each dedicated to one of the use cases, are structured in the same way: (i) the security requirements are introduced in form of **system properties** (defined at a system level), (ii) one assurance case per property is formulated, to argue that the property holds in the system, (iii) a system AADL model is described, together with the properties in the Resolute language, (iv) the assurance cases are translated to the Resolute language and the AADL model is extended to support the concepts necessary to build the arguments, (v) verification through the Resolute tool is carried out and the analysis results are reported.

5.2 Summary of Secure SW Update Security Properties

An important step that needs to be performed prior to the definition and tracing of the fulfilment of the assurance cases in the context of REWIRE is to summarise the **security properties** for the SW update process, which will guide the ratification and validation of the fulfilment of the underlying requirements. As described in D3.2, the **Trusted Computing Group (TCG)** cyber-resilience documentation [72] explicitly defines the need for two properties regarding the SW update process, specifically:

1. **SW Update Authenticity and Integrity:** No unauthorised party should be able to install a malicious SW update. Thus, there should be the required trustworthiness guarantees in order to ensure that each SW update originates from the intended entity (e.g., Manufacturer or Service Provider), it is installed to the correct recipient device, and the unmodified and original update has been installed on the device. In the context of REWIRE, this is achieved through the use of LR-BC-2, the verification process of the signed update within a TEE enclave. In addition, a Configuration Integrity Verification (CIV) attestation process is performed both before the update is installed, and after the update in order to verify that the post-installation state of the device is correct and expected.

2. **App State Confidentiality and Integrity:** This dictates that no unauthorised party can read or modify the App state during the SW update process. This may apply to the state restoration process within the device, i.e., when moving to a new enclave containing the updated version of the application. According to the TCG, a recovery process is needed for detecting and recovering from failed or malicious updates, as well as a guarantee that the recovery process will not roll back to vulnerable firmware. These are provided in REWIRE through the use of the Keystone TEE. [1]

[1] We are aware of a limitation of the current Keystone implementation: at one stage, the App state is processed in the

As aforementioned, the SW update process of REWIRE offers two modalities, namely (i) the **1-to-1**, which is the core process followed in realistic *automotive scenarios*, and (ii) **1-to-many**, which is more practically applicable in *smart cities scenarios*. Here, we translate the two previously defined security properties, which will be the basis for the analysis provided in the following regarding the Automotive and Smart Cities use case.

Specifically, in the context of the **1-to-1 modality**, consider the existence of a **Manufacturer** who may be either a Tier 1 or Tier 2 automotive vendor, and is responsible for sharing a pre-established symmetric key (instantiated in REWIRE as an LR-BC-2 Key), which is used for the **Authenticated Encryption (AE)** of the update which is shared and pushed from the Manufacturer to the **Electronic Control Unit (ECU)** of the vehicle. Thus, considering that we employ AE for protecting the security and authenticity of the update, the core issue is to identify how this is translated when applied to the overall flow.

On the other hand, in the case where multiple devices need to obtain the same update through the **1-to-many modality**, as in the Smart Cities use case, *the Manufacturer or Service Provider is not able to encrypt the update with the different preshared keys of each device*, since the encryption process cannot be scaled to such a scenario. In this case, while the update cannot be considered confidential, it can be *signed with the Private Key (PK) of the Service Provider in order to ensure its integrity*. Then, when it is pushed through the **Blockchain Infrastructure**, all recipient devices of the update can validate its authenticity and ensure that it originates from the correct Service Provider. In this regard, the question is, *how can we be sure that, when the update is distributed by the Policy Orchestrator through the Facility Layer of the device (which orchestrates the update process), it has not been compromised?* To this end, as demonstrated in D3.2, we utilise the pre-established LR-BC-2 Key in order to securely communicate the output of the attestation process to the Service Provider using AE, so that it can be ensured that the correct update has been successfully installed.

In the following Sections, we provide the **assurance cases for both the Automotive and Smart Cities use cases**. Afterwards, using AADL, we demonstrate how these can be achieved when AE is used in different phases of the Secure SW update process. Specifically, considering all the above, the key difference between the assurance cases in the two modalities is as follows: **In the Automotive use case, the authenticity of the update is achieved using the LR-BC-2 Key which is used to protect the update itself**, while **in the Smart Cities use case, the LR-BC-2 Key is used in order to verify and validate the correct output of the update process**.

5.3 Automotive

In recent years, there have been increasing needs in automotive applications with regards to aspects such as smart mobility, connectivity, electrification, etc. These developments have led to rapid advancements in automotive technologies and the implementation of increasingly complex software running within the **Electronic Control Units (ECUs)** in vehicular architectures, such as **Autonomous Driving (AD)**, **Advanced Driving Assistance Systems (ADAS)**, **Augmented Reality (AR)**, etc. Thus, the ever-increasing complexity of the automotive stack intensifies the need for verified SW and HW co-designs that can guarantee the assurance-by-design quality, especially when it comes to safety-critical subsystems or vehicles.

One such technology, which is becoming increasingly important for the future of vehicle connectivity, is the capability for **Over-the-Air (OTA) updates**, which will allow for remote upgrades and patches in vehicle functionalities, thus introducing significant benefits both to the automotive **Original Equipment Manufacturers (OEMs)**, and the end users. Specifically, while recalls have so far been a common practice for addressing SW malfunctions, this process induces significant repairing expenses and overall financial burden to the OEM. To address this issue, OTA updates will enable *delivery of SW patches*

untrusted world in unencrypted form, which could allow information leakage. We anticipate this issue will be addressed in the next release of Keystone.

directly to the cars requiring updates, upgrades, and bug fixes. Thus, OTA updates will not only *alleviate the need for the owner to bring the car to the dealership*, but it will also *allow warranties to be kept intact*, *minimise the number of recalls*, and *enable frequent patching in order to keep vehicle functionalities up-to-date*.

Current automotive electrical/electronic (E/E) architectures typically follow a decentralised architecture, where each specific vehicular function is managed by an individual ECU, and a common bus network connecting to various ECUs within the vehicle. However, this approach presents various drawbacks, particularly regarding scalability and communication. Thus, the KENOTOM Automotive Use Case of REWIRE follows a **centralised ECU network architecture**, focused on grouping functions into a single ECU. However, while a basic assumption is that if an ECU supports firmware decryption and authentication the update file can be sent directly to the OTA Manager of the vehicle, *not all ECUs have secure key storage and HW-accelerated security*. Thus, **the secure SW update functionality of REWIRE aims to address this issue, by providing the required security, integrity, and authenticity guarantees to the update process**. In the following, we provide the assurance cases and their validation for the SW update process in the context of the Automotive use case, and the validation of the properties highlighted in Section 5.2.

5.3.1 Security Properties and System Description

Here, we provide a high-level description of the system model that will guide the definition of the Assurance Cases and their verification in the context of the Automotive use case. Note that this is based on the use case architecture that has been presented in D6.1, but given the complexity of the processes taken into account, the scope of the modelling process, and the level of detail of the processes that the models are based on, *the construction of the AADL models only considers the components which are necessary for the formulation of the assurance cases*. Thus, the model is constructed with the level of detail in the system representation required for the successful verification of the properties of interest, as presented in Section 5.2.

With respect to the use case sequence diagram reported in Fig. 5.1, the system can be seen as the interaction of the following entities, from the left to the right:

- **ECU Manufacturer**, designer of the security policies;
- **High-Level ADAS Functions and ADAS ECU** (ADAS, for brevity), that exchange information on the vehicle state with App1 and App2;
- **Policy Orchestrator (PO)**, that receives and deploys security policies and SW updates;
- **Automotive App1**, a first instance of a Powertrain controller application, that can be compromised by an attacker;
- **Enclavised Data Sender (EDS)**, residing in App1 enclave, that manages encryption of the App state and transmission to Enclavised Data Sender Update;
- **REWIRE TCB**, the REWIRE trusted computing base that provides functionalities such as attestation;
- **Enclavised Data Sender Update (EDSU)**, counterpart of EDS residing in App2 enclave;
- **Automotive App2**, a second instance of the controller application, that is more secure thanks to a SW update;
- **REWIRE Risk Assessment Component and SW/FW Validation Component**, that process the report received upon failed attestation;

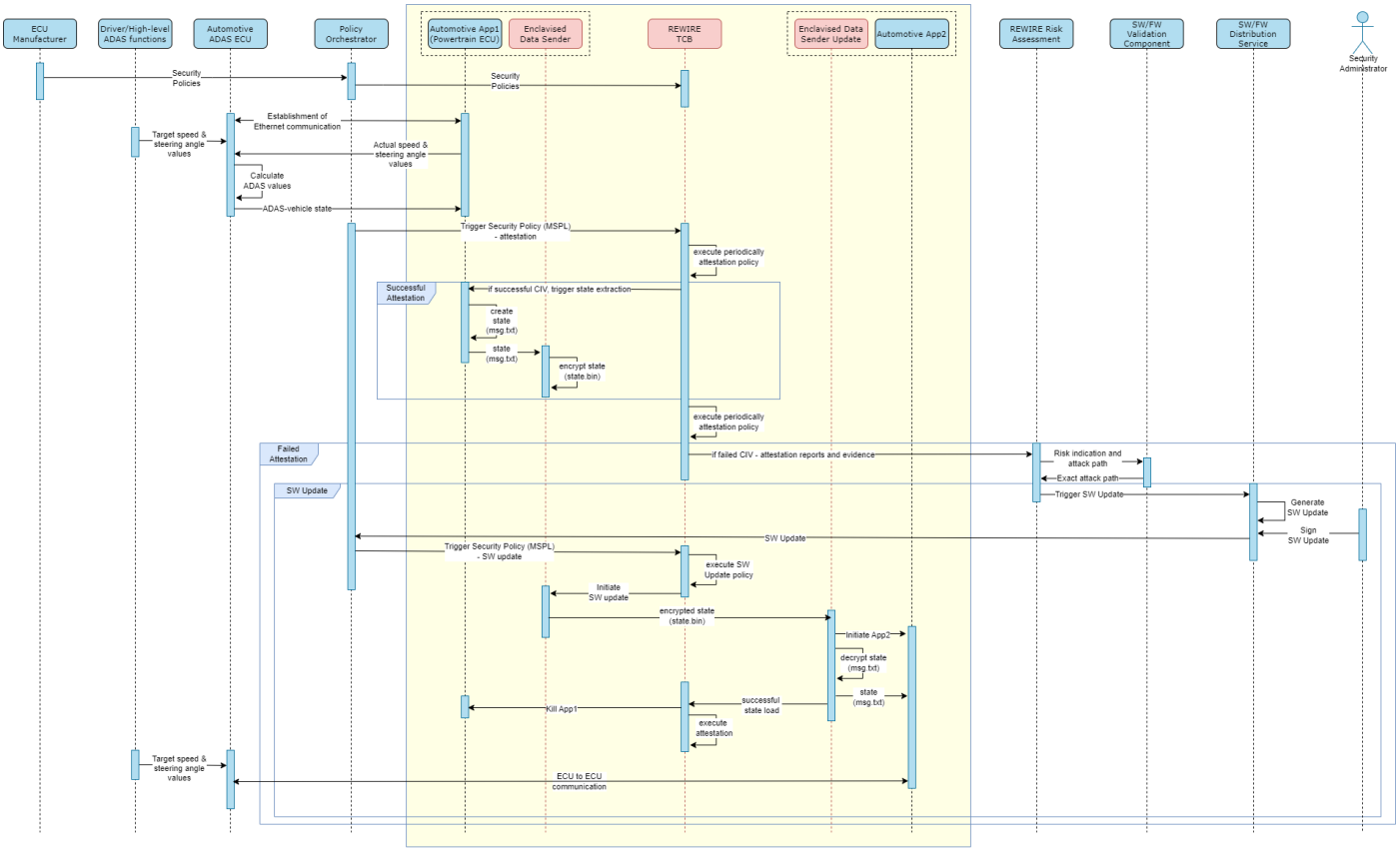


Figure 5.1: Automotive Use Case Sequence Diagram

- **SW/FW Distribution and Security Administrator**, the latter being responsible for developing and signing the SW update, that is transmitted by the former.

In the following, all entities are assumed to be functionally correct. From a security perspective, it is important to consider both the entities' behaviour and the way they communicate. Entities are distinguished between trusted, whose behaviour matches their specification, and untrusted, subject to vulnerabilities exploited by an attacker, that might try to exploit them for malicious purposes or obtain illicit access to them; communication channels (IPC, Ethernet, Internet) are assumed to be reliable in delivering a message to the intended recipient (i.e., denial-of-service-like attacks are excluded), but do not prevent an attacker from reading and/or modifying the message; a communication protocol (whose design is outside of the scope of REWIRE, and is always aligned with what is recommended in the standards), adopted on a given channel, might employ means to enforce properties such as confidentiality, integrity, and authenticity of the exchanged messages.

We assume Manufacturer, ADAS, EDS, EDSU, REWIRE TCB, and App2 to belong to the "trusted world", while the others to the "untrusted world". In HW-rooted security, the separation between trusted and untrusted world refers to the separation of computing environments based on the level of trust and security. Specifically, the **trusted world** encompasses the components and processes that are considered reliable and secure, often including hardware-based security features like Trusted Execution Environments (TEEs) and cryptographic modules. The **untrusted world** includes the rest of the system, such as the operating system, applications, and potentially even the network, which may be susceptible to vulnerabilities and attacks.

Note that, while App1 is untrusted, App2 is trusted to be correct due to being initiated after a correct SW update, and restoring a state that was previously successfully attested. This is not a heavy assumption, because we are relying on attestation capabilities enhanced with the crypto protection measures provided by REWIRE, thus we can be certain that the attested state is correct. This allows to assert that the state

restored by App2 was not compromised and corresponds to the most recent attested version. However, it may be compromised after state restoration.

As the focus in the following is on the far edge device, in order to facilitate the validation of the assurance cases, the **REWIRE Risk Assessment Component**, **SW/FW Validation Component**, **SW/FW Distribution Service**, and **Security Administrator** are merged into a single untrusted entity, called **"Other Entities" (OE)** for simplicity, as the type of entity does not affect the validation process.

First, let us consider the untrusted entities, and the potential effect of an inappropriate behaviour from their side. App1 might send obsolete or wrong values to the ADAS (note that these may not always be the result of an attack, but may also be caused by a malfunctioning sensor), with a chain impact on the whole state saving and restoring process; this risk is mitigated by ADAS capability of detecting such a situation with some degree of certainty, by comparison with values provided by other sensors in the system, as well as observing the fluctuation in the data sent by App1. Thus, in the case of values that fluctuate, this will trigger a flag of a possible misbehaviour.

Note that **the PO is not inherently trusted**; however, its role is mainly to forward policies and SW updates encrypted and authenticated by other entities, so its margins for jeopardizing the communications are negligible. In fact, while it can withhold a security policy that was issued after the design phase was completed, such an attack is easily detectable, and falls outside the scope of the threat model here adopted. The PO cannot interfere with the attestation process either. Attestation, at least in the chosen implementation, is in fact initiated by the REWIRE TCB with a fixed periodicity, as specified by the received security policy, rather than on demand by the PO, in turn notified of a change of App state by the monitoring Tracer. In case of a failed attestation, **the OE receive failure evidence from the REWIRE TCB and provide an updated SW package**. We assume that the SW package is correctly developed based on failure evidence analysis and take care of any vulnerability detected in the running App1.

Second, let us focus on the communication protocols and the security mechanisms put in place. The ECU, the ADAS, and the PO are executed outside of the enclave. The ECU and the ADAS are trusted, and outgoing messages enjoy confidentiality (especially relevant when it comes to state information), integrity and authenticity, which is specified in the **ETSI** [36] and **Car-to-Car Communication Consortium (C2C-CC)** [20] documentation on the protection of the CAN bus. The EDS, the EDSU, App2, and the REWIRE TCB are trusted and executed in isolation instantiated as separate enclaves; communication among these entities is mediated by the **Security Monitor (SM)** of the REWIRE TCB, which guarantees confidentiality, integrity, and authenticity. Essentially, leveraging Keystone [50], the SM provides the foundation for secure communication by managing memory isolation and enforcing access control, and cryptographic protocols ensure the confidentiality, integrity, and authenticity of the communication itself. Keystone also leverages **Physical Memory Protection (PMP)**, a RISC-V feature, to isolate enclaves from each other and the host OS. This prevents unauthorized access to enclave memory, ensuring confidentiality and integrity. In addition, mutual attestation allows enclaves to verify each other's trustworthiness before engaging in sensitive operations.

App1 is also wrapped inside an enclave, but, for what concerns enforcing the main execution logic, belongs to the untrusted world. Specifically, App1 has two counterparts, one running inside the trusted world for managing the persistent state (e.g., Keystore, if any), and one running in the untrusted world, responsible for enforcing the main execution logic; thus, integrity of the data being transmitted between the trusted and the untrusted world, could, in principle, be compromised by an attacker. Confidentiality and integrity are ensured in this case by the use of a key, protected by secure storage, to encrypt exchanged messages. Note that this is an assumption that we are making in our implementation, and in a realistic scenario where App1 is running as part of an ECU, then all ECUs (during maintenance and onboarding of the ECUs in the overall in-vehicle network) are considered to have been equipped with all application keys from the other ECUs. Therefore, this can be leveraged in order to verify a form of access control in the receiving entity. Regarding the communication between App1 and the ADAS, it takes place between ECU and ECU with an appropriate protocol [46]. As aforementioned, the PO can decide to

not forward the policies through, but the sending entities are responsible for enforcing authenticity and integrity of the messages PO forwards around.

The OE are executed outside of an enclave. We assume that communications from REWIRE TCB to OE and from OE to PO have authenticity and integrity, thus preventing an attacker from successfully (i) modifying the failed attestation evidence, based on which the SW update packaged is developed, and from (ii) tampering with the SW update package itself or impersonating the distribution service in sending a malicious update package (the possibility of replaying an old package is discussed later in the text). In particular, a message from TCB to OE undergoes two signatures, one by the Attestation Agent in the REWIRE TCB, the second from the Security Monitor (leveraging the underlying HW-based key of the Root-of-Trust), while a message from OE to PO benefits from employing the LR-BC-2 mode of operation.

5.3.2 Assurance Cases

The following sections examine each of the two target properties separately, providing arguments to support their validity in the system. The arguments follow step-by-step, in a top-down fashion, the communication flow outlined in the sequence diagram of Fig. 5.1, where a step corresponds to a message (data, notification, etc.) exchanged between two entities; each step is briefly described and evaluated in terms of its relevance to the property under account.

We assume the security policies to have been preliminarily deployed by the PO to the REWIRE TCB; any attempt to modify or replay data is expected to be detected, thanks to digital signature and encryption. Note that this only refers to the communication of the actual SW update, which is signed by the distribution service through the LR-BC-2 Key, and not for all data in general. In the context of REWIRE, while all data exchanged between entities is generally signed, **Authenticated Encryption (AE) is much stronger than simple authentication** which is achieved, for instance, for the sharing of policies.

5.3.2.1 App State Confidentiality and Integrity

Among the entities identified in this use case, the ones that are expected to have access to App1 state in plaintext are EDS, EDSU, REWIRE TCB, that performs attestation, and App2, that performs state restoration.

App1 → ADAS. App1 sends vehicle values to the ADAS; confidentiality and integrity are ensured by communication mechanisms between ECU and ECU. App1 might display a malicious behaviour in sending falsified data to the ADAS; the ADAS is assumed able to recognize bad data as previously explained.

ADAS → App1. The ADAS sends vehicle state to App1. The ADAS is trusted; confidentiality and integrity are ensured, as per above.

PO → REWIRE TCB. The Policy Orchestrator sends an attestation request to the TCB through the facility layer, according to schedule. The PO is not trusted, but communications from the PO are authenticated and have integrity. Note that, according to the pre-defined policies, if the schedule is not followed, this can be checked by an external verifier, as the policies are also deployed/stored on the Blockchain as public information. Considering that the policies are signed by an external trusted entity (output of the design-phase), matching of the scheduling of the attestation requests can be ratified.

TCB. The TCB performs attestation on App1. The TCB is trusted, the attestation process is safeguarded to be executed correctly, and guarantees state confidentiality, thanks to CIV features. Note that, in this context, confidentiality refers to the guarantee that no details on the raw evidence are shared outside of the device, thus achieving the zero-knowledge property.

TCB → App1. If attestation is successful, the TCB sends a state extraction request to App1. Communication is managed through the SM, so it is signed by either the main HW-based key or another dedicated key for this process. No impersonation of the TCB can occur, and the integrity of the request cannot be compromised.

App1 → EDS. App1 performs state extraction and sends the state to the EDS; App1 has just been successfully attested, so it can be considered trusted. Confidentiality and integrity of the communication are ensured by the use of a sealing key by App1. Note that we make an assumption on the secure communication between App1 and the secure key storage, which requires the existence of a mechanism (e.g. SPDM protocol) for ensuring that the state does not stay unprotected in the untrusted world. Then, the EDS encrypts and stores the state.

. . . SW update takes place . . .

EDS → EDSU. Upon update trigger from the TCB, the EDS sends the encrypted state to EDSU. The EDS is trusted and communication takes place within the trusted world. The update is also signed with the sealing key of the EDS, which can then be verified by the EDSU for ensuring authenticity.

EDSU. The EDSU initiates App2. Note that the EDSU is trusted, since it is part of the TCB.

EDSU → App2. The EDSU decrypts the state, and sends the state to App2, that runs in the trusted world, but in a different enclave. Thus, the EDSU is trusted and communication takes place within the trusted world, but the EDSU and App2 are protected through different enclaves. However, there is the risk of a fork attack, as shown below.

App2 → EDSU. App2 restores the state and notifies the EDSU. App2 is executed within an enclave, and consists of code cleaned of detected vulnerabilities by the software update (however, there is the risk of a replay attack, see below); the restored state was attested to be correct. Communication also takes place within the trusted world.

EDSU → TCB. The EDSU notifies the TCB of App initiation. The EDSU is trusted and communication takes place within the trusted world.

TCB. The TCB performs attestation on App2. The TCB is trusted. App state integrity could still be compromised, if a replay or a fork attack were successfully performed: in the first case, the SW update could have been based on a correct but obsolete package with known vulnerabilities; in the second case, the update could have been circumvented, to initiate a second instance of App2 running the vulnerable software, that would play the role of the original App2 during the execution.

TCB → PO. The output of the final attestation is signed with the attestation key by the TCB, and sent through the PO to a trusted external verifier for an independent assessment of the correctness of the update process.

5.3.2.2 SW Update Authenticity and Integrity

PO → REWIRE TCB. The Policy Orchestrator sends an attestation request to TCB. See previous section.

REWIRE TCB. The TCB performs attestation on App1. See previous section.

TCB → OE. If attestation fails, the TCB sends a failed attestation report to the Other Entities. The data is not encrypted, but undergoes two signatures, the first performed with the *Attestation Key*, and the second with the *HW-based key of the Security Monitor*; any modification by an attacker is detected, and integrity is ensured.

OE → PO. The OE are assumed to overall behave in the expected way, in developing a SW update package and in sending it to the PO. The package is subject to authenticated leakage-resistant encryption by means of the LR-BC-2 mode of operation.

PO → TCB. The PO forwards the SW update package to the TCB. Authenticated encryption prevents successful tampering by the PO or by an attacker.

TCB → EDS. The TCB performs App SW update and sends an update trigger to the EDS. The TCB is trusted and communication takes place within the trusted world.

5.3.3 System Model

AADL is a very rich language, and provides the means to detail both software and hardware of a target system; in addressing the Automotive use case, the modeling activity focused on describing the high-level communication aspects of the framework, and the control and data flows realized by the entities' interactions expressed by the sequence diagram.

The nature of the properties to be verified did not require distinguishing the entities into different categories, rather it was enough to model each entity as a generic agent, with a well-defined communication interface. The following systems, introduced in Section 5.3.1, were included: ADAS ECU, PO, App1, EDS, TCB, EDSU, App2, and Other Entities. Each interaction, corresponding to a message (data, notification, trigger,...) being sent by a sender to a recipient, was individually modeled by (i) specifying the communication means (wired ethernet, wireless internet, IPC), (ii) respectively defining an outgoing and an ingoing port in the sender and in the recipient systems, and (iii) adding a correspondent directed connection in the overall system combining the entities.

However, the presence of pairwise connections is not sufficient to represent complex processes such as software update with state restoration, consisting of a sequence of interactions with branching points. To this aim, the notion of flow in AADL proved essential: the language allows to encode observable flows of logic at any level in a model hierarchy, either originating within a system (source), ending within a system (sink) or passing through a system, from input ports to output ports (path), possibly including subcomponents and connections among them.

In particular, two main flows were identified and modeled at top level: (i) one characterized by successful attestation, that begins with App1 receiving vehicle data from the ADAS, proceeds with successful attestation, and ends with the EDS encrypting and storing the state sent by App1; (ii) one characterized by failed attestation, that begins as before, but proceeds with failed attestation, software update, App2 initiation and state restoration, and ends with the TCB performing attestation and reporting to the PO. A visual representation of the AADL model is given in Fig. 5.2, while Fig. 5.3 shows a portion of the textual model that includes the flows.

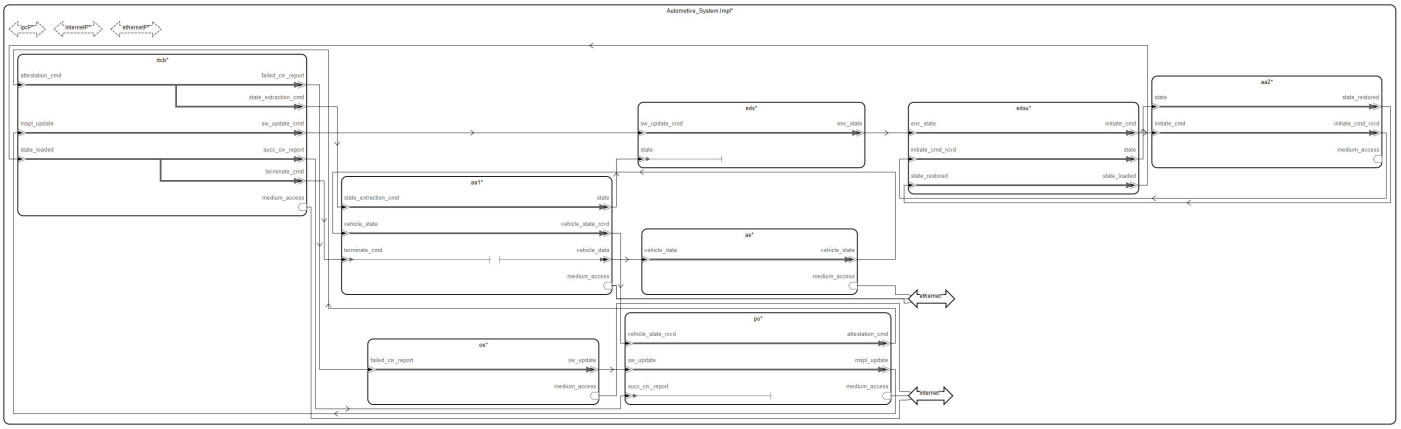


Figure 5.2: Automotive Use Case AADL Diagram

```
    c15: port edsu.state_loaded -> rtcb.state_loaded;
    c16: port rtcb.terminate_cmd -> aa1.terminate_cmd;
    c17: port rtcb.succ_civ_report -> po.succ_civ_report
        { System_Properties::Security_Mechanism => Signature; };
flows
    -- attestation success: save state
    f1: end to end flow aa1.f0 -> c0 -> ae.f1 -> c1 -> aa1.f3 -> c2 -> po.f2 -> c3 ->
                        rtcb.f2 -> c4 -> aa1.f1 -> c5 -> eds.f1;
    -- attestation failure: update policy, start App2 and restore state, check update and inform PO
    f4: end to end flow aa1.f0 -> c0 -> ae.f1 -> c1 -> aa1.f3 -> c2 -> po.f2 -> c3 ->
                        rtcb.f3 -> c6 -> oe.f1 -> c7 -> po.f1 -> c8 -> rtcb.f4 -> c9 -> eds.f2 -> c10 -> edsu.f1 -> c11 ->
                        aa2.f1 -> c12 -> edsu.f2 -> c13 -> aa2.f2 -> c14 -> edsu.f3 -> c15 ->
                        rtcb.f6 -> c17 -> po.f3;
properties
    Actual_Connection_Binding => (reference (internet)) applies to internetP;
    Actual_Connection_Binding => (reference (ethernet)) applies to ethernetP;
    --
```

Figure 5.3: Automotive Use Case AADL Model

5.3.4 Assurance Cases Encoding

Assurance cases in Resolute are structured as a hierarchy of goals and sub-goals, corresponding to predicates returning a true/false value, and auxiliary functions. The target requirements were formalized as main goals, respectively stating that the SW update with state restoration process ensures authenticity and integrity of the SW update package, and confidentiality and integrity of the App state. The main goals were associated with the two AADL flows outlined in the previous section, the first goal with flow (i), the second goal with flow (ii); each goal is supported by a conjunction of sub-goals, structurally matching the steps the flows are made of. Fig. 5.4 reports the Resolute encoding of the goal pertaining to SW package authenticity and integrity.

```
-- SW update authenticity and integrity
goal a_SW_update_with_state_restoration_ensures_package_authenticity_and_integrity(s: system) <=
** "SW update with state restoration ensures package authenticity and integrity" **
    debug("Appl attestation request transmission from PO to TCB is correct") and
    attestation_request_transmission_from_PO_to_TCB_is_correct(s)
    and debug("Appl attestation execution by TCB is correct")
    and attestation_execution_by_TCB_is_correct(s)
    and debug("failed attestation report transmission from TCB to OE has authenticity and integrity")
    and failed_attestation_report_transmission_from_TCB_to_OE_has_authenticity_and_integrity(s)
    and debug("SW update package development by OE is correct")
    and SW_update_package_development_by_OE_is_correct(s)
    and debug("SW update package transmission from OE to PO has confidentiality and authenticity and integrity")
    and SW_update_package_transmission_from_OE_to_PO_has_confidentiality_and_authenticity_and_integrity(s)
    and debug("SW update package transmission from PO to TCB has authenticity and integrity")
    and SW_update_package_transmission_from_PO_to_TCB_has_authenticity_and_integrity(s)
    and debug("SW update by TCB is correct")
    and SW_update_by_TCB_is_correct(s)
    and debug("SW update notification by TCB to EDS is correct")
    and SW_update_notification_by_TCB_to_EDS_is_correct(s)
```

Figure 5.4: Automotive Use Case Assurance Case Main Goal

In accordance with the assurance cases, sub-goals are used to argue either that the functions executed by the system entities are correct, or that the communication steps satisfy the necessary security properties for the information being exchanged. These sub-goals also serve as the tracing point to specific sub-requirements. For instance, the first goal is supported by evidence demonstrating the correctness of the CIV attestation protocol, as well as by the LR-BC-2 mode of operation's ability to provide authenticated

encryption, thereby ensuring the integrity and authenticity of the software update package (Fig. 5.5). The second goal is backed by the fact that CIV operates as a zero-knowledge protocol, which guarantees the confidentiality of the App's state. This supporting evidence may come from probabilistic or formal analysis techniques, such as those outlined in Section 3.

```
goal SW_update_package_transmission_from_OE_to_PO_has_confidentiality_and_authenticity_and_integrity(s: system) <=
** "SW update package transmission from OE to PO has confidentiality and authenticity and integrity" **
    let c1: system = get_component("oe", s);
    let c2: system = get_component("po", s);
    OE_behavior_is_correct(s)
    and communication_has_lrbc2_encryption(c1, c2)

goal communication_has_lrbc2_encryption(c1: system, c2: system) <=
** "Communication between " c1 " and " c2 " has LR-BC-2 encryption" **
    let con: connection = get_connection(c1, c2);
    communication_has_authenticated_encryption(c1, c2)
    and has_property(con, System_Properties::LRBC2_AE_Evidence)
```

Figure 5.5: Automotive Use Case Assurance Case Evidence Linkage

5.3.5 Analysis Results

Two claims were placed in the system AADL model encoding, each instantiating one of the main goals (Fig. 5.6).

```
    annex Resolute {**
        prove a_SW_update_with_state_restoration_ensures_package_authenticity_and_integrity(this)
        prove a_SW_update_with_state_restoration_ensures_state_confidentiality_and_integrity(this)
    **};
end Automotive_System.Impl;
```

Figure 5.6: Automotive Use Case Assurance Case Claims

Resolute was able to prove the claims in negligible time; the tool reports the chains of reasoning that, respectively, provide arguments in support of the capability of the SW update process to ensure SW package authenticity and integrity, on one side, and App state confidentiality and integrity, on the other side (Fig. 5.7).

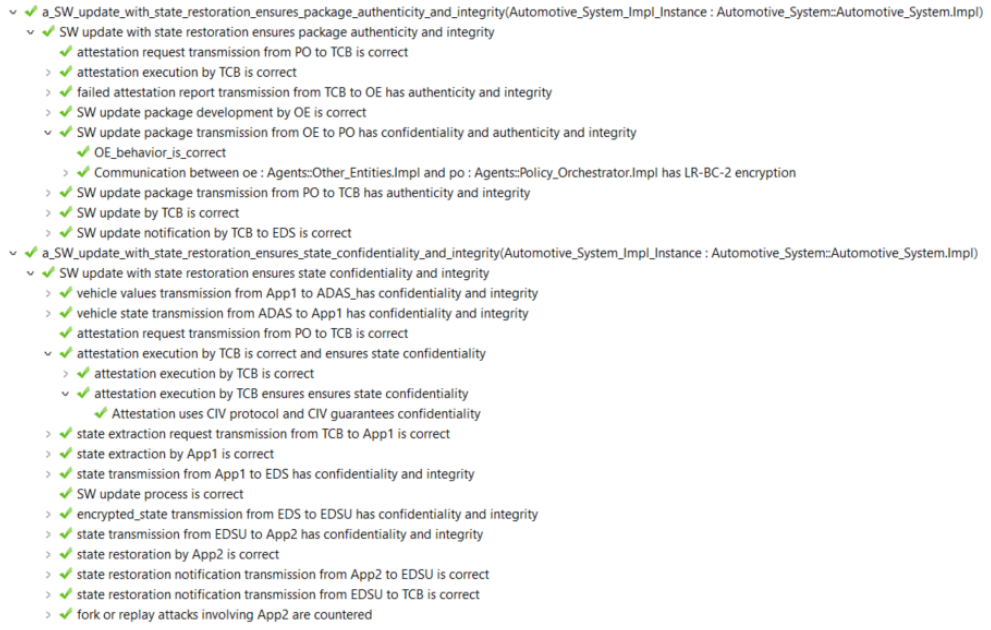


Figure 5.7: Automotive Use Case Assurance Case Analysis Results

5.4 Smart Cities

Modern urban scenarios typically involve the deployment of **large-scale Internet-of-Things (IoT) networks** aiming to perform a wide range of functionalities, such as Intelligent Transportation Systems,

Smart Agriculture, Industry 4.0, etc. Thus, there are expectations of rapid growth in such infrastructures, incorporating various technological paradigms, including **Infastructure-to-Vehicle (I2V)**, **Vehicle-to-Vehicle (V2V)**, **Industrial IoT (IIoT)**, and **remote surveillance**. However, the expansion of the scope of such environments may lead to various issues, such as the compatibility between heterogeneous IoT systems and devices originating from different vendors, and the introduction of vulnerabilities stemming from the interconnectivity between such different devices.

As in the automotive case, the capability to perform updates to such devices in a secure and authenticated manner is critical towards addressing such vulnerabilities. In Smart Cities infrastructures, a typical **Firmware Update Over-the-Air (FUOTA)** scenario involves the rollout of an updated firmware version for end-devices through a **purpose-specific service** to a **centralised server** through external APIs, indicating which devices the update must be distributed to. Next, the centralised servers of the Smart City deployment trace a new route for the firmware to be transmitted, and it is sent through the **backhaul network** towards the **radio edge communication components**. Finally, the firmware is transmitted to the end-devices through the employed **radio access technology**, and the update is validated and installed by the intended devices.

In this regard, the **Secure SW Update** functionality of REWIRE is able to offer strong security, integrity, and authenticity guarantees to the process of the deployment of such patches. Specifically, REWIRE enables the installation of OTA updates, without the need for regular human supervision, and guarantees that the update has been installed correctly. It also offers **authentication of the SW/FW update origin**, as it ensures that only authorised users can trigger firmware updates and prevents malicious parties from installing unauthorised software, and enables **post-management of updates**, thus preventing unexpected malfunctions in critical infrastructure. For instance, as a traffic light presenting an unexpected behaviour after an unsuccessful update is likely to cause a traffic accident, REWIRE provides the required guarantees for the prevention of such incidents.

In the following, we provide the assurance cases and their validation for the SW update process in the context of the Smart Cities use case, and the validation of the properties highlighted in Section 5.2. First, the 1-to-1 scenario is examined and compared to the Automotive use case. Then, the 1-to-many scenario is presented, with emphasis on how involving multiple devices in the software update process affects both the communication flow and its modeling.

5.4.1 1-to-1 Scenario

This section is dedicated to the 1-to-1 scenario where an update is distributed to a single device, and its application in the context of the Smart Cities use case. In this regard, we present the set of security properties and the assurance cases for their verification.

5.4.1.1 Security Properties and System Description

The sequence diagram of Fig. 5.8 illustrates the system components and their interactions in the context of the Smart Cities use case that will guide the definition of the assurance cases and their verification. When compared to Fig. 5.1, we observe that the core architecture of the Smart Cities and Automotive use cases is fundamentally the same. Specifically, on the right-hand side, both show how the Other Entities communicate with the REWIRE TCB and the Policy Orchestrator to deliver the SW update package. The EDS and EDSU are each associated with different versions of the App, while the Policy Orchestrator is responsible for deploying policies and initiating App attestation.

The main distinction lies on the left-hand side of 5.8. Specifically, instead of featuring the ECU Manufacturer and the ADAS as in the Automotive use case, the Smart Cities use case introduces a **Keystore**. This component securely stores an MQTT broker access password, encrypted using a pre-installed AES key. Rather than capturing vehicle measurements, the state in this context - saved upon successful attestation - includes the encrypted access password.

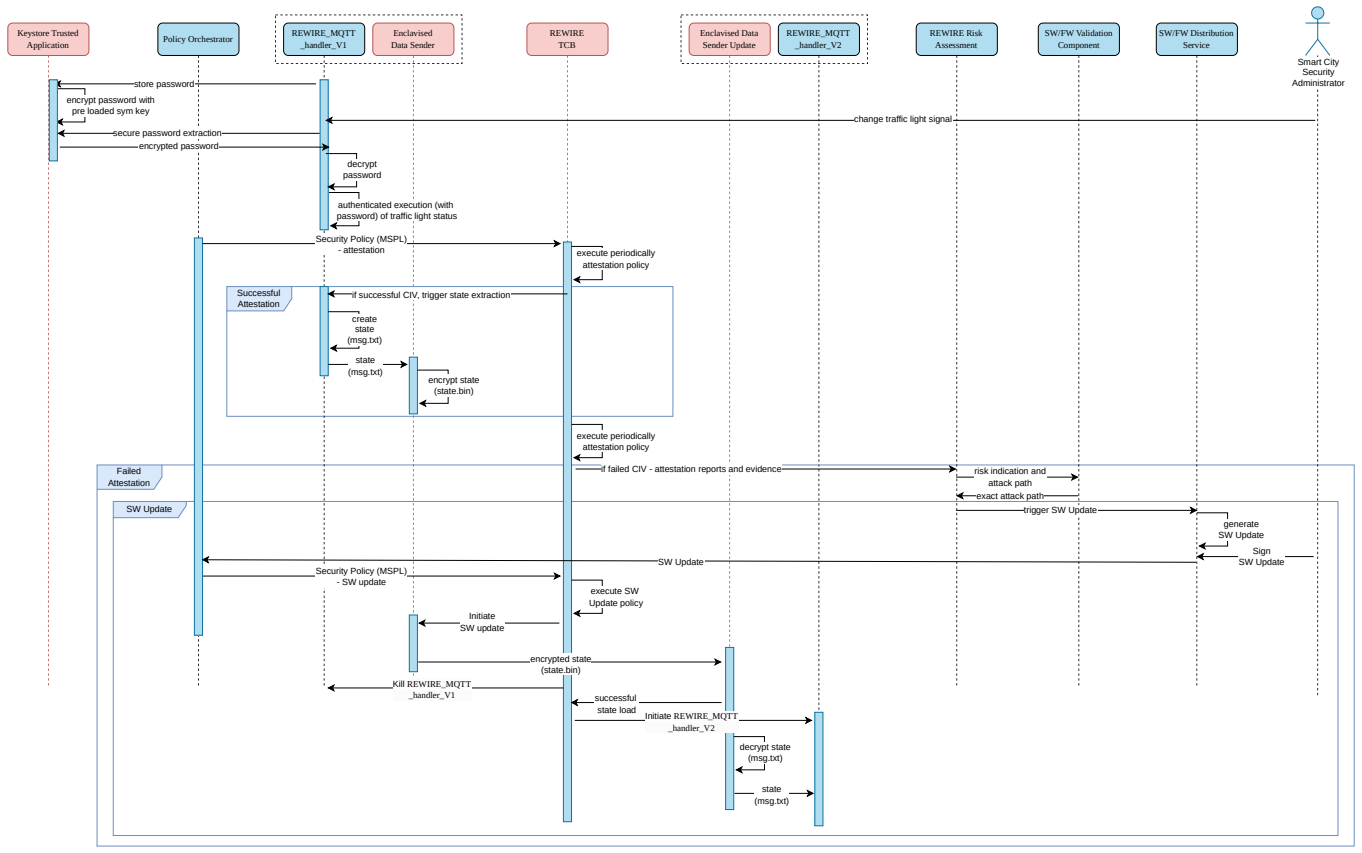


Figure 5.8: Smart Cities Use Case Sequence Diagram

The analysis so far indicates that the security requirements for the Smart Cities use case are the same as those identified for the Automotive case. These are: (i) **ensuring authenticity and integrity of software updates**, and (ii) **preserving confidentiality and integrity of the App's state**, noting that the definition of state differs slightly between the two contexts.

The assumptions regarding the trustworthiness of the system entities, as outlined in Section 5.3.1, remain valid here. The Keystore is considered a trusted component, and due to its use of authenticated encryption, the confidentiality, integrity, and authenticity of its output messages are guaranteed. Unlike in the Automotive scenario, the Smart Cities case does not involve any direct communication between the (untrusted) App and the Keystore.

5.4.1.2 Assurance Cases

Given that the Smart Cities use case shares security goals, key entities, and communication patterns with the Automotive use case, their assurance cases largely overlap in structure and reasoning. Thus, in the following, we provide the assurance cases for the Smart Cities use case, focusing on the differences between the two.

App State Confidentiality and Integrity. The App State Confidentiality and Integrity assurance case begins with this interaction:

Keystore → App. The Keystore sends state to App. The Keystore is trusted; confidentiality and integrity are ensured by authenticated encryption of the state.

From this point onward, the case proceeds as described in Section 5.3.2, from "PO → REWIRE TCB".

SW Update Authenticity and Integrity. The assurance case for Software Update Authenticity and Integrity remains unchanged.

5.4.1.3 System Model

The AADL model for the Smart Cities use case was developed similarly to that of Section 5.3.3, as it shares the same core architecture as the Automotive use case. The main difference lies in the absence of the ADAS ECU component, replaced by the Keystore; instead of a bi-directional communication between ADAS ECU and App1, there is a unidirectional flow between the Keystore and App1, where the former sends the encrypted state to the latter (Fig. 5.9).

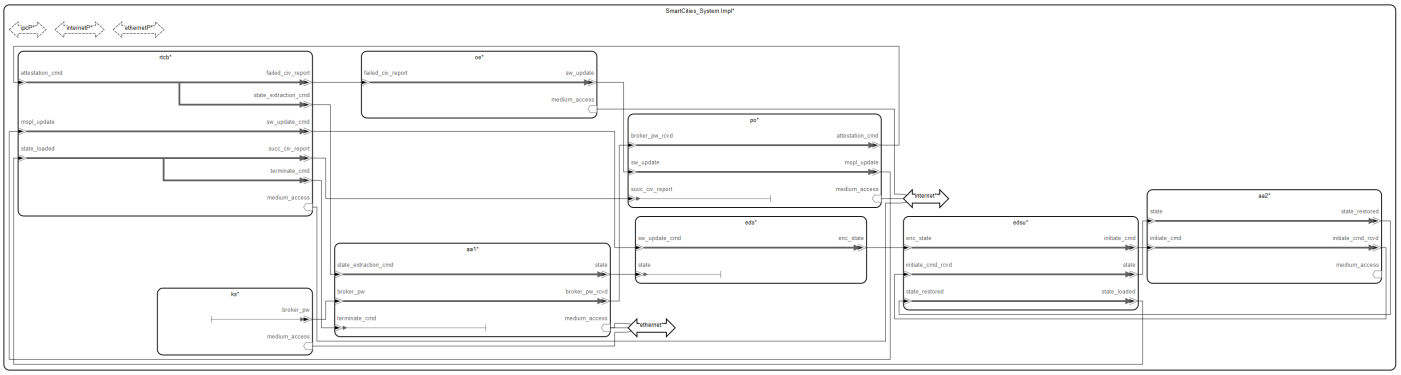


Figure 5.9: Smart Cities Use Case AADL Diagram: 1-to-1

5.4.1.4 Assurance Cases Encoding

No significant modifications are made to the encoding described in Section 5.3.4, except for those related to the App State Confidentiality and Integrity assurance case; a part of it is illustrated in Fig. 5.10, where the first sub-goal relies on the use of authenticated encryption in the communication between the Keystore and App1.

```
-- App state confidentiality and integrity
goal sc_SW_update_with_state_restoration_ensures_state_confidentiality_and_integrity(s: system) <=
** "SW update with state restoration ensures state confidentiality and integrity" **
    debug("MQTT broker pw transmission from Keystore to App1 has confidentiality and integrity")
    and MQTT_broker_pw_transmission_from_Keystore_to_App1_has_confidentiality_and_integrity(s)
    and debug("App1 attestation request transmission from PO to TCB is correct")
    and attestation_request_transmission_from_PO_to_TCB_is_correct(s)
    and debug("App1 attestation execution by TCB is correct and ensures state confidentiality")
    and attestation_execution_by_TCB_is_correct_and_ensures_state_confidentiality(s)
    and debug("state extraction request transmission by TCB to App1 is correct")
    and state_extraction_request_transmission_from_TCB_to_App1_is_correct(s)
    and debug("state extraction by App1 is correct")
    and state_extraction_by_App1_is_correct(s)
    and debug("state transmission from App1 to EDS has confidentiality and integrity")
    and state_transmission_from_App1_to_EDS_has_confidentiality_and_integrity(s)
    and debug("SW update process is correct")
    and SW_update_process_is_correct(s)
    and debug("encrypted state transmission from EDS to EDSU has confidentiality and integrity")
    and encrypted_state_transmission_from_EDS_to_EDSU_has_confidentiality_and_integrity(s)
    and debug("state transmission from EDSU to App2 has confidentiality and integrity")
    and state_transmission_from_EDSU_to_App2_has_confidentiality_and_integrity(s)
    and debug("state restoration by App2 is correct")
    and state_restoration_by_App_2_is_correct(s)
    and debug("state restoration notification transmission from App2 to EDSU is correct")
    and state_restoration_notification_transmission_from_App2_to_EDSU_is_correct(s)
    and debug("state restoration notification transmission from EDSU to TCB is correct")
    and state_restoration_notification_transmission_from_EDSU_to_TCB_is_correct(s)
    and debug("fork or replay attacks involving App2 are countered")
    and fork_or_replay_attacks_involving_App2_are_countered(s)


    goal MQTT_broker_pw_transmission_from_Keystore_to_App1_has_confidentiality_and_integrity(s: system) <=
    ** "MQTT broker pw transmission from Keystore to App1 has confidentiality and integrity" **
        let c1: system = get_component("ks", s);
        let c2: system = get_component("aa1", s);
        component_is_trusted(c1)
        and communication_has_authenticated_encryption(c1, c2)
```

Figure 5.10: Smart Cities Use Case Assurance Case Main Goal: 1-to-1

5.4.1.5 Analysis Results

Similarly to the Automotive use case, Resolute was able to verify the claims in seconds. Fig. 5.11 emphasizes the difference introduced by the App State Confidentiality and Integrity assurance case in the first reasoning step, as outlined in the previous section.

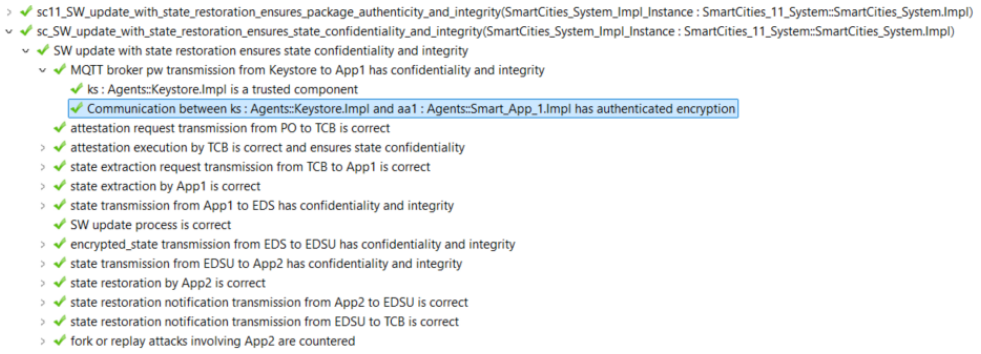


Figure 5.11: Smart Cities Use Case Assurance Case Analysis Results: 1-to-1

5.4.2 1-to-Many Scenario

This scenario is defined by the fact that several devices, rather than a single one, are undergoing the same software update. Although this may initially appear to be a minor distinction, we will examine how it affects both the communication flow and the assurance cases.

5.4.2.1 Security Properties and System Description

In the 1-to-1 scenario, the presence of a shared pre-established symmetric key - instantiated in REWIRE as an LR-BC-2 key - allows to guarantee the authenticity and the integrity of the SW update package by means of authenticated encryption; this is a key factor in preventing the untrusted Policy Orchestrator or an attacker to compromise the package, while being transmitted from the Other Entities to the PO, both residing in the untrusted world, or forwarded from the PO to the REWIRE TCB, the latter in the trusted world.
In the 1-to-many scenario, it is not feasible to employ a single SW update package encryption key, to be shared among the Service Provider and all the devices to be updated[2], making it necessary to rely on an alternative approach. The Service Provider (part of the OE) does not apply authenticated encryption to the SW update package, instead signs it with its private key, to ensure authenticity and integrity, and then sends the package to the PO. In turn, the PO distributes the package to all the devices. In each device, the TCB performs signature verification on the package, and the update process with state restoration follows as in the 1-to-1 scenario.
At this point, the TCB executes a second round of attestation on the newly initiated instances of App2; then, the attestation reports, produced as output, are subject to authenticated encryption by means of LR-BC-2 and transmitted to the Service Provider, for a final validation of the overall update process of each device (see Section 5.2).

The 1-to-many scenario shares the same security properties as the 1-to-1 scenario: SW Update Authenticity and Integrity, and App State Confidentiality and Integrity.

[2]Note that the use of a distinct key, for each of the involved devices, would correspond to multiple 1-to-1 scenarios, rather than to one 1-to-many scenario.

5.4.2.2 Assurance Cases

In the following, we highlight the changes to the 1-to-1 scenario assurance cases, necessary to accommodate the presence of multiple devices, based on the observations presented in Section 5.4.2.

App State Confidentiality and Integrity The assurance case is the same as in the 1-to-1 scenario.

SW Update Authenticity and Integrity In this case, as aforementioned, while the LR-BC-2 key is also used for the verification of the authenticity and integrity of the SW update package, it is utilised in different phases of the process. Thus, while the assurance case is similar to the corresponding one in the 1-to-1 scenario, it is formulated as follows in order to reflect this difference, considering that *the LR-BC-2 Key validates the correctness of the output of the update process via the attestation report, rather than the update itself*:

PO → REWIRE TCB; REWIRE TCB; TCB → OE. See Section 5.3.2.

OE → PO. The OE are assumed to overall behave in the expected way, in developing a SW update package and in sending it to PO. Note that, conversely to the 1-to-1 scenario, in the 1-to-many scenario this is not performed with authenticated encryption using the LR-BC-2 key, but the OE (specifically, the SW Service Provider) sign the SW update package with their own private key in order to ensure its authenticity and integrity.

PO → TCB. The PO forwards the SW update package to TCB.

TCB → EDS. The TCB verifies the SW update package with the public key of the OE, executes the App SW update, and sends an update trigger to the EDS.

. . . App2 initiation and state restoration take place . . .

TCB. Upon notification from the EDSU that state restoration is complete, the TCB performs attestation on App2, producing an attestation report, and applies authenticated encryption with the LR-BC-2 key.

TCB → PO. The TCB sends the encrypted attestation report to the PO.

PO → OE. The PO forwards the report to the OE, which verify that the output of the update process is correct.

5.4.2.3 System Model

The AADL model for the 1-to-many scenario adapts that of the 1-to-1 scenario to represent the revised communication flow discussed in the previous section. In particular: (i) the SW update package is signed, rather than encrypted via LR-BC-2, by the OE, before being sent to the PO; (ii) the TCB applies LR-BC-2 authenticated encryption to the attestation report performed on App2 after update and state restoration, and transmits the encrypted report to the PO.
Fig. 5.12 shows a portion of the textual model that captures the changes in the communication security mechanisms. No significant modifications appear in the corresponding AADL diagram.

```
c7: port oe.sw_update -> po.sw_update
    { System_Properties::Security_Mechanism => Signature; };
c8: port po.mspl_update -> rtcb.mspl_update
    { System_Properties::Security_Mechanism => Signature; };
c9: port rtcb.sw_update_cmd -> eds.sw_update_cmd;
c10: port eds.enc_state -> edsu.enc_state;
c11: port edsu.initiate_cmd -> aa2.initiate_cmd;
c12: port aa2.initiate_cmd_rcvd -> edsu.initiate_cmd_rcvd;
c13: port edsu.state -> aa2.state;
c14: port aa2.state_restored -> edsu.state_restored;
c15: port edsu.state_loaded -> rtcb.state_loaded;
c16: port rtcb.terminate_cmd -> aa1.terminate_cmd;
c17: port rtcb.succ_civ_report -> po.succ_civ_report
    { System_Properties::Security_Mechanism => AuthenticatedEncryption;
      System_Properties::LRBC2_AE_Evidence  => "link_to_LR-BC-2_evidence_data"; };
```

Figure 5.12: Smart Cities Use Case AADL Model: 1-to-many

5.4.2.4 Assurance Cases Encoding

The Resolute encoding of the App State Confidentiality and Integrity assurance case is the same as in the 1-to-1 scenario.
The encoding of the SW Update Authenticity and Integrity case reflects the impact of the different security mechanisms employed in the 1-to-many scenario, in particular (i) the use of digital signature instead of authenticated encryption on the SW update package, and (ii) the use of authenticated encryption instead of digital signature on the report obtained from attesting App2.

```
-- SW update authenticity and integrity
goal sc1m_SW_update_with_state_restoration_ensures_package_authenticity_and_integrity(s: system) <=
** "SW update with state restoration ensures package authenticity and integrity" **
    debug("App1 attestation request transmission from PO to TCB is correct") and
    attestation_request_transmission_from_PO_to_TCB_is_correct(s)
    and debug("App1 attestation execution by TCB is correct")
    and attestation_execution_by_TCB_is_correct(s)
    and debug("failed attestation report transmission from TCB to OE has authenticity and integrity")
    and failed_attestation_report_transmission_from_TCB_to_OE_has_authenticity_and_integrity(s)
    and debug("SW update package development by OE is correct")
    and SW_update_package_development_by_OE_is_correct(s)
    and debug("SW update package transmission from OE to PO has authenticity and integrity")
    and SW_update_package_transmission_from_OE_to_PO_has_authenticity_and_integrity(s)
    and debug("SW update package transmission from PO to TCB has authenticity and integrity")
    and SW_update_package_transmission_from_PO_to_TCB_has_authenticity_and_integrity(s)
    and debug("SW update by TCB is correct")
    and SW_update_by_TCB_is_correct(s)
    and debug("SW update notification by TCB to EDS is correct")
    and SW_update_notification_by_TCB_to_EDS_is_correct(s)
    and debug("App2 state restoration process is correct")
    and App2_state_restoration_process_is_correct(s)
    and debug("App2 attestation execution by TCB is correct")
    and attestation_execution_by_TCB_is_correct_and_ensures_state_confidentiality(s)
    and debug("attestation report transmission from TCB to PO has confidentiality and authenticity and integrity")
    and attestation_report_transmission_from_TCB_to_PO_has_confidentiality_and_authenticity_and_integrity(s)
```

Figure 5.13: Smart Cities Use Case Assurance Case Main Goal: 1-to-many

5.4.2.5 Analysis Results

Resolute completed the verification task without difficulties. Fig. 5.14 emphasizes the difference introduced by the SW Update Authenticity and Integrity assurance case in the reasoning, as outlined in the previous section.

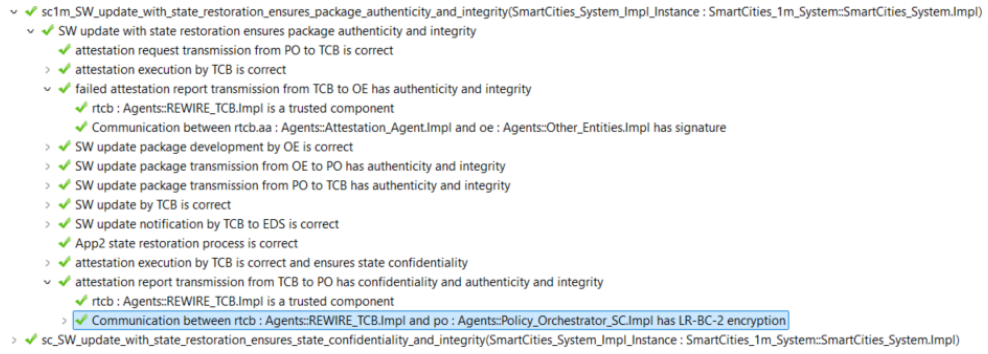


Figure 5.14: Smart Cities Use Case Assurance Case Analysis Results: 1-to-many

Chapter 6

Instantiation of REWIRE MSPL-based Security Policies

6.1 REWIRE Policy Orchestration and Device Lifecycle Management

One core aspect of the REWIRE architecture pertains to the *identification of boundaries of the protocols and schemes to be synthesized prior to their deployment to the respective devices, in order to define and narrow down the set of device properties which need to be attested dynamically during runtime*. This is achieved through the **Compositional Verification and Validation Component**, whose core target is the formal verification of the operational profiles of devices, mapped to the underlying security requirements of the devices and the operational domains where they should be deployed, towards the definition of the **trust boundary** of the system. This entails the identification of the device properties that can be formally verified during design-time, and those that need to be verified dynamically during runtime through the available **security controls**.

In the context of REWIRE, the latter is expressed through the definition of **security policies**, which are defined as mitigation actions that can safeguard the operation of the system during runtime. Specifically, these policies dictate the **actions that need to be performed** in order to ensure the correctness of the properties that have been defined during the design-time phase, as well as the **conditions for triggering the execution of the policy** and the **types of evidence that need to be collected** in order to verify the correctness of the identified properties through the defined **security controls**. In the context of REWIRE, two types of policies are defined: (i) **security policies**, which dictate or predicate the execution of a security enabler (e.g., attestation process) for verifying the correctness of the identified security properties, and (ii) **operational policies**, which dictate the tasks that need to be performed for mitigating identified threats or vulnerabilities following a failed attestation (e.g., **secure software update** or **live device migration**).

One core component of REWIRE in this regard is the **Policy Orchestrator**, whose purpose is to *act as a bridge between the outside world and the device*. Specifically, it is responsible for receiving the security and operational policies given as output of the design-time phase of the REWIRE framework and providing them to the **Facility Layer** of the device, so that they can be translated from the specified **policy expression language** into **execution logic** for performing the specified actions, considering also their **periodicity** and **triggering conditions**. In case the specified action is an attestation enabler, the specified actions are handled by the **Attestation Agent**. In case of a Secure Software Update, which is typically triggered in response to an event (such as a detected security bug or feature extension), the Policy Orchestrator verifies the update policy on the old enclave and validates that an update is permitted, and also receives an updated host and enclave file. Note that each update action is associated with a **Configuration Integrity Verification (CIV)** action both before and after the update is performed, in order to verify the correctness of the device state. In case of a Secure Live Migration action, the **Migration**

Services of both the source and target device are responsible for the actions required for the migration of the enclave to the one device to the other. Note that all aforementioned processes are described in further detail in D2.2.

As aforementioned, the policies provided as output of the Design-time Phase are formulated using a policy expression language. As described in detail in D2.2, the **Medium Security Policy Language (MSPL)** was selected in the context of REWIRE, as it exhibits the required properties for complementing the operation of both the Compositional Verification and Validation component, and the Policy Orchestrator. Specifically, it offers both the required **level of granularity** for expressing the types of attributes to be attested during runtime, as well as the **periodicity of the tasks** defined by the policies (e.g., attestation, software update, or live migration). MSPL is also appropriate for various types of application domains, as it is able to express the required **abstractions** in the formulation of policies in a **device-independent format**.

6.2 Security Policy Operational Codes

In this section, we provide concrete examples of policies expressed in the MSPL language, tailored to the user stories and functionalities of REWIRE targeted towards each use case demonstrator. Recall that details on the MSPL vocabulary, as well as policy modelling with MSPL considering all pertinent fields to be included in the policy structure in the context of REWIRE, have been described in detail in D2.2. Thus, here we build upon these descriptions in order to describe the instantiation of the required policies corresponding to each use case demonstrator.

6.2.1 Use Case 1: Smart Cities for Empowering Public Safety

As detailed in D6.1, one core purpose of this use case is to ensure citizen safety using **vehicular and pedestrian traffic systems** in urban sensor networks, with a high volume and density of associated IoT devices. At a high level, the architecture of this use case consists of a large number of end-devices, each one of which is responsible for **traffic control in a single block**, and acts as a controller of the corresponding radio cell of an IoT network. In addition, it implements the traffic control logic and powers various associated electromechanical components, sensors, and actuators. In the evaluation activities carried out in D6.1, this device was instantiated with the **VisionFive2** RISC-V-based Single Board Computer (SBC). Then, these devices provide traffic-related data to a **Network Gateway (NGW)**. In turn, the NGW forwards the data to the backend **Application Servers**, which are responsible for applications such as the **Smart City management platform**, the **Software distribution platform**, and the **City network onboarding platform**.

As dictated by user story **ODINS.US.1**, in this use case, there is a need for the remote deployment of update packages over secure and authenticated communication channels, so that technicians do not need to be physically present in order to perform the update. Considering the large-scale nature of urban networks, the core functionality of REWIRE is the **One-to-Many Software Update** modality, which is able to deploy the software patch to multiple control devices (VisionFive2) simultaneously. An example of such an MSPL-expressed policy is provided in Listing 6.1.

Listing 6.1: Example of a One-to-Many Software Update policy expressed in MSPL

```
<ITResource xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <configuration xsi:type="RuleSetConfiguration">
        <capability xsi:type="TaskCapability">
            <Name>Software Update Service</Name>
        </capability>
        <DeviceCondition>
```

```
            <isCNF>false</isCNF>
            <DeviceIDList>
                <DeviceID>R</DeviceID>
                <DeviceID>K</DeviceID>
                <DeviceID>L</DeviceID>
                <DeviceID>M</DeviceID>
                <DeviceID>N</DeviceID>
            <VerifierIDList>
        </DeviceCondition>

        <AttestationTaskList>
            <AttestationTask>
                <AttestationTaskID>0</AttestationTaskID>
                <AttestationUniqueidentifier>XX</AttestationUniqueIdentifier>
                <AttestationTaskName>CIV</AttestationTaskName>
                <ResToBeAttested>Binary unpacking process</ResToBeAttested>
            </AttestationTask>
            <AttestationTask>
                <AttestationTaskID>2</AttestationTaskID>
                <AttestationUniqueidentifier>KK</AttestationUniqueIdentifier>
                <AttestationTaskName>CIV</AttestationTaskName>
                <ResToBeAttested>Device SW Stack</ResToBeAttested>
            </AttestationTask>
            <AttestationTask>
                <AttestationTaskID>3</AttestationTaskID>
                <AttestationUniqueidentifier>YY</AttestationUniqueIdentifier>
                <AttestationTaskName>CFA</AttestationTaskName>
                <ResToBeAttested>Data Chunk Encryption</ResToBeAttested>
            </AttestationTask>
        </AttestationTaskList>

        <ConfigurationRule>
            <AttestationAction>
                <TaskActionType>EXECUTE</TaskActionType>
                <TaskActionParameters>
                    <TaskParameters>
                        <ProcessID>XX</ProcessID>
                        <ExecutionTime>1</ExecutionTime>
                        <VerifierID>R</VerifierID>
                    </TaskParameters>
                </TaskActionParameters>
            </AttestationAction>
        </ConfigurationRule>

        <ConfigurationRule>
            <TaskAction>
                <TaskActionType>EXECUTE</TaskActionType>
                <TaskActionParameters>
                    <ActionParameters>
                        <ProcessID>SW Update Process</ProcessID>
                        <TaskType></TaskType>
```

```
                    <ExecutionTime>2</ExecutionTime>
                </ActionParameters>
            </TaskActionParameters>
        </TaskAction>
    </ConfigurationRule>

    <ConfigurationRule>
        <AttestationAction>
            <TaskActionType>EXECUTE</TaskActionType>
            <TaskActionParameters>
                <TaskParameters>
                    <ProcessID>KK</ProcessID>
                    <ExecutionTime>3</ExecutionTime>
                    <VerifierID>R</VerifierID>
                </TaskParameters>
            </TaskActionParameters>
        </AttestationAction>
    </ConfigurationRule>

    <ConfigurationRule>
        <AttestationAction>
            <TaskActionType>EXECUTE</TaskActionType>
            <TaskActionParameters>
                <TaskParameters>
                    <ProcessID>YY</ProcessID>
                    <ExecutionTime>4</ExecutionTime>
                    <VerifierID>R</VerifierID>
                </TaskParameters>
            </TaskActionParameters>
        </AttestationAction>
    </ConfigurationRule>

  </configuration>
</ITResource>
```

As demonstrated in the Listing, a **DeviceIDList** is defined, containing five devices, with **DeviceID** R, K, L, M, and N, each one of which corresponds to a different End-device where an update needs to be securely deployed. Then, an **AttestationTaskList** is defined, containing the different types of actions to be performed as dictated by the policy, each one corresponding to a different **AttestationTaskID** with a unique identifier and name **AttestationUniqueIdentifier** and **AttestationTaskName**, respectively. In this case, we define three different types of AttestationTaskID: (i) XX, the **Configuration Identity Verification (CIV)** attestation action that verifies the correctness of the old enclave and update patch, (ii) KK, the CIV attestation of the Device SW stack, and (ii) YY, the **Control Flow Attestation (CFA)** process for the encryption action of the attestation report, after the completion of the update process.

After the AttestationTaskList has been defined, the next step is the definition of the actual actions dictated to be performed by the policy, containing information about their ordering, periodicity, and resource where they should be executed. Specifically, each action is defined under a **ConfigurationRule**, the order of execution of the action is defined under the **ExecutionTime** field, and the **ProcessID** and **VerifierID** fields define the type of action and resource, respectively. In this example, as shown in the Listing, the first action is the CIV attestation ”XX”, which is responsible for verifying the correctness of the device state prior to the deployment of the update. The second action is the ”SW Update Process”, which refers to the

deployment of the update patch itself. The third action is the ”KK” attestation, which is the CIV attestation on the Device SW stack after the update has been deployed. The fourth and final action is the ”YY” CFA attestation, which verifies the correctness of the **Data Chunk Encryption**, referring to the encryption of the **attestation report** generated after the execution of the CIV process.

6.2.2 Use Case 2: Adaptive In-Vehicle SW & FW Patch Management & Software Functions Migration

The core motivation behind the Adaptive In-Vehicle SW (Automotive) use case of REWIRE is the trend of the automotive domain followed by many OEMs towards the adoption of centralized **Electronic Control Unit (ECU)** network architectures. In this regard, REWIRE aims to address the various potential ECU Risks and vulnerabilities which may stem from this approach. To this end, as detailed in D6.1, the employed evaluation architecture consists of two **Zonal Control Units (ZCUs)**, the first one of which (ZCU1) functions as a **Powertrain controller**, and the second (ZCU2) acts as an additional **control unit**. In addition, an ECU is connected to ZCU1 for performing ADAS automotive functions. In the experimental architecture of REWIRE, the ZCUs were instantiated with **VisionFive2** boards based on the RISC-V architecture, while the ADAS ECU was instantiated with an NVIDIA DRIVE Orin.

Considering the above, as demonstrated by the defined user stories, REWIRE aims to provide the capability for mitigating the types of aforementioned vulnerabilities affecting automotive applications and ensuring the secure and trustworthy operation of in-vehicle devices. For instance, as documented in user story **KEN.US.1**, the **Secure SW Update** capability of REWIRE aims to ensure that, through the deployment of the appropriate software patch, the functionality of the ECU will be maintained, even after an indication of risk or fault. This can be instantiated through a **One-to-One SW Update**, where a patch is deployed to a single ECU or ZCU, and the corresponding policy is shown in Listing 6.2. This policy follows a similar structure as the One-to-many policy of Listing 6.1 which was extensively described in Section 6.2.1. The difference in this case is that there is a single DeviceID listed, corresponding to the ECU where the update is intended to be deployed.

Listing 6.2: Example of a One-to-One Software Update policy expressed in MSPL

```
<ITResource xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <configuration xsi:type="RuleSetConfiguration">
        <capability xsi:type="TaskCapability">
            <Name>Software Update Service</Name>
        </capability>
        <DeviceCondition>
            <isCNF>false</isCNF>
            <DeviceIDList>
                <DeviceID>R</DeviceID>
            <VerifierIDList>
        </DeviceCondition>

        <AttestationTaskList>
            <AttestationTask>
                <AttestationTaskID>0</AttestationTaskID>
                <AttestationUniqueidentifier>XX</AttestationUniqueIdentifier>
                <AttestationTaskName>CIV</AttestationTaskName>
                <ResToBeAttested>Binary unpacking process</ResToBeAttested>
            </AttestationTask>
            <AttestationTask>
                <AttestationTaskID>2</AttestationTaskID>
                <AttestationUniqueidentifier>KK</AttestationUniqueIdentifier>
```

```
            <AttestationTaskName>CIV</AttestationTaskName>
            <ResToBeAttested>Device SW Stack</ResToBeAttested>
        </AttestationTask>
        <AttestationTask>
            <AttestationTaskID>3</AttestationTaskID>
            <AttestationUniqueidentifier>YY</AttestationUniqueIdentifier>
            <AttestationTaskName>CFA</AttestationTaskName>
            <ResToBeAttested>Data Chunk Encryption</ResToBeAttested>
        </AttestationTask>
    </AttestationTaskList>

    <ConfigurationRule>
        <AttestationAction>
            <TaskActionType>EXECUTE</TaskActionType>
            <TaskActionParameters>
                <TaskParameters>
                    <ProcessID>XX</ProcessID>
                    <ExecutionTime>1</ExecutionTime>
                    <VerifierID>R</VerifierID>
                </TaskParameters>
            </TaskActionParameters>
        </AttestationAction>
    </ConfigurationRule>

    <ConfigurationRule>
        <TaskAction>
            <TaskActionType>EXECUTE</TaskActionType>
            <TaskActionParameters>
                <ActionParameters>
                    <ProcessID>SW Update Process</ProcessID>
                    <TaskType></TaskType>
                    <ExecutionTime>2</ExecutionTime>
                </ActionParameters>
            </TaskActionParameters>
        </TaskAction>
    </ConfigurationRule>

    <ConfigurationRule>
        <AttestationAction>
            <TaskActionType>EXECUTE</TaskActionType>
            <TaskActionParameters>
                <TaskParameters>
                    <ProcessID>KK</ProcessID>
                    <ExecutionTime>3</ExecutionTime>
                    <VerifierID>R</VerifierID>
                </TaskParameters>
            </TaskActionParameters>
        </AttestationAction>
    </ConfigurationRule>

    <ConfigurationRule>
```

```
            <AttestationAction>
                <TaskActionType>EXECUTE</TaskActionType>
                <TaskActionParameters>
                    <TaskParameters>
                        <ProcessID>YY</ProcessID>
                        <ExecutionTime>4</ExecutionTime>
                        <VerifierID>R</VerifierID>
                    </TaskParameters>
                </TaskActionParameters>
            </AttestationAction>
        </ConfigurationRule>

    </configuration>
</ITResource>
```

As documented in user story **KEN.US.3**, REWIRE aims to ensure that partial or full vehicle functionality is maintained, even in case of a security breach on a given ECU. This is achieved through the **Secure Live Migration** functionality of REWIRE, which enables the migration of an entire enclave from a host device which may be considered untrustworthy or compromise to a different device, which is in a trustworthy state, so that the enclave can continue its operation normally. An example of an MSPL expression of such a policy is provided in Listing 6.3. In this example, two types of **DeviceID**s are defined, corresponding to the **Source Migration Node (SMN)**, i.e., the compromised host device where the enclave is originally located, and the **Target Migration Node (TMN)**, which corresponds to the device where the enclave will be migrated. Also, note that both the SMN and the TMN possess a **Migration Service**, which is responsible for executing the logic of the migration operation.

In the Secure Live Migration functionality of REWIRE, as documented in detail in D2.2, the process is initiated by the TMN, which sends a migration request to the SMN, which in turn verifies the authenticity of the migration challenge and verifies the existence and operating status of the application to be migrated, through a secure enclave registry service. Afterwards, the process continues with the establishment of a secure communication channel between the SMN and the TMN for executing the remainder of the action workflow pertaining to the migration of the enclave from the one device to the other. Thus, considering the above, we define two processes: The first, with **ProcessID** ”TM”, refers to the initiation of the migration by the TMN, and the second, with **ProcessID** ”SM”, refers to the initiation of the process at the side of the SMN as a response to the migration request.

Listing 6.3: Example of a Live Migration policy expressed in MSPL

```
<ITResource xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <configuration xsi:type="RuleSetConfiguration">
        <capability xsi:type="TaskCapability">
            <Name>Software Update Service</Name>
        </capability>
        <DeviceCondition>
            <isCNF>false</isCNF>
            <DeviceIDList>
                <DeviceID>SMN</DeviceID>
                <DeviceID>TMN</DeviceID>
            <VerifierIDList>
        </DeviceCondition>

        <AttestationTaskList>
            <AttestationTask>
                <AttestationTaskID>0</AttestationTaskID>
```

```
                <AttestationUniqueidentifier>SM</AttestationUniqueIdentifier>
                <AttestationTaskName>Live Migration Source</AttestationTaskName>
                <ResToBeAttested>Binary unpacking process</ResToBeAttested>
            </AttestationTask>
            <AttestationTask>
                <AttestationTaskID>1</AttestationTaskID>
                <AttestationUniqueidentifier>TM</AttestationUniqueIdentifier>
                <AttestationTaskName>Live Migration Target</AttestationTaskName>
                <ResToBeAttested>Device SW Stack</ResToBeAttested>
            </AttestationTask>
        </AttestationTaskList>

        <ConfigurationRule>
            <AttestationAction>
                <TaskActionType>EXECUTE</TaskActionType>
                <TaskActionParameters>
                    <TaskParameters>
                        <ProcessID>TM</ProcessID>
                        <ExecutionTime>1</ExecutionTime>
                        <VerifierID>TMN</VerifierID>
                    </TaskParameters>
                </TaskActionParameters>
            </AttestationAction>
        </ConfigurationRule>

        <ConfigurationRule>
            <AttestationAction>
                <TaskActionType>EXECUTE</TaskActionType>
                <TaskActionParameters>
                    <TaskParameters>
                        <ProcessID>SM</ProcessID>
                        <ExecutionTime>2</ExecutionTime>
                        <VerifierID>SMN</VerifierID>
                    </TaskParameters>
                </TaskActionParameters>
            </AttestationAction>
        </ConfigurationRule>

    </configuration>
</ITResource>
```

6.2.3 Use Case 3: Smart Satellites Secure SW Updates for Spacecraft Applications & Services

The Smart Satellites use case of REWIRE entails the provision of runtime assurance capabilities to a satellite communication infrastructure, referred to as **SatNOGS**, consisting of multiple ground stations situated in different locations worldwide, connected through a cloud infrastructure. These base stations communicate with satellites, referred to as **FlatSats**, which are designed to be compact due to limited available space and the need to minimize weight, thus allowing easy access to all subsystems and facilitating quick and efficient troubleshooting. In the context of the evaluation activities detailed

in D6.1, REWIRE leveraged the FlatSat implementation provided by the Libre Space Foundation utilizing the **SatNOGS-COMMS** telecommunication subsystem and a **VisionFive2** RISC-V board. Thus, the SatNOGS-COMMS system is responsible for managing communication with the ground stations, while the VisionFive2 board handles satellite data processing activities.

The core challenge in this case is the *limited communication window and low data rate between the ground station and the satellite*, thus necessitating the segmentation of the transmitted data (e.g., a firmware patch distributed as part of a secure software update) by the mission control centre. This, in tandem with the need to transmit data over communication channels that may pose challenges regarding the integrity of the transmitted data, highlights the need for the implementation of **attestation processes** (CFA and CIV), not only for verifying the correctness of the device state and software execution processes, but also to support the execution of the Onboarding and Secure Software Update processes.

Considering the above, as detailed examples of MSPL policies for the update and live migration processes were provided in Sections 6.2.1 and 6.2.2, here we focus on the aforementioned attestation processes, which are used in order to support the Secure SW update process in the context of user story **LSF.US.1**, as well as the secure onboarding process in user story **LSF.US.2**. Note that, while attestation processes have been included as part of the software update policy examples provided in the previous sections, here we provide attestation policy examples demonstrating the capability to include additional, more detailed information can be included, related to the vulnerability to be addressed by the attestation process.

Specifically, an example of a CFA attestation policy is provided in Listing 6.4, in order to verify the correctness of the execution of a software process compared against a listed of trusted reference values corresponding to correct control flows, representing the correct system behaviour. Here, the **TaskParameters** field can contain various metadata attributes, such as **category** and **GDPR**. Furthermore, the **TaskParameters** field contains a **RiskLevel** field, in this example marked as H (High), due to being associated with the *CAPEC-242* vulnerability in the **VulnerabilityID** field, which can be exploited by an adversary to execute malicious code on the edge device. In addition, the attested resource is specified in the **ResToBeAttested** as the pointer to a valid, pre-computed secure control flow graph.

The policy provided in this example also includes a set of constraints in the **configurationCondition** field, specifying the desired level of risk after the attestation process is complete as L (Low) in the **DesiredLevel** field. The **TaskDependencyList** field contains a list of prerequisite process identifiers for tasks that need to be executed before the CFA attestation through their **PolicyID**s. Another constraint that can be included is **isInstalled**, which shows which devices the process must be installed on for the CFA process to be executed.

Listing 6.4: Example of a CFA attestation policy expressed in MSPL

```
<ITResource>
    <configuration xsi:type="RuleSetConfiguration">
        <capability xsi:type="TaskCapability">
            <Name>TaskCapability</Name>
        </capability>
        <configurationRule>
            <configurationRuleAction xsi:type="TaskAction">
                <TaskActionType>EXECUTE</TaskActionType>
                <TaskActionParameters>
                <TaskParameters xsi:type="TaskParameters" />
                    <ProcessID>0</ProcessID>
                    <category>Infrastructure Service/Edge Service</category>
                    <GDPR>false</GDPR>
                    <RiskAssessmentID>1714</RiskAssessmentID>
                    <RiskLevel>H</RiskLevel>
                    <ExecutionTime>15</ExecutionTime>
```

```
                <Threats xsi:type="TaskThreats">
                    <Threat>
                        <Name>DEFAULT-I</Name>
                        <Level>VH</Level>
                        <VulnerabilitiesList>
                            <Vulnerability>
                             <VulnerabilityID>CAPEC-242</VulnerabilityID>
                             <Level>VL</Level>
                             <Impact>VH</Impact>
                             <RiskLevel>H</RiskLevel>
                             <PrivacyFuncImpactScore>0
                                                 </PrivacyFuncalImpactScore>
                               <PrivacyImpactScopeScore>0
                                                  </PrivacyImpactScopeScore>
                               <PrivacyDataTypes>[dat1, dat2]
                                                  </PrivacyDataTypes>
                               <PrivacyScore>0</PrivacyScore>
                               <PrivacyRiskScore>0</PrivacyRiskScore>
                               <cvssScore>6</cvssScore>
                               <cvss>H</cvss>
                               <TaskID>0</TaskID>
                               <AttestationTaskParameter>
                                   SecureUpdate
                               </AttestationTaskParameter>
                               <ResToBeAttested>
                                   ValidControlFlowGraphsPointer
                               </ResToBeAttested>
                            </Vulnerability>
                        </VulnerabilitiesList>
                    </Threat>
                </Threats>
                <ControlFlowParameters xsi:type="ControlFlowParameters">
                    <ProcessID>software_process</ProcessID>
                    <ValidControlFlows>
                        <ControlFlowList>
                            <Block>main,software_process</Block>
                            <Block>printf,libc-2.27.so</Block>
                        </ControlFlowList>
                        <ControlFlowList>
                            <Block>main,software_process</Block>
                            <Block>write,libc-2.27.so</Block>
                        </ControlFlowList>
                    </ValidControlFlows>
                </ControlFlowParameters>
            </TaskActionParameters>
        </configurationRuleAction>
        <configurationCondition xsi:type="TaskCondition">
            <isInstalled>[0,1]</isInstalled>
             <DesiredLevel>L</DesiredLevel>
             <DeviceCondition>0/<DeviceCondition>
            <TaskDepndencyList>
```

```
                    <TaskDepndency>5</TaskDepndency>
                    <TaskDepndency>15</TaskDepndency>
                </TaskDepndencyList>
            </configurationCondition>
            <Name>"TaskRule"</Name>
        </configurationRule>
        <Name>"TaskConfiguration"</Name>
    </configuration>
</ITResource>
```

Conversely, the Listing 6.5 contains an example of a policy corresponding to a CIV process, considering the possibility that a CIV attestation enabler is used for the mitigation of the same vulnerability defined in the CFA example of Listing 6.4. While the MSPL expression of this policy is similar, there are some differences, corresponding to the particular needs of the CIV process. Specifically, the **ResToBeAttested** field specifies the attribute to be attested, as defined by the Compositional Verification and Validation component. In addition, the configuration parameters of the CIV attestation enabler specified in the **ConfigurationIntegrityParameters** field within the **TaskActionParameters** is a list of expected hash values for code pages in libraries and binaries loaded in the device memory against which the configuration state of the device will be evaluated.

Listing 6.5: Example of a CIV attestation policy expressed in MSPL

```
<ITResource>
    <configuration xsi:type="RuleSetConfiguration">
        <capability xsi:type="TaskCapability">
            <Name>TaskCapability</Name>
        </capability>
        <configurationRule>
            <configurationRuleAction xsi:type="TaskAction">
                <TaskActionType>EXECUTE</TaskActionType>
                <TaskActionParameters>
                <TaskParameters xsi:type="TaskParameters" />
                    <ProcessID>0</ProcessID>
                    <category>Infrastructure Service/Edge Service</category>
                    <GDPR>false</GDPR>
                    <RiskAssessmentID>1714</RiskAssessmentID>
                    <RiskLevel>H</RiskLevel>
                    <ExecutionTime>15</ExecutionTime>
                    <Threats xsi:type="TaskThreats">
                        <Threat>
                            <Name>DEFAULT-I</Name>
                            <Level>VH</Level>
                            <VulnerabilitiesList>
                                <Vulnerability>
                                 <VulnerabilityID>CAPEC-242</VulnerabilityID>
                                 <Level>VL</Level>
                                 <Impact>VH</Impact>
                                 <RiskLevel>H</RiskLevel>
                                 <PrivacyFuncImpactScore>0
                                                    </PrivacyFuncalImpactScore>
                                    <PrivacyImpactScopeScore>0
                                                     </PrivacyImpactScopeScore>
                                    <PrivacyDataTypes>[dat1, dat2]
```

```
                                                    </PrivacyDataTypes>
                                <PrivacyScore>0</PrivacyScore>
                                <PrivacyRiskScore>0</PrivacyRiskScore>
                                <cvssScore>6</cvssScore>
                                <cvss>H</cvss>
                                <TaskID>1</TaskID>
                                <ResToBeAttested>
                                    ConfigurationIntegrityPointer
                                </ResToBeAttested>
                            </Vulnerability>
                        </VulnerabilitiesList>
                    </Threat>
                </Threats>
                <ConfigurationIntegrityParameters
                    xsi:type="ConfigurationIntegrity">
                    <ValidHashes>
                        <Hash>libc-2.27.so,0,
                            9ac411bef0479d2ebd6ee2126243b8b4a5d7c3d4ed5f2899
                        </Hash>
                        <Hash>software_process,
                            0721718c63188fe6ed74462222b4af4177e518e3ac2
                        </Hash>
                    </ValidHashes>
                </ConfigurationIntegrityParameters>
            </TaskActionParameters>
        </configurationRuleAction>
        <configurationCondition xsi:type="TaskCondition">
            <isInstalled>[0,1]</isInstalled>
             <DesiredLevel>L</DesiredLevel>
             <DeviceCondition>0/<DeviceCondition>
            <TaskDepndencyList>
                <TaskDepndency>5</TaskDepndency>
                <TaskDepndency>15</TaskDepndency>
            </TaskDepndencyList>
        </configurationCondition>
        <Name>"TaskRule"</Name>
    </configurationRule>
    <Name>"TaskConfiguration"</Name>
  </configuration>
</ITResource>
```

6.3 REWIRE Policy Enforcement APIs

Chapter 7

REWIRE Continuous and Modular Risk Assessment

One of the main targets of REWIRE is *to enhance the secure lifecycle management and guide the calculation of the trustworthiness level of embedded systems featuring RISC-V architectures*, operating in the context of large-scale **Systems-of-Systems (SoS)**, and to ensure the correctness of all software and hardware artefacts belonging to such domain infrastructures. Towards this direction, as detailed in D2.2 and D3.2, the **Risk Assessment (RA)** component is a core aspect of the REWIRE framework, which is responsible for maintaining an up-to-date **risk graph** of the entire ecosystem, based on the threats and vulnerabilities identified for the devices participating in the target domain infrastructure. One core innovation of REWIRE in this regard is the calculation of the **Required Trust Level (RTL)** of the devices, meaning the baseline level of trust that needs to be achieved so that they can be considered trustworthy, based on the risks affecting each device and the available **security enablers as mitigation measures against these risks**.

As detailed in D2.2, the Risk Assessment component participates in both the **Design-time** and **Runtime** phases of the REWIRE framework. In the Design-time phase, after the extraction of the security and privacy requirements based on the device manufacturers, an initial risk quantification is performed based on the most prominent types of risks identified for the types of devices considered. During the Runtime phase, after the deployment of the device to the target operational domain, the Risk Assessment component aims to capture all types of asset topologies, namely (i) **SW-SW relationships** (e.g., a process receives input from a different process), (ii) **HW-HW relationships**, (e.g., an asset is connected to a different one), and (iii) **SW-HW relationships** (e.g., a process is executed on a device), and is responsible for maintaining an up-to-date view of the RTL of the device, which translates to the *types of evidence and security controls that can be deployed during runtime*. This is performed by receiving input from components and processes that are able to identify new risks, such as the **AI-based Misbehaviour Detection engine**.

Considering all the above, throughout the remainder of this Chapter we will provide a detailed documentation of the functionality of the Risk Assessment component, focusing on the updates compared to D3.2, and we will highlight how the Risk Assessment can be used in order to guide a **Trust Assessment Framework (TAF)** through the calculation of the RTL, so that the achievement of the baseline required level of trust is achieved. Note that, while the provision of such a TAF falls outside the scope of REWIRE, we however aim to provide all the required tools and functionalities for guiding the operation of the TAF, through the monitoring the trust level of a device via the available **Trust Sources** and the mitigation of risks using the available security enablers.

7.1 Overview and Updates on Second Release

In general, the operation of **Trust Assessment Frameworks (TAFs)** is based on the accurate characterization of all trust relationships between components comprising the target system or SoS under evaluation. This, at its core, relies on a **risk analysis** that needs to take place not only at design-time considering the device in isolation, but also throughout its operational lifecycle as part of the overall domain infrastructure, to obtain an up-to-date view of the risks affecting the system. The endmost goal of the TAF is the calculation of the **Actual Trust Level (ATL)** of the device and, if it exceeds its RTL, the device is considered to be in a trustworthy state. As aforementioned, the RA component is responsible for the calculation of the RTL, for enabling the trust characterization of the device and the system as a whole.

Thus, the importance of accurately characterizing the risk associated with the target system within the overall pipeline of the TAF is underscored by the structure of the RA lifecycle. For instance, consider the **Threat Analysis and Risk Assessment (TARA)** Framework, which is a risk assessment methodology appropriate for use in the automotive domain. In this methodology, the Security Administrator must first provide a detailed component diagram that models the system under evaluation, and involves the identification of all relevant assets and interdependencies, as well as the specification of all plausible attack paths leading to one or more forms of system damage. This process leads to the derivation of concrete risk scenarios associated with the automotive system.

After these risk scenarios are derived, the Security Administrator must then determine an appropriate risk treatment strategy, which entails the *identification of the most prominent risks that require mitigation through the enforcement of security controls*, eventually leading to the specification of the accepted (residual) risk level under which the device is permitted to operate. In this regard, RTL constraints serve to formalize the baseline level of trust that defines the minimum acceptable runtime conditions. Simultaneously, the ATL values computed at runtime offer an evidence-based estimation of the trustworthiness level of the device, thus enabling the Security Administrator to determine (within a margin of uncertainty) whether the system continues to operate within the bounds of the accepted risk level.

This highlights the need for the definition of a **REWIRE RA Engine** that not only provides the baseline functionalities for performing the necessary risk assessment tasks, but also enables the realization of the ATL and RTL calculations for providing runtime trust assurance to the target operational domain. Note that, as aforementioned, the calculation of the ATL falls outside the scope of REWIRE, but the REWIRE framework offers all necessary elements to guide the operation of a TAF. In the following, we provide the final iteration of the REWIRE RA component, building upon the initial version detailed in D3.2, and introducing significant improvements in terms of both functionalities and integration capabilities, with a particular focus on the enablers for the calculation of the RTL.

7.2 Functional Specifications

In Table 7.1, we summarize the list of functional specifications of the RA component presented in D3.2, and report the progress made in the second release of the REWIRE RA Engine. In general, we verify that the final release of the RA component performs all envisioned functionalities, thus fulfilling its role in the overall REWIRE framework, as outlined in the previous sections.

First, specification **FR_RA_6** refers to the ability of the REWIRE RA engine to carry out attack path assessment and identify cascading attacks across the asset topology. In this regard, one of the core features of the second version of the REWIRE RA engine is the realization of the **Attack Path Calculator** component, which as analysed in Section 7.4 is responsible for processing the available threat analysis knowledge and provide a list of possible attack paths that may allow an adversary to leverage one asset in order to compromise another. For instance, in the automotive use case, a malicious party may aim to exploit a vulnerability in one ECU and use it in order to attack the main in-vehicle computer. Thus,

the attack path calculation of REWIRE alleviates some of the threat analysis required in the context of a full-blown risk assessment process which may be required by a methodology such as TARA. The results of the Attack Path Calculator component may assist practitioners to pre-populate the list of attack paths with concrete attack scenarios and update the existing list with additional cases based on their knowledge and expertise.

In addition, as dictated by **FR_RA_7** and **FR_RA_8**, the REWIRE RA Engine should be able to consume various **Indicators of Compromise (IoC)** reported by the operational environment, for instance through the available REWIRE enablers. One such example is the **AI-based Misbehaviour Detection Engine (AIMDE)**, which receives the traces collected from a failed attestation action and is responsible for identifying the vulnerability or cause behind the failure. In such a case, the REWIRE RA Engine is able to update the risk graph and perform risk recalculation of the target operational domain. To this end, the final release provides two possible ways to enable dynamic risk recalculation: (i) Authenticated and authorized OEMs and Security Administrators are able to access the **Graphical User Interface (GUI)** of the REWIRE RA Engine and update the risk graph based on new information from available IoCs. (ii) **Integration with well-known repositories** that contain up-to-date information on threats and vulnerabilities, such as the **National Vulnerability Database (NVD)**, in order to enable the dynamic recalculation of the risk graph of the domain infrastructure. This is triggered both when new entries are published, and when existing vulnerabilities are re-evaluated in terms of criticality (e.g., when a vulnerability initially assessed as low-risk is later upgraded to higher severity due to newly discovered exploitation methods or real-world attacks). The interfaces implemented in the REWIRE RA Engine, including those that enable performing a revised risk analysis, are provided in Section 7.6.

Finally, a core update in the final version of the RA Engine of REWIRE pertains to the fulfilment of **FR_RA_10** regarding the harmonization of different risk assessment methodologies. Specifically, as shown in Section 7.7, considering two methodologies with concrete risk quantification equations to convert the threat analysis knowledge into a score per risk scenario, the REWIRE RA approach enables expressing the available risk scores with a common scale, thus increasing the robustness of the RTL equations and enhancing the interplay between the risk assessment pipeline and the relaxation (or strengthening) of RTL constraints.

ID	Title	I want to <Action>	So that <Action>	Implementation Status
FR_RA_1	Modeling assets	model various types of assets (i.e., software and hardware assets)	I capture the complexity and diversity of the Systems-of-Systems continuum	Done
FR_RA_2	Modelling asset relationships	model various types of assets interdependencies	I can model all relationships and data flows associated with the identified assets	Done
FR_RA_3	Visualise asset graph	visualise the entire asset cartography	I can execute the necessary threat analysis for the risk assessment methodologies	Done
FR_RA_4	Modelling system configuration and attributes	capture assurance claims, and properties that need to be attested to during the runtime phase.	parametrise and fine-tune the risk assessment calculations to focus on the critical properties that have not been reflected in the formal verification process at design phase.	Done
FR_RA_5	Up-to-date open intelligence	have access to the latest open intelligence information available related to threats and vulnerabilities	compute the risk level of the assets (and the topology in general) based on valid and fresh open intelligence information	Done

FR_RA_6	Identifying attack paths	identify the attack paths comprising vulnerabilities that allow the propagation of attacks across multiple assets	I can capture the cascading effects that enable an adversary to pivot from one asset to another through the exploitation of the associated vulnerabilities.	Done
FR_RA_7	Continuous and dynamic risk assessment	perform risk assessment periodically as well as asynchronously	I can have the latest risk values for the entire topology right after an incident takes place.	Done
FR_RA_8	Modelling security controls (Mitigation strategies)	associate assets with security controls that address - fully or partially - specific risks	I can re-perform the risk analysis and evaluate its effectiveness in the overall risk of the topology.	Done
FR_RA_9	Automation of risk assessment	provide an automated and seamless risk assessment analysis	I facilitate the assessment of the trustworthiness the integration of trust assessment methodologies	Done
FR_RA_10	Convergence of risk results	converge outputs from both qualitative and quantitative risk quantification models	I can provide a more comprehensive risk assessment framework for the asset graph of an Edge Service, tailored to the set of properties of interest.	Done

Table 7.1: Functional Specifications of REWIRE Risk Assessment

7.3 REWIRE Risk Assessment Automotive Use Case Instantiation

As detailed in D3.2, the REWIRE Risk Assessment component is built upon the OLISTIC Risk Assessment platform developed by UBITECH, which provides a wide variety of features and capabilities. One of these, which enables the fulfilment of the functional specification **FR_RA_10**, is that while it implements a generic CVSS-based methodology, it also supports the capability for the *integration of different risk assessment methodologies, depending of the particular needs of the domain under consideration*. For instance, consider the **Adaptive In-Vehicle SW & FW Patch Management & Software Functions Migration** use case which pertains to the automotive domain to which, as aforementioned, the TARA risk assessment methodology is particularly suited. Thus, throughout this Section, we will present the **conceptual analysis of the REWIRE RA Engine**, considering also the **integration of the TARA framework for better fulfilling the needs of the automotive domain** in the context of this use case.

Specifically, the final release of the REWIRE RA framework offers two key advancements: (i) the addition of new components, namely the **Attack Path Calculator** and the **Attack Feasibility Recommendation Engine**, which support various aspects of the TARA threat analysis process, and (ii) enhancements to existing components to **more accurately model the threat analysis information** within the TARA context. In the following, we provide a high-level description of the flow of actions of the REWIRE RA Engine, considering also the architectural details related to the integration of the TARA framework and the calculation of the RTL.

7.3.1 High-level Flow of Actions

Figure 7.1 depicts the positioning of the RA Engine within the overall REWIRE framework, for which detailed action workflows have been provided in D2.2 and D6.1. Conversely, the internal architecture of the RA Engine considering the latest updates provided in this deliverable are depicted in Figure 7.2. In this regard, the REWIRE RA Engine is capable of collecting all threat analysis details provided by the relevant

OEMs and Security Administrators, and deriving the risk graph of the target topology (or the in-vehicle topology with the assistance of the TARA framework, in case of the automotive use case). Consolidating both the above, the action workflow of the REWIRE RA Engine within the REWIRE architecture can be summarized as follows:

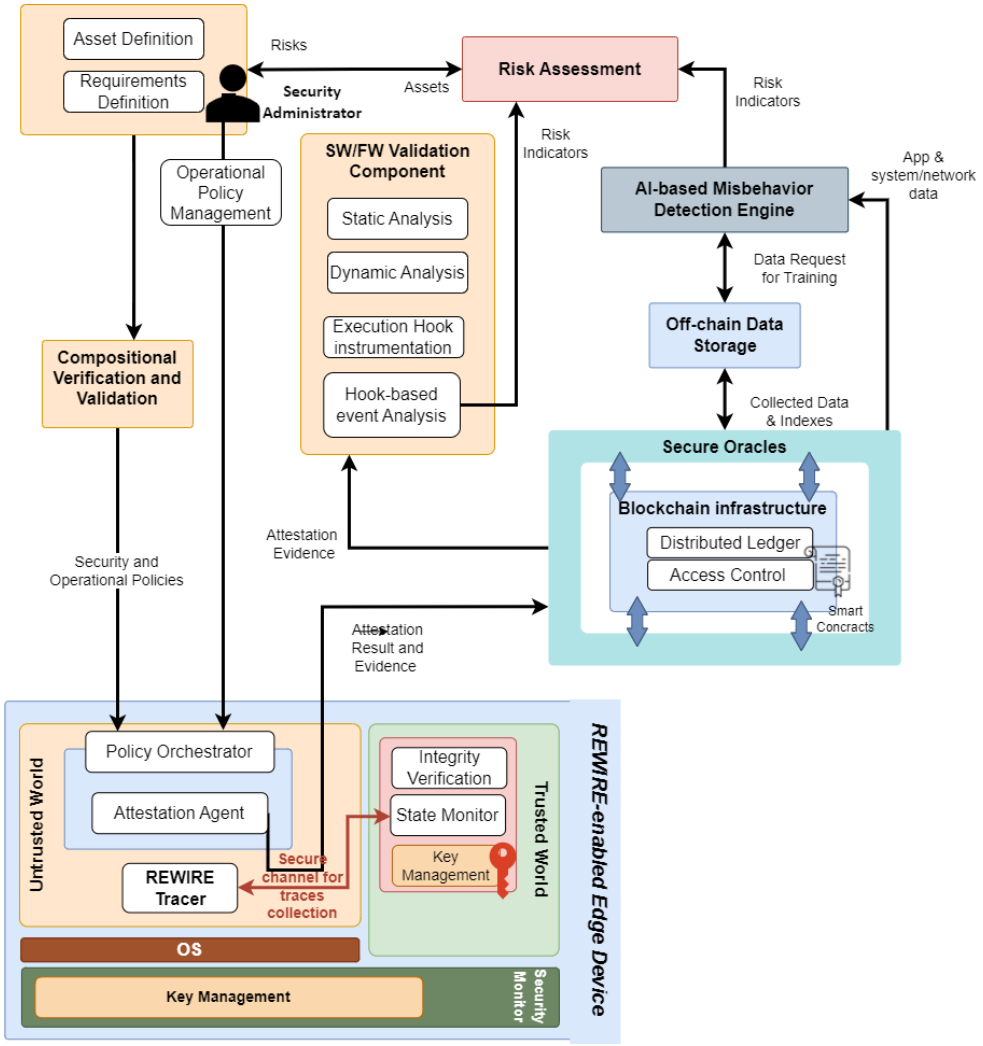


Figure 7.1: Positioning of Risk Assessment within REWIRE architecture

1. During the **Design-time phase**, the RA is populated with the assets, interdependencies, and the risk quantification model comprising the overall service graph chain in order to calculate the risk quantification graph. Note that, in the following, we will consider that a combination of the TARA methodology and the CVSS-based methodology of OLISTIC is employed in the context of the automotive use case.

2. As depicted in Figure 7.2, the risk analysis process is initiated with a new **Request For Risk Assessment (RFRA)**, which can be triggered either **manually** (e.g., through an action performed by the Security Administrator) or **automatically** (e.g., through a periodic, scheduled task).

3. The RFRA invokes the REWIRE RA Engine to aggregate all the threat analysis knowledge provided by the Security Administrator (e.g., asset topology, associated threats, attack paths and threat scenarios) and quantify the risk score for each risk scenario that is specified. Not that this task is not executed only once, but according to the TARA framework once all the risk scenarios have been identified and their risk is quantified, the security administrator decides on the most appropriate treatment strategy for each entry in the report.

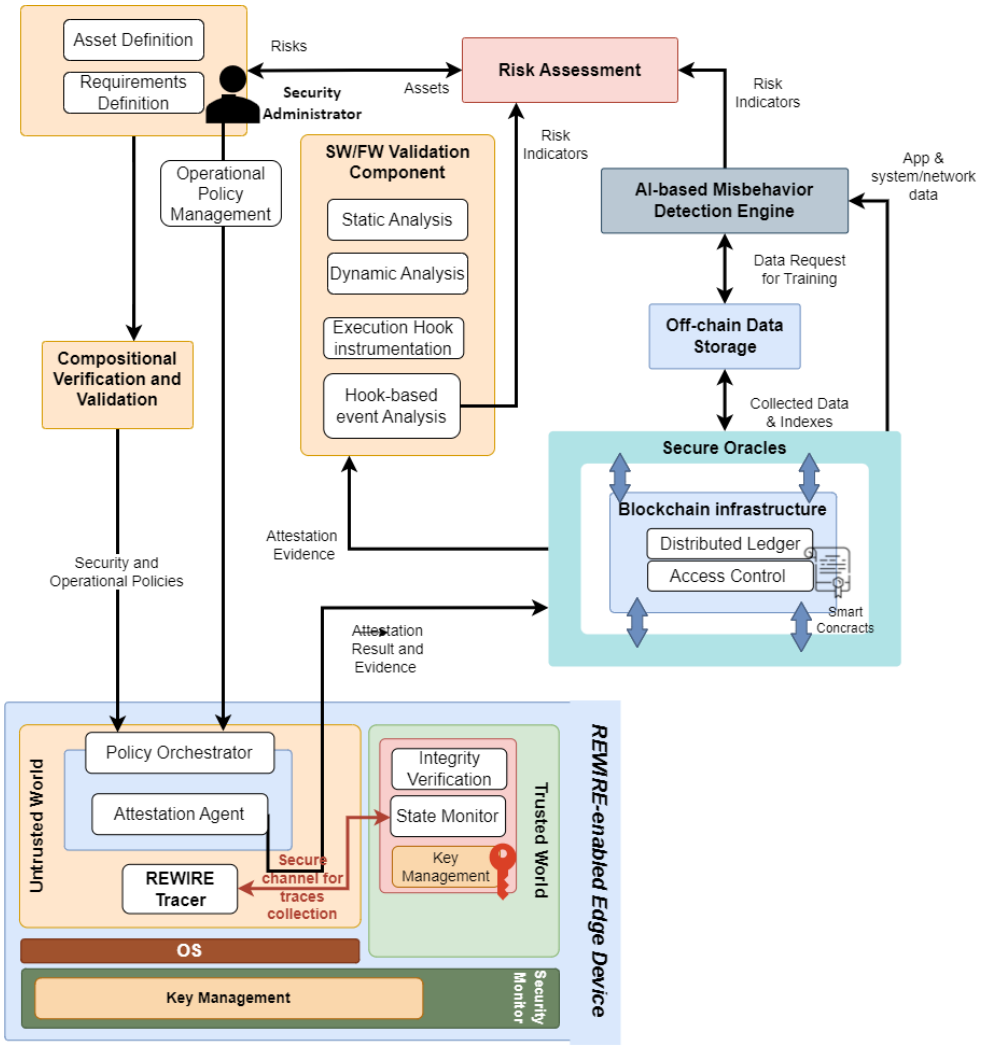


Figure 7.2: REWIRE Risk Assessment architecture

4. For the mitigation of the risk scenarios, the security administrator specifies the security control that addresses a particular risk and quantify the decrease in its attack feasibility. Thus, the collection of the results for all the iterations of the TARA process (including also the available security controls) provides a holistic view of the risk posture of the entire target topology.

5. Based on the filtering capabilities of the REWIRE RA Engine, it is possible to group risk scenarios per security property (e.g., Integrity, Confidentiality) and per asset or data item. This capability enables the realization of the RTL calculations for a specific context.

6. After the output of the RA Engine is provided to the **Compositional Verification and Validation Component** as described in detail in D2.2, the output of the Design-time phase is the set of **security and operational policies** dictating the actions that need to be taken and the security enablers that need to be executed in order to address and mitigate the risks and vulnerabilities, as identified by the RA Engine.

7. During the **Runtime Phase** and in order to establish trust, the Security Administrator, in a supervised manner, evaluates the risk quantification graph generated during the Design-time Phase and identifies the necessary controls to be deployed to minimise the impact of the previously calculated risk, considering also domain-specific requirements. The RA also calculates the **RTL**, based on the identified risks and the corresponding available security enablers.

8. After the execution of a security or operational policy, it is possible for an **Indication of Risk** to arise during runtime, for instance following a failed attestation process or an anomaly identified

by the **AI-Based Misbehaviour Detection Engine (AIMDE)**. Specifically, in this case, the raw traces collected as **trustworthiness evidence** are forwarded to the **Secure Oracle Layer**, which is notified of the existence of new traces and verifies that they originate from an authenticated and authorised device.

9. Data then could be stored on the **Blockchain Infrastructure** (in the **Besu Blockchain** in case of application-related data and the **Fabric Private Chaincode** in case of attestation-related data), while larger sized data is stored on the **Off-chain Data Storage**, while only pointers are securely stored on the ledger.

10. The **Security Context Broker** receives all necessary information (e.g., the output of the AIMDE based on this data) and shares it with the RA Engine to recalculate all risk quantification graphs, and the **SW/FW Validation Component** is able to identify the exact attack execution path.

11. The SW/FW Validation component retrieves the raw traces, and the Security Administrator converts them manually to fuzzy test cases so that the SW/FW Validation component is able to identify the exact attack execution path that could lead to an existing or zero-day vulnerability. The identified attack execution path is then forwarded to the RA Engine.

12. Then, based on the available information, the RA Engine performs the calculation of the RTL To this end, the RA Engine needs to know the most prominent risks, as well as the likelihood and the impact of the risk in order to identify the security/attestation controls that need to be deployed to minimise the impact. The process differentiates depending on whether the detected vulnerability is already existing or not:

    (a) In the case of an existing vulnerability, the Risk Assessment updates the risk graph, recalculates the risk quantification with the end goal to calculatre the RTL of the device through this up-to-date view risk quantification graph (in a supervised manner).

    (b) If a zero-day vulnerability has been detected, then it is mapped to an existing one in order to be able to proceed with the Risk Assessment process and calculate the RTL.

13. The calculated RTL will eventually be converted to MSPL security policies via the Compositional Verification and Validation component (as detailed in D2.2) that will be enforced on the device for the reduction of the impact of the identified risks.

7.3.2 Component Analysis

As aforementioned, Figure 7.2 depicts the main aspects of threat analysis (e.g., Asset Modelling and Visualisation, Vulnerability and Threat Modelling) as part of the REWIRE RA Engine. Throughout this Section, we provide an architectural blueprint of all internal elements of the REWIRE RA Engine, including a description per subcomponent and a report on the latest updates that have been implemented in the final version of the RA Engine.

7.3.2.1 OLISTIC Backend and Frontend

The OLISTIC backend, which provided the basis for the development of the REWIRE RA Engine, provides the necessary APIs to orchestrate all the operations pertaining to the execution of the risk assessment logic and works in tandem with the frontend to offer the necessary functionalities to a security analyst. The OLISTIC backend is also responsible for enforcing the access control rules, so that only the users with the appropriate permissions are able to use the available API. Finally, it provides the necessary interfaces for participating (i.e., publishing and subscribing) to Kafka topics for enabling the asynchronous consumption of security incidents reported by other REWIRE components. Conversely, the OLISTIC

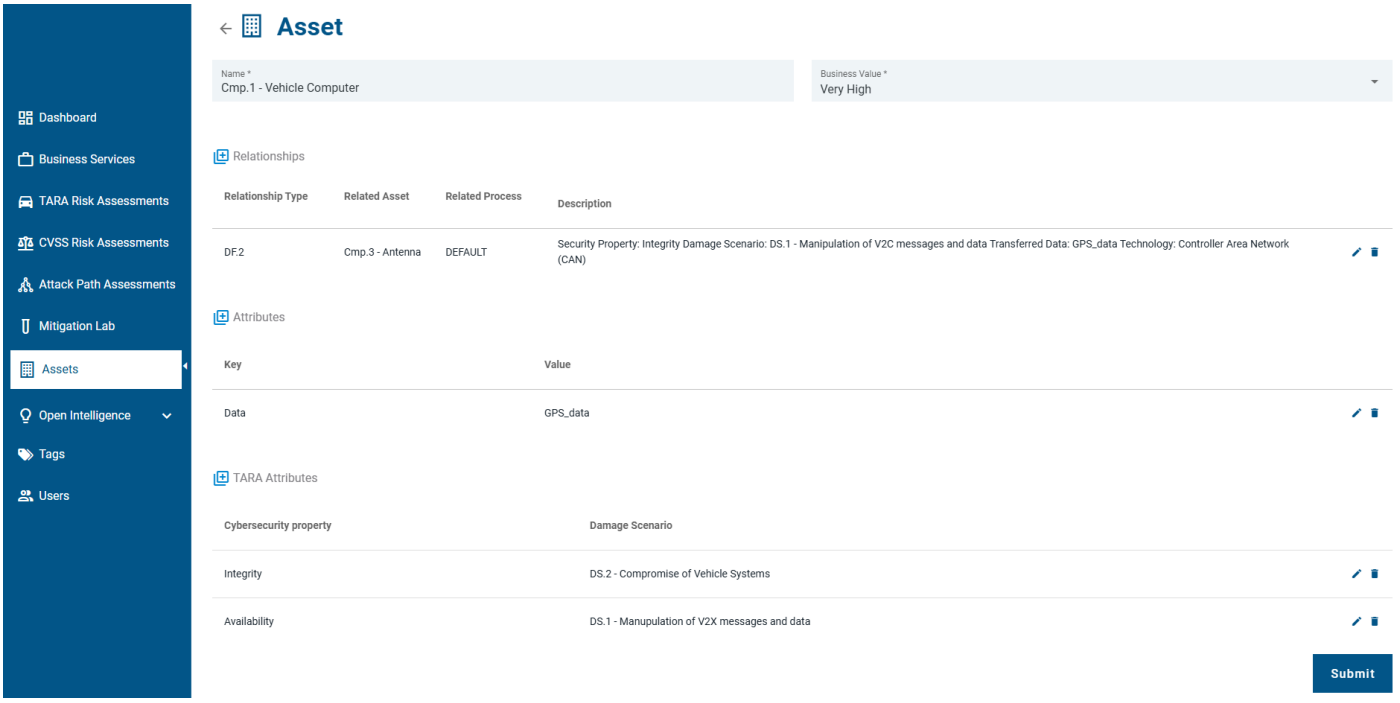


Figure 7.3: Asset and relationships data modelling

frontend provides an interactive dashboard with a GUI for visualizing the digital representation of the target domain infrastructure.

In the final version of the REWIRE RA Engine, some critical updates have been performed in both the backend and frontend logic. With regards to the backend, the corresponding API services have been updated to expose the endpoints providing all the risk scores for the RTL belief and disbelief equations. This includes not only the implementation of new endpoints, but also the support for filtering functionality to isolate the most pertinent risk scenarios for particular trust propositions (e.g., querying for all the risk scenarios that relevant for the main vehicular on-board unit and pose an integrity risk). The OLISTIC Frontend is also supplied with additional capabilities that allow the Security Administrator to provide more information pertaining to the threat analysis as part of the TARA methodology. In the following, we provide further details on the GUI enhancements per component of the REWIRE RA Engine.

7.3.2.2 Asset Modelling and Visualization Component

This component is responsible for modelling the assets landscape of the target domain infrastructure. Specifically, it enables the security administrator to model the component diagram that represents the device whose trustworthiness should be assessed. In this regard, assets need to be considered as abstract as possible in order to represent both tangible (e.g., hardware equipment like ECUs and in-vehicle sensors) and intangible entities (e.g., data items, data flows or software services). Thus, this enables the correspondence between threats, attack paths, and security controls in each asset of the target topology.

As part of the final release of the REWIRE RA Engine, this component has been updated to allow the definition of fine-grained information per asset and the interdependencies between them. Figure 7.3 depicts the level of detail in the information that can be provided by the security administrator. Thus, the final version of this component allows users to define custom asset relationships in order to accurately represent the data flows and interconnectivity between them. It also enables the specification of metadata, such as the technologies used to establish links between assets.

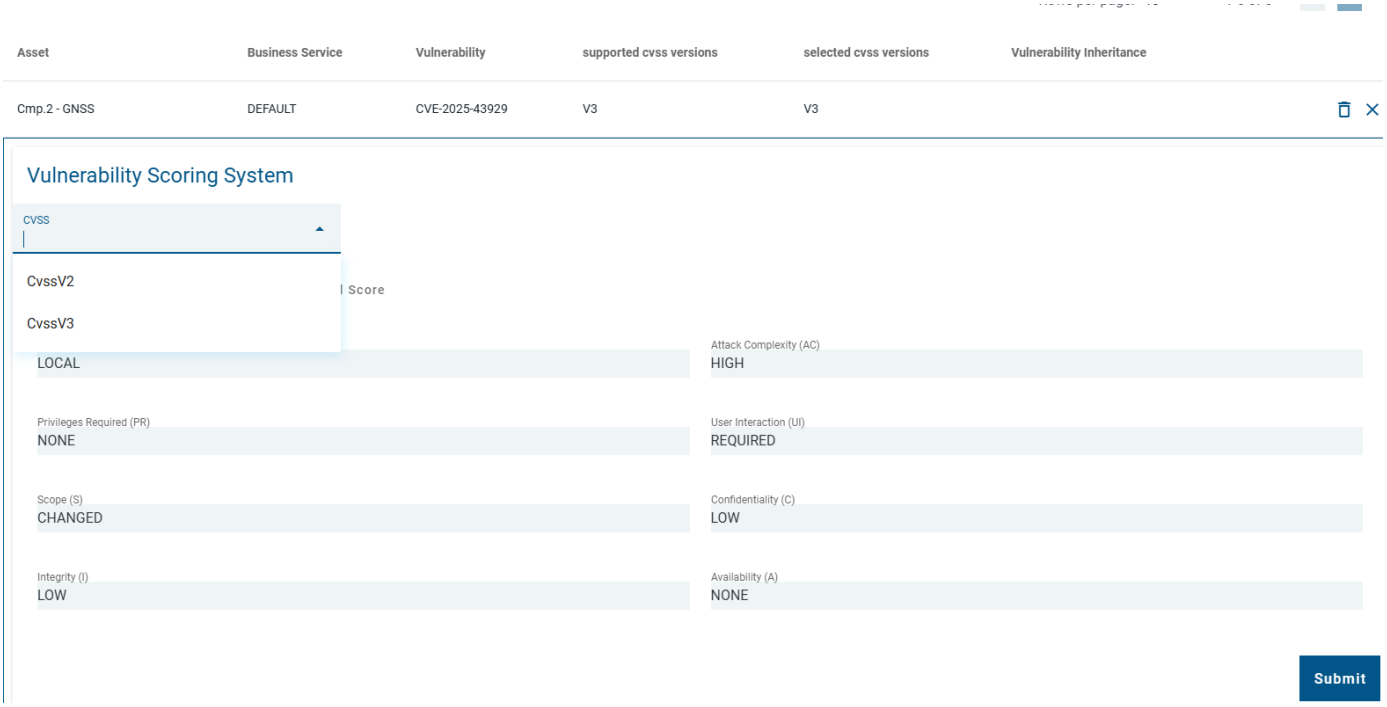


Figure 7.4: Extendability of vulnerability profiles

7.3.2.3 Vulnerability and Threat Modelling Component

The main functionality of this component pertains to the data modelling of vulnerabilities and threats which are either specified by the Security Administrators based on their knowledge and expertise, or retrieved by well established threat intelligence platforms, such as the NVD. Thus, the Vulnerability and Threat Modelling Component of the REWIRE RA Engine has been equipped with the logic that enables it to capture vulnerability characteristics and threat knowledge based on well established data models, namely the Common Vulnerability Scoring System and STRIDE, respectively.

In the final release of the REWIRE RA Engine, the vulnerability model has been redesign in order to enable the expression of multiple CVSS profiles for a single vulnerability entry, as shown in Figure 7.4). This will enable the expression of vulnerability characteristics originating from different versions of the already adopted versions of CVSS, namely v2 and v3.1. This will simultaneously facilitate the future support of the latest v4.0 specification.

7.3.2.4 Damage Scenario Management Component

The definition of damage scenarios is a core aspect that enables the calculation of risk associated with threat scenarios within the TARA methodology. Specifically, the damage scenario management component facilitates the fine-tuning of the impact of each identified damage scenario, thus enabling security administrators to refine their risk assessment analysis. According to the TARA specification, each damage scenario is characterized by its impact in four concrete dimensions, namely **Financial**, **Operational**, **Safety**, and **Privacy impact**. For each risk assessment process, Security Administrators are able to select the target dimension of the impact of each damage scenario to be considered in the risk quantification equation.

As part of the second release, considering that all interfaces are already specified and implemented already at the first release, enhancements have been performed in the various GUI views in order to accommodate the different pieces of information related to the damage scenarios. Figure 7.6 depicts the creation of a new damage scenario in the REWIRE RA Engine, while Figure 7.6 depicts the association of additional information with damage scenarios, such as the associated threat scenarios.

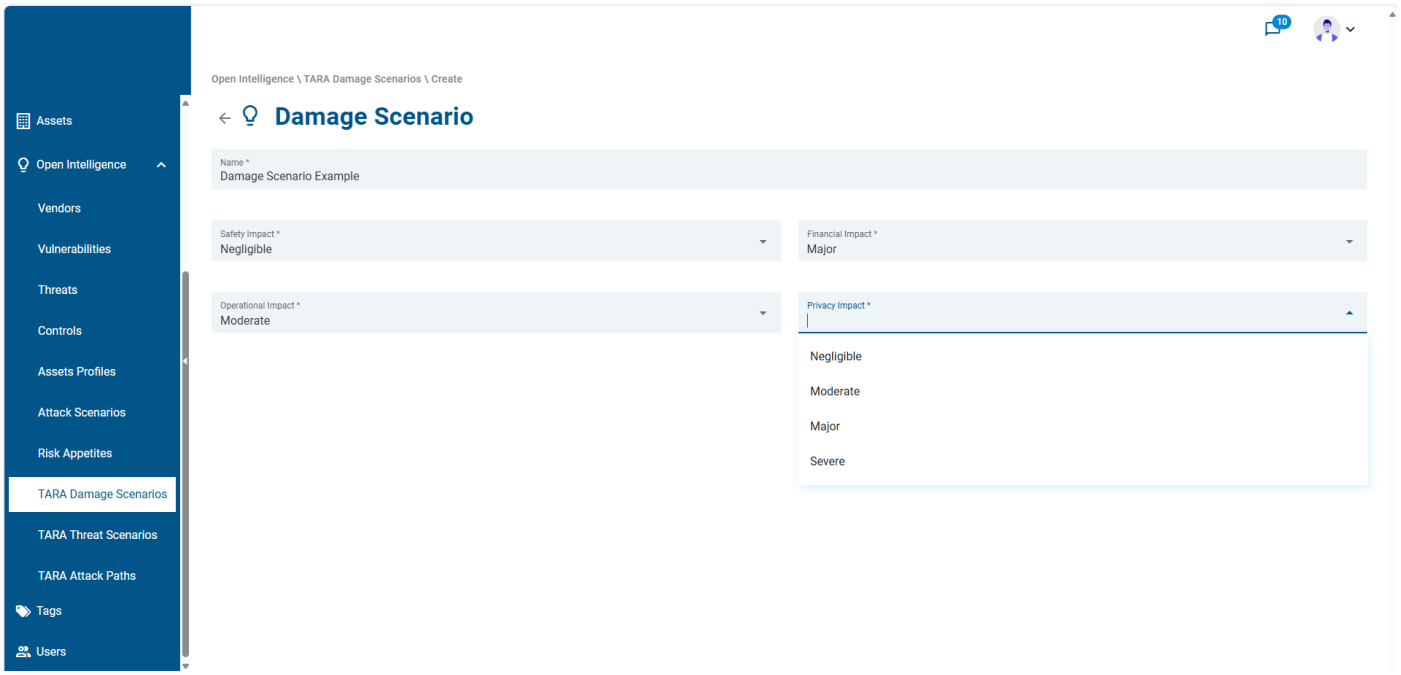


Figure 7.5: Creating a new damage scenario entry

7.3.2.5 Attack Path Calculator

One of the most computationally intensive tasks of the TARA threat analysis process is the **definition of all possible attack paths** that can lead to a threat scenario in the target domain infrastructure, which can eventually lead to a threat scenario that can result in a damage scenario (e.g., in the case of the automotive domain, for the entire vehicle or its passengers). The **Attack Path Calculator** component aims to alleviate the computational intensity of this task by leveraging the available knowledge for each asset, as well as the interdependencies between assets across the target topology. Thus, it is able to provide a set of possible attack paths from the already identified vulnerabilities associated with each asset.

The Attack Path Calculator component, implemented as an expert system using the **Drools** engine, analyses the characteristics of vulnerabilities associated with each asset. Specifically, by leveraging a predefined set of rules that demonstrate the conditions under which a vulnerability can be exploited so that an asset can be used as an entry point in order to compromise a different asset, it identifies potential cascading attacks that a malicious party could perform.

7.3.2.6 Attack Feasibility Recommendation Engine

After the possible attack paths have been determined, provided either by the Security Administrator or by the Attack Path Calculator component, it is necessary to enable the identification of their attack feasibility and eventually their overall risk. This is performed by the Attack Feasibility Recommendation Engine, which has been implemented as part of the Attack Path Calculator. As part of the TARA process, the calculation of the attack feasibility rating for an attack path is based on **seven key parameters** that can be specified by the security administrator, as shown in Figure 7.7. Regarding the overall risk level of the identified attack path, a new equation is defined based on the Individual Risk Level (IRL) equation, and calculates the Cumulative Risk Level (CRL) of an attack path as follows:

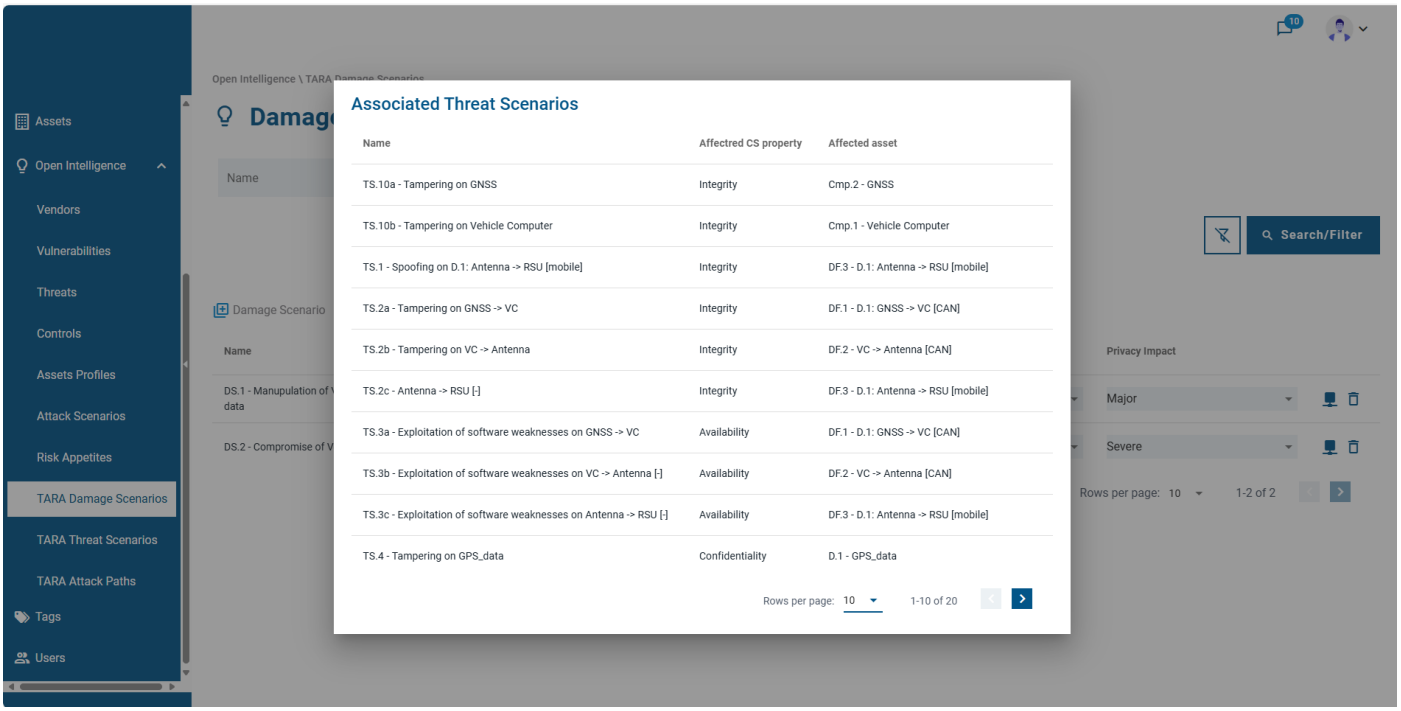


Figure 7.6: Displaying the threat scenarios associated with a particular damage scenario

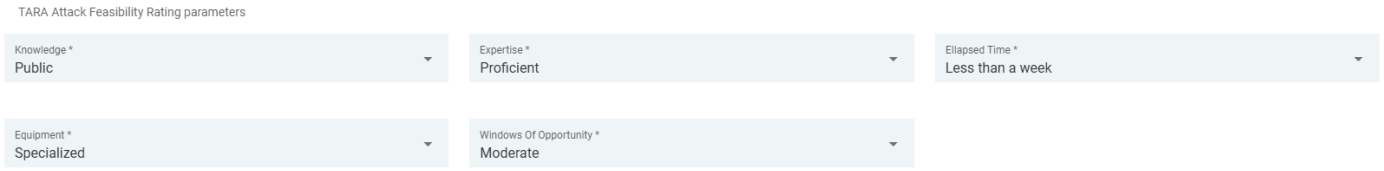


Figure 7.7: Attach feasibility rating parameters

$$CRL(Asset_1, Asset_i, \ldots, Asset_n) = IRL(Asset_1) \times IRL(Asset_i) \times \ldots \times IRL(Asset_n)$$

$$\text{where} \quad IRL(Asset_i) = IRL(Asset_1, Vulnerability_j, Threat_k)$$
$$= ThreatLevel \times VulnerabilityImpact$$

7.3.2.7 Risk Quantification Engine

The Risk Quantification Engine is responsible for consuming the threat analysis performed automatically or manually by the security administrators, and compute risk values for the monitored asset cartography. This component is designed in a modular manner, so that it enables the inclusion of diverse risk assessment methodologies. Note that, in the first version of the REWIRE RA Engine, both the CVSS-based and the TARA methodologies were implemented. In the final release of the REWIRE RA Engine, the goal of the risk quantification engine is to provide the necessary reporting capabilities so that it allows the derivation of the RTL constraints.

In the context of the final release, the goal of the instantiation of the REWIRE RA Engine in the automotive use case is to showcase how the available information from the TARA risk assessment processes can be leveraged to calculate the risk quantification of the target domain, and make them available for the execution of the RTL calculations.

Figure 7.8: Impact of applying a security control (e.g., software patch applied in the in-vehicle computer) in the number of risk scenarios considered.

7.3.2.8 Mitigation Strategy Component

The Mitigation Strategy Component supports the modelling of all types of security enablers provided by the REWIRE framework as mitigation measures against the identified vulnerabilities, and allows security administrators to signal the effectiveness of each security control by configuring the various parameters that contribute to the overall risk level for an asset or a trust relationship. A version of this component is available in the final release of the REWIRE RA Engine, which enables security administrators to evaluate the impact of enforcing a particular security control to the overall risk graph, before it is actually applied in the risk analysis. Note that, while this has been initially developed for the CVSS-based methodology, it also provides a simulation environment where users can observe the overall impact of a security control in the IRL/CRL values and the relevant risk scenarios, as shown in Figure 7.8.

7.3.3 REWIRE Risk Assessment Framework

Based on the final release of the REWIRE RA Engine, here we provide details on the action workflows realised within the REWIRE RA architecture. Specifically, in the following, *we position the sequence of actions considering how the resulting outcomes can enable the calculation of the RTL constraints*. In general, the sequence diagrams provided in this Section demonstrate how the security administrator is able to perform the CVSS-based methodology and leverage the inferred knowledge in order to enhance the threat analysis performed as part of the TARA risk framework. Eventually, as part of the risk treatment decision, the security administrator has the capability to enforce security controls for the unacceptable risk scenarios that need to be mitigated. A mitigation strategy consisting of one or more security controls is referred to as a **control scenario**, the enforcement of which may lead to a revised risk quantification process for minimising the identified risk to an acceptable level.

Figure 7.9 demonstrates the action workflow of the revised CVSS-based methodology, including also the functionalities integrated in the final version of the REWIRE RA Engine, namely the **Attack Path Calculator** component and the **derivation of the CRL risk score** for the identified attack paths. Specifically, the sequence of actions can be summarised as follows:

1. After the Security Administrator has provided the necessary threat analysis (asset topology and

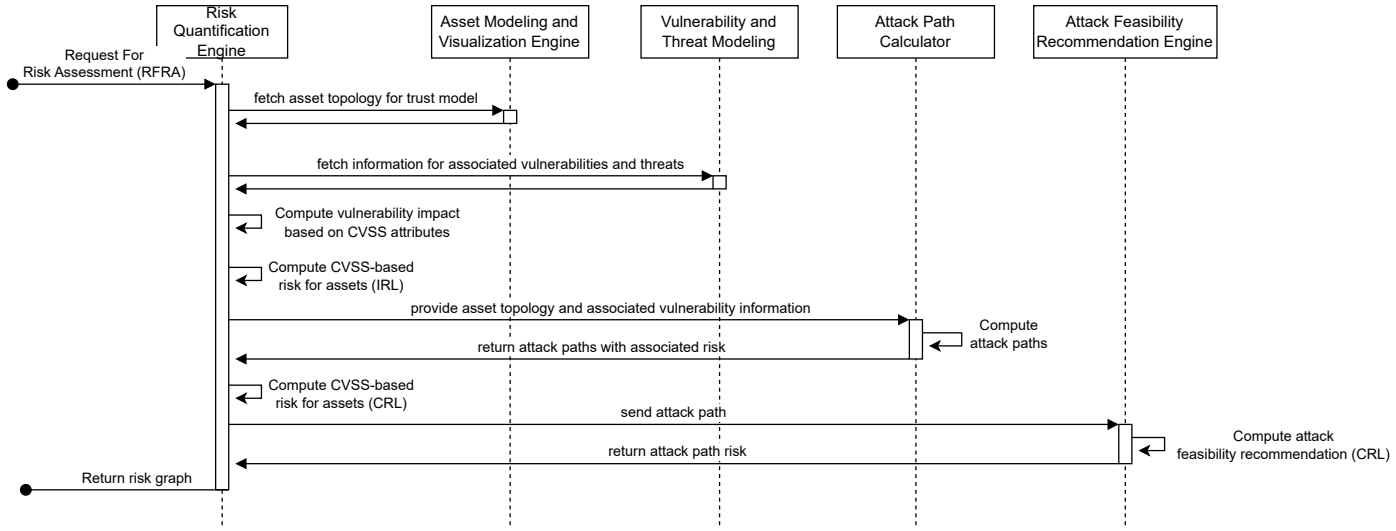


Figure 7.9: CVSS-based methodology with IRL and CRL calculations

relevant threats and vulnerabilities), a new **Request For Risk Assessment (RFRA)** can be issued.

2. This, in turn, triggers the risk quantification engine to fetch the component diagram from the asset modelling and visualization engine, and the relevant attack scenarios that allow the derivation of the IRL score. Specifically, based on the vulnerabilities and their CVSS characteristics, it is possible to derive the vulnerability impact, while the associated threats are assigned a threat likelihood factor that can be parameterised by the Security Administrator.

3. After the IRL scores have been quantified per risk scenario on a device, the Attack Path Calculator is invoked in order to automatically infer the attack paths based on a set of rules that dictate the conditions under which a vulnerability can be exploited by an adversary to carry out a cascading attack using one asset as an entry point to compromise a different asset. In each attack path, one vulnerability is considered at a time (even though there may be multiple attack paths leading from one asset to a different one).

4. After this calculation is complete, it is possible for the Attack Feasibility Recommendation Engine to assign a CRL score for each attack path that is identified. Finally, all collected risk results are finally aggregated and sent back to the Security Administrator.

The aforementioned CVSS-based methodology offers automated mechanisms that may facilitate the collection or even enhance the threat analysis required by the TARA framework, as depicted in the first two steps of Figure 7.10. The iterative TARA process for the enforcement of control scenarios in the risk quantification process consists of the following steps:

1. For the initiation of the threat analysis process, the Security Administrator provides the detailed specification of the attack steps for each attack path, and identifies the **Attack Feasibility Rating** as part of the TARA process.

2. Subsequently, the Security Administrator initiates a TARA risk assessment process to perform their risk analysis, as specified in [1]. Note that, as this process may be performed multiple times throughout the operational lifecycle of the target domain infrastructure, we define the term **TARA iteration** which describes the single execution of the TARA steps.

3. After specifying the component diagram characterizing the domain infrastructure under evaluation, the Security Administrator specifies the relevant damage scenarios and their associated impact through the **Damage Scenario Management** component.

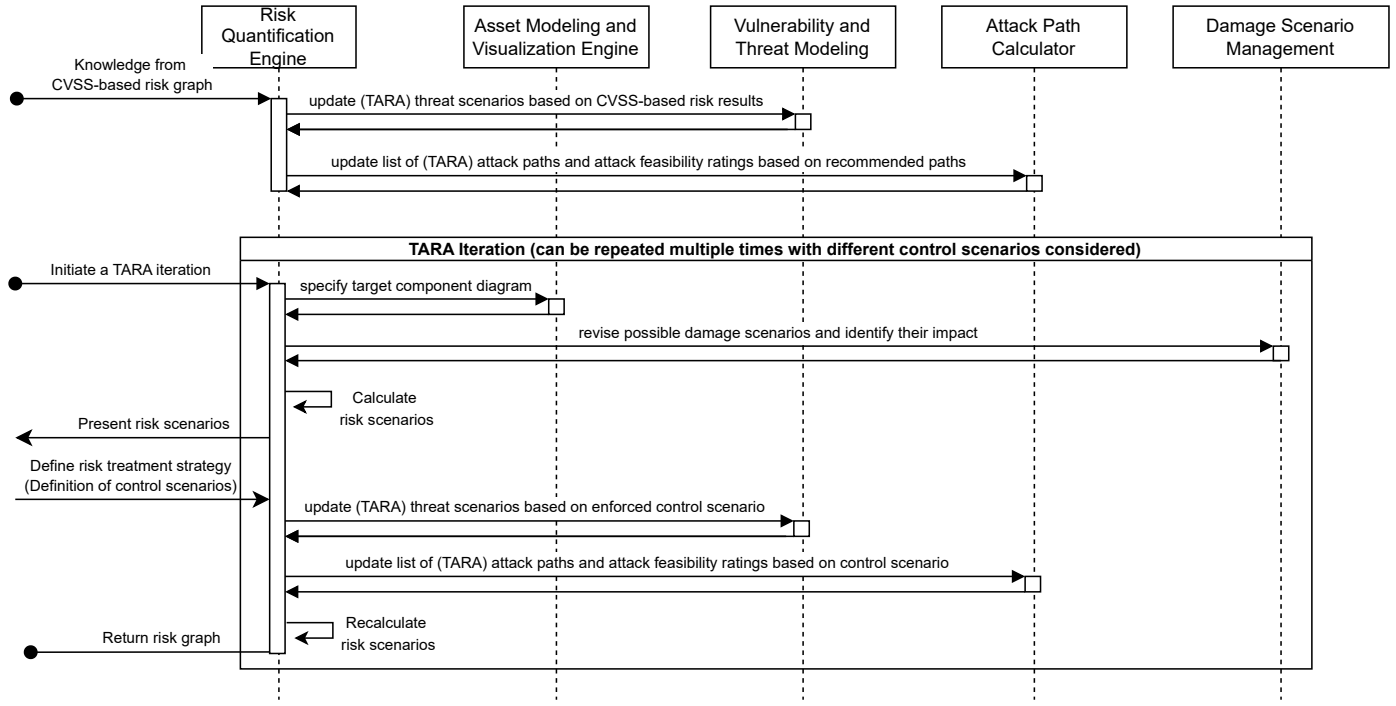


Figure 7.10: TARA iterative process encompassing the enforcement of control scenarios in the risk quantification process

4. After this process has been performed, the **Risk Quantification Engine** is able to quantify the risk based on the equations defined and implemented in D3.2.

5. The risk scenarios are then made available to the Security Administrator.

In the context of the risk treatment decision process, it is necessary to decide the management strategy of the identified risk scenarios: **Accept**, **Avoid**, **Reduce**, **Transfer**. When deciding to reduce the risk of one or more risk scenarios, it is necessary that the Security Administrator specifies the set of **security controls** as mitigation measures through a **control scenario**. The impact of an enforced control scenario is typically reflected as *reduced attack feasibility or complete elimination of previously identified attack paths*. However, note that introducing a control scenario may also lead to new attack vectors within the system. However, the simultaneous enforcement of multiple controls as part of a control scenario can result in diminished effectiveness due to dependencies or conflicts between them. To this end, the Security Administrator may need to iteratively execute the TARA process until the residual risks are reduced to an acceptable level.

After the risk of the system has been reduced to an acceptable level following the process outlined above, it is possible to proceed with the calculation of the RTL. Specifically, as shown in Table 7.17, the Risk Quantification Engine exposes an API to enable querying for risk scenarios and their associated risk values across different TARA iterations, where different control scenarios are taken into consideration. Thus, the RTL Calculator is able to identify the **maximum residual risk** within the system, thus deriving RTL constraints based on belief and disbelief values. This is also demonstrated in Section 7.5, where the querying capabilities that enable the RTL calculations are demonstrated through an example of two consecutive TARA iterations showcasing the filtering capabilities resulting RTL constraints for a specific context (i.e., security property and target trust object).

7.4 Streamlining TARA with Automated Attack Path Calculation

The Attack Path Calculator sub-component, as aforementioned, enhances the capability of the REWIRE RA engine to analyse the risk for a particular asset through the identification of possible attack paths as-

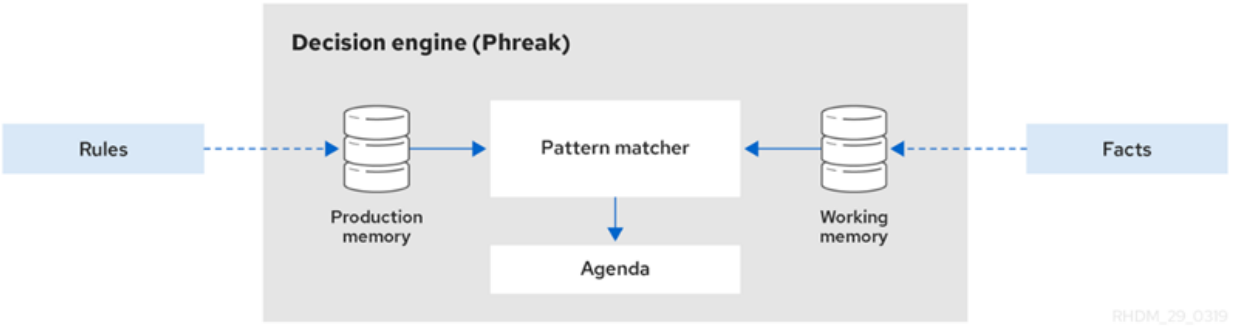


Figure 7.11: The Drools Engine internal architecture

sociated with this asset. The attack path calculation analysis does not require any additional information, beyond what is already used in the context of the risk assessment. Specifically, it takes into consideration the asset topology, considering relationships between assets, as well as the vulnerabilities affecting each asset. Leveraging this information, the attack path calculation component is able to derive a set of attack paths that an attacker can leverage in order to infiltrate the infrastructure and use an assets as an entry point to access another by exploiting one associated vulnerability at a time.

Note that not all vulnerabilities enable an adversary to pivot from one asset to another. Thus, it is important that a security administrator defines the rules dictating the conditions based on which a vulnerability can be exploited by a malicious user to leverage an identified attack path between assets. These rules can be then provided to the **Drools** expert system [58], which combines the rules with the existing topology information and derives the identified attack paths. Drools is a business-rule management system with a forward-chaining and backward-chaining inference-based rules engine, which enables the fast and reliable evaluation of business rules and complex event processing. As demonstrated in Figure 7.11, the Drools engine can receive two types of information as input, i.e., **rules** and **facts**.
In the context of the attack path calculation, rules are defined as **static information** that describes the conditions for an asset relationship be a valid link of an attack path. These rules are expressed in the **Drools Rule Language (DRL)**, are stored in .drl files, and can be configured by an administrator during the launch of the REWIRE RA engine. An example of such a file is presented in Table 7.2. It utilizes the CVSS attributes in order to define the conditions of a valid attack path edge. Simultaneously, the facts are loaded during runtime, and describe the asset topology information, as well as the associated vulnerabilities per asset. Using this information, the Drools engine is able to perform multiple rounds of processing to identify the available attack paths. For each round, the Drools engine uses the input data to derive multiple intermediate facts, which are provided as input to the subsequent round. When no additional fact is produced for a specific round, the Drools Engine terminates its execution, and the resulted facts are placed in the Drools Agenda. Then, the Attack Path Calculator component can retrieve the calculated attack paths, store them in the NoSQL database, and report them back to the user.

Rule Id	Rule Description	Rule expressed in DRL

1	Input Validation Rule	`rule "rule-ValidationMandatoryInputCheck"`
		`when`
		`not(exists(APAsset(entryPoint==true) && APAsset(target==true) && APAttacker()))`
		`then`
		`insert(new ExceptionStructural("At least one entry point one target point and an attacker profile should exist"));`
		`End`

Rule Id	Rule Description	Rule expressed in DRL

2	Rule for creating an attack path for vulnerabilities that require Network access	`rule "rule-PropagationFromEntryPointsRemote"`
		`when`
		`$assetto:APAsset(isEntryPoint() == true)`
		`$attacker:APAttacker()`
		`$vuln: APVulnerability(vector == "NETWORK" || vector == "ADJACENT_NETWORK", AttackerSkillType.isAttackerCapableBasedOnComplexity(complexity, $attacker.skill)) from $assetto.vulnerabilities`
		`not(exists(APPath(fromAssetId==APPath.REMOTE_ADVERSARY_ID, toAssetId==$assetto.getId() , vuln==$vuln )))`
		`then`
		`insert(new APPath($vuln, APPath.REMOTE_ADVERSARY_ID , $assetto.getId()));`
		`end`

3	Rule for creating an attack path for vulnerabilities that don't require Network access	`rule "rule-PropagationFromEntryPointsLocal"`
		`when`
		`$assetto: APAsset(isEntryPoint() ==true)`
		`$attacker: APAttacker()`
		`$vuln: APVulnerability(vector=="LOCAL", AttackerSkillType.isAttackerCapableBasedOnComplexity(complexity, $attacker.skill)) from $assetto.vulnerabilities`
		`not(exists(APPath(fromAssetId==APPath.LOCAL_ADVERSARY_ID, toAssetId==$assetto.getId(), vuln==$vuln )))`
		`then`
		`insert(new APPath( $vuln, APPath.LOCAL_ADVERSARY_ID , $assetto.getId()));`
		`end`

4	Rule for expanding an attack path for vulnerabilities that require Network access	`rule "rule-PropagationFromPivotingLocal"`
		`when`
		`$path: APPath($assetfromid: toAssetId)`
		`$assetfrom: APAsset(id==$assetfromid)`
		`$dependency: APDependency(fromAssetId==$assetfrom.id, $assettargetid: toAssetId)`
		`$assetto:APAsset(id==$assettargetid)`
		`$attacker:APAttacker()`
		`$vuln: APVulnerability(vector == "LOCAL", AttackerSkillType.isAttackerCapableBasedOnComplexity(complexity, $attacker.skill)) from $assetto.getVulnerabilities()`
		`not(exists(APPath(fromAssetId==$assetfrom.getId(), toAssetId==$assetto.getId(), vuln==$vuln)))`
		`then`
		`insert(new APPath($vuln, $assetfrom.getId() , $assetto.getId()));`
		`end`

Rule Id	Rule Description	Rule expressed in DRL

5	Rule for expanding an attack path for vulnerabilities that don't require Network access	`rule "rule-PropagationFromPivotingRemote"`
		`when`
		`$path: APPath($assetfromid:toAssetId)`
		`$assetfrom: APAsset(id==$assetfromid)`
		`$dependency: APDependency(fromAssetId==$assetfrom.id , $assettargetid:toAssetId)`
		`$assetto:APAsset(id==$assettargetid)`
		`$attacker:APAttacker()`
		`$vuln : APVulnerability(vector == "NETWORK" || vector == " ADJACENT_NETWORK", AttackerSkillType. isAttackerCapableBasedOnComplexity(complexity, $attacker. skill)) from $assetto.getVulnerabilities()`
		`not(exists(APPath(fromAssetId==$assetfrom.getId(), toAssetId== $assetto.getId(), vuln==$vuln)))`
		`then`
		`insert(new APPath($vuln, $assetfrom.getId() , $assetto.getId())) ;`
		`end`

6	Rule for obtaining the final attack chain	`rule "rule-ChainGeneration-EntryPoint"`
		`when`
		`$path:APPath(fromAssetId==APPath.REMOTE_ADVERSARY_ID)`
		`not(exists(APChain(getEntryPointPath()==$path)))`
		`then`
		`insert(new APChain($path));`
		`end`

Table 7.2: Rules for obtaining attack chains, expressed in DRL format

In addition to the management of the Drolls sessions for calculating the attack paths for the target topology, the Attack Path Calculator component is also responsible for managing the entire lifecycle of the attack path assessment tasks within the REWIRE RA engine. Note that, in the calculations of the Drools engine, the attacker capabilities need to be taken into account, as demonstrated in Figure 7.12. This can be seen as a parameter that enables the Drools engine to decide whether a vulnerability can be exploited based on the attacker's skills, quantified on a 5-tier scale: **Very Low**, **Low**, **Medium**, **High**, and **Very High**. Each attack path corresponds to a different combination of vulnerabilities that are used by an adversary to pivot through these assets. Finally, in the context of Attack Path Assessment, a risk value is assigned to each one of these attack paths, as shown in Figure 7.13, following the rules defined in Table 7.2.

7.5 Transitioning from TARA to RTL

In Section 7.3.3, we provided a systematic overview of the operation of the REWIRE RA Engine through detailed sequence diagrams and step-by-step descriptions. In order to better illustrate the availability of risk-related information supporting RTL calculations, here we provide further details on the TARA process from the perspective of a Security Administrator interacting with the REWIRE RA GUI. To this end, we present an example from the **automotive use case** involving a simplified in-vehicle topology which receives **Global Navigation Satellite System (GNSS)** data which is utilized for determining the position of the vehicle, and providing this data to the onboard computer of the vehicle.

Following the steps of the sequence diagram in Figure 7.10, the Security Administrator logs into the

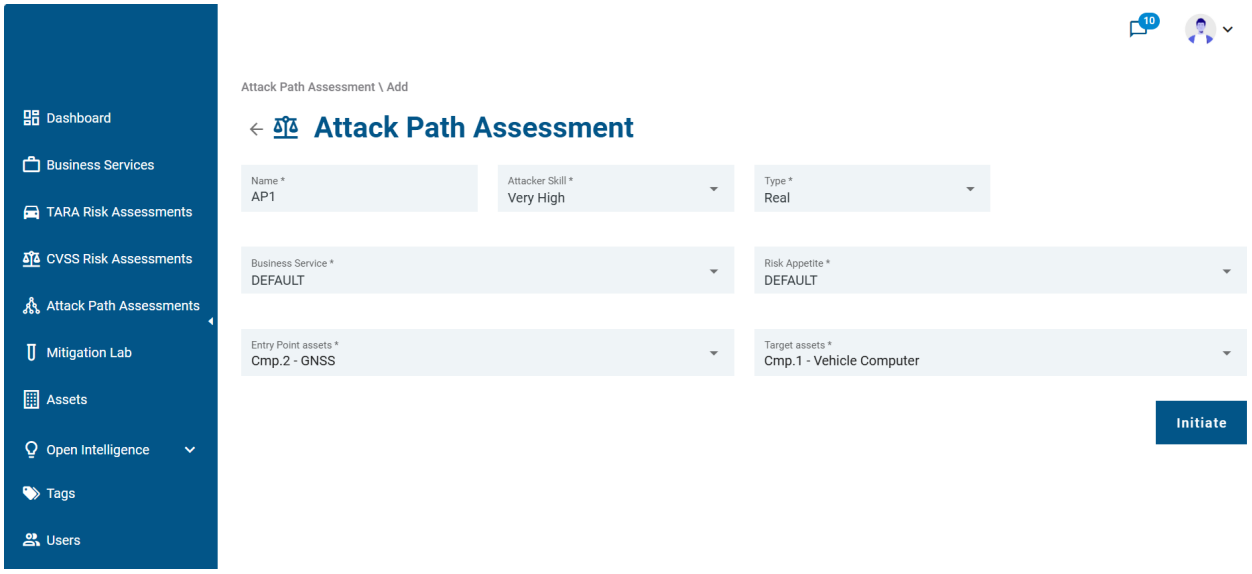


Figure 7.12: Initializing a new Attack Path Assessment

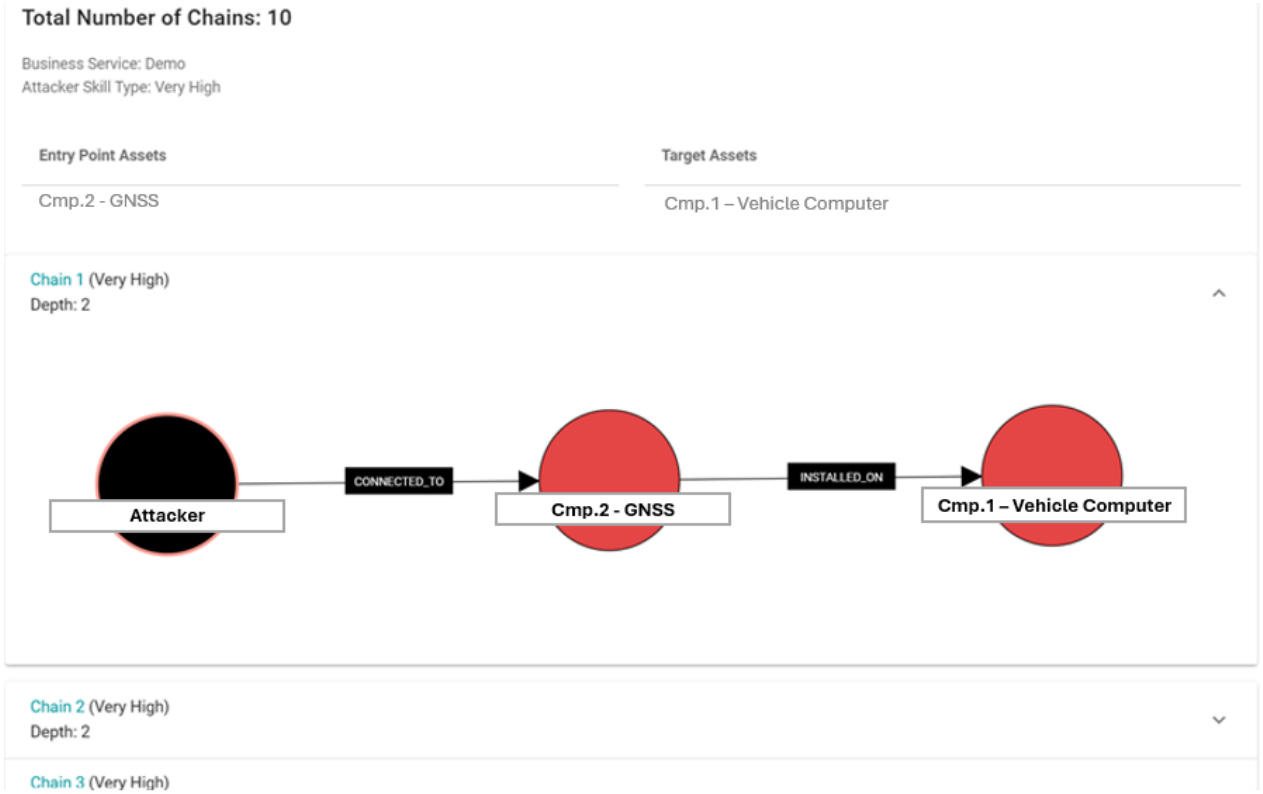


Figure 7.13: Visualizing results of Attack Path Assessment

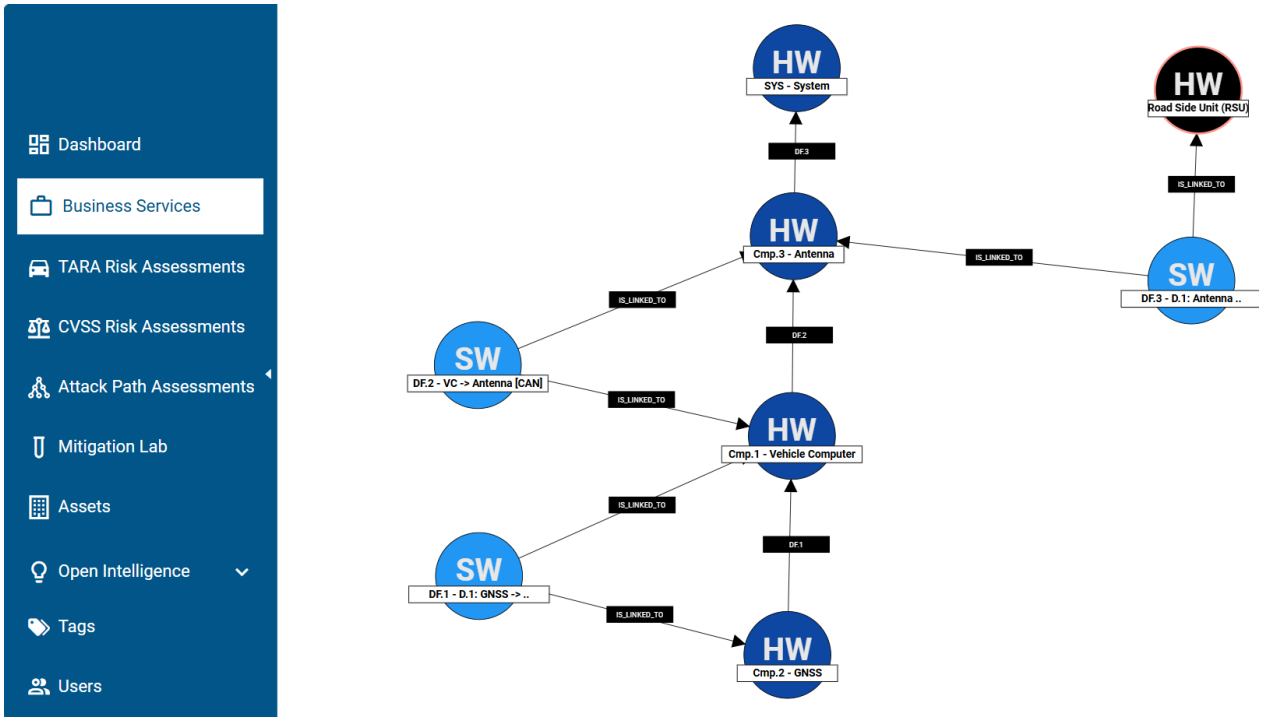


Figure 7.14: Step 1: Specify target component diagram

REWIRE RA Engine and specifies the component diagram corresponding to the GNSS data transmission. In Figure 7.14, we provide the visualization of all assets and interdependencies that characterize the domain infrastructure (i.e., the in-vehicle system), including software, hardware, and data flow entities, thus allowing for a fine-grained identification of attack paths.

When navigating to the ”TARA Damage Scenario” tab (left sidebar), the administrator is able to specify the relevant damage scenarios associated with this item function. As shown in Figure 7.15, the Security Administrator is able to specify the name of each damage scenario and the corresponding impact in terms of **Safety**, **Economical**, **Operational**, and **Privacy** aspects.

Afterwards, the Security Administrator is able to identify the threat scenarios that may target any of the assets in the component diagram. As shown in Figure 7.16, each threat scenario can be associated with a particular asset (e.g., Tampering on GNSS sensor or Spoofing on the data flow from the sensor to the vehicle computer), as well as the cybersecurity property compromised by such an attack. This provides the capability for linking the types of threats that affect a specific trust property with the calculations of the RTL constraints.

The next step, as shown in Figure 7.17, is the definition of the attack paths for the TARA risk assessment process. In this regard, each path may consist of multiple intermediate steps that help realize a more complex and cascading attack scenario. The attack feasibility of each scenario is reflected in the reporting of the attack scenarios and can be dynamically configured from the six values specified in [1], namely **Knowledge**, **Expertise**, **Elapsed Time**, **Equipment**, and **Window Of Opportunity**.

By clicking the second rightmost button shown in Figure 7.17, the Security Administrator is able to inspect all intermediate steps that need to be followed as part of a particular attack path, and evaluates the intermediate attack feasibility ratings per attack step.

Before performing the risk quantification process, it is also possible for the Security Administrator to define the number of security controls to be considered in the risk assessment iterations. In this regard, as shown in Figure 7.18, the Security Administrator is able to click the ”Controls” tab in order to manage the list of available controls and associate them with specific attack paths that aim to mitigate.

In the first iteration, which is depicted in Figure 7.19, the Security Administrator calculates the baseline risk, which refers to the risk scenarios and their associated scores without considering the existence of

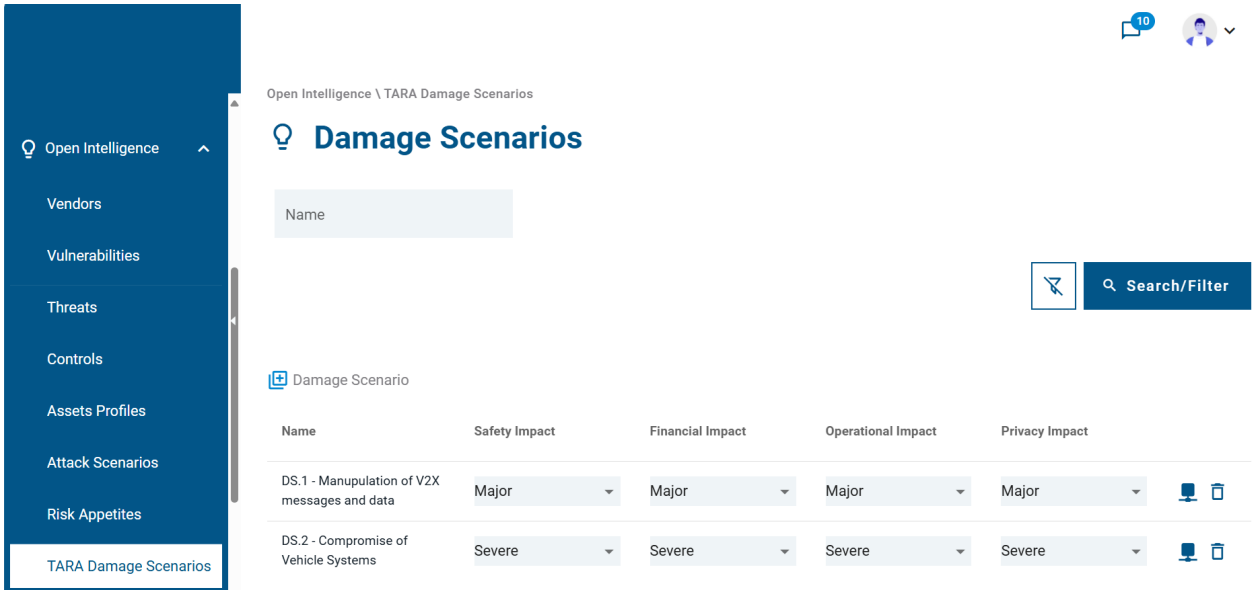


Figure 7.15: Step 2: Specify TARA damage scenarios

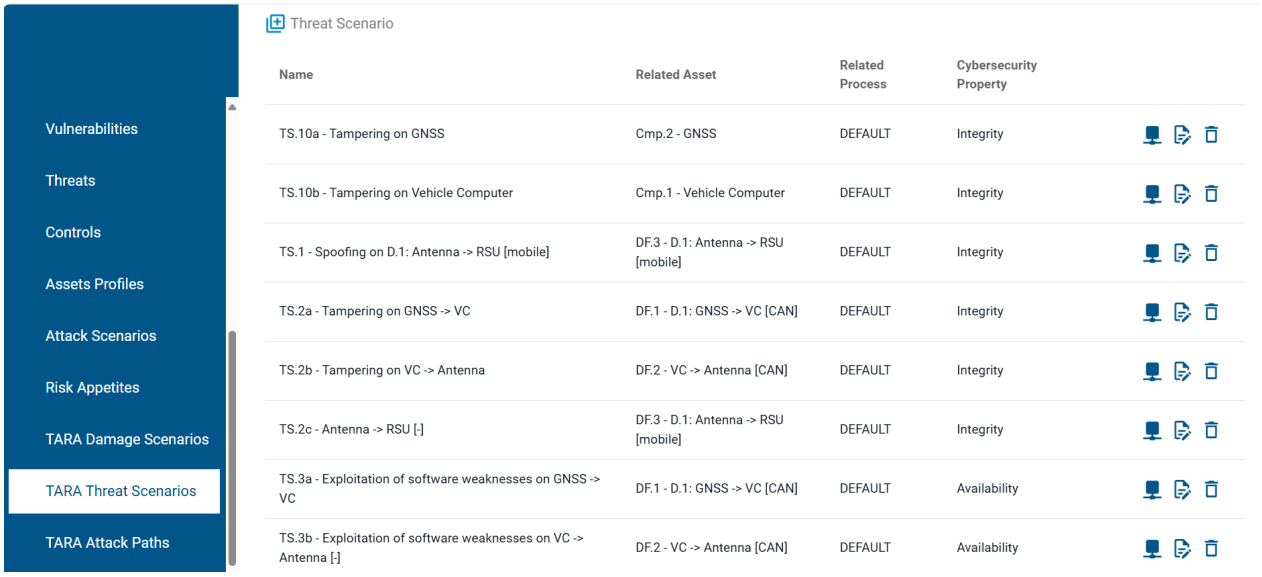


Figure 7.16: Step 3: Specify TARA threat scenarios

any security controls. At this step, the Security Administrator has concluded the first iteration of the TARA risk assessment process by making a decision on the mitigation strategy to be applied per risk scenario.

Figure 7.20 depicts the management panel of each iteration where the users can instantiate new iterations by taking into account different control scenarios, and monitor the progress of the tasks. In this example, it is clear that the second iteration refers to a control scenario where access control mechanisms are available.

The execution of multiple TARA iterations enables the comparison of different control scenarios. Specifically, as depicted in Figure 7.21, this comparison enables the Security Administrators to determine the impact of each control scenario in the overall risk posture of the topology. As shown in the Figure, this panel offers filtering capabilities that allow narrowing down those risk scenarios to ones that are relevant for a particular context. This is important for the derivation of both the **RTL constraints**, and the **optimal selection of the weights** to be specified as part of the ATL calculations. For instance, assuming the need to derive the RTL constraints pertaining to the in-vehicle integrity, the filtering options would allow the three first risk scenarios but exclude the impact of the Antenna - Road Side Unit risk scenario, as it falls outside the scope of the trust proposition to be evaluated.

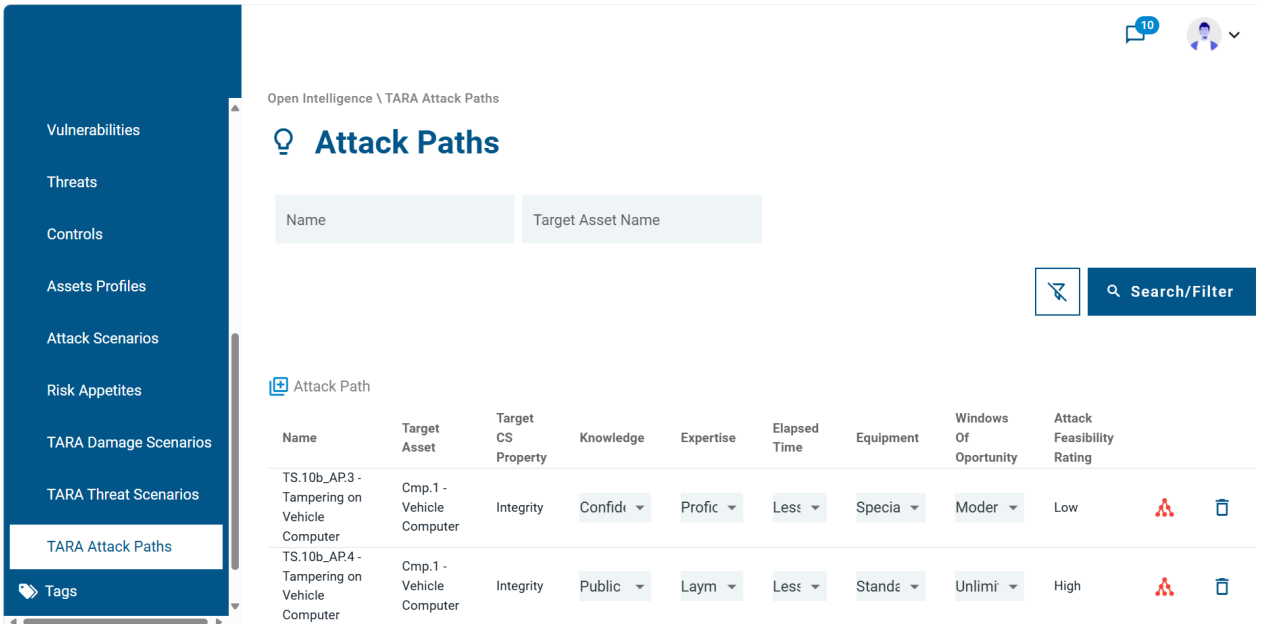


Figure 7.17: Step 4: Specify TARA attack paths

7.6 Specification of Implemented Interfaces

In this Section, we provide the specification of the REWIRE RA interfaces implemented in the final release. These include the execution of concrete CVSS-based and TARA-based risk assessment processes, as well as all **Create**, **Update**, **Read**, and **Delete (CRUD)** operations for the necessary data models (e.g., assets, threats, vulnerabilities, controls).

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Addition of an process - /api/v1/processes	
Inputs & Outputs	Inputs	Outputs
	Process name	Id of the process as persisted in the REWIRE RA database.
Interaction with components	1. Asset modeling and visualization component 2. Risk Quantification Engine, 3. OLISTIC Backend, 4. OLISTIC Frontend, 5. External OEM integration activities	

Table 7.3: Add a process in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Addition of an asset - /api/v1/assets	
Inputs & Outputs	Inputs	Outputs

	1. Asset name 2. Attributes (key-value pairs) 3. Relationships (i.e., interdependencies with other assets, data flows, participating functions) 4. Linked cybersecurity properties	1. Id of the asset as persisted in the REWIRE RA database.
Interaction with components	1. Asset modeling and visualization component 2. Risk Quantification Engine, 3. OLISTIC Backend, 4. OLISTIC Frontend, 5. External OEM integration activities	

Table 7.4: Add an asset in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Addition of a vulnerability - /api/v1/vulnerabilities	
Inputs & Outputs	Inputs	Outputs
	1. Vulnerability name, 2. Vulnerability description, 3. Vulnerability library, 4. Published date, 5. (Optional) CVSS attributes (v2.0/v3.1)	1. Id of the vulnerability as persisted in the REWIRE RA database.
Interaction with components	1. Vulnerability and Threat modeling component 2. Risk Quantification Engine, 3. OLISTIC Backend, 4. OLISTIC Frontend, 5. External OEM integration activities	

Table 7.5: Add a vulnerability in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Addition of a threat - /api/v1/threats	
Inputs & Outputs	Inputs	Outputs

	1. Threat name, 2. Threat description, 3. Threat library, 4. Published date	1. Id of the threat as persisted in the REWIRE RA database.
Interaction with components	1. Vulnerability and Threat modeling component 2. Risk Quantification Engine, 3. OLISTIC Backend, 4. OLISTIC Frontend, 5. External OEM integration activities	

Table 7.6: Add a threat in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Addition of a mitigation control - /api/v1/controls	
Inputs & Outputs	Inputs	Outputs
	1. Control name, 2. Control description, 3. Control library, 4. Published date, 5. Attributes	1. Id of the control as persisted in the REWIRE RA database.
Interaction with components	1. Mitigation Strategies component 2. Risk Quantification Engine, 3. OLISTIC Backend, 4. OLISTIC Frontend, 5. External OEM integration activities, 6. TAM	

Table 7.7: Add a control in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Addition of a TARA damage scenario - /api/v1/tara/damage-scenarios	
Inputs & Outputs	Inputs	Outputs

	1. Damage scenario name, 2. Safety impact, 3. Financial impact, 4. Operational impact, 5. Privacy impact, 6. Overall impact	1. Id of the TARA damage scenario as persisted in the REWIRE RA database.
Interaction with components	1. Risk Quantification Engine, 2. OLISTIC Backend, 3. OLISTIC Frontend, 4. External OEM integration activities, 5. TAM	

Table 7.8: Add a TARA Damage Scenario in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Addition of a TARA attack path - /api/v1/tara/attack-paths	
Inputs & Outputs	Inputs	Outputs
	1. Attack path name, 2. Chain of intermediate asset id forming the attack path, 3. Target asset id, 4. Target Cybersecurity property, 5. Attack path Elapsed Time, 6. Attack path Expertise, 7. Attack path Knowledge, 8. Attack path Windows of opportunity, 9. Attack path Equipment, 10. Attack path Attack Feasibility Rating	1. Id of the TARA attack path as persisted in the REWIRE RA database.
Interaction with components	1. Risk Quantification Engine, 2. OLISTIC Backend, 3. OLISTIC Frontend, 4. External OEM integration activities, 5. TAM	

Table 7.9: Add a TARA Attack Path in REWIRE RA

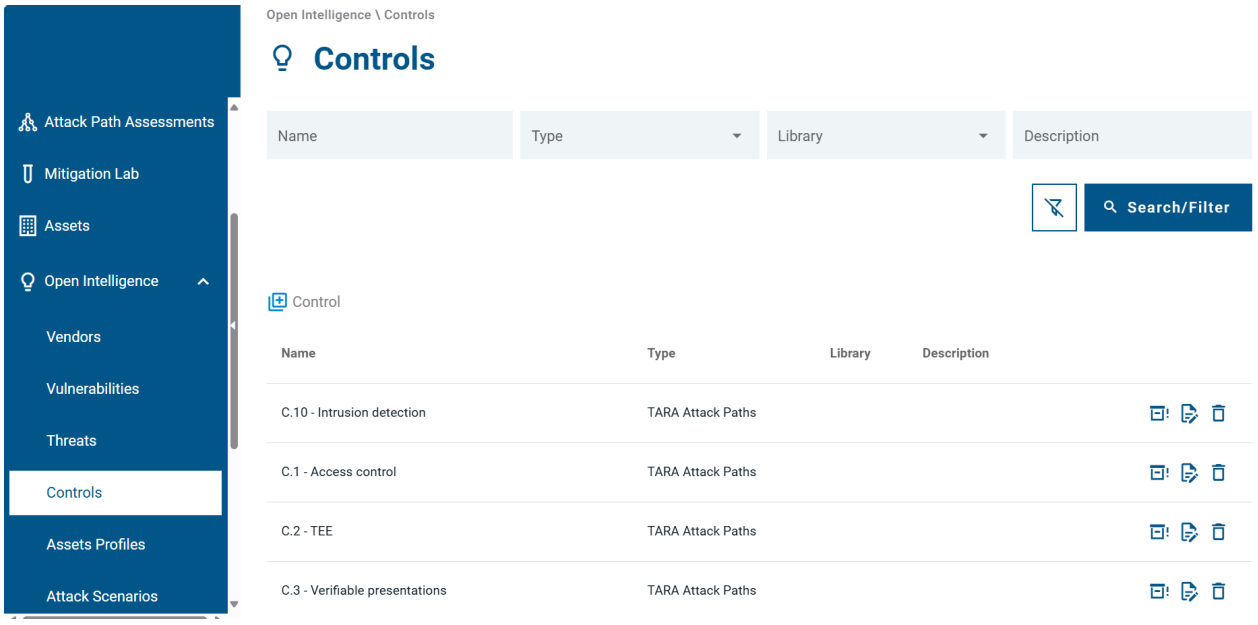


Figure 7.18: Step 5: Specify TARA security controls

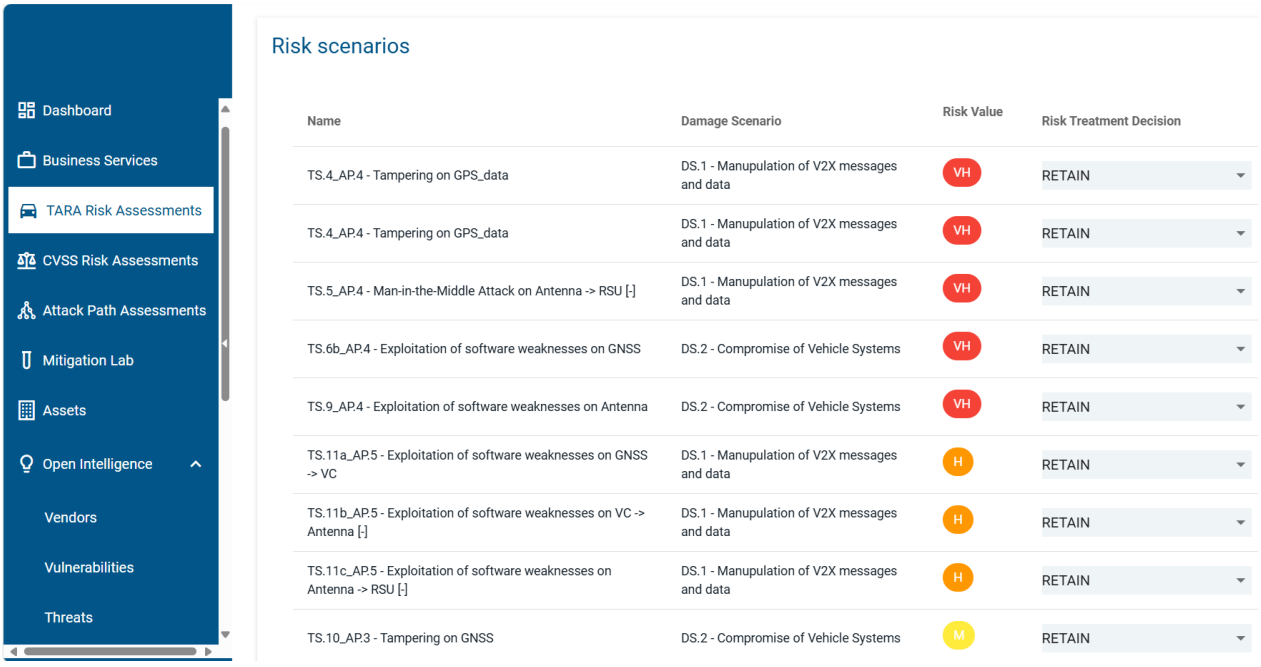


Figure 7.19: Step 6: Results of a risk assessment with no security controls enforced

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Addition of a TARA threat scenario - /api/v1/tara/threat-scenarios	
Inputs & Outputs	Inputs	Outputs
	1. Threat scenario name, 2. Related (target) asset id, 3. Related process id, 4. Related cybersecurity property	1. Id of the TARA threat scenario as persisted in the REWIRE RA database.

Interaction with components	1. Risk Quantification Engine, 2. OLISTIC Backend, 3. OLISTIC Frontend, 4. External OEM integration activities, 5. TAM

Table 7.10: Add a TARA Threat Scenario in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Creation of a TARA risk assessment task - /api/v1/tara/risk-assessments	
Inputs & Outputs	Inputs	Outputs
	1. Risk assessment name, 2. Related process id	1. Id of the TARA risk assessment task as persisted in the REWIRE RA database.
Interaction with components	1. Risk Quantification Engine, 2. OLISTIC Backend, 3. OLISTIC Frontend, 4. External OEM integration activities, 5. TAM	

Table 7.11: Add a Risk Assessment task in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Adjustment of a TARA attack path inside a risk assessment task - /api/v1/tara/attack-paths/{id}/attack-path-profiles	
Inputs & Outputs	Inputs	Outputs

	1. TARA Attack path id within a risk assessment task, 2. Chain of intermediate asset id forming the attack path, 3. Target asset id, 4. Target Cybersecurity property, 5. Attack path Elapsed Time, 6. Attack path Expertise, 7. Attack path Knowledge, 8. Attack path Windows of opportunity, 9. Attack path Equipment, 10. Attack path Attack Feasibility Rating	1. HTTP OK
Interaction with components	1. Risk Quantification Engine, 2. OLISTIC Backend, 3. OLISTIC Frontend, 4. External OEM integration activities, 5. TAM	

Table 7.12: Update TARA Attack Path in a Risk Assessment task in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Adjustment of damage scenario within a risk assessment - /api/v1/tara/risk-assessments/{raId}/damageScenario	
Inputs & Outputs	Inputs	Outputs
	1. Risk assessment id, 2. Damage scenario id, 3. Safety impact, 4. Financial impact, 5. Operational impact, 6. Privacy impact, 7. Overall impact	1. HTTP OK
Interaction with components	1. Risk Quantification Engine, 2. OLISTIC Backend, 3. OLISTIC Frontend, 4. External OEM integration activities, 5. TAM	

Table 7.13: Update TARA Damage Scenario in a Risk Assessment task in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Inclusion of a control in a risk assessment task - /api/v1/tara/risk-assessments/{raId}/damageScenario	
Inputs & Outputs	Inputs	Outputs
	1. Risk assessment id, 2. Control id	1. HTTP OK
Interaction with components	1. Risk Quantification Engine, 2. OLISTIC Backend, 3. OLISTIC Frontend, 4. External OEM integration activities, 5. TAM	

Table 7.14: Update control in a Risk Assessment task in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Execute a risk assessment task - /api/v1/tara/risk-assessments/{raId}/execute	
Inputs & Outputs	Inputs	Outputs
	1. Risk Assessment id	1. HTTP OK
Interaction with components	1. Risk Quantification Engine, 2. OLISTIC Backend, 3. OLISTIC Frontend, 4. External OEM integration activities, 5. TAM	

Table 7.15: Execute a Risk Assessment in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP GET	
Purpose	Query for TARA risk results - /api/v1/tara/risk-assessments/102/risk-scenarios	
Inputs & Outputs	Inputs	Outputs

	1. Risk assessment id 2. Query parameter 1: Attack path name 3. Query parameter 2: Damage scenario name 4. Query parameter 3: Risk level 5. Query parameter 4: Risk assessment name	Array list where each entry is a risk scenario, containing the following: 1. Risk assessment id 2. Attack path id 3. Attack path name 4. Attack path target asset id 5. Attack path target cybersecurity property 6. Attack path feasibility rating 7. Damage scenario id 8. Damage scenario name 9. Damage scenario overall impact 10. Risk scenario level
Interaction with components	1. Risk Quantification Engine, 2. OLISTIC Backend, 3. OLISTIC Frontend, 4. External OEM integration activities 5. TAM	

Table 7.16: Get Risk Assessment results in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP GET	
Purpose	Query for TARA comparison results per risk scenario - /api/v1/tara/risk-assessments/102/compare-results	
Inputs & Outputs	Inputs	Outputs
	1. Base Risk assessment id (i.e., with no controls in place) 2. Query parameter 1: List of risk assessment ids 3. Query parameter 2: Asset name 4. Query parameter 3: Cybersecurity property 5. Query parameter 4: Damage Scenario name 6. Query parameter 4: Initial Risk Level	Array list where each entry is a risk scenario, containing the following: 1. Risk assessment id 2. Attack path id 3. Attack path name 4. Attack path target asset id 5. Attack path target cybersecurity property 6. Attack path feasibility rating 7. Damage scenario id 8. Damage scenario name 9. Damage scenario overall impact 10. Risk level per risk assessment id

Interaction with components	1. Risk Quantification Engine, 2. OLISTIC Backend, 3. OLISTIC Frontend, 4. External OEM integration activities 5. TAM

Table 7.17: Compare Risk Assessment results in REWIRE RA on a risk-scenario level

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP POST	
Purpose	Execute an attack path assessment task - /api/v1/attack-path-assessment/{apId}/execute	
Inputs & Outputs	Inputs	Outputs
	1. Attack Path Assessment id	1. HTTP OK
Interaction with components	1. Risk Quantification Engine, 2. Attack Path Calculator, 3. OLISTIC Backend, 4. OLISTIC Frontend, 5. External OEM integration activities, 6. TAM	

Table 7.18: Execute an Attack Path Calculator task in REWIRE RA

REWIRE RA	Interface Technical Details	
Type of Interface	REST HTTP GET	
Purpose	Query for TARA risk results - /api/v1/tara/attack-path-assessments/103/visualize	
Inputs & Outputs	Inputs	Outputs
	1. Attack path assessment id	Array list where each entry is a attack path, containing the following: 1. Attack path assessment id 2. Attack path id 3. Attack path name 4. Attacker skill level 5. Attack step id (Asset id and vulnerability id) 6. Overall CRL for the identified path

Interaction with components	1. Risk Quantification Engine, 2. Attack Path Calculator, 3. OLISTIC Backend, 4. OLISTIC Frontend, 5. External OEM integration activities 6. TAM

Table 7.19: Get Attack Path Assessment results in REWIRE RA

7.7 Reinforcing RTL Calculation by Abstracting the Underlying Risk Quantification Engine

In the final release of the REWIRE RA Engine, we focused on the the features pertaining to the realization of the RTL constraints. Specifically, the goal of the release of the integrated REWIRE Architecture is to evaluate the implemented features of the REWIRE RA Engine in the context of the use cases, and particularly the automotive use case, as highlighted throughout this Chapter. This will provide key insights for the evaluation of the robustness and potential fine-tuning that needs to be perormed on the RTL equations.

A core factor that affects the final RTL constraints is the quantification of risk derived from the TARA methodology. However, note that the TARA specification does not dictate a single, universal risk quantification equation applicable across all domains and requirements, but the risk equation presented in this Chapter is an example originating from the ISO/SAE 21434 standard, which provided the basis for the TARA methodology.

Thus, the performance evaluation of the final release of the REWIRE RA Engine in the context of D6.2 is envisioned in two core dimensions:(i) experimentation setup based on a more complex component diagram describing a typical automotive domain infrastructure, and (ii) evaluation of different risk quantification methodologies, specifically the baseline TARA risk quantification equation compared to the CVSS-based equation. These two dimensions aim to evaluate the robustness of the RTL equations, and the assessment of how the selected risk quantification methodology influences the strictness and accuracy of the RTL constraints.

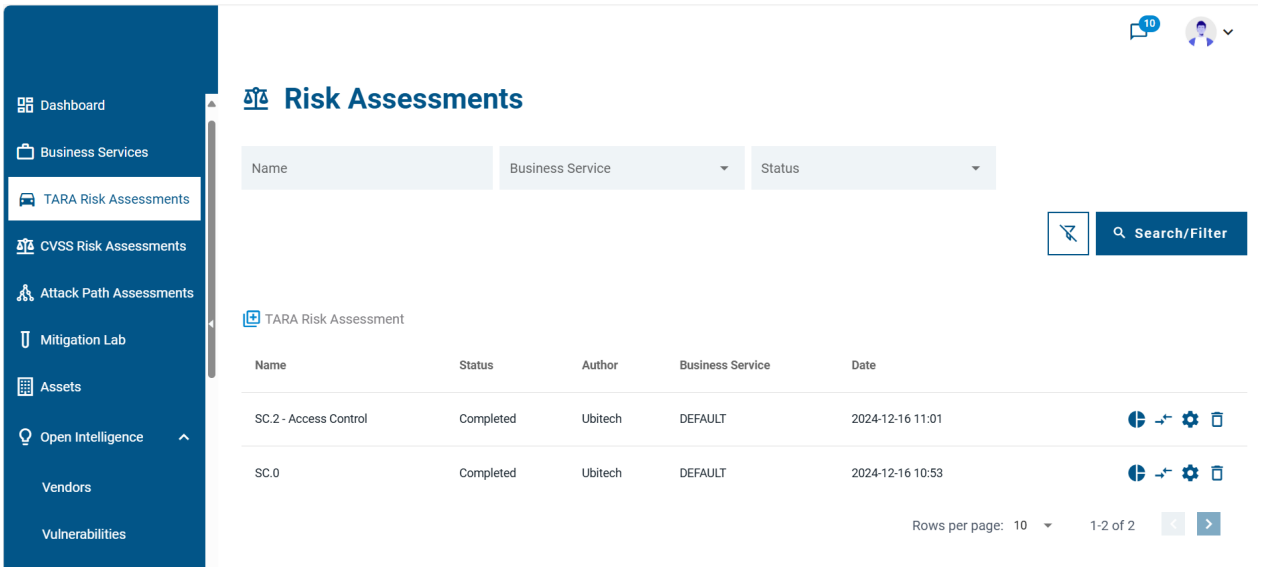


Figure 7.20: Step 7: List of risk assessment tasks: One with no controls applied and one with a single security control (Access control mechanisms enforced).

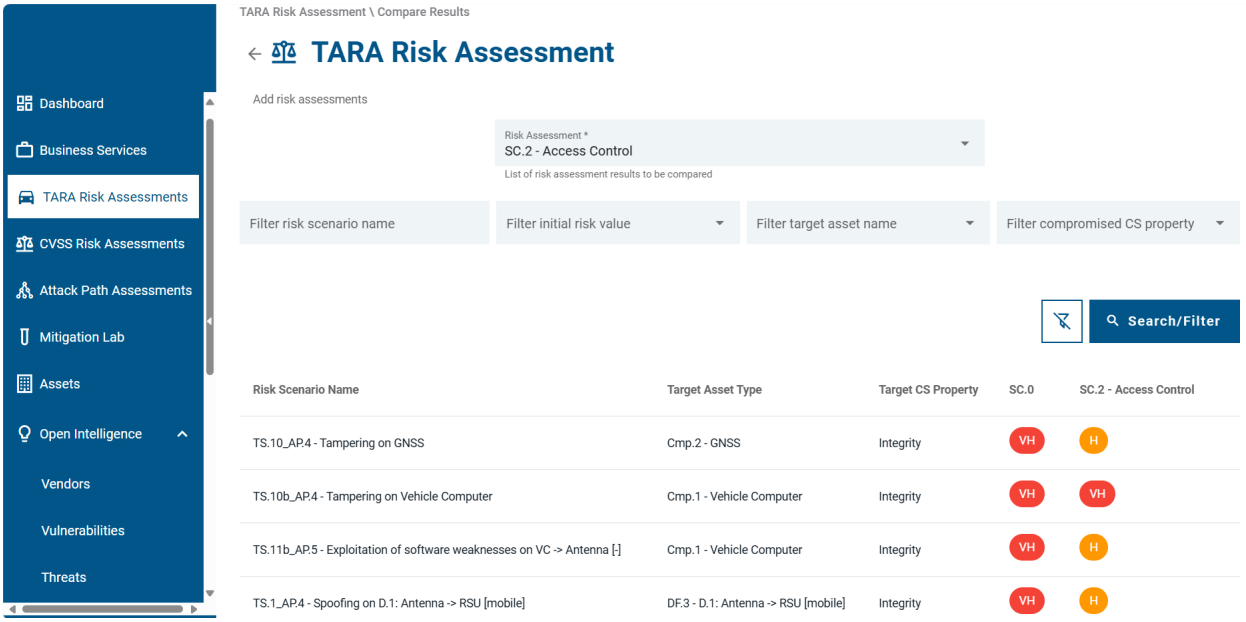


Figure 7.21: Step 8: Comparison of results from different risk assessment tasks per risk scenario.

Chapter 8

Establishment of Authenticated Channels

As detailed in Deliverable D2.2 with regards to the system model and security requirements considered for the formulation of the threat landscape of REWIRE, we considered the concept of a **domain** whose infrastructure comprises multiple heterogeneous devices subject to different security and privacy requirements. In this regard, we defined two different types of communication profiles that need to be supported through the REWIRE TCB [61], namely (i) **Inter-domain communication**, i.e., communication between devices belonging to the same domain, and (ii) **Intra-domain communication**, i.e., communication between devices belonging to different domains. Thus, in the context of the continuous authentication and authorization services of REWIRE, we need to consider communication not only between devices belonging to the same domain, but also devices belonging to different domains with different privacy requirements. Regarding the latter, there may be applications (e.g., trusted path routing, communication between different hospitals in the healthcare domain) may be needed in order to exchange data between different domains in an anonymous manner, while also being lightweight enough to operate within the computational restraints of the devices belonging to these domains.

Thus, in order to deal with communications between entities in different domains while ensuring the fulfilment of the overarching privacy requirements and preventing replay attacks, anonymous encryption channels are necessary to create secure communication channels. The conventional approach to offering anonymous encryption channels requires several cryptographic building blocks: First, two parties agree on a key, then derive a proof-of-possession, and finally execute a privacy-protecting signature and a verification protocol on the proof-of-possession token. This approach has some disadvantages: Multiple message exchanges are needed, and even more importantly, the device has to generate fresh keys for each encryption session. Therefore, conventional privacy-protecting key agreement incurs much overhead compared to classical asymmetric encryption schemes between known parties.

To provide for agreement of (symmetric) encryption keys, we are inspired by the scheme proposed by Schermann *et al.* [64], which can be seen as an anonymous authenticated credential key agreement by combining the Camenisch-Lysyanskaya (CL) credentials with elliptic curve Diffie-Hellman key agreement. This scheme is called anonymous authenticated credential key agreement (AACKA). Assume that there are n attributes, comparisons in terms of signature size and the number of pairings for verification between the CL signature and BBS signature are shown in Table 8.1. We can conclude that the signature size and the number of pairings for verification in the CL signature are both linear with the number of attributes. While in BBS signature, the signature size is constant, i.e., $1\mathbb{G}_1 + 1\mathbb{Z}_p$ and only 2 pairings are needed for verification. Therefore, BBS signature is more efficient than the CL signature. Moreover, considering the credential used in the ZTO process is a BBS signature, we design a BBS-based AACKA protocol for the REWIRE project. Further, we make use of the basename as a link token to realize the key agreement.
We need to consider two cases: (1) The device can do heavy computation, including the proof of the secret key, the randomization of credentials and the proof of the randomized credential. (2) The device has limited computation power that delegation of computation is necessary. To address these two different cases and make our BBS-based AACKA protocol adapted to REWIRE project, we proposed two

Signature scheme	Signature size	Number of pairings for verification
CL	$\mathcal{O}(n)$	$\mathcal{O}(n)$
BBS	$1\mathbb{G}_1 + 1\mathbb{Z}_p$	2

Table 8.1: Comparisons between the CL and BBS signatures

versions of BBS-based AACKA protocols. One is the device acts as a single signer, who takes over all the computation, the other one is that the device acts as the principle signer and the domain manager acts as the help signer. In the following, we will give syntax and security model a generic AACKA protocol firstly, followed by the two versions of AACKA protocols.

8.1 Syntax and Security model for a AACKA protocol

All notations used in the security model of a generic AACKA protocol are listed in Table 8.2. In an AACKA

Table 8.2: Notations used in the AACKA protocol

Notation	Meaning
$\mathcal{D}_1$	A device in domain 1
$\mathcal{D}_2$	A device in domain 2
$\mathcal{CA}$	Privacy CA
λ	The security level
pp	System public parameters
sk	$\mathcal{CA}$'s secret key
pk	$\mathcal{CA}$'s public key
n	The number of attributes for a device
$\{a_i\}_{i\in[1,n]}$	Attribute set for the device $\mathcal{D}_1$
VC	Verifiable credential
sk_{VC}	$\mathcal{D}_1$'s secret key associated with the VC
PK_{VC}	$\mathcal{D}_1$'s public key associated with the VC
(J, K)	Link token
bsn	Basename
Z	Shared secret key
k	Derived encryption key

protocol, there are three entities, i.e., a device $\mathcal{D}_1$ in domain 1, another device $\mathcal{D}_2$ in a different domain 2 and a Privacy CA $\mathcal{CA}$. The AACKA protocol consists of following algorithms/protocols:

- Setup$(\lambda) \rightarrow (pp, sk, pk)$: This setup algorithm is run by the Privacy CA $\mathcal{CA}$, taking the security level λ as input and outputting the system parameters pp, issuer's secret/public key pair (sk, pk).
- Join$\{\lambda; sk, pk\} \rightarrow \{\{a_i\}_{i\in[1,n]}, VC\}$: This join protocol is run between the device $\mathcal{D}_1$ and a Privacy CA $\mathcal{CA}$. $\mathcal{D}_1$ requests a credential from $\mathcal{CA}$ by proving the possession of the associated secret/public key. After verifying the proof successfully, $\mathcal{CA}$ issues a set of attributes $\{a_i\}_{i\in[1,n]}$ and an associated verifiable credential VC for $\mathcal{D}_1$.

- AACKA$(pp, VC) \rightarrow (Z, k)$: This key agreement protocol is run between $\mathcal{D}_1$ and $\mathcal{D}_2$. $\mathcal{D}_1$ sends the randomized verifiable credential VC' and the associated proof, which also includes a link token (J, K) associated with the basename bsn, to $\mathcal{D}_2$. After receiving the proof, $\mathcal{D}_2$ verifies the proof. After the successful verification, $\mathcal{D}_\in$ and $\mathcal{D}_1$ will generate the shared key Z and derived encryption key k.

The security of an AACKA protocol can be captured by *correctness*, *confidentiality*, *anonymity*, *linkability* and *unforgeability*. These notions are defined as follows:

- *Correctness*: A device $\mathcal{D}_1$ with a valid secret/public key pair can successfully create a shared key with another device $\mathcal{D}_2$ in domain 2.
- *Confidentiality*: For key-confidentiality, the secret key that is used to establish the shared key Z between the device $\mathcal{D}_1$ in domain 1 and another device $\mathcal{D}_2$ in domain 2 shall remain secret. To ensure the data-confidentiality, the derived encryption key k of both parties shall only persist in the entities ($\mathcal{D}_1$ and $\mathcal{D}_2$) that have previously negotiated the shared key k.
- *Anonymity*: Given two key agreement instantiations, an adversary or a service provider $\mathcal{D}_2$ cannot distinguish whether both instantiations have been computed with the same secret key or with different ones.
- *Linkability*: Given two pairs of link tokens, (J, K) and (J', K') generated by the same basename bsn are linkable. On the other hand, if this two pairs of link tokens (J, K) and (J', K') are generated by two different basenames bsn and bsn', are not linkable.
- *Unforgeability*: An adversary is not able to perform a key agreement with a verifiable credential without having the corresponding secret key sk_{VC}. For the link token, an adversary not being in possession of a valid credential VC, an associated secret key sk_{VC} and a basename bsn is not able to perform a key agreement with a different basename bsn'.

8.2 The first BBS-based AACKA protocol with a single signer

In this section, we introduce the first BBS-based AACKA protocol, where the device acts as the single signer. We are going to give an overview of the proposed AACKA protocol followed by detailed descriptions of all algorithms/protocols. Following the notation in Table 8.2, we list some extra notations used in this proposed BBS-based AACKA protocol in Table 8.3.

Overview of the proposed BBS-based AACKA protocol. In this protocol, the device $\mathcal{D}_1$ wants to create the key agreement with the device $\mathcal{D}_2$ The workflow of this protocol is as follows:

- Setup$(\lambda) \rightarrow (pp, sk, pk)$: Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ denote three cyclic groups of prime order p, $h : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, a type III bilinear map. Let $g_2 \leftarrow \mathbb{G}_2$ The Privacy CA chooses a random values x as secret keys sk and computes the corresponding public key $pk = X$, where $X = g_2^x$. The public parameters are $pp = (p, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$.
- Join: This join protocol is run between the device $\mathcal{D}_1$ and the Privacy CA $\mathcal{CA}$, which has been realized during the VC issuance stage in the ZTO scheme described in subsection 6.1.1 in deliverable D4.3.
- AACKA: This protocol is run between the device $\mathcal{D}_1$ in domain A and another device $\mathcal{D}_2$ in domain B, shown in Figure 8.1.
  - ✓ If $bsn = \perp$, $\mathcal{D}_1$ selects a group generator J from $\mathbb{G}_1$; else compute $J \leftarrow H(bsn)$.
  - ✓ Then $\mathcal{D}_1$ selects two random values a, b to randomize the credential A to get A'. Further computes D, B, which are associated with A' and e.

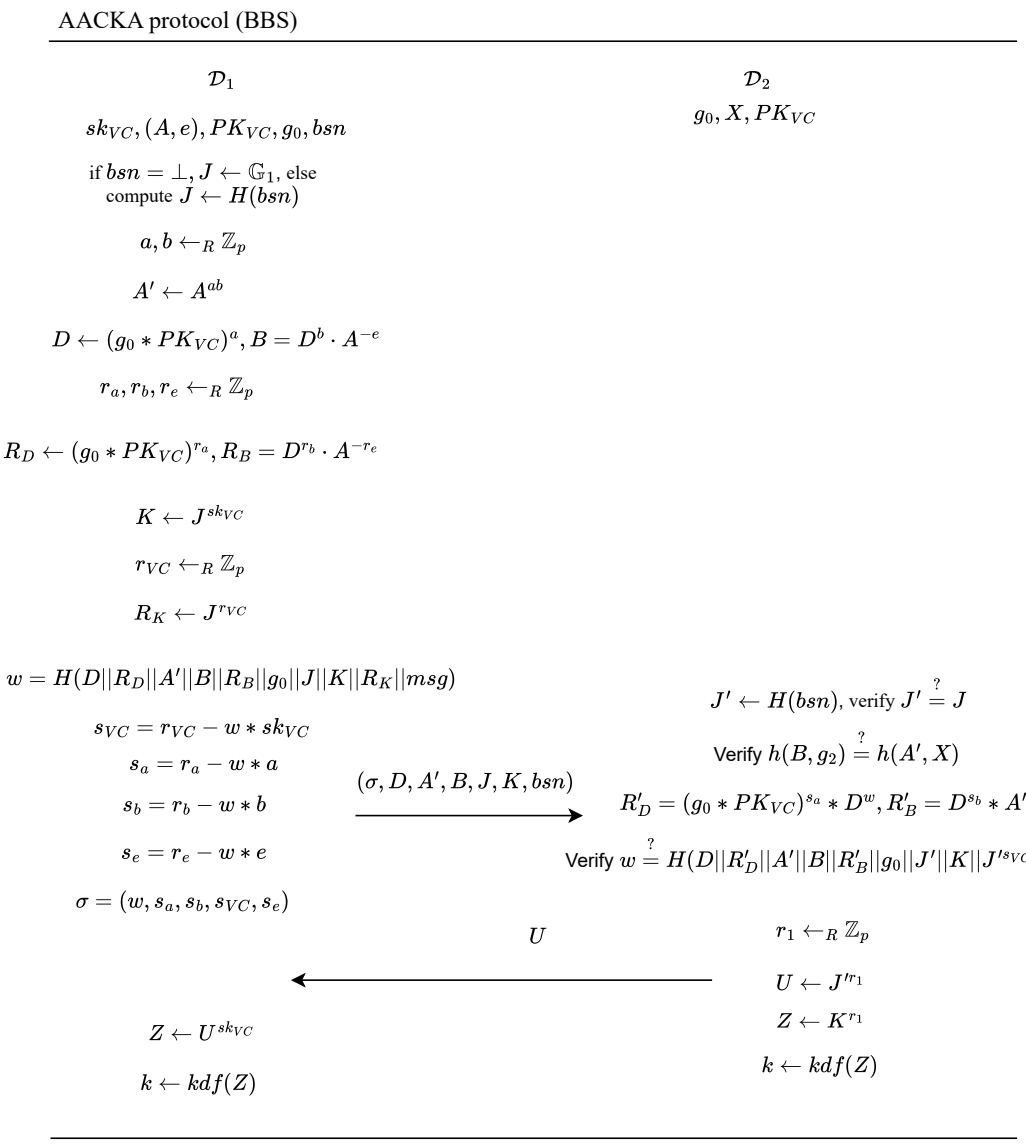


Figure 8.1: The BBS-based AACKA protocol with a single signer

Table 8.3: Notation used in BBS-based PACKA protocol

Notation	Meaning
(A, e)	BBS Credential
X	CA's public key
H	Hash function, to $\mathbb{G}_1$
a, b	Random values for randomizing the credential
D, A', B	Two values to be proved associated with the randomized credential
r_a, r_b, r_e	Random values used in Schnorr signature proof
R_D, R_B	Two commitments associated with D, B
r_{VC}	Random value used in Schnorr signature
R_K	Commitment for K
w	Challenge generated by $\mathcal{D}$ used in Schnorr signature proof
s_{VC}	Element in Schnorr signature for $\mathcal{D}$ related to f
s_a, s_b, s_e	Elements in Schnorr signature related to a, b, e
σ	Final Schnorr signature
$J^{s_{VC}} * K^w$	This is supposed to be equal to R_K
r_1	Random value
U	Commitment to J
Z	Shared ephemeral key between $\mathcal{PP}$ and $\mathcal{D}$
kdf	Key derivation function
k	key for encryption and decryption

✓ Computes the commitments R_D, R_B on D, B and link tokens (J, K), the associated commitment R_K

✓ Then generates a Schnorr signature $\sigma = (w, s_a, s_b, s_{VC}, s_e)$, The whole proof is denoted as follows:

$$\pi = \text{NIZKP}\{(g_0, PK_{VC}, J, K, D, A', B); (b, a, e, sk_{VC}) | \\ K = J^{sk_{VC}}, D = (g_0 * PK_{VC})^a, B = D^b \cdot A^{-e}\}$$

$K = J^{sk_{VC}}$ is computed by the device $\mathcal{D}_1$. The details of π are as follows:

* The $\mathcal{D}_1$ selects three random numbers $r_b, r_a, r_e \leftarrow \mathbb{Z}_p$, and computes $R_D = (g_0 * PK_{VC})^{r_a}$ and $R_B = D^{r_b} \cdot A^{-r_e}$.
* Then computes a challenge $w = H(D||R_D||A'||B||R_B||g_0||J||K||R_K||msg)$.
* $\sigma = (w, s_{VC}, s_b, s_a, s_e)$, where $s_b = r_b - w \cdot b$, $s_a = r_a - w \cdot a$ and $s_e = r_e - w \cdot e$. Then $\mathcal{D}_1$ sends $(\sigma, D, A', B, J, K, bsn)$ to $\mathcal{D}_2$.
* During verification, $\mathcal{D}_2$ executes as follows:
  - Computes $J' = H(bsn)$ and verifies $J' \stackrel{?}{=} J$ and $h(B, g_2) \stackrel{?}{=} h(A', X)$.
  - Computes $R'_D = (g_0 * PK_{VC})^{s_a} * D^w$, $R'_B = D^{s_b} * A'^{-s_e} * B^w$.
  - Verifies $w \stackrel{?}{=} H(D||R'_D||A'||B||R'_B||g_0||J'||K||J'^{s_{VC}} * K^w)||msg)$.

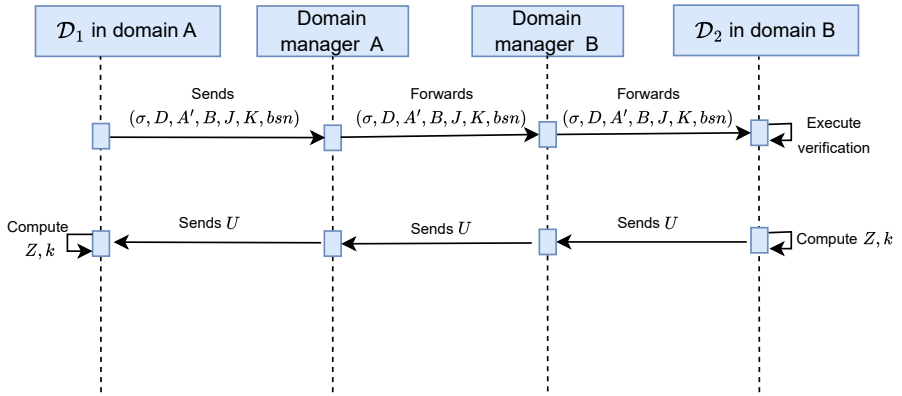


Figure 8.2: Workflow of the AACKA protocol with a single signer in REWIRE

- ✓ After the successful verification, $\mathcal{D}_2$ computes U using a random value r_1 and sends it to $\mathcal{D}_1$.
- ✓ Then the shared key Z and the encryption/decryption key k can be locally computed by $\mathcal{D}_1$ and $\mathcal{D}_2$.

Apply the AACKA protocol to REWIRE project. Here, we introduce how to apply the AACKA protocol to REWIRE project. As shown in Figure 8.2, the workflow of the AACKA protocol with a single signer in REWIRE is as follows:

1. Firstly, the device $\mathcal{D}_1$ in domain A, as a single signer, generates the proof $(\sigma, D, A', B, J, K, bsn)$ to the domain manager A, who verifies the $\mathcal{D}_1$. If the verification fails, aborts. Otherwise, go to the next step. For simplicity, we omit the verification of $\mathcal{D}_1$ performed by the domain manager A in Figure 8.2.
2. The proof $(\sigma, D, A', B, J, K, bsn)$ is forwarded via domain manager A and B to the device $\mathcal{D}_2$ in domain B.
3. After receiving the proof $(\sigma, D, A', B, J, K, bsn)$, the device $\mathcal{D}_2$ perform the verification of the proof.
4. If the verification fails, aborts. Otherwise, $\mathcal{D}_2$ sends U via domain manager A and B to $\mathcal{D}_1$. The domain B and A perform verification of $\mathcal{D}_2$, for simplicity, which is omitted in Figure 8.2.
5. $\mathcal{D}_1$ and $\mathcal{D}_2$ all computes the Z and final shared k locally. Then the key agreement is done.

8.3 Security analysis for the first AACKA protocol

In this section, we are going to give proofs of security properties mentioned in subsection 8.1. Note that because the randomized credential and the corresponding proof include a link token, which is generated by the secret key sk_{VC}. Therefore, in this BBS-based AACKA protocol, the unconditional anonymity cannot be ensured because the link token (J, K) generated by the secret key sk_{VC} may leak some information. The conditional anonymity can be ensured by the Schnorr signature. Further, the confidentiality can be achieved since only both communication parties can compute the shared secret key Z and the encryption key k. Therefore, in the following, we give security analysis on correctness, unforgeability and linkability.Before giving security analysis, some assumptions used in following proofs should be introduced. Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ be three cyclic groups of prime order p. In particular, two generators g_1 and g_2 are associated with $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively.

Definition 13. (Decisional Diffie-Hellman (DDH)). Given two elements $A = g_1^a, B = g_1^b$, where $(a, b) \in \mathbb{Z}_p^2$, and a random element $X \in \mathbb{G}_1$, an adversary $\mathcal{A}$ is able to decide whether $X = g_1^{ab}$ only with negligible probability.

Definition 14. (q-Strong Diffie-Hellman ($q - SDH$)). Given as input a $(q + 3)$-tuple of elements $(g_1, g_1^x, g_1^{x^2}, ..., g_1^{x^q}, g_2, g_2^x) \in \mathbb{G}_1^{q+1} \times \mathbb{G}_2^2$, output a pair $(c, g_1^{1/(x+c)}) \in \mathbb{Z}_p \times \mathbb{G}_1$, for $c \in \mathbb{Z}_p$ and $c \neq -x$.

An adversary $\mathcal{A}$ solves the $q - SDH$ problem in the bilinear group pair $(\mathbb{G}_1, \mathbb{G}_2)$ with advantage ε if

$$\mathrm{SDHAdv}_{q,\mathcal{A}} := \Pr\left[\mathcal{A}\left(g_1, g_1^x, \ldots, g_1^{(x^q)}, g_2, g_2^x\right) = \left(c, g_1^{\frac{1}{x+c}}\right)\right] \geq \varepsilon \tag{8.1}$$

Where the probability is over the random choices of generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$, the random choice of $x \in \mathbb{Z}_p^\times$ and the random bits consumed by $\mathcal{A}$.

Definition 15. (Computational co-Diffie-Hellman (co-CDH) on $(\mathbb{G}_1, \mathbb{G}_2)$). Given $g_2, g_2^a \in \mathbb{G}_2$ and $h \in \mathbb{G}_1$ as input, compute $h^a \in \mathbb{G}_1$.

Definition 16. (Decision co-Diffie-Hellman (co-DDH) on $(\mathbb{G}_1, \mathbb{G}_2)$). Given $g_2, g_2^a \in \mathbb{G}_2$ and $h, h^b \in \mathbb{G}_1$ as input, output yes if $a = b$ and no otherwise. When the answer is yes we say thay (g_2, g_2^a, h, h^a) is a co-Diffie-Hellman tuple.

Definition 17. (Gap co-Diffie-Hellman (co-GDH)). Two groups $(\mathbb{G}_1, \mathbb{G}_2)$ are a (τ, t, ϵ)-Gap co-Diffie-Hellman (co-GDH) group pair if they satisfy the following properties:

- The group operation on both $\mathbb{G}_1$ and $\mathbb{G}_2$ and the map ψ from $\mathbb{G}_2$ to $\mathbb{G}_1$ can be computed in time at most τ.
- The Decision co-Diffie-Hellman problem on $(\mathbb{G}_1, \mathbb{G}_2)$ can be solved in time at most τ.
- No algorithm (t, ϵ)-breals computational co-Diffie-Hellman on $(\mathbb{G}_1, \mathbb{G}_2)$.

Theorem 2. *The proposed BBS-based AACKA protocol is correct.*

Proof. Based on the BBS-based AACKA protocol, the correctness of the verifiable credential $VC = (A, e)$ and the randomized verifiable credential (A', D, B) can be ensured due to the correctness of the BBS signature, i.e., $\mathcal{SP}$ checks as follows:

- $h(B, g_2) = h(D^b * A^{-t}, g_2) = h((g_0 * PK_{VC})^{abx/(x+e)}, g_2) = h(A', X)$.
- $R_0' = D^{s_{a'}} * g^{s_{VC}} * g_0^w = R_0, R_B' = D^{s_b} * A'^{-s_e} * B^w = R_B$, then $ch' = ch, w = H(ch'||g_0||R_0'||J'||K||J'^{s_{VC}} * K^w)||msg)$.

For the generated shared key, $Z = K^{r_1} = J^{sk_{VC} * r_1} = (J^{sk_{VC}})^{sk_{VC}} = U^{sk_{VC}}$. Therefore, the correctness of the proposed BBS-based AACKA protocol can be offered. □

Theorem 3. *The proposed BBS-based AACKA protocol is unforgeable if zero-knowledge proof system (Schnorr signature) offers zero-knowledge, the BBS signature ($q-SDH$ hard problem) and BLS signature (co-GDH) are both unforgeable.*

Proof. Firstly, as confirmed, the zero-knowledge proof system, i.e., the Schnorr signature, can offer zero-knowledge. Here, we omit the detailed proof of the zero-knowledge proof system, the Schnorr signature. Moreover, it has been proved in [71] that a verified BBS credential (A', D, B) must be a randomization of a genuine BBS credential (A, e), both of which are associated with the same secret key sk_{VC}. The BBS signature offers one-more unforgeability. The proposed protocol reduces to a static-ephemeral Diffie-Hellman over some unknown base point. It does not matter that the base point is not known to either $\mathcal{D}_1$ or $\mathcal{D}_2$ because $\mathcal{D}_1$ does not need the base point and $\mathcal{D}_2$ has constructed U in a way that looks like

random group element for $\mathcal{D}_1$. This is due to the unforgeability of the BBS signature. The link token, (J, K), where $J = H(bsn)$, $K = J^{sk_{VC}}$, is a BLS signature [17]. The unforgeability of the BLS signature has been proved, which relies on the co-GDH hard problem. Specifically, the shorter BBS signature was newly proposed in 2023 [71], therefore, we introduce the proofs of unforgeability of [71].

Proof of unforgeability of the BBS signature: The unforgeability game $\text{Game}_{Unf}^{\mathcal{A}(\lambda)}$ is defined as follows:

- Setup: In the setup stage, public parameters pp is generated and sent to the adversary $\mathcal{A}$.
- CA's key generation: The Privacy CA generates his secret/public key pair (x, X) and the public key is known by the adversary $\mathcal{A}$.
- Sign query: In this stage, there is a signing oracle, which takes $\mathcal{A}$'s signing query by inputting the message **m** (**m** can act as a device's secret key sk_{VC} and attributes in our AACKA protocol) and return a BBS signature to $\mathcal{A}$.
- Forgery: $\mathcal{A}$ outputs a message-signature forgery $(\mathbf{m}^*, \sigma^*)$. If the signature passes the verification, $\mathcal{A}$ wins the game; otherwise, fails.

We define the advantage of the adversary winning the unforgeability game as $Adv^{Unf}_{\text{Setup,eG,eS}}(\mathcal{A}, \lambda)$. This unforgeability is under the algebraic group model (AGM) [39]. We introduce a theorem about the AGM security of BBS.

Theorem 4. *AGM security of BBS Let eG and eS record the state of output of Setup and the state of random value e for the BBS signature (A, e). For every algebraic adversary $\mathcal{A}$ issuing at most q signing queries, there exists adversaries $\mathcal{B}_1$ and $\mathcal{B}_2$ such that*

$$Adv^{Unf}_{\textit{Setup,eG,eS}}(\mathcal{A}, \lambda) \leqslant Adv^{q-dl}_{\textit{Setup}}(\mathcal{B}_1, \lambda) + Adv^{dl}_{\textit{Setup}}(\mathcal{B}_2, \lambda) + \delta_{\textit{eG,eS}}(q, \lambda) + \frac{1}{p(\lambda)} \tag{8.2}$$

Where "dl" means the DL hard problem. The adversaries $\mathcal{B}_1$ and $\mathcal{B}_2$ are given explicitly in the proof, and have running times comparable to that of $\mathcal{A}$. The adversary $\mathcal{B}_1$ needs to additionally factor a polynomial of degree (at most) q. The requirement is that $\delta_{\text{eG},textsfeS}(q, \lambda)$ is small. Let $e_1, .., e_q \in \mathbb{Z}_p$ be distinct, let $\boldsymbol{y}\mathbb{Z}_p^l$, and let $\mathbf{m}_1, ..., \mathbf{m}_q \in \mathbb{Z}_p^l$. Further, let $(\boldsymbol{m}^*, e^*) \notin \{(\boldsymbol{m}_i, e_i)\}_{i\in[q]}$. Then, assume that there exist $\lambda_1, \ldots, \lambda_q, \gamma \in \mathbb{Z}_p$ such that

$$\varphi^{\boldsymbol{y}}_{\boldsymbol{m}^*,\boldsymbol{e}^*}(\mathrm{X}) = \sum_{i=1}^{q} \lambda_i \cdot \varphi^{\boldsymbol{y}}_{\boldsymbol{m}_i,e_i}(\mathrm{X}) + \gamma. \tag{8.3}$$

Where $\varphi^{\boldsymbol{y}}_{\boldsymbol{m}^*,e^*}(\mathrm{X}) = \frac{1+\langle \boldsymbol{y},\boldsymbol{m}^*\rangle}{\mathrm{X}+e^*}$, $\langle \boldsymbol{x}, \boldsymbol{y}\rangle$ denotes inner product in $\mathbb{Z}_p$. Then, one of the following two conditions must be true:
(i) There exists $i \in [q]$ such that $e^8 = e_i$ and $1 - \lambda_i + \langle \boldsymbol{y}, \boldsymbol{m}^* - \lambda_i \cdot \boldsymbol{m}_i\rangle = 0$.
(ii) We have $e^* \notin \{e_1, \ldots, e_q\}$, but $1 + \langle \boldsymbol{y}, \boldsymbol{m}^*\rangle = 0$.
To verify (i), assume that $e^* \in \{e_1, ..., e_q\}$, and w.l.o.g., let $e^* = e_1$. We multiply both sides of equation 8.3 by $p(x) = \prod_{i=1}^{l}(X + e_i)$, and we can get

$$(1 - \lambda_i + < \boldsymbol{y}, \mathbf{m}^* - \lambda_1 \cdot \mathbf{m}_1 >) \cdot p_1(X) = \gamma \cdot p(X) + \sum_{i=2}^{q} \lambda_i \cdot \varphi^{\boldsymbol{y}}_{\boldsymbol{m}_i,e_i}(\mathrm{X}) \cdot p_i(X) \tag{8.4}$$

Let us consider instead the case $e^* \notin \{e_1, ..., e_q\}$. For notational convenience, we let $e_{q+1} = e^*$, $p'(X) = \prod_{i\in[q+1]}(X + e_i)$, and $p'_k(X) = \prod_{i\in[q+1]\setminus\{k\}}(X + e_i)$. Then multiplying both sides of equation 8.3 by $p'(X)$, we can get

$$(1 + < \boldsymbol{y}, \mathbf{m}^* >) \cdot p'_{q+1}(X) = \gamma \cdot p'(X) + \sum_{i=2}^{q} \lambda_i \cdot \varphi^{\boldsymbol{y}}_{\boldsymbol{m}_i,e_i}(\mathrm{X}) \cdot p'_i(X) \tag{8.5}$$

We notice that if $(1 + < \boldsymbol{y}, \mathbf{m}^* >) \neq 0$ the LHS is non-zero, and not divisible by $X + e_{q+1}$, as $p'_{q+1}(e_{q+1}) \neq 0$. In contrast, the RHS is always divisible by $X + e_{q+1}$. A contradiction.

Overview of the reduction. Let $\mathcal{A}$ be an algebraic adversary in the unforgeability game. For each signing query, except the public parameters, the adversary can also gets $A_i\mathbb{G}_1$. Finally, when producing the forgery $(\mathbf{m}^*, (A^*, e^*))$, by virtue of being algebraic, $\mathcal{A}$ also provides a representation $(\gamma_0, ..., \gamma_l, \lambda_1, ..., \lambda_q) \in \mathbb{Z}_p^{q+l+1}$ of A^* such that

$$A^* = g_1^{\gamma_0} \prod_{i=1}^{l} h_1[i]^{\gamma_i} \prod_{i=1}^{q} A_i^{\lambda_i} = (C^*)^{1/(x+e^*)} = g_1^{\varphi^{\boldsymbol{y}}_{\mathbf{m}^*,e^*}(x)} \tag{8.6}$$

where $\boldsymbol{y}[i] = DL_{g_1}(h_1[i])$ for all $i \in [l]$. Further, we have $A_i = g_1^{\varphi^{\boldsymbol{y}}_{\mathbf{m}^*,e^*}(x)}$. Therefore, setting $\gamma = \gamma_0 + \sum_{i\in[l]} \gamma_i \cdot \boldsymbol{y}[i]$, this implies that

$$\gamma + \sum_{i=1}^{q} \lambda_i \cdot \varphi^{\boldsymbol{y}}_{\mathbf{m}_i,e_i}(x) - \varphi^{\boldsymbol{y}}_{\mathbf{m}^*,e^*}(x) = 0 \tag{8.7}$$

Then assume that two conditions (i) and (ii) do not hold, and $e_1, ..., e_q$ are distinct. Moreover, we can get x is zero.
To formalize the above analysis, we consider three cases as follows:

- Forge: This is the event that $\mathcal{A}$ outputs a successful forgery and wins the game;
- Rel: This is the event that the forgery is for a message $\mathbf{m}^*$ such that either (i) or (ii) holds;
- Is the event that there exit distinct $i, j \in [q]$ with $e_i = e_j$.

Then we can get

$$\begin{aligned} Adv^{unf}_{\mathsf{Setup,eG,eS}}(\mathcal{A}, \lambda) &= Pr[Forge \wedge \bar{Rel}] + Pr[Forge \wedge Rel] \\ &\leq Pr[Forge \wedge \bar{Rel} \wedge Col] + Pr[Forge \wedge \bar{Rel} \wedge \bar{Col}] + Pr[Rel] \\ &\leq Pr[Col] + Pr[Forge \wedge \bar{Rel} \wedge \bar{Col}] + Pr[Rel] \end{aligned} \tag{8.8}$$

$Pr[Col] = \delta_{\mathsf{eG,eS}}(q, \lambda)$. We give $\mathcal{B}_1$ and $\mathcal{B}_2$ such that

$$Pr[Forge \wedge \bar{Rel} \wedge \bar{Col}] \leq Adv^{q-dl}_{\mathsf{Setup}}(\mathcal{B}_1), Pr[Rel] \leq Adv^{dl}_{\mathsf{Setup}}(\mathcal{B}_2) + 1/p \tag{8.9}$$

To simplify, for the q-DL adversary $\mathcal{B}_1$, it can be deferred that the adversary succeeds with probability at least $Pr[Forge \wedge \bar{Rel} \wedge \bar{Col}]$.
For the adversary $\mathcal{B}_2$, the probability can be reduced to $1/p$.
In summary, this concludes the proof. □

Theorem 5. *The proposed BBS-based AACKA protocol can offer linkability in the random oracle model and if the DDH problem holds.*

Proof. In the BBS-based AACKA protocol, different randomized BBS credentials using different link tokens cannot be linked, i.e., if a basename bsn is changed, this results in different J and K. □

8.4 The second AACKA protocol with two separate signers

Except that the device $\mathcal{D}_1$ acts as the single signer, if the device has limited computation power, the delegation of generating the proof to other entities is necessary. In REWIRE, the domain manager is fully trusted by all devices in the corresponding domain that we can delegate most of the computation during generating the proof to the domain manager. In this case, we designed a new AACKA protocol with a device acting as the principle signer and the domain manager acting as the helper signer.

Overview of the proposed BBS-based AACKA protocol. We follow the notation in Table 8.2 and Table 8.3. In this protocol, the device $\mathcal{D}_1$ in domain A acts as the principle signer, holding the secret key sk_{VC}, to generate a proof of the secret key ownership. The associate domain manager A acts as the helper signer, holding the BBS credential, to generate the randomized credential and link token. Then combines them with the proof of the secret key ownership to generate a whole proof, which can be sent to the target device $\mathcal{D}_2$. We omit the domain manager B for the device $\mathcal{D}_2$ in the protocol temporarily because it only forwards information during communications. In the end, we will involve the domain manager B during discussing how to apply this AACKA protocol into REWIRE. The whole protocol are as follows:

- Setup$(\lambda) \rightarrow (pp, sk, pk)$: Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ denote three cyclic groups of prime order p, $h : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, a type III bilinear map. Let $g_2 \leftarrow \mathbb{G}_2$ The Privacy CA chooses two random values x as secret keys sk and computes the corresponding public key $pk = X$, where $X = g_2^x$. The public parameters are $pp = (p, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$.
- Join: This join protocol is run between the device $\mathcal{D}_1$ and the Privacy CA $\mathcal{CA}$, which has been realized during the VC issuance stage in the ZTO scheme described in subsection 6.1.1 in deliverable D4.3.
- AACKA: This protocol is run among the device $\mathcal{D}_1$, the associated domain manager A and the device $\mathcal{D}_2$ in domain B, which is shown in Figure 8.3. Here we omit the domain manager B temporarily, because B just forwards information during the communications among other three entities.
  - ✓ If $bsn = \perp$, the domain manager A selects a group generator $J \leftarrow \mathbb{G}_1$; else computes $J \leftarrow H(bsn)$.
  - ✓ Then A selects two random values a, b to randomize the credential A to get A'. Further computes D, B, which are associated with A' and e.
  - ✓ A selects three random values r_a, r_b, r_e and computes R_D, R_B on D, B. Using the tuple (D, A', B, R_D, R_B), computes a challenge $ch = H(D||A'||B||R_D||R_B)$ and sends (msg, J, ch) to $\mathcal{D}_1$.
  - ✓ After receiving (msg, J, ch), $\mathcal{D}_1$ computes $K = J^{sk_{VC}}$. The to generate the proof of the secret key and the link token, selects $r_{sk_{VC}}$ randomly associated with sk_{VC} and computes $R_K = J^{r_{sk_{VC}}}$. Then generate a Schnorr signature by computing $w = H(ch||J||K||R_K||msg)$ and $s_{sk_{VC}} = r_{sk_{VC}-w*sk_{VC}}$.
  - ✓ $\mathcal{D}_1$ sends $(w, s_{sk_{VC}}, K)$ to the domain manager A.
  - ✓ After receiving $(w, s_{sk_{VC}}, K)$, A computes $s_{a'}, s_b s_e$ and forms the combined proof $\pi = (w, s_{a'}, s_b s_e, s_{sk_{VC}}$ Finally, A sends (π, D, A', B, bsn) to $\mathcal{D}_2$.
  - ✓ After receiving the tuple (π, D, A', B, bsn), $\mathcal{D}_2$ verifies the proof by checking the following equations:
    - $J' \leftarrow H(bsn)$, verify $J' \stackrel{?}{=} J$;
    - Verify $h(B, g_2) \stackrel{?}{=} h(A', X$;
    - $ch' = H(D||A'||B||D^{s_b} * A'^{-s_e} * B^w)$;
    - Verify $w \stackrel{?}{=} H(ch'||J||K||J^{s_{sk_{VC}}} * K^w)||msg)$
  - ✓ The verification fails, aborts; otherwise, $\mathcal{D}_2$ selects a random value r_1 and computes U. Then sends U to the domain manager A, who forwards it to $\mathcal{D}_1$.
  - ✓ Finally, $\mathcal{D}_1$ and $\mathcal{D}_2$ can locally computes Z, k.

Apply the AACKA protocol to REWIRE project. Here, we introduce how to apply the AACKA protocol to REWIRE project. As shown in Figure 8.4, the workflow of the AACKA protocol with two signers in REWIRE is as follows:

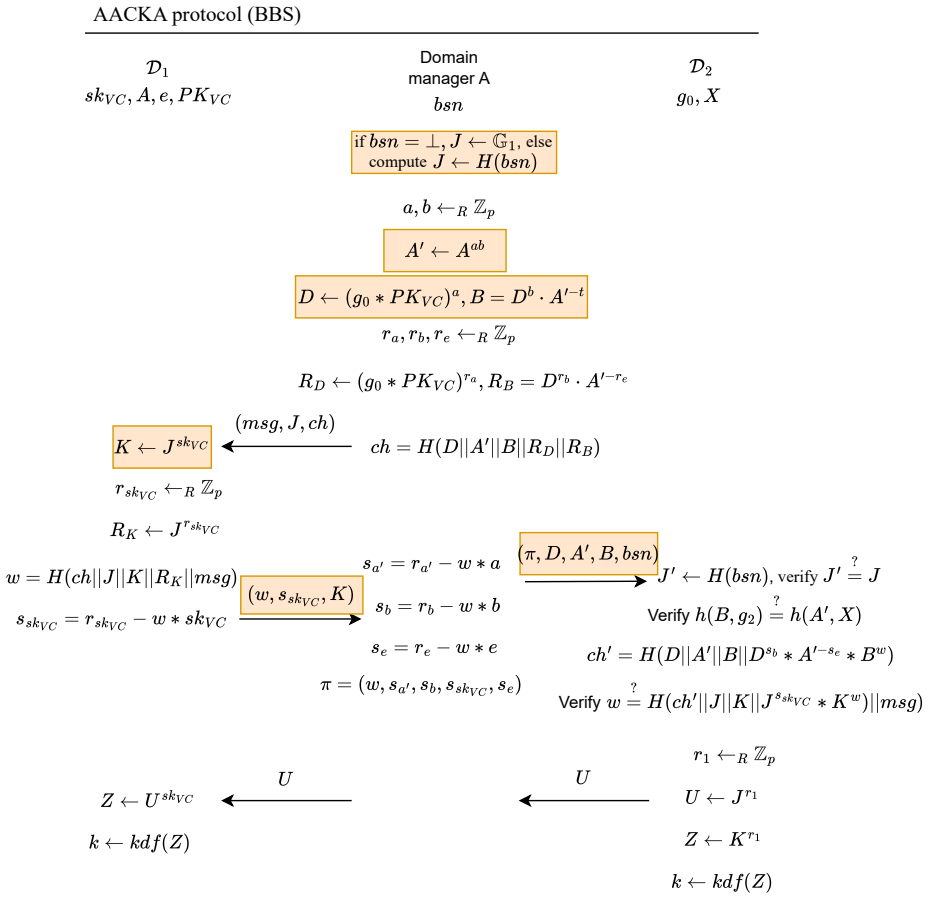


Figure 8.3: The BBS-based AACKA protocol with two signers

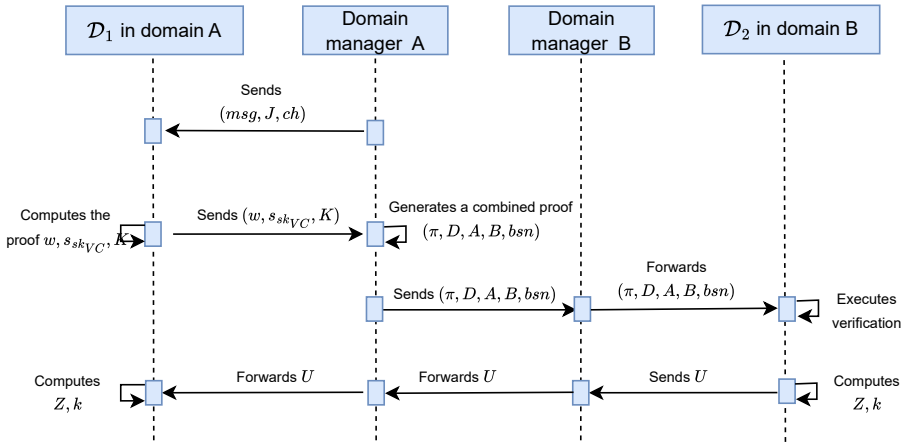


Figure 8.4: Workflow of the AACKA protocol with two signers in REWIRE

1. The domain manager A sends (msg, J, ch), in which J is associated with the basename bsn and ch is challenge, to the device $\mathcal{D}_1$.
2. After receiving the tuple (msg, J, ch), $\mathcal{D}_1$ computes a proof of the secret key and the link token, i.e. $(w, s_{sk_{VC}}, K)$, which is sent to the domain manager A.
3. A can computes a combined (credential and secret key) proof (π, D, A, B, bsn), which is sent to the domain manager B. Then B forwards the tuple (π, D, A, B, bsn) to $\mathcal{D}_2$ in the domain B.
4. $\mathcal{D}_2$ executes the verification of the proof. If the verification fails, aborts; otherwise sends U to B. In this process, B and A just forwards U to $\mathcal{D}_1$.
5. Finally, $\mathcal{D}_1$ and $\mathcal{D}_2$ computes Z, k locally.

Discussion about the security analysis. In this AACKA protocol, two signers, one principle signer $\mathcal{D}_1$, and the helper signer A, are used. But in REWIRE project, the domain managers A and B are fully trusted. When we discuss about the security of this AACKA protocol with two signers, we can regard them as one single signer. Therefore, the security model can be the same as that in the AACKA protocol with one single signer. Further, this AACKA protocol can also offer the same properties, unforgeability, confidentiality, anonymity, linkability. Because we can regard two signers as a single one, the security proof are also the same. Here, we omit it.

8.5 The running time for randomization of credential

Having described the two AACKA protocols, we can find a common phase of the randomization behind the credential (BBS and CL signatures). In what follows, we give th detailed performance footprint for both cases.

For our experimental setup, we implemented the AACKA schemes in C, leveraging the mbedTLS cryptographic library atop the Keystone framework—a trusted execution environment (TEE) for RISC-V platforms. All evaluations were conducted on a StarFive VisionFive 2 board, equipped with a quad-core 1.5 GHz CPU and 4 GB of RAM. The protocol was executed 1,000 times to ensure consistency in the performance measurements. We implemented the scheme with the 10 attributes and tested the running time of signing process over a commitment, the verification, key establishment in BBS and CL based schemes, respectively. The results are shown in Tables **??** and **??**, respectively. From these two tables, we can see

that the BBS-based version outperforms the CL-based version, especially for the signing over a commitment. The signature generation over a commitment requires $213ms$ (when BBS-type of signatures are leveraged) against $704, s$ required for the case of CL-based credentials (when 5 attributes are disclosed). In the same context, verification takes around $234ms$ (BBS-based version) and $417ms$ (CL-baed version) whereas the key establishment takes $23ms$ and $46ms$, respectively.

Chapter 9

Conclusions

Throughout this deliverable, we provided a detailed description and documentation of the final versions of the **artefacts**, **schemes**, and **methods participating in the design-time phase of the REWIRE framework**, and their role towards the achievement of the overall goal of REWIRE regarding the secure lifecycle management of embedded systems through a set of security and trust enablers (considering the increasing adoption of OpenHW architectures, such as RISC-V). Recall that the core output of the design-time phase is the definition of the **trust boundary** of the system, i.e., the device properties for which the system administrator can have the required trustworthiness guarantees, and which need to be verified during runtime through the available security enablers.

In Chapter 2, we focused on the **Secure SW Update through Authenticated Encryption (AE)** process of REWIRE, which aims to ensure the integrity of the SW or FW running on IoT devices, and includes both **1-to-1** and **1-to-many** modes, depending on the number of devices the update needs to be deployed to. One core consideration in the design of this component was protection against **Side-Channel Attacks (SCA)**, which is a core concern in practical systems and will become increasingly prevalent with the advent of quantum cryptography. To this end, we have developed the **LR-BC-2** scheme for providing resilience against key leakage, and we leveraged **ASCON** for providing a solution tailored to devices requiring with limited computational capabilities. We also provided detailed evaluation and benchmarking results in order to demonstrate the efficiency of the proposed scheme.

Next, in Chapter 3, we focused on the **Compositional Verification and Validation** component of REWIRE, and specifically the Formal Verification of the CIV protocol leveraging the **VerifPal** tool. Note that *the Formal Verification of local (implicit) attestation, which does not require the transmission of sensitive information and traces to the Verifier, is a core innovation of REWIRE*. To this end, we followed a layered approach that entailed the evaluation of increasingly difficult scenarios (i.e., attackers with increasingly strong capabilities). Through this approach, we identified a counterexample corresponding to a potential exploitation path by an attack, and we demonstrated how this can greatly assist towards both the **concrete and accurate definition of all assumptions and requirements**, as well as the **addressing of any potential vulnerabilities in the evaluated scheme**.

Chapter 4 was dedicated to the **SW/FW Vulnerability Analysis** component of REWIRE, which enables the partially automated analysis of embedded software, as well as software patches to be deployed, through **partial emulation of the firmware image and fuzzing of the emulation**. The fuzzing process entails the generation of semi-random inputs and their adaptation to the software of the device, and the observation of the device behaviour when these inputs are provided. Any potentially unexpected behaviour (e.g., unintended functionality or software crashes) indicate the need for further analysis. We also provided detailed benchmarking results, in order to evaluate both the performance, and the effectiveness of the component.

A core consideration towards the fulfilment of the target of REWIRE regarding the secure lifecycle management of embedded devices is the need to **trace the achievement of the trust requirements characterising the technical components of REWIRE**, which in turn provide the basis for the fulfilment of the

overarching security goals and requirements of the REWIRE use cases. To this end, the **Architecture Analysis & Design Language (AADL)** component, detailed in Chapter 5, provides a methodology for specifying both software and hardware configurations, and building a representation of the system to be validated. Then, using the **RESOLUTE** extension of the **Open-Source AADL Tool Environment (OSATE)**, we develop assurance cases to break down the requirements of the REWIRE use cases, and we apply the developed methodology to the **Automotive** and **Smart Cities** use case.

In order to express the security policies provided as output of the design-time phase of REWIRE, the **Medium Security Policy Language (MSPL)** has been selected, as it possesses the required level of **granularity** for expressing the types of attributes to be attested during runtime, as well the **periodicity of the tasks to be performed**. In Chapter 6, we provided example implementations of such policies in the context of all use cases, specifically (i) a **one-to-many software update** and a **live migration policy** in the **Smart Cities** use case, (ii) a **one-to-one software update** in the **Automotive** use case, and (iii) **CIV and CFA attestation policies** in the Smart Satellites use case.

Chapter 7 is dedicated to the **Risk Assessment** component, whose main target is the calculation of the **Required Trust Level (RTL)** of devices, which is essentially the baseline level of trust above which a device can be considered trustworthy, and can guide the runtime trust evaluation of devices. In this deliverable, we provided a detailed technical description of the Risk Assessment component and the underlying Functional Specifications. We also provide details on the instantiation of the Risk Assessment framework in the automotive use case, by leveraging and integrating the **TARA framework**, which is tailored to the automotive domain. We detail how we transition from TARA to the calculation of the RTL, and the reinforcement of RTL calculation through abstracting the underlying Risk Quantification engine.

Finally, Chapter 8 is dedicated to the methodology leveraged by REWIRE in order to establish secure and authenticated communication, not only between devices within the same domain, but also **devices belonging to different domains**, which is a core consideration of the authentication and authorization services of REWIRE. The proposed methodology leverages the **Anonymous Authenticated Credential Key Agreement (AACKA)** scheme, which can be seen as an anonymous authenticated credential key agreement scheme by combining **Camenisch-Lysyanskaya (CL)** credentials with **Elliptic Curve Diffie-Hellman (EDCH)** key agreement. We also provide comparisons between CL and BBS signatures, and we design a BBS-based AACKA protocol to be applied in the REWIRE framework.

Overall, this deliverable provided a detailed description and evaluation of the final version of all technical components participating in the design-time phase of REWIRE. These will provide the basis of their integration into all REWIRE use cases, and their evaluation in the context of those use cases as part of D6.2.

Bibliography

[1] ISO/SAE 21434:2021. Road vehicles Cybersecurity engineering. *ISO/TC 22/SC 32 Technical Standard*, 2021. https://www.iso.org/standard/70918.html.

[2] Tomer Ashur, Orr Dunkelman, and Atul Luykx. Boosting authenticated encryption robustness with minimal modifications. In *CRYPTO (3)*, volume 10403 of *Lecture Notes in Computer Science*, pages 3–33. Springer, 2017.

[3] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber algorithm specifications and supporting documentation. *NIST PQC Round*, 2(4):1–43, 2019.

[4] Melissa Azouaoui, Davide Bellizia, Ileana Buhan, Nicolas Debande, Sébastien Duval, Christophe Giraud, Éliane Jaulmes, François Koeune, Elisabeth Oswald, François-Xavier Standaert, and Carolyn Whitnall. A systematic appraisal of side channel evaluation strategies. In *SSR*, volume 12529 of *Lecture Notes in Computer Science*, pages 46–66. Springer, 2020.

[5] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Tobias Schneider, Markus Schönauer, François-Xavier Standaert, and Christine van Vredendaal. Protecting dilithium against leakage revisited sensitivity analysis and improved implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(4):58–79, 2023.

[6] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François -Xavier Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In *ESORICS (1)*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

[7] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre- Yves Strub. Verified proofs of higher-order masking. In *EUROCRYPT (1)*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485. Springer, 2015.

[8] Julien Béguinot, Wei Cheng, Sylvain Guilley, Yi Liu, Loïc Masure, Olivier Rioul, and François-Xavier Standaert. Removing the field size loss from duc et al.'s conjectured bound for masked encodings. In *COSADE*, volume 13979 of *Lecture Notes in Computer Science*, pages 86–104. Springer, 2023.

[9] Sonia Belaïd, Fabrice Benhamouda, Alain Passelè gue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 616–648. Springer, 2016.

[10] Sonia Belaïd, Vincent Grosso, and François-Xavier Standaert. Masking and leakage-resilient primitives: One, the other(s) or both? *Cryptogr. Commun.*, 7(1):163–184, 2015.

[11] Sonia Belaïd, Fabrizio De Santis, Johann Heyszl, Stefan Mangard, Marcel Medwed, Jörn-Marc Schmidt, François-Xavier Standaert, and Stefan Tillich. Towards fresh re-keying with leakage-resilient prfs: cipher design principles and analysis. *J. Cryptogr. Eng.*, 4(3):157–171, 2014.

[12] Davide Bellizia, Olivier Bronchain, Gaëtan Cassiers, Vincent Grosso, Chun Guo, Charles Momin, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. Mode-level vs. implementation-level physical security in symmetric cryptography - A practical guide through the leakage-resistance jungle. In *CRYPTO (1)*, volume 12170 of *Lecture Notes in Computer Science*, pages 369–400. Springer, 2020.

[13] Davide Bellizia, Balazs Udvarhelyi, and François-Xavier Standaert. Towards a better understanding of side-channel analysis measurements setups. In *CARDIS*, volume 13173 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2021.

[14] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. Gimli : A cross-platform permutation. In *CHES*, volume 10529 of *Lecture Notes in Computer Science*, pages 299–320. Springer, 2017.

[15] Francesco Berti, Olivier Pereira, Thomas Peters, and Franç ois-Xavier Standaert. On leakage-resilient authenticated encryption with decryption leakages. *IACR Trans. Symmetric Cryptol.*, 2017(3):271–293, 2017.

[16] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina K önighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In *EUROCRYPT (2)*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, 2018.

[17] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[18] Olivier Bronchain, Charles Momin, Thomas Peters, and Franç ois-Xavier Standaert. Improved leakage-resistant authenticated encryption based on hardware AES coprocessors. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):641–676, 2021.

[19] Olivier Bronchain, Charles Momin, Thomas Peters, and François-Xavier Standaert. Improved leakage-resistant authenticated encryption based on hardware aes coprocessors. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):641–676, Jul. 2021.

[20] CAR 2 CAR Communication Consortium. C2C-CC Basic System Profile, 2016. Accessed: 2025-07-10.

[21] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Trans. Inf. Forensics Secur.*, 15:2542–2555, 2020.

[22] Gaëtan Cassiers and François-Xavier Standaert. Provably secure hardware masking in the transition- and glitch-robust probing model: Better safe than sorry. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):136–158, 2021.

[23] Gaëtan Cassiers and Olivier Bronchain. Scalib: A side-channel analysis library. *Journal of Open Source Software*, 8(86):5196, 2023.

[24] Gaëtan Cassiers, Barbara Gigerl, Stefan Mangard, Charles Momin, and Rishub Nagpal. Compress: Generate small and fast masked pipelined circuits. Cryptology ePrint Archive, Paper 2023/1600, 2023. https://eprint.iacr.org/2023/1600.

[25] Gaëtan Cassiers, François-Xavier Standaert, and Corentin Verhamme. Low-latency masked gadgets robust against physical defaults with application to ascon. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):603–633, Jul. 2024.

[26] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[27] Marios O. Choudary and Markus G. Kuhn. Efficient, portable template attacks. *IEEE Trans. Inf. Forensics Secur.*, 13(2):490–501, 2018.

[28] Joan Daemen and Vincent Rijmen. *The Design of Rijndael - The Advanced Encryption Standard (AES), Second Edition*. Information Security and Cryptography. Springer, 2020.

[29] Eloi de Chérisey, Sylvain Guilley, Olivier Rioul, and Pablo Piantanida. Best information is most successful mutual information and success rate in side-channel analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):49–79, 2019.

[30] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. Isap v2.0. *IACR Trans. Symmetric Cryptol.*, 2020(S1):390–416, 2020.

[31] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *J. Cryptol.*, 34(3):33, 2021.

[32] Christoph Dobraunig and Bart Mennink. Leakage resilient value comparison with application to message authentication. In *EUROCRYPT (2)*, volume 12697 of *Lecture Notes in Computer Science*, pages 377–407. Springer, 2021.

[33] Yevgeniy Dodis and Krzysztof Pietrzak. Leakage-resilient pseudorandom functions and side-channel attacks on feistel networks. In *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2010.

[34] Alexandre Duc, Sebastian Faust, and François-Xavier Standaert. Making masking security proofs concrete - or how to evaluate the security of any leaking device. In *EUROCRYPT (1)*, volume 9056 of *Lecture Notes in Computer Science*, pages 401–429. Springer, 2015.

[35] Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehl é. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 238–268, 2018.

[36] ETSI EN 302 890-2 V2.1.1 (2020-10). ETSI EN 302 890-2 V2.1.1 (2020-10) Intelligent Transport Systems (ITS); Facilities Layer function; Part 2: Position and Time management (PoTi); Release 2, 2020. Accessed: 2025-07-10.

[37] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[38] Sebastian Faust, Krzysztof Pietrzak, and Joachim Schipper. Practical leakage-resilient symmetric cryptography. In *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 213–232. Springer, 2012.

[39] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In *CRYPTO*, pages 33–62, 2018.

[40] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *FOCS*, pages 464–479. IEEE Computer Society, 1984.

[41] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for sidechannel resistance validation. *NIST non-invasive attack testing workshop*, 2011.

[42] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In *CHES*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.

[43] Hannes Groß, Rinat Iusupov, and Roderick Bloem. Generic low-latency masking in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):1–21, 2018.

[44] Chun Guo, Olivier Pereira, Thomas Peters, and François- Xavier Standaert. Authenticated encryption with nonce misuse and physical leakage: Definitions, separation results and first construction - (extended abstract). In *LATINCRYPT*, volume 11774 of *Lecture Notes in Computer Science*, pages 150–172. Springer, 2019.

[45] Chun Guo, Olivier Pereira, Thomas Peters, and François- Xavier Standaert. Towards low-energy leakage-resistant authenticated encryption from the duplex sponge construction. *IACR Trans. Symmetric Cryptol.*, 2020(1):6–42, 2020.

[46] Hari Parmar. Introduction to MACsec in the Automotive E/E Architecture, 2021. Accessed: 2025-07-10.

[47] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[48] Akira Ito, Rei Ueno, and Naofumi Homma. On the success rate of side-channel attacks on masked implementations: Information-theoretical bounds and their practical usage. In *CCS*, pages 1521–1535. ACM, 2022.

[49] Marc Joye and Michael Tunstall, editors. *Fault Analysis in Cryptography*. Information Security and Cryptography. Springer, 2012.

[50] Keystone Secure Enclave. Keystone Security Monitor, 2021. Accessed: 2025-07-10.

[51] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In *ASIACRYPT (1)*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.

[52] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[53] Thorben Moos, Amir Moradi, Tobias Schneider, and François -Xavier Standaert. Glitch-resistant masking revisited or why proofs in the robust probing model are needed. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):256–292, 2019.

[54] Nicolai Müller and Amir Moradi. PROLEAD A probing-based hardware leakage detection tool. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):311–348, 2022.

[55] nicolai müller, david knichel, pascal sasdrich, and amir moradi. transitional leakage in theory and practice unveiling security flaws in masked circuits. *iacr trans. cryptogr. hardw. embed. syst.*, 2022(2):266–288, 2022.

[56] Stjepan Picek, Guilherme Perin, Luca Mariot, Lichao Wu, and Lejla Batina. Sok: Deep learning-based physical side-channel analysis. *ACM Comput. Surv.*, 55(11):227:1–227:35, 2023.

[57] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2013.

[58] Red Hat. Drools: Business rule management system, 2025. Accessed: 2025-05-30.

[59] Mathieu Renauld, François-Xavier Standaert, Nicolas Veyrat-Charvillon, Dina Kamel, and Denis Flandre. A formal study of power variability issues and side-channel attacks for nanoscale devices. In *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2011.

[60] REWIRE. Rewire design time secure operational framework - initial version. Deliverable D3.2, The REWIRE Consortium, 18 2024.

[61] REWIRE. D4.3 rewire runtime assurance framework - final version. Deliverable D4.3, The REWIRE Consortium, 30 2025.

[62] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-order masking and shuffling for software implementations of block ciphers. In *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 171–188. Springer, 2009.

[63] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 373–390. Springer, 2006.

[64] Raphael Schermann, Rainer Urian, Ronald Toegl, Holger Bock, and Christian Steger. Enabling anonymous authenticated encryption with a novel anonymous authenticated credential key agreement (aacka). In *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 646–655. IEEE, 2022.

[65] Tobias Schneider and Amir Moradi. Leakage assessment methodology - extended version. *J. Cryptogr. Eng.*, 6(2):85–99, 2016.

[66] Amol H. Shinde and A. J. Umbarkar. Analysis of cryptographic protocols aki, arpki and opt using proverif and avispa. *International Journal of Computer Network and Information Security*, 8:34–40, 2016.

[67] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with GDB. *Free Software Foundation*, 675, 1988.

[68] François-Xavier Standaert. *Side-Channel Analysis and Leakage-Resistance*. Version 1.2, September 2024.

[69] François-Xavier Standaert, Olivier Pereira, Yu Yu, Jean-Jacques Quisquater, Moti Yung, and Elisabeth Oswald. Leakage resilient cryptography in practice. In *Towards Hardware-Intrinsic Security*, Information Security and Cryptography, pages 99–134. Springer, 2010.

[70] Stefano De Luca. Cryptographic security: Critical to europe's digital sovereignty, 2024. Accessed: 2025-6-18.

[71] Stefano Tessaro and Chenzhi Zhu. Revisiting bbs signatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 691–721. Springer, 2023.

[72] Trusted Computing Group. TCG guidance for secure update of software and firmware on embedded systems, 2020. Accessed: 2025-06-05.

[73] Balazs Udvarhelyi, Olivier Bronchain, and François-Xavier Standaert. Security analysis of deterministic re-keying with masking and shuffling: Application to ISAP. In *COSADE*, volume 12910 of *Lecture Notes in Computer Science*, pages 168–183. Springer, 2021.

[74] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: A generic power/em analysis on post-quantum kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):296–322, 2022.

[75] C. Verhamme, C. Momin, and F.-X. Standaert. Primitive-Level vs. Implementation-Level DPA Security: a Certified Case Study. *iacr trans. cryptogr. hardw. embed. syst.*, 2025, 2025.

[76] Corentin Verhamme, Gaëtan Cassiers, and François- Xavier Standaert. Analyzing the leakage resistance of the nist's lightweight crypto competition's finalists. In *CARDIS*, volume 13820 of *Lecture Notes in Computer Science*, pages 290–308. Springer, 2022.

[77] Andrew Waterman and Krste Asanovic. The RISC-V instruction set manual, volume I: User-level ISA, Document Version 20191213. *RISC-V Foundation*, December 2019.

[78] Carolyn Whitnall and Elisabeth Oswald. Robust profiling for dpa-style attacks. In *CHES*, volume 9293 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2015.

[79] Yu Yu and François-Xavier Standaert. Practical leakage-resilient pseudorandom objects with minimum public randomness. In *CT-RSA*, volume 7779 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 2013.