DELIVERABLE D1.2

ASTRID ARCHITECTURE

Grant Agreement number:	786922
Project acronym:	ASTRID
Project title:	AddreSsing ThReats for virtualIseD services
Start date of the project:	01/05/2018
Duration of the project:	36 months
Type of Action:	Research & Innovation Action (RIA)
Project Coordinator:	Name: Orazio Toscano Phone: +39 010 600 2223 E-mail: orazio.toscano@ericsson.com

Due Date of Delivery:	M10 (28/02/2019)
Actual Date of Delivery:	22/05/2019
Work Package:	WP1 – Reference Architecture
Type of the Deliverable:	R
Dissemination level:	PU
Editors:	CNIT, POLITO
Version:	1.0

List of Authors	
CNIT	CONSORZIO NAZIONALE INTERUNIVERSITARIO PER LE TELECOMUNICAZIONI
Alessandro Carrega, Matteo Repetto	
POLITO	POLITECNICO DI TORINO
Fulvio Valenza, Fulvio Risso	
ETI	ERICSSON TELECOMUNICAZIONI
Orazio Toscano	
DTU	DANMARKS TEKNISKE UNIVERSITET
Athanasios Giannetsos	

Disclaimer

Copyright

Table of Contents

List of Acronyms

Acronym	Definition
API	Application Program Interface
CAPEX	Capital Expenditure
CERT	Computer Emergency Response Team
CLI	Command Line Interface
CMS	Content management system
CPU	Central Processing Unit
CSIRT	Computer Security Incident Response Team
DEVOPS	Development and operations
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
ECA	Event-Condition-Action
eBPF	Extended Berkeley Packet Filter
eNodeB	Evolved Node B
EPC	Evolved Packet Core
ETSI	European Telecommunications Standards Institute
GRE	Generic Routing Encapsulation
HTTP	HyperText Transfer Protocol
I2NSF	Interface to Network Security Functions
IAAS	Infrastructure as a Service
IDM	Identity management
IETF	Internet Engineering Task Force
IoT	Internet of Things
IDS	Intrusion detection systems
IPS	Intrusion prevention systems
IPSec	IP Security
IT	Information Technology
LAN	Local Area Network
LXC	Linux Containers
M2M	Machine to Machine
MANO	Management and Orchestration
MPLS	Multiprotocol Label Switching

Acronym	Definition
NSF	Network Security Function
OPEX	Operating Expense
PKI	Public Key Infrastructure
RAM	Random Access Memory
RAS	Remote Access Service
SFC	Service Function Chaining
SQL	Structured Query Language
TEE	Trusted execution environment
TOSCA	Topology and Orchestration Specification for Cloud Applications
TPM	Trusted Platform Module
VF	Virtual Function
VLAN	Virtual Local Area Network
VoIP	Voice Over IP
VPN	Virtual Private Network

1 Executive Summary

This document reports the intermediate outcome from Task 1.5, by describing the initial version of the ASTRID architecture that will drive the technical activity in the second year of the project. The description considers the expected workflow to automate security management for a virtualized service, including the involved actors and their roles, and highlights the evolutionary and innovative steps with respect to current practice. The workflow identifies two main scopes, one for adapting the service to the ASTRID framework (*pre-deployment*) and the other for orchestrating security features (*run-time*). The architecture mostly covers the second scope, which is suitable for being engineered, and provides more generic indications for the first one, which is expected to be different for every orchestration model.

The description of the ASTRID orchestrator is largely based on the identification of three logical planes, namely the Data Plane, the Control Plane, and the Management Plane, with an explicit parallelism with the architectures adopted for software-defined networking. A number of logical components have been identified to implement the ASTRID workflow, which are mapped to research and innovation activities in WP2 and WP3. The document also analyses the progress towards the project objectives, by discussing to what extent the proposed architecture and its logical components fulfil the functional requirements previously identified in D1.1.

Finally, the ASTRID architecture is compared with other frameworks in the same context (I2NSF from IETF), to identify possible exploitation opportunities and technological distinction. Future work in WP5 will also consider commercial aspects and competitive advantages.

2 Introduction

The usage of virtualization technologies simplifies the creation of dynamic execution environments, removing the need for long hardware reconfiguration. In case the amount of resources and duration of the usage is unknown or uncertain, public cloud services avoid the need for large investments in the hardware, hence reducing both CAPEX and OPEX through pay-as-you-go models. Despite extreme flexibility and large cost-effectiveness, virtual environments wipe physical barriers away, entrusting hypervisors or other purely-software mechanisms to assure isolation between applications and networks.

There is a general awareness of the additional threats that come when adopting cloud technologies, but the presence of multiplicity of technical and business models hinders the development of general solutions. The main evolutionary trends in virtualization paradigms and their implications on cyber-security aspects have already been thoroughly examined in D1.1. The same document also discusses new directions from improving situational awareness, as inferred by the analysis of security reports and whitepapers from major vendors in this domain. Based on the evolving scenario, there is a clear need for more effective detection (especially for unknown, stealthy, and zero-day attacks), more efficient and pervasive inspection (encompassing hardware, cloud, and IoT), and improved automation (in both the detection and management processes). The ASTRID concept entails enhanced visibility on virtualized services and improved awareness of attacks and new threats, through a modular and programmable framework that leverages latest advances in automation paradigms.

The scope of the ASTRID project embraces different technological domains: virtualization and cloud computing, network function virtualization, and cyber-security. Section 3 gathers the main technical terms that are used throughout the paper, giving a brief yet exhaustive explanation for each of them. It represents a good reference for those readers that are not deeply skilled in all the involved fields or are not interested to read all technical documentation from the ASTRID project.

In the transition from concept to implementation, the definition of a reference architecture is the preliminary step to translate objectives into concrete functional elements and their relationships. The definition of the architecture starts from delineating the technological and business areas (i.e., the scope), because the whole context of virtualization and cloud services is too vast and generic for a common approach in the mid-term. Then, it is important to take into account the main limitations of existing approaches and technologies, so that proper solutions can be considered since design time to overcome them. Finally, the technical and business objectives dictate what functions and characteristics must be present in the architecture. Though the scope, motivations, and objectives for the project were already clear at the proposal stage and are (at least) partially expanded in D1.1, it is worth collecting and briefly reviewing them also in this document (Section 0), so to keep together all the relevant background behind the definition of the ASTRID architecture. This Section can be safely skipped by those readers that are already familiar with the main ASTRID concepts.

The ASTRID conceptual workflow (Section 0) describes the main actors involved in the process of secure orchestration and the sequence of operations that they are expected to perform. It compares the expected workflow with current practice, mostly to give a complete picture to readers which are not familiar with software orchestration. The conceptual workflow identifies the need for the main elements in the ASTRID architecture. In brief, it identifies the need for a pre-processing phase, which analyses the service and identifies a set of enrichments to provide security services, and a run-time phase, which implements the security policies. The ASTRID architecture is then described in Section 6. It follows the separation between a pre-deployment and a run-time scope identified in the conceptual workflow. The pre-deployment scope is more a list of tasks and activities to be carried out according to different practice, so it is not going to be implemented as a software component in ASTRID. The run-time scope describes the ASTRID orchestrator, which includes all components to collect the security context, process it, and trigger response and mitigation actions. Since this document is conceived to organized the internal work, this Section also assigns the design and implementation of the discrete components to the different Tasks in WP2 and WP3. In addition, Section 7 reviews the requirements already identified in D1.1 and assesses the degree of fulfilment for each of them, so to better understand which one still need to be addressed by the following activities.

It is worth underlining that the ASTRID concept follows under a more general effort to bring programmability and autonomicity in the management of security for virtual services. To this aim, the proposed architecture is compared with a parallel framework (Section 8), the I2NSF, being developed by the IETF. The comparison highlights similar concepts and conceptual workflows, as well as the unique aspects of ASTRID that distinguish the project and make it more challenging.

The ASTRID framework leverages software orchestration for automatic deployment of its components, as well as for quick and effective reaction to threats and attacks. Since the development of orchestration tools does not fall under the scope of the Project, it will not deliver technical documentation on these aspects. Unfortunately, orchestration is a relatively new branch of research and there is a general lack of books, articles, and whitepapers on this topic. To fill this gap, this document includes a brief appendix (Annex A) on alternative orchestration models that are of interest for ASTRID, including the main initiatives from standardization bodies. It is intended both as internal reference for the Consortium as well as more detailed information for the interested reader. For similar reasons, an additional an additional appendix discusses the I2NSF framework (Annex B), which currently represents the only reference and comparison term for ASTRID.

3 Terminology

A number of terms and expressions are used in ASTRID that may differ in different domains. To avoid ambiguities, Table 1 lists the most relevant terminology and the meaning in this Project.

Table 1. ASTRID terminology.

Term	Meaning
ASTRID Security Framework	The ASTRID Security Framework is a collection of software tools, protocols, and technologies that are used to build situational awareness on virtual services and react to attacks. All the components are organized in three planes: data, control, and management. It is very important to note that, though we use a typical network terminology, the security framework is not related to a network stack.
Cloud Management Software	Cloud Management Software effectively support provisioning of resources in a virtualization environment. It implements a common and unified interface towards heterogeneous virtualization technologies (i.e., hypervisors, virtual switches, storage systems), for creation, modification, configuration, and removal operations. Cloud Management Software also manages identity and access control throughout the virtualization environment. Typical examples of Cloud Management Software are OpenStack and VMware vSphere.
Control plane	The control plane is a collection of functions for monitoring, discovery, negotiation, and configuration of capabilities and properties of an underlying data plane. The control plane implements the necessary logic to react to the evolving context in a short timeframe. The control plane orchestrates the flow on the data plane according to imperative guidance (i.e. configuration) received via the management plane. Typical functions of a control plane include discovery of peers and their capabilities, monitoring of performance, negotiation of dynamic properties and operation modes, configuration of operational parameters. The concept origins and is mostly used in the networking domain for packet forwarding, where the control plane encompasses all functions to automatically compute and update paths for all possible destinations. In the ASTRID Security Framework, the control plane selects the detection algorithms, configure the data plane to collect the security context, manages authentication, authorization and access control. It operates according to behavioural policies provided by the management plane.
Cyber-security professional	See IT Security professional.
Data plane	The data plane is the set of functions to process data. This may include several tasks as inspection, enforcement, collection, aggregation, and analysis. It operates under imperative and deterministic models, according to configurations and parameters set by the control plane. The concept origins and is mostly used in the networking domain, where it denotes packet forwarding operations (including classification and filtering), according to the rules computed by the control plane. In the ASTRID Security Framework, this plane includes all functions for monitoring, inspection, enforcement, and analysis of the security context (events, logs, statistics), therefore including detection and correlation algorithms.

Term	Meaning
End user	He is the actual user of a virtual service. He may deploy himself the service through the orchestration software, or use the service provided by a service provider. Only in the first case it takes actively part in the software lifecycle.
Event-Condition-Action (ECA) policy	An imperative paradigm to define a conditional behaviour under the following model: • an Event triggers the evaluation of a Condition clause; • a Condition evaluates the current context and possibly selects an Action to be performed; • an Action clause defines the sequence of operations to be performed.
Flow-based policies	Flow-based security policies define the behaviour of a system according to constructs that reflect the typical operation of flow-based network security functions. Flow-based NSFs are based on stateful processing, i.e., they consider both the packet content (headers and payload) and context (session state). ECA policies are an example of flow-based security policies.
Identity management	A framework that establishes the rules to access resources and enforces their application. It is responsible for Authentication, Authorization, and Accounting (AAA) operations. Rules are often expressed as policies that take into account multiple authentication and authorization factors. While originally the concept was more focused on users, it has then evolved to include mutual authentication of applications, services, and resources.
IT Security professional	He is responsible for IT security, including networks, computer systems, and all digital equipment in the enterprise. The roles and job titles in the security sector often involve somewhat overlapping responsibilities and can be broad or specialized depending on the size and special needs of the organization. Typical job titles are security analyst, security engineer, security administrator, security architect, security specialist, and security consultant. The responsibilities vary from defining security plans and policies, analysing vulnerabilities and evaluating risks, deploying and configuring firewalls, intrusion detection/prevention systems, deep packet inspectors, security information and event management systems.
Management plane	The management plane defines the system behaviour in the mid/long term. It includes tasks as instantiation and configuration of components in the data/control planes, translation of high-level intents and goals into commands and control policies, monitoring of operational parameters to detect deviations and malfunctioning, interaction with other systems/components. The term originates and is commonly used in the networking domain, where it denotes the set of interfaces and protocols to configure the behaviour of devices and protocols (e.g., select and configure the routing protocol, set device name, set date and time, create virtual LANs, etc.). In general, interaction on the management plane is less frequent and less regular than on the control plane. While a system can fulfil its purpose without continuous input from the management plane (and this is a typical situation when there is no need to change the services implemented by that system), without continuous availability of control plane functions a typical component could not function properly (i.e., it would not know how to behave in case of unexpected events). The management plane can require manual operations (e.g. through a CLI or web interface), or it can be automated through orchestration tools. In the ASTRID

Term	Meaning
	Security Framework, this plane is responsible to assist security providers. Typical management actions include but are not limited to: dynamic adaptation of the service graph to on-going attacks, enhanced trustworthiness in deployment and business chains, graphical representation to users, information sharing, definition of reaction and mitigation policies, etc.
Network Security Function	Software that provides a set of security-related services. Examples include detecting unwanted activity and blocking or mitigating the effect of such unwanted activity in order to fulfil service requirements. The NSF can also help in supporting communication stream integrity and confidentiality [16].
PKI	Public Key Infrastructure is a set of roles, policies, and procedures needed to create, manage, distribute, use, store & revoke digital certificates and manage public-key encryption.
Policy rule	An imperative statement that is used to define the behaviour of a system. This includes monitoring and control of the state of one or more managed objects. For example, a policy rule may be described with the ECA pattern.
Security orchestrator	A tool that automates most tasks concerning security management, including monitoring and inspection, detection, reaction, notification. The effectiveness of a security orchestrator depends upon the presence of programmable and software-defined components in the execution environments, which can be controlled remotely. A security orchestrator is therefore expected to largely rely on the presence and capability of a service orchestrator.
Security provider	In the ASTRID architecture, he is the IT Security professional that flanks the service providers to guarantee secure and reliable operation of the virtual service(s). He is responsible to respond to attacks, investigate new threats, keep data with legal validity as evidence in court.
Service developer	He creates Virtual Services, by chaining together Virtual Functions into Service Graphs. The service developer usually builds common service templates, which are then tailored to the specific needs of different customers by the inclusion of proper policies, to drive the orchestration process in configuration, deployment, and lifecycle management.
Service description	This is an abstract representation of a service, suitable to be parsed and processed by (semi-)autonomous orchestration tools. A service description includes the Virtual Functions that compose the service, and their logical relationships. Different models can be used for this purpose. Cloud applications are usually described by service graphs according to model-driven engineering (e.g., TOSCA), while network services usually employee forwarding graphs, which better capture the packet forwarding behaviour (e.g., ETSI MANO and IETF SFC). The ASTRID framework should remain agnostic of the orchestration model.
Service graph	A service graph depicts the logical topology of a service. It is usually represented as a (not necessarily connected) graph, which nodes are the elementary components of the service and links are associated with some logical relationship (e.g., flow of data, configuration dependencies).
Service orchestrator	A tool that automates most tasks to operate a service (cloud application or network function). Orchestration typically includes provisioning of virtual

Term	Meaning
	resources (VMs, virtual networks, virtual storage), software deployment and configuration, management of life-cycle events (start/stop, scale, recover from errors or failures).
Service provider	He deploys Virtual Services by orchestration software on virtualized infrastructure. He provides services to end-users.
Software developer	He develops Virtual Functions. In ASTRID, the software developer may write himself the program code, or he may build Virtual Functions by wrapping existing applications with proper orchestration metadata (i.e., service name and description, deployment and execution constraints, provided/required functionalities, management hooks).
Software orchestration	This is the process to automate the deployment and lifecycle management of software applications and services over virtualized environments. The main purpose is to adapt the same software to the dynamic execution context, considering both different operational conditions and workload. There exist different orchestration paradigms, based on different models to describe the service, its components, execution constraints, lifecycle management rules, and so on.
User (security) policy	A user policy defines the expected system behaviour. It is often formulated by end users, so it uses non-technical semantics. User policies usually express the goal or intent, without considering the specific control interfaces of the underlying system. Its application requires a refinement into commands and configurations that can be understood by the controlled system; sometimes, intermediary technical representations are used to better cope with the presence of heterogeneous devices.
Virtual Service	A virtual service is a combination of multiple (virtual) functions deployed in a virtualization infrastructure. ASTRID will primarily address the Infrastructure-as-a-Service paradigm, where applications are run in custom execution environments. A virtual service also includes virtual networking to enable communication among the functions. ASTRID specifically considers virtual services deployed over multiple clouds (i.e., cross-cloud deployments).
Virtual Function (VF)	A VF is a standalone software unit that can be combined with others and orchestrated to create virtual services. Each VF implements an elementary service and provides suitable interfaces to be chained with other components. A VF is composed of software programs and metadata for orchestration. Examples of VF include both applications (e.g., web server, data base, network functions (e.g., firewall, load balancer, router, traffic shaper, classifier, DNS, Enhanced Packet Core).
Virtualization environment	A virtualization environment creates software-based instances of resources. There exist different paradigms based on the type of resources to virtualize. In ASTRID, the focus is on the Infrastructure-as-a-Service model, hence a virtualization environment provides computing, storage, and networking resources. The main purpose of virtualization is to share the same hardware among multiple users, giving them the perception of being the unique owner of the resources; this is typical for virtual LANs (VLANs), hypervisors, shared file systems. In addition, being software-based technologies, provisioning of virtual resources becomes extremely easy with respect to the traditional processes to buy, deploy, and configure real hardware.

4 Scope, motivations, and objectives

The ASTRID project tackles the structural limitations of legacy cyber-security paradigms with respect to evolving business and computing models. Hardware appliances are still largely used for most cyber-security services in large enterprises, whereas software implementations are commonly used in small offices and homes environments (SOHO). In both cases, the typical assumption is the physical segmentation between internal and external resources through a security perimeter, an assumption that does not hole anymore when external cloud and IoT resources are integrated into business processes.

The growing level of competitiveness and low profitability in the software industry have increasingly pushed for ever-shorter times-to-market and software lifecycles, from design to implementation and deployment. Agile software development methodologies have embraced the *devops* approach to reduce release times by massive recourse to automation in the deployment and operation process. Model-driven engineering envisions the creation of high-level process models, which are then dynamically and automatically mapped (*orchestrated*) into software functions and infrastructural resources based on the evolving context. This evolutionary process undoubtedly brings unprecedented opportunities for the software industry, but the tight integration among diverse business roles and the need to share infrastructure and data bring additional security and privacy concerns that have not been addressed in a satisfactory way yet.

A thorough analysis of the major trends behind the virtualization wave and their implications on cyber-security aspects is reported in D1.1, which also explains in details the Project concept and target application scenarios. Here, we complement that description with additional information that drove the definition of the ASTRID architecture. We therefore elaborate on the specific scope of the Project (the whole context of cloud and service orchestration is too vast and generic for a common approach in the mid-term), the main motivations, and the specific objectives, which together dictate the fundamental traits of the ASTRID architecture.

4.1 Scope

The cloud paradigm entails multiple virtualization models: Infrastructure-as-a-Service, Platform-as-a-Service, Service-as-a-Service, Network-as-a-Service, and even Everything-as-a-Service. Every model provides different kind of resources (virtual machines, containers, storage space, web servers, operating systems, virtual networks, …), which customers acquire by software APIs.

This Project explicitly consider the Infrastructure-as-a-Service (IaaS) model. IaaS is a very effective and agile paradigm that provides computing, storage, and networking resources corresponding to physical instances (servers, switches, disk arrays, network links). All the physical infrastructure is installed and configured only once with cloud management software and shared among multiple projects (tenants) and users, thus removing the need for hardware purchase, configuration, management, and disposal each time a new service is created.

Figure 1 depicts the typical deployment of a cloud service in an IaaS environment. Each tenant corresponds to a virtual execution environment (light green cloud), which includes virtual machines (VMs), block storage, and network connectivity (locally and towards the Internet). A cloud service is composed of several components that interact to implement a business logic; each component is often installed in a separate VM for isolation and quick restoration purposes.

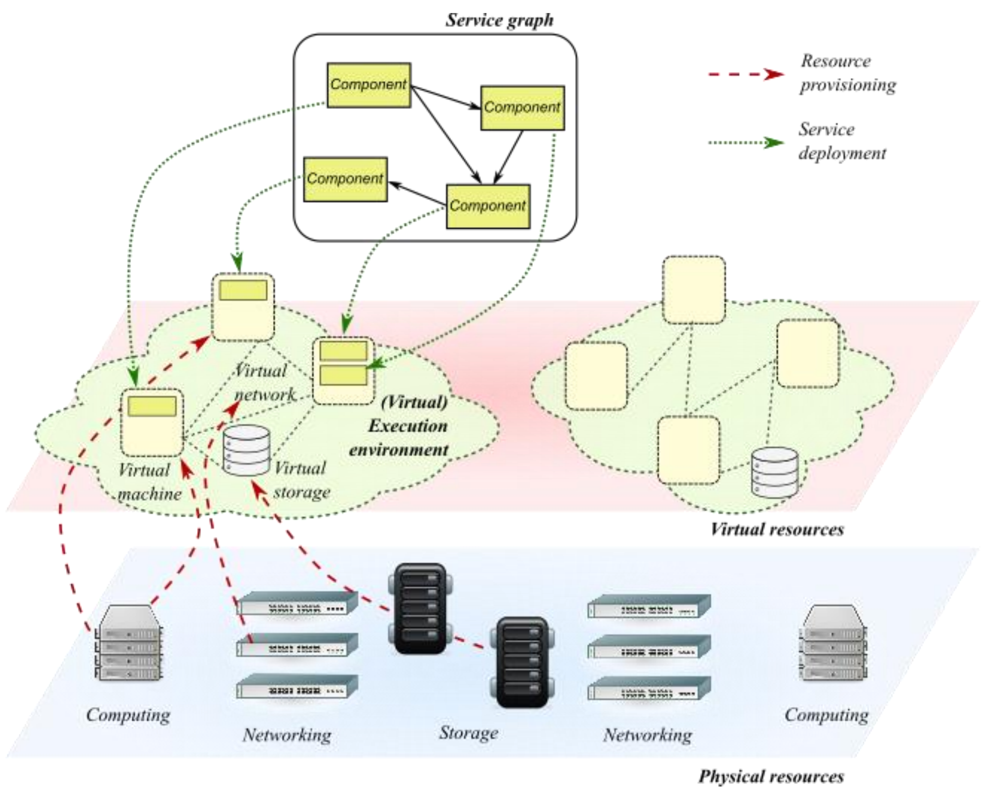


Figure 1. Virtual services are designed as graphs of applications. They are deployed in virtualised resources provisioned over a physical infrastructure.

Despite of the speed and agility in resource provisioning, software installation and configuration still require time and delays service deployment. In addition, the same service is often replicated for multiple customers with minimal re-configuration, and the deployment burden significantly affects the final cost. The need to easily replicate repetitive tasks for multiple installations is already tackled by industry-level configuration tools like Puppet, Chef, Ansible; however, such tools are static in nature and cannot react to the evolving context (time, utilization, errors, workload). Software orchestration greatly increases the level of automation, pursuing the implementation of reactive systems[1].

The starting point for software orchestration is an abstract model that describes the application topology and its basic components. The model consists in a semantics that describes individual software components, deployment and execution constraints (CPU, RAM, storage), connectivity requirements, software dependencies (operating system, system libraries, software packages), interfaces, configurations, management hooks. The semantics usually covers both each single software component and the entire topology as a whole. The topology is often indicated as "service graph" or "forwarding graph", which substantially differences in the formal representation (i.e., application-modelling engineering or data-driven topologies) and the target application domain (i.e., cloud or network function virtualization). While nodes are usually associated to basic applications and software functions, links have different meaning in different orchestration models: for example, they represent logical configuration dependencies (e.g., client/server, publisher/subscriber) in TOSCA [5] and data paths in ETSI NFV [7] and IETF SFC [9].

[1] The Reactive manifesto. URL: https://www.reactivemanifesto.org/

The orchestration process is then responsible to provision the virtual resources, select appropriate software implementations for each graph node, deploy and configure each function in a consistent way, manage lifecycle operations for the whole service.

The orchestrator is a smart engine that adapts the service graph to the current context, by exploiting specific metadata embedded by the developers (concerning requirements, constraints, and logical characteristics) and policies settled by service providers. The orchestrator is made of several components, which are used to provision the necessary virtual resources (CPU, RAM, disk, and network), to download software images and boot them into virtual containers (Virtual Machines, LXC, Docker, unikernel), to install required software and libraries, to configure each component in order to interact with the others (IP addresses, usernames, passwords, …), to monitor the execution of each single component. Orchestration may also include the deployment of additional components, according to specific service requirements (e.g., DNS, DHCP servers, load balancers, firewalls, antivirus, etc.) Finally, the orchestrator is responsible for whole life-cycle management of the application (start, stop, re-start, scale up and down or scale in and out, reconfiguration, de-provision). ETSI NFV [8] defines a refence architecture for orchestration (MANO – Management and Orchestration), which serves as a blueprint for interoperability between virtual network functions and NFV orchestrators. Nevertheless, multiple orchestration solutions have been developed both for the cloud and NFV (see D1.1).

4.2 Motivations

The ground-breaking shift in development and operation paradigms entailed by the usage of virtualization and automation tools has required a great effort in recent years to move from concept to real implementations. The complexity and multiplicity of the technical challenges to be solved have moved security in the background, also endorsed by the consideration that the large correspondence between the IaaS model and physical infrastructures would have permitted the application of virtualized implementations of existing cybersecurity appliances. However, de-coupling the software from the underlying hardware raises new security concerns about the mutual trustworthiness, lack of visibility, and potential threats about these two layers [11].

The automation of the operation tasks implies that also security aspects should be properly considered at the stage of graph design. However, agile and devops methodologies are progressively shrinking the number of professional figures involved. As a matter of fact, all the process workflows proposed so far only envision the roles of software developer, service developer, service provider, and infrastructure provider, but do not explicitly consider the presence of any cyber-security professionals (analysts, engineers, architects, officers). Security staff designs complex security policies and integrates security appliances into business processes, taking into account opposite needs of security and usability. They play a crucial role for safe and reliable IT operation and cannot be properly replaced by software/service developers. As a matter of fact, by recognizing the difficulty in effectively tackling ever more complex cyber-threats, enterprises are increasingly looking at cloud-based services and externalization as cost-effective solutions to the growing challenges and complexity in maintaining an up-to-date secure infrastructure that complies with regulatory requirements.

Based on the above considerations, the following motivations for a new approach were already identified at the proposal stage:

- **Lack of physical perimeter**: multi-tenancy and resource sharing are the main technical drivers for cost-effective cloud services. Isolation of tenants and segmentation of resources are the main pillars for any virtualization mechanism but are based on software mechanisms that provide weaker defence against cyber-attacks than physical boundaries. The number of known vulnerabilities and attacks in the cloud demonstrate that malicious users might be able to sneak in someone's else virtual environment (virtual machines, containers, virtual networks, etc.).

- **Lack of visibility**. As extensively discussed in D1.1, there is lack of security services exposed by cloud management software to the tenants (usually limited to basic firewalling functions), which means that potential vulnerabilities, breaches, and threats of the virtualised resources are not visible to owners of the virtual services, maybe giving a false sense of security. Even if more appliances were available, relying on correct configuration and operations (including security updates) from third parties might not be an acceptable policy for most critical services.
- **Limited expertise in security aspects**: even if portability and adaptation to multiple contexts of the same service is the peculiarity of devops methodologies, security implications might be very different. Pushing virtual instances around the service graph may give a false confidence of security, if they are not placed, configured and correlated in the right way.
- **Overhead on the service graph**. Security appliances as antiviruses, firewalls, and intrusion detection/prevention systems usually perform deep analysis on network packets, system calls, and logs, often replicated in each virtual function. Such computation may delay the execution of main processes, hence reducing the responsivity and processing capabilities of the application, to not mention the additional requirements on computing and storage resources. The overhead is particular heavy for virtual services of limited size (which are expected to be the majority).
- **Increased attack surface**. Security appliances are themselves sources of potential threats, as witnessed by the continuous vulnerabilities reported to national CERTs/CSIRTs (see D1.1). Just pushing security appliances in the service graphs without any effective response and mitigation plan in place might lead to a misleading perception of security.
- **Difficulty in forensics investigations**. Saving relevant events and data for a posteriori investigation and evidence collection is not a straightforward process and may be cumbersome to be implemented for all virtual services.

4.3 Objectives

ASTRID seeks a novel a novel cyber-security paradigm that, according to the Project proposal, "*provides better awareness about cyber-security threats of virtualised services, referred to each single component (i.e., each specific application) as well as the service as a whole (i.e., the entire service graph), and facilitate (possibly automate) the detection and reaction to sophisticated cyber-attacks*." The implicit focus is on enhanced visibility of what happens in each software component, with correlation at the graph level. Semi-autonomous reaction and mitigation is also envisioned to reduce the impact of attacks.

ASTRID aims at making virtual environment more secure than today exploiting the agility of modern orchestration systems, by designing new tools for automating security management and integrating these environments with existing procedures. It is worth noting that ASTRID does not develop new procedures for sharing knowledge in national or international context and does not specifically target a wide range of threat detection algorithms. Rather, ASTRID will provide an overarching framework for dynamically collecting pervasive and thorough situational knowledge in virtualised environments and will show how this enhanced information can be exploited to feed a new generation of effective and efficient detection algorithms.

The cardinal pillars of the ASTRID architecture come from the technical objectives settled in the proposal:

- **To decouple the service business logic from security management**. The implementation of security is a complex task, which starts from the definition of policies and comes down to technical solutions and procedures. Security management of service graphs is even more challenging, since the context continuously changes (e.g., scaling and life-cycle management).

Though automation of mechanical tasks is an effective way to reduce human errors in software deployment and configuration, the definition of security policies and technical design should remain a prerogative of security professionals, which harmonize the whole process at the enterprise level. That means security appliances should not be incorporated in the graph topology at design time, but a side layer should be present to "enrich" the service graph with a broad set of security properties, including inspection, monitoring, detection, enforcement, reaction, and mitigation tasks. In the ASTRID architecture, security properties of each graph component as well as the whole service are defined by proper models and policies, which are then used at deployment time to properly configure the execution environment. A dedicated interface is expected to this purpose, that facilitate the integration with existing tools and procedures at the enterprise level (e.g., visualization of events and alerts, risk assessment, virtual networks).

- **Automate security management and response to security incidents and attacks**. Despite the need of human abilities in defining the correct security strategies and policies, many repetitive tasks may be largely automated to avoid errors and misconfigurations. In this respect, the ASTRID architecture is expected to take advantage of recent orchestration models, by defining policies and constraints that describe "what" is required rather than "how" to implement it. Similarly, common mitigations and response actions can be often undertaken according to specific security strategies. Beyond the mere application of firewalling or access control rules, orchestration enables to automatically remove or replace compromised functions, change forwarding rules, isolate suspicious services for investigation. This will substantially improve the effectiveness of responses, which today often come late because of the need for human intervention.
- **Reduce the run-time overhead of security processing**. Most security appliances require monitoring and inspection operations (on network packets, software behaviour, system calls) that consume CPU cycles and may deteriorate the overall service performance, especially in virtualised environment where hardware acceleration is rarely available; in addition, security appliances are not totally immune to attacks, so they increase the attack surface. In this case, the objective is two-fold: a) increase the processing efficiency of (at least) the most frequently executed portion of monitoring and detection tasks, and b) to protect security appliances from attacks. The ASTRID architecture is therefore expected to leverage local programmability to dynamically offload lightweight computation tasks, such as aggregation, filtering, fusion. Local inspection and monitoring capabilities should cover all relevant events and data for security analysis: data packets, system calls, application logs. The detection logic should instead be placed in remote and secure locations, hence shrinking the attack surface of the service graph.
- **Support legal and forensics investigation in virtualized environments**. The growing number of virtual applications and services is also expected to be subject to illegal usage and cyber-crimes. Specific technical mechanisms are therefore required to facilitate forensics investigation for identification of crimes and criminals. In this respect, tight integration with the orchestration software will enable to dynamically change the service topology for interception or obscuration as required by law. In the meanwhile, the presence of a PKI and identity management will support digital signing and anonymization compliant with forensics practice and privacy rules.

5 ASTRID conceptual workflow

The conceptual workflow describes the main steps that are needed to automatically manage secure services over virtualized environments. The definition of the workflow starts from emerging practice in software orchestration and insert additional elements for security that are currently neglected or overlooked. This approach has been given the top priority in the architectural definition, since it is considered a strategic point for effective exploitation of the Project results.

5.1 Software orchestration

The term "orchestration" is a buzzword used in many domains, often with very difficult meanings, ranging from control to management operations. In ASTRID, software orchestration refers to the typical concept in the domains of cloud computing and network function virtualization. It therefore entails the process of automating the deployment, configuration, and lifecycle management of software, based on a declarative service template.

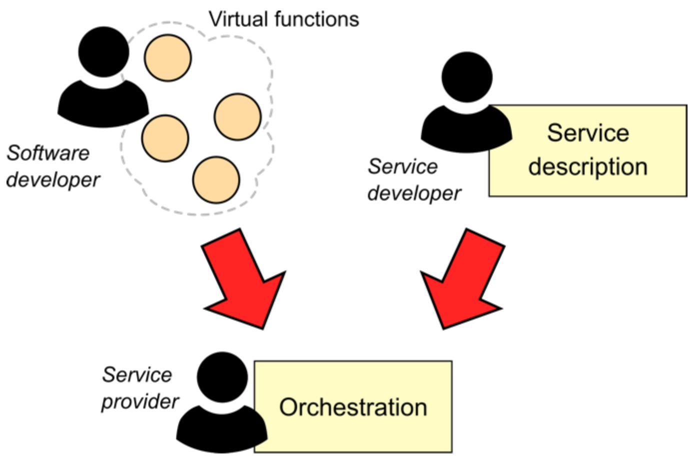


Figure 2. Orchestration of virtual services.

Figure 2 shows the main elements involved in software orchestration. The ability to easily adapt applications to the evolving context calls for a transition from "imperative" (i.e., procedural languages) to declarative models, an on-going process both for cloud applications [1][2] and network function virtualization [3]. A declarative model defines the application as a logical topology of elementary functions (i.e., the service "graph"), together with a set of rules and constraints for deployment and operation, provided as additional metadata. The functions may refer to bootable software images, software packages, or generic services (e.g., relational database, web server, domain name server). The creation of a virtual service requires the identification of functional and non-functional requirements, the design of a processing pipeline, the definition of rules for deployment and life-cycle management. This includes the specification of monitoring flows on the service execution and the conditions (e.g., thresholds) for triggering specific actions (scaling, healing, replacing, removing, etc.). A service is usually provided as a sort of *template*, which has to be instantiated and initialized at run-time by the orchestration process. The set of information that describes how to instantiate, configure, and manage the service is denoted as *metadata*. It includes the name and version of the software, vendor, description (including licensing and usage terms), entry points, deployment constraints, and management hooks (for instance, to start, stop, reload, or reset the service, to collect measurements, data, events, log). The design of a virtual service roughly corresponds to the evolution of existing tasks for software architects; however, this role is often indicated as "Service Developer" or "Service Designer" by current standards and technical literature.

The elementary components of any service topology are (virtual) functions. Virtual functions are developed by programmers (hereby indicated as “software developers”) and delivered in different forms. The simplest form is a (compressed) archive, including either source or object code. This is a straightforward approach, but it is mostly unsuitable for automatic deployment and management in heterogeneous contexts. A slightly more sophisticated approach is the creation of *packages*, including dependencies and configuration scripts. This is currently a very common way to distribute and install software on UNIX-like systems, where a package manager maintains a system-wide database of installed packages and takes care of dependencies. This approach ensures that the correct software version is already installed for every dependency, but it is not easy to guarantee the stability and reliability of every environment that is created by the selection of different applications and their updates. Finally, the last option is to deliver bootable images, with pre-installed software. These can be disk images, suitable for bare VMs, or container images to be started in pre-installed VMs. In this approach, each virtual function can be thoroughly tested and certified by the vendor. Clearly, this requires additional roles than software developers, but we will not go in further details here. Though the size of the final objects is far larger than any other approach, this is the only solution that can support high-reliability (up to 4 or 5 nines). Virtual functions should be enriched with metadata as well to drive automatic deployment and orchestration. Metadata typically includes the name of the component (i.e., trademark and vendor), its description (including licensing and usage terms), provided functionality (e.g., web server, database, DNS, EPC, eNodeB, RAS), required services (e.g., database, authentication server), deployment constraints (e.g., number of cores, CPU speed, RAM, disk space, network bandwidth, hardware acceleration), measured performance metrics (e.g., packet latency and throughput, dropped packets, packet statistics), and management hooks (for instance, to start, stop, reload, or reset the function). This information is used by orchestration tools to provision the proper set of resources, set up and configure the execution environment, and perform life-cycle management actions (e.g., scale the function upon indication from the orchestrator, recover from failure).

The actual instantiation of a service is done by the Service Provider. He deploys the service for his own usage (e.g., a billing application) or to provide services to end users (e.g., a virtual mobile network). Starting from the declarative service template, orchestration is responsible to start the provisioning process for virtual resources, deploy and configure the software, start all functions and execute any lifecycle management operation. The whole process may be totally automated, or there may remain a number of functions that should be carried out by human staff. We can therefore identify additional sub-roles within the Service Provider, which correspond to human specialists in case automation is not present (see Figure 3). For example, when the service model only indicates the type of function but does not dictate a specific implementation/version, and an automatic selection process is not deemed appropriate or reliable, a Supply Chain Specialist may be required to recommend or identify virtual functions suitable for the service, also considering existing commercial relationships, cost, reliability, and trust aspects. For mission-critical applications, a further Acceptance Specialist may be required to validate, certificate, and on-board the service and its functions, guaranteeing the compliance with dependability policies for the specific application. The Service Deployment Manager is the sub-roles that is better suited for automation, given the large availability of virtualization and configuration management tools.

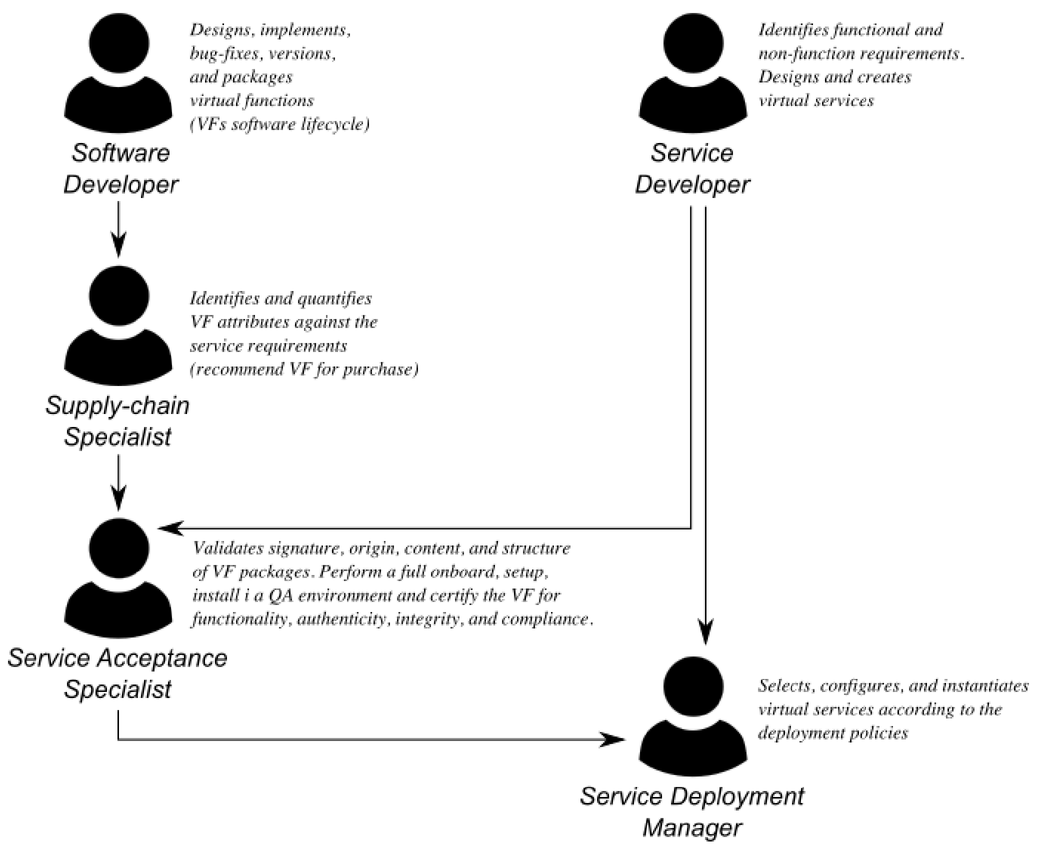

Software Developer
Designs, implements, bug-fixes, versions, and packages virtual functions (VFs software lifecycle)
Service Developer
Identifies functional and non-function requirements. Designs and creates virtual services
Supply-chain Specialist
Identifies and quantifies VF attributes against the service requirements (recommend VF for purchase)
Service Acceptance Specialist
Validates signature, origin, content, and structure of VF packages. Perform a full onboard, setup, install i a QA environment and certify the VF for functionality, authenticity, integrity, and compliance.
Service Deployment Manager
Selects, configures, and instantiates virtual services according to the deployment policies

Figure 3: Sub-roles in the service deployment chain.

Depending on the specific service model and virtual function delivery method, the instantiation of virtual functions may be limited to booting a given image or may require booting a vanilla operating system and installing the required software. Typical orchestration tools monitor at run-time the execution of the graph, by collecting measurements about used resources (CPU, RAM, disk, network), workload, performance (processed requests, latency) as defined in the service/function metadata. This data is then used to trigger lifecycle management actions, according to the policies defined by the service designer or the service provider. Such policies define how the system should react to events, either related to monitored data or triggered by the service provider. In most cases, the actions consist in running management scripts provided by the service/software developers.

An overview of the main descriptive models and their impact on service orchestration is given in Annex A, for both cloud applications and network function virtualization.

5.2 ASTRID workflow

Enterprises are today considering the usage of security functions hosted and managed by external providers, due to the growing challenges and complexity in designing, deploying, and maintaining up-to-date security infrastructures that complies with regulatory requirements in a cost-effective way. To meet this demand, more and more service providers are providing hosted security solutions to deliver cost-effective managed security services to enterprise customers, hence creating new business models. One of the main objectives for the ASTRID Project is therefore direct involvement of security experts in the whole orchestration workflow, which are not present in current models. They should be able to integrate security aspects at design time, to gain enhanced visibility on service execution, and to perform quick reactions at run-time. In this respect, we introduce the role of Security Provider, which complements the technical roles already described in the previous Section.

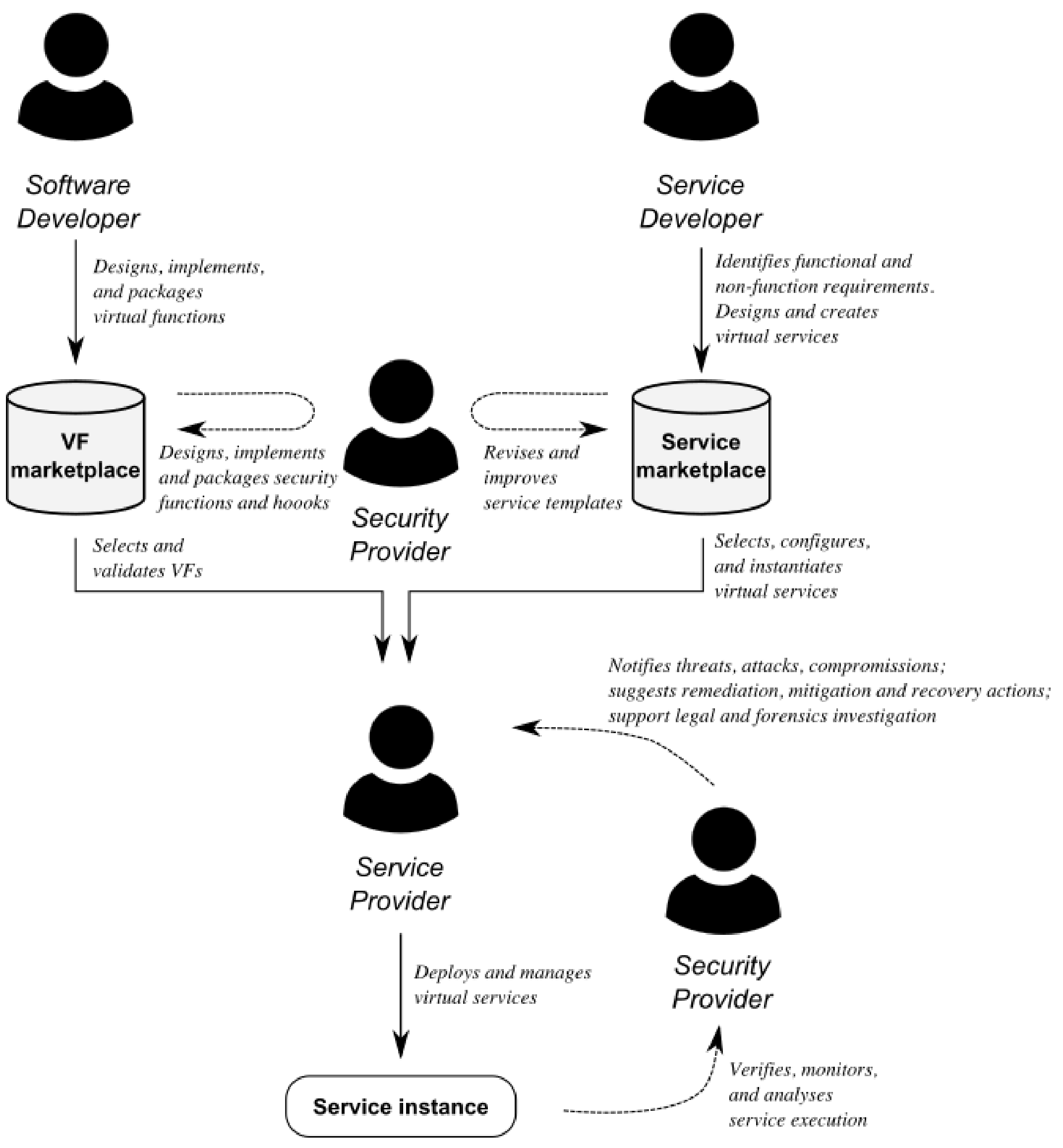

Software Developer
Designs, implements, and packages virtual functions
Service Developer
Identifies functional and non-function requirements. Designs and creates virtual services
VF marketplace
Designs, implements and packages security functions and hoooks
Security Provider
Revises and improves service templates
Service marketplace
Selects and validates VFs
Selects, configures, and instantiates virtual services
Notifies threats, attacks, compromissions; suggests remediation, mitigation and recovery actions; support legal and forensics investigation
Service Provider
Security Provider
Deploys and manages virtual services
Service instance
Verifies, monitors, and analyses service execution

Figure 4. The orchestration business chain enhanced with ASTRID roles.

The Security Provider operates in multiple points of the orchestration business chain, as shown in Figure 4. We remark that additional sub-roles may be involved in a real chain, and that different orchestration models imply different workflows. For the sake of brevity, we now consider a specific chain, without any loss of generality. In the considered model, Software Developers design, implement, and package VFs and sell them in a public or private marketplace. Service Developers are mostly analysts and software architects, which create service templates and sell them in a public or private marketplace. Security Providers take VFs and service templates from the marketplace and modify them. They can act as on their own initiative, by selecting popular or critical services, so to add value to the original version by including security functions that they know to be of general interest. Otherwise, they can operate on mandate of a Service Provider, who has already selected (or asks them to select) a specific service. In this phase, Security Providers assess the degree of trustworthiness of VFs and their providers, verify the robustness of software applications when taken alone or combined in complex topologies, insert additional functions and monitoring hooks to implement security services at run-time, and so on. Basically, they laid the foundations for security operations that will be available on the running service.

After getting the “revised” version of the service template and related virtual functions, the Service Provider deploys the service over a virtualization infrastructure. Security features that have been incorporated in the graph can now be activated and configured to implement several security services (e.g., firewalling, IPS/IDS, malware detection, remote attestation). The most general workflow for converting user policies into low-level configurations is shown in Figure 5. Many Service Providers and End Users might not have the requisite skills to identify security architectures, behavioural policies, and functional configurations. Instead, they are expected to express their expectations (i.e., goals or intents) in terms of high-level guidelines or “user security policies” (e.g., “protect the service from network threats”, “deploy the service in trusted infrastructures”, “protect against the injection of malware”, “bind the service to legal obligations for crime investigation and data protection”, etc.). The scope of security services may cover monitoring, response, mitigation, and investigation. Such expectations must then be refined into security services (e.g., firewalling, deep packet inspection, software execution tracing, antivirus, intrusion detection) and flow-based policies (e.g., if <average bandwidth usage is greater than 50 Mbps> then <drop incoming packets from *x.x.x.x*>, when <file *X* is requested> if <user is not authorized> then <log possible privilege escalation attempt>), including their integration with the service graph and behavioural rules in the ECA form. Finally, flow-based policies must be mapped to a set of commands and configuration directives, which are activated and modified at run time according to the evolving context (e.g., drop packets with source address *x.x.x.x*, log packets with destination address *y.y.y.y*, log requests to port *Y*). The refinement process can be fully automated for the simpler user policies, but human intervention is still expected in the majority of cases.

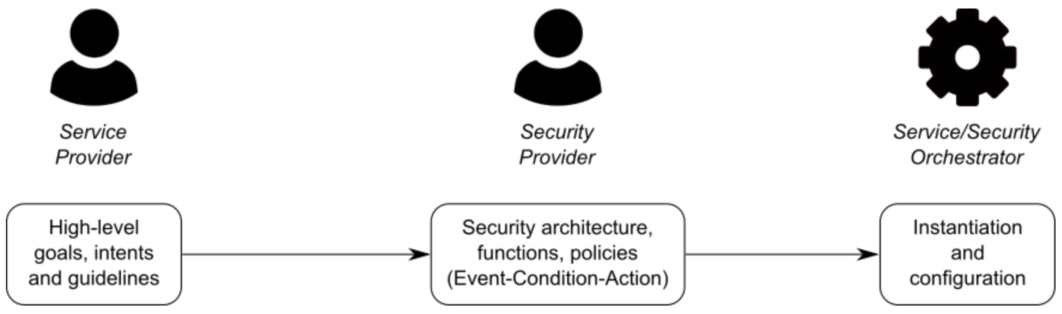

Service Provider
Security Provider
Service/Security Orchestrator
High-level goals, intents and guidelines
Security architecture, functions, policies (Event-Condition-Action)
Instantiation and configuration

Figure 5. The ASTRID “security policies chain.”

From a business perspective, externalization of security management could be an effective option, especially in case of critical services. The Security Provider can also undertake the responsibility to manage security aspects of the running service. This goes beyond the translation of user policies into security services and functional policies. Indeed, the Security Provider becomes in charge of assuring correct operation of security functions, supervising automated procedures, initiating mitigation, suggesting the Service Provider possible response and recovery actions on the deployed service. Clearly, the presence of security features in the service graph is an inescapable requirement for effective operation of the Security Provider. The extent to which the Security Provider operates autonomously of upon authorization of the Service Provider should be settled in the contractual agreements.

Based on the preliminary considerations on the business roles and potential business models, Figure 6 shows the main technical aspects of an enhanced ASTRID orchestration workflow. There are two scopes of the ASTRID framework, tightly corresponding to the business opportunities identified so far for the Security Provider, that are inserted between existing processes in the orchestration workflow (i.e., the orange soft shapes). The *pre-deployment scope* enhances the plain graph design with security aspects, including the verification and certification of the service as well as the inclusion of additional components to be deployed. The *run-time scope* covers monitoring, detection, and reaction operations that are necessary for secure and trustworthy execution.

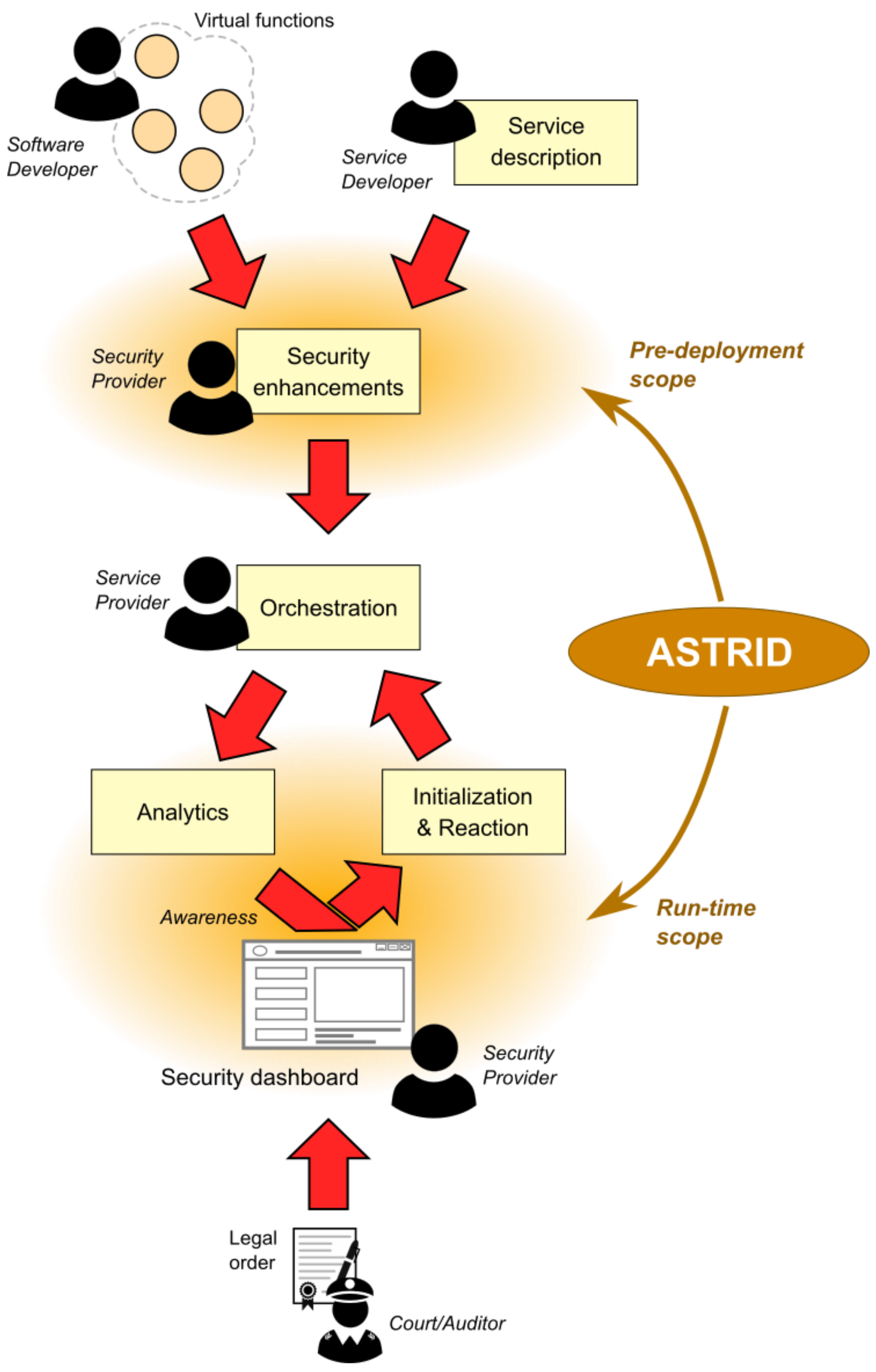


Figure 6. Overall workflow of the ASTRID framework.

5.2.1 Pre-deployment scope

The pre-deployment scope complements the plain service design with security aspects. While the Service Developer focuses on the business logic of the service, a Security Provider should revise the whole graph and its components and "*enhance*" the description with security functions, either on his own initiatives or on commission of a Service Provider. Such functions are conceived to support the implementation of security policies at run-time, in the same way the original metadata enables automatic deployment and life-cycle management.

There are multiple complementary ways to enhance the service description, acting on either the service itself or the constituent virtual functions; most of them target the availability of specific features at run time. First, the ASTRID concept entails the **enrichment of the service graph** with dedicated components for supporting multiple features:

- building the logical fabric to create, collect and analyse the security context from the set of independent virtual functions, also including the possibility to retain data for legal investigation and forensics;
- the deployment of additional components for identity management and access control, to enable seamless and secure interconnection with external components;
- the inclusion of dynamic components for legal interception of the traffic;
- segregation or obfuscation of data and traffic for privacy issues;
- the definition of automatic reaction policies, in terms of actions to be executed when specific events occur or conditions are met, for instance according to a typical `if-then-else' scheme.

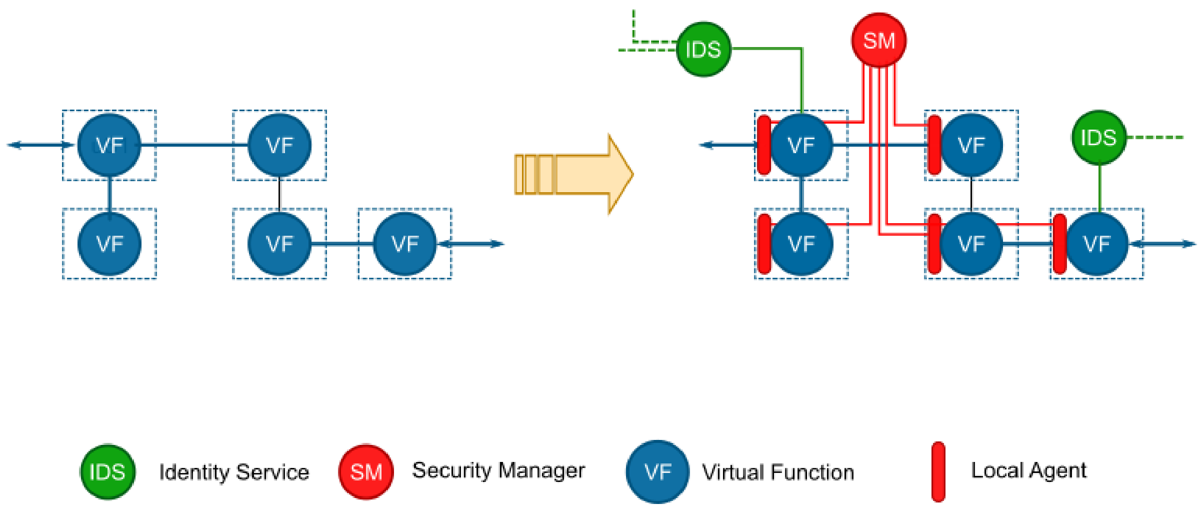


Figure 7. Service description enriched with ASTRID Security Orchestrator and Identity Services to connect to external functions/devices.

Figure 7 pictorially illustrates the concept of graph enrichment. There are basically two types of components that can be used to enrich the graph:

- stand-alone virtual functions, roughly corresponding to software versions of legacy security appliances (i.e., the red and green circles in the picture);
- software co-located in the same execution environment of the original functions, which may be integrated in the kernel, installed as shared libraries, or deployed as standalone applications or daemons (i.e., the red ellipses along by the virtual functions).

As the picture shows, the enrichment also covers (logical) communication channels to create a sort of security overlay above the business logic of the service graph. The purpose of this overlay will be further discussed in Section 6, while its practical definition and implementation falls under the objectives of WP2.

The second main enhancement in the service description will be the **validation and certification** of the involved virtual functions. There are multiple actions that may be undertaken in this respect. First of all, the origin and integrity of the software should be verified, and the trustworthiness of the vendor and distribution channels should be assessed. A second step may consist in scanning the software to seek out viruses, malware and known signatures. These processes could be applied to any delivery methods, including both source code or executables. However, depending on the output of the virtual function that is available (either source code or only the executable binary), different types of static vulnerability analysis approaches can be integrated ranging from traditional "fuzzing" techniques [24] to more sophisticated and dynamic concolic analysis systems [25][26].

A more advanced enhancement is the analysis of the source code, whenever available, targeting more advanced dynamic *control-flow attestation* solutions that aim to check software behaviour during run-time; software that will be running as expected by verifying the integrity of specific control flows. Towards this direction, identification of the correct (i.e., as expected from the normal execution of the virtual function) execution paths that will be checked during run-time for any deviations, need to be identified at the pre-deployment phase coupled with adequate control-flow properties and policies. We define a set of properties and policies to be attested as the minimal required set of execution related attributes that will allow the correct verification of the virtual function during run-time by attesting the function's control-flow graph against these policies. Obviously, this generates additional data that must be included in the service description (i.e., control-flow policies and graphs) and digitally signed to guarantee its origin and integrity.

After the enhancement process, the service template is ready for instantiation. As already mentioned, one design target is to avoid the need for major modifications in existing tools. In this respect, the enhancement process is expected to modify the service description according to existing models, without introducing additional formalisms or capabilities.

5.2.2 Run-time scope

The run-time scope of the ASTRID framework gives deep visibility on the execution of the service and provides the means for quick reaction and mitigation actions. As already pointed out for the pre-deployment scope, one main objective is the involvement of security experts, which become responsible to supervise security aspects (this is the reason why this role is indicated as "security provider" in Figure 6). Again, we remark that multiple activities and professional figures may be entailed by this role, but this is largely dependent on the specific business model adopted.

In the run-time scope, the graph enhancements applied before deployment are used for three purposes:

- initialization of the security functions that have enriched the graph;
- building awareness about attacks, anomalies, and suspicious activities;
- interacting with the graph to implement protection, reaction, recovery, and healing actions.

All the above operations should be centrally managed by a **security dashboard**, a GUI that facilitates the visual representation and control of the on-going situation. Through the security dashboard, the security provider gets a real-time picture of relevant events and data, pin-pointed to the relative position in the service graph. The dashboard is also used to define policies, which may be expressed in different forms depending on the skills of the specific user. Service providers and end users will select high-level user policies, whereas security providers will typically define ECA policies; the latter can be easily refined in specific commands and operations (for example, close a given port on the firewall when the amount of traffic suggests a possible DoS, freeze a virtual function if there is evidence of corruption). The dashboard can also be used to conduct investigation on suspicious activities, which may be symptomatic of new threats or unknown attack vectors. In this respect, the security provider can inject different configurations and programs for monitoring and inspection hooks, so to increase the verbosity and capillarity of the security context according to the current needs, and can manually trigger orchestration actions (for example, removing or replacing a compromised function, deploy additional inspection or mitigation functions, steer packet flows, isolate the service graph in a honeypot, duplicate traffic for interception). The same type of intervention is also required upon requests from the court, authorized law enforcement offices, or auditors, as pictorially shown in the bottom part of Figure 6.

The dashboard is conceived as the main operational tool for the security provider, but it may also be used for sending tailored notifications to end-users. As a matter of fact, due to legal obligations or internal transparency policies, a service provider may be concerned to keep his customers aware about possible threats to their personal data or service availability, so that proper countermeasures could be implemented on their side (for instance, remove sensitive records, suspend or move away critical activities).

Run-time analytics are responsible for automatic processing of the security context. Through orchestration, all security hooks defined by the enrichment process are deployed in the execution environment. The security hooks monitor the virtual functions and the network traffic exchanged in between; logs, statistics, events, and other relevant information are therefore collected and delivered to a central location for analysis. Decoupling data collection from data processing is one of the main pillars of the ASTRID concept, and enables the definition of a great range of detection algorithms.

Run-time analytics are therefore a collection of heterogeneous algorithms, including both legacy appliances (e.g., IDS/IPS, antivirus, anti-DoS) and more advanced anomaly detection techniques that could leverage cross-correlation of data from multiple heterogeneous sources. Part of these algorithms will be able to detect known attacks from their signatures or behavioural patterns, hence triggering an alarm. Other algorithms will only detect anomalies, so to raise a warning in case of suspicious behaviour that needs further investigation. In both cases, the dashboard will be notified.

Initialization and reaction is a compound process that, as the name implies, entails complementary actions in different phases of the service lifecycle. Initialization is conceived to assist the main orchestration process in the preliminary configuration prior to starting the service. Typical tasks for initialization include:

- configuration of firewalling rules to protect the virtual functions and links;
- installation of monitoring and inspection programs in the service graph according to security policies;
- configuration of the sink(s) for collection of monitoring information;
- configuration of access points for programmable hooks in the service graph;
- instantiation of the security services defined in the enhancement process;
- instantiation of identity management and access control.

Reaction encompasses several actions as mitigation, reconfiguration, restoration, and healing that are undertaken in response to particular events or conditions, usually aimed to respond to (possible) attacks. Reaction can work in three alternative ways:

- *fully automated*: the framework reacts to specific conditions based on pre-defined rules, without any intervention from humans. This is only possible for well-known threats. For example, a packet filter may be installed when the traffic streams grow beyond a given threshold. Another example is the request to isolate or remove a virtual function upon indication of intrusion.
- *semi-automated*: in case of unknown or complex attacks, pre-defined policies might not be able to cover all possible situations or variants, so the system may only partially respond automatically and wait for further inputs from humans. This may be the case of anomalous (yet not overwhelming) flows of packets that are temporarily blocked while waiting for additional actions from the security provider.

- *supervised*: the system is able to react autonomously, but the likelihood or impacts of possible errors suggest confirmation from humans. In the same example as the previous point, the security provider is asked the permission to block the traffic, so to avoid disrupting any critical activity.

Automatic reaction shortens response times and unburden humans from mechanical and repetitive tasks. However, full awareness and the need for post-mortem analysis recommend keeping track and report any action to the dashboard, at least to give visibility of the occurrence of attacks. This aspect is graphically implied in Figure 6 by the folded arrow between the analytics and reaction blocks, which always involves the security dashboard.

6 System architecture

Consistently with the workflow outlined in Section 0, the ASTRID architecture enhances emerging orchestration practices without demanding full re-design of existing tools. That means that the architecture can be described in terms of additional components, corresponding to the specific ASTRID scopes in the workflow.

6.1 *Legacy architecture (Business-as-Usual)*

Following the same approach used in the previous Section, we start by elaborating on the main elements for service orchestration, which are depicted in Figure 8. We already discussed the main elements of service orchestration, and we are now going to give more details about these elements and their implications on the service model and instance by comparing them side by side in Figure 8.

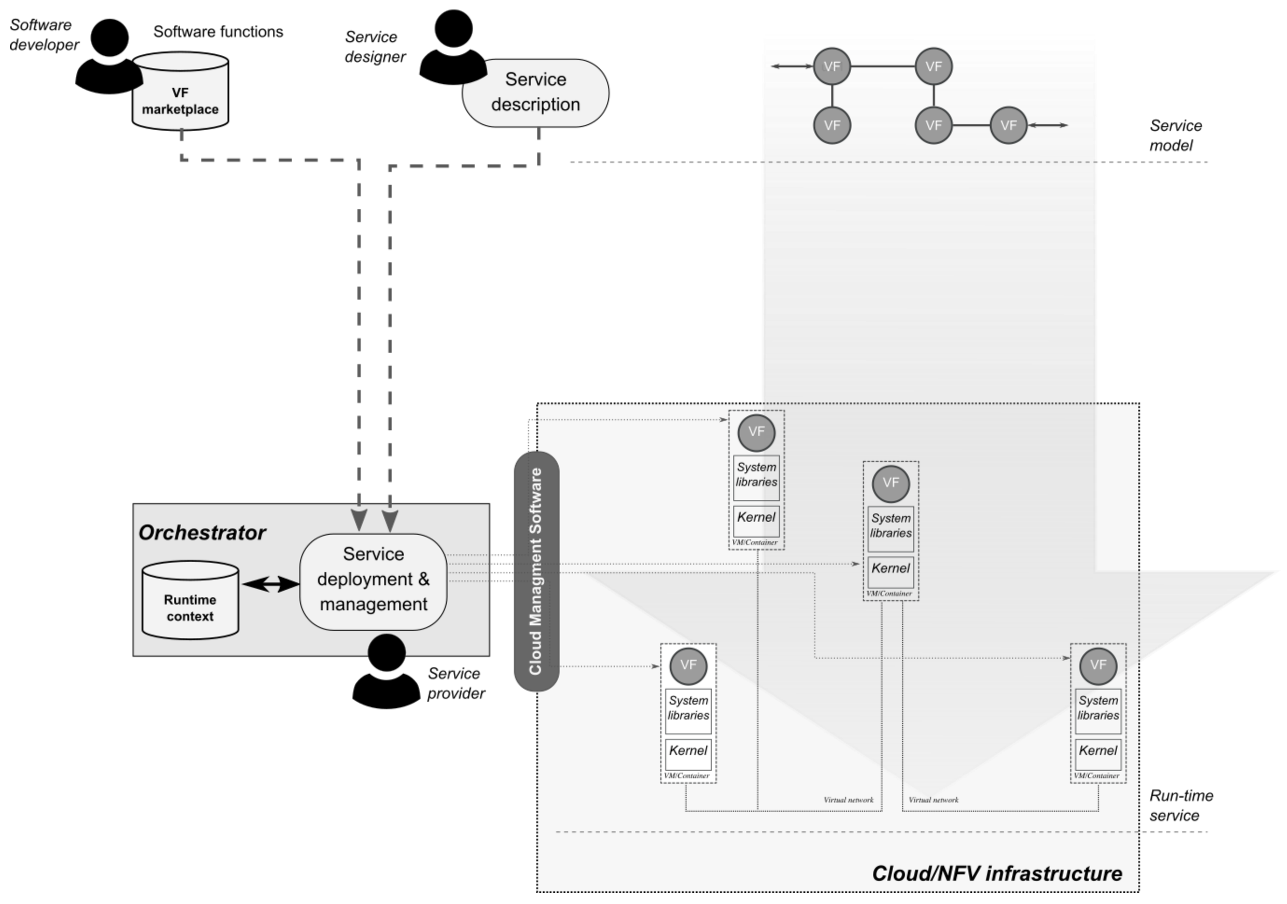


Figure 8. A typical architecture for software orchestration.

A service description includes the constituent elements of the service: virtual functions, relationships, deployment constraints, and management policies. The service topology is perhaps the most important part of this description, since it represents the logical relationships between the functions. The nodes of the graphs correspond to virtual functions. They can be given either as unambiguous references to software objectives (for instance: “Apache HTTP Server v. 2.4.38”, “PostgreSQL rel. 11.2”, “pfsense v. 2.4.4”) or more abstract function types or classes (for instance: “Web server”, “Relational database”, “Stateful Packet Firewall”). While the association between nodes and virtual functions is rather similar in different models, the relationships given by links may be very different. In TOSCA [5], the links represent configuration dependencies and imply a coherent configuration of some parameters at the endpoints. They can also indicate communication paradigms like client-server and publish-subscribe. For instance, one node may be an HTTP server and the other node a SQL database. A link between these components means: “HTTP server needs a SQL database”, and implies the configuration of database’s IP address and port, username and password, database name, etc. This approach is typical of Model-Driven Engineering and is mostly used for cloud applications. It is the model followed by orchestration tools as Juju[2], CloudML[3], and Cloudiator[4]. In the NFV domain, the links correspond to communication channels: physical transmission lines, virtual links, VLANs, MPLS links, GRE/IPSec tunnels, etc. They often imply the configuration of packet forwarding/filtering rules, based on the values of some header fields (destination, QoS, source, TTL, identification, tags, etc.) and external conditions (e.g., queue size, time, location). Both ETSI MANO [8] and IETF SFC [9] follow this model. A few examples of compliant orchestration tools include: OpenBaton[5], OpenNFV[6], and OpenMANO[7]. Some more details on orchestration models for the interested readers are given in Annex A.

Some models support the design of very generic templates for virtual services, allowing the definition of multiple topologies, scaling policies, management actions, and forwarding rules (this is the case for the ETSI NFV framework [7]). This is intended to sell a common service (e.g., virtual mobile network, M2M private network, VoIP service, VPN, gaming platform, Smart Grid) that may be tailored by service providers to their different customers. This business case is relevant for very common services and large vendors, with a potential market of thousands or millions of instances. However, the software industry has already demonstrated the remunerative of other business models, where many competitive services are designed, developed, and directly commercialized by a plethora of SMEs to their customers. In this second case, the worth of commercialization of such descriptions is marginal. Given the diversity of business models, the delivery of virtual services is not explicitly shown in Figure 8 for the sake of generality.

The diversity of orchestration paradigms, models, and standard is also reflected on the creation of virtual functions. As a matter of fact, the same software needs different metadata to be handled by the different orchestrators. We briefly recall that, in addition to identification and versioning, metadata are used to expose capabilities, configurable parameters, management hooks and scripts; these elements are very different for every orchestration model, as can be easily inferred by the very synthetic description above. Differently from virtual services, virtual functions are expected to be largely reused in different services. The need for the widest availability, the possibility to select the best implementation at run-time, the lower profitability in building this kind or products, and the larger expected competition motivate the presence of a public marketplace for trading virtual functions. As already noted in the previous Section, the selection and on-boarding processes may be fully automated or rely on human skills.

[2] https://jujucharms.com/.
[3] https://github.com/SINTEF-9012/cloudml.
[4] http://cloudiator.org/.
[5] https://openbaton.github.io/.
[6] https://www.opnfv.org/.
[7] http://www.tid.es/long-term-innovation/network-innovation/telefonica-nfv-reference-lab/openmano.

The main tasks for a service orchestrator are service deployment and management. Based on the service description and the policies set by the service provider, the orchestrator creates the execution environment and instantiates the software. The creation of an execution environment is facilitated by cloud paradigms and pay-as-you-go models, which provide management APIs for dynamic provisioning of computing, storage, and networking resources[8]. Open-source and commercial CMS (OpenStack, Microsoft Azure, Google Cloud Services, AWS, VMware vSphere) currently uses very different APIs, without considering the still evolving landscape for network slicing. However, the heterogeneity of the underlying virtualization infrastructure is totally managed by the orchestrator and does not have a direct impact on the ASTRID framework. What is indeed relevant for ASTRID, is the instantiation of the virtual service in the virtual resources.

As pictorially shown in Figure 8, each virtual function typically corresponds to a different execution sandbox. This facilitates management of the virtual functions. An execution sandbox may be a VM or a software container. The distinction is not relevant at this stage, so we will typically refer to VMs without any explicit preclusion of other forms of virtualization. The execution of a VF would likely require some libraries or daemons, and, in case of VM, an operating system. The deployment of the VF depends on the software distribution method. If the VF is packaged as bootable disk image, the orchestrator is only responsible to feed the CMS with such image. If the package only contains the executable code of the VF (or a link to it), the orchestrator has to boot a vanilla kernel (fulfilling the possible constraints in the VF package) and to install the required software and its dependencies (typically leveraging some package management tool). In any case, after deployment, the orchestrator is still responsible for the configuration of the VFs according to the service template specification. This may include IP addresses, port numbers, usernames and passwords, number of replicas and threads to activate, and so on.

Most of the automation implied by the processes described so far is already possible through configuration management tools as Puppet, ANSI, Chef. The distinctive trait of a modern orchestrator is the capability to monitor software execution and react to the evolving context. The necessary awareness is built by monitoring a number of different parameters, partially provided by the virtualization environment and the guest OS (for instance, CPU/RAM/disk/network usage) and partially provided by the VF itself (this may include, for example, number of processed requests, average service time, dropped requests, failures or internal errors, number of objects stored in a database, number of active users). The presence and structure of the former are known given the specific CMS/OS, whereas the presence and structure of the latter must be explicitly defined in the metadata of the VF. All this data feeds a *run-time context* database, which content may be further processed and compared with conditions (e.g., thresholds) to trigger management actions. The abstraction and implementation of life-cycle management is very different for different orchestration model and is not relevant for the definition of the ASTRID architecture, so we will not go into more details here.

6.2 ASTRID architecture

The ASTRID architecture is logically organized in three layers, corresponding to different operations and timing requirements. The layers are named data plane, control plane and management plane; they are taken from common networking architectures and are adopted to facilitate the identification of logical functions and the comparison with other frameworks (see Section 8). Figure 9 shows the relationship between the three planes and the main logical functions of the ASTRID framework.

[8] We recall that, despite of the existence of alternative cloud models, the ASTRID project only focuses on IaaS.

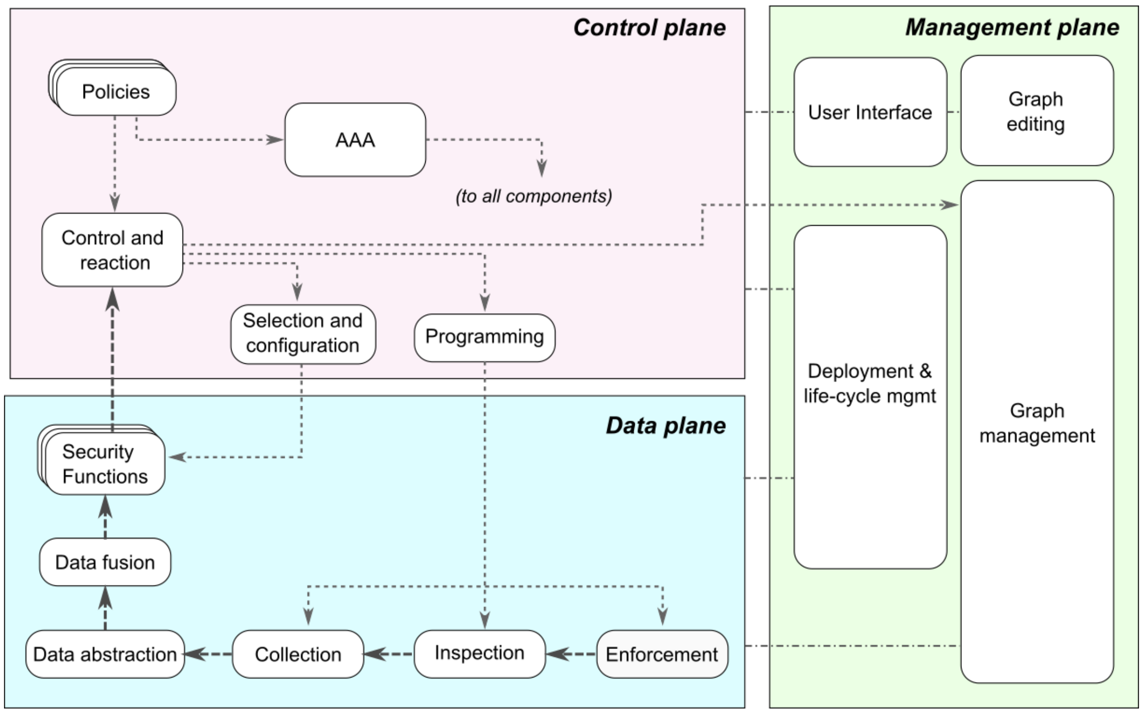


Figure 9. The three layers of the ASTRID framework.

At the bottom of the framework stack, the **data plane** creates and processes the security context (events, logs, statistics), including inspection, enforcement, collection, aggregation, and analysis operations. It operates in a deterministic way, according to imperative behavioural statements, configurations and parameters set by the control plane. In ASTRID, the data plane is partially implemented within the service, by embedding monitoring, inspection, aggregation, and enforcement hooks in the execution environment of the virtual functions, and partially implemented as external component, by abstraction, fusion, and analysis operations on the set of collected data.

On top of the data plane, the **control plane** takes decisions based on the evolving context. The control plane programs the data planes, selects security functions and configures them, reacts to system events according to behavioural policies set by the management plane, manages authentication, authorization and access control to the different components of the data plane. The scope of reactions is not limited to the data plane, but also extends to the management plane, in case some modifications to the running service are requested. In ASTRID, the control plane is mostly an external component, though some it is also partially present in the execution environment of virtual functions to configure the embedded security hooks.

Finally, the **management plane** is a side component alongside both the control and data planes. The management plane defines the behaviour of the system in the mid/long term. It includes tasks as instantiation and configuration of components in the data/control planes, translation of high-level intents and goals into commands and control policies, monitoring of operational parameters to detect deviations and malfunctioning, interaction with other systems/components. In ASTRID, the management plane includes a User Interface to interact with the whole system; this enables direct investigation of unknown or complex attacks. Graph editing is typically a manual operation to modify the original service description so to account for security features, as described in the pre-deployment scope (Section 5.2.1); it includes the definition of both additional components (data plane) and reaction policies (control plane). Graph management will provide the ability to modify the service at run-time, so to stop, mitigate, or react to attacks; this will largely leverage existing orchestration tools and will partially integrate with the BAU architecture shown in Section 6.1. Finally, deployment and life-cycle management takes care of those components that are not directly managed by the service orchestrator. For example, this may consist of management tools for big data infrastructures or cloud security services. This component might not be present when all ASTRID components are deployed and managed by the same service orchestration tools (see deployment and instantiation options in Section 6.6).

Based on the logical functions identified in each layer, Figure 10 shows the ASTRID architecture in terms of functional components and how it positions with respect to the BAU scenario. We point out the distinction between the ASTRID components and legacy orchestration tools. This architectural choice is motivated by the effort to ensure compatibility and portability to existing solutions. ASTRID remains agnostic of specific orchestration models: the general framework will be developed outside of any orchestration tool, while specific binding (acting like *drivers*) will allow interaction with those tools to dynamically adapt the graph to the evolving context and to react to incidents and attacks. To demonstrate the portability of the whole framework, different orchestration models have already been selected at the proposal stage for the Use Cases; the implementation will evaluate the effort to develop the specific drivers.

To help distinguish the ASTRID components from those already present in the BAU scenario, the latter are greyed. In the middle of the picture we find the BAU components. The left side of the picture depicts the ASTRID architectural components and their relationship, and the right side shows their implication on the service model and instance. The components of the run-time subsystem below the Security Controller (with the exception of the Idm) are part of the data plane, whereas the Idm, Security Controller and upper components are the control plane. In the execution environment we have both control and data plane for ASTRID agents, corresponding to configuration and processing tasks, respectively. The Security Dashboard, the service orchestrator, and the pre-deployment subsystem are part of the management plane. The last element is considered management because it defines security services that will be available at run-time but does not execute them.

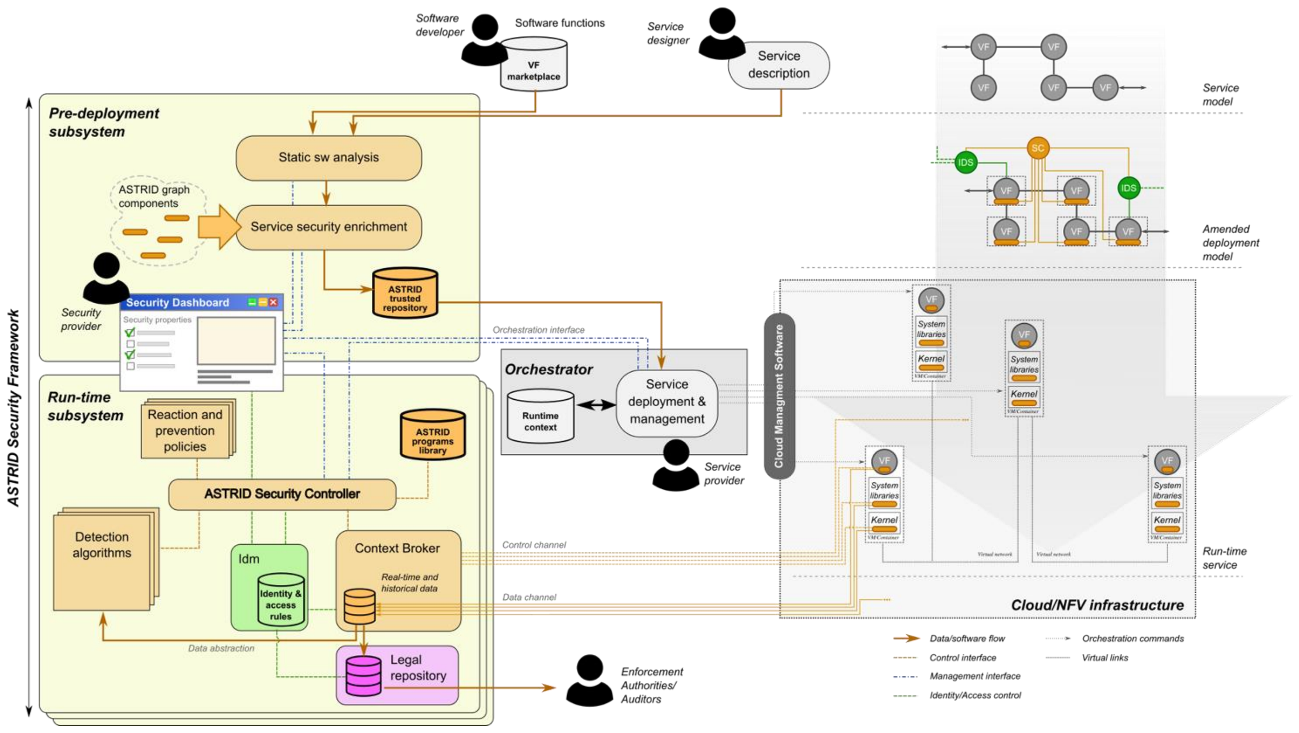


Figure 10. ASTRID architecture.

The preliminary consideration is that the two scopes of intervention in the conceptual workflow are reflected in the presence of two separate subcomponents. This separation is necessary to reflect the different level of automation that can be achieved in the two scopes, and to support multiple business models. The Security Dashboard is represented as a single component, but it entails a number of management functions that could be split in multiple tools for the different players; again, this should support multiple business models. These aspects will be analysed in WP5, where some possible business cases will be analysed and linked to exploitation plans for the Consortium partners.

According to the conceptual workflow, the description of virtual services and VFs are elaborated by the ASTRID pre-deployment subsystem, before being made available for deployment. This phase is largely transparent to service providers; the only meaningful difference is the place where such descriptions are retrieved from. As discussed in the previous Section, the pre-deployment subsystem is expected to modify the service graph, but its output must be compliant with the service model the orchestration understands.

Once the enhanced service has been deployed by the service provider, the run-time subsystem is automatically connected to the service and retains control over security features. Some remediation and mitigation actions can be implemented directly via the ASTRID framework (i.e., enforcement hooks), whereas topology changes on the service graph must be asked to the service orchestrator.

Finally, we note that the presence of the Security Dashboard in between the two subsystems also points out that they cannot operate independently. Indeed, the pre-deployment stage inserts additional metadata to the service, including security capabilities added to the graph, security policies to be fulfilled, and results from static source code analysis. This information is transparent to the service orchestrator, but must be shared in some way between the two subsystems.

6.3 Pre-deployment subsystem

The pre-deployment subsystem *enhances* the original service graph under the supervision of a security expert. Through the security dashboard, the security provider revises the whole graph and its individual components. The main purposes are the evaluation of the software and the inclusion of additional components for run-time control and management.

6.3.1 Software analysis

As aforementioned, to keep up with the amount of services that must be vetted for vulnerabilities, a hybrid approach is needed capable of performing vulnerability analysis of the virtual function software and executable both before its deployment but more importantly during run-time.

For the former, the most straightforward and immediate analysis of software versions is against already known vulnerabilities and exploits as have been identified in threat intelligence markets (e.g., Common Vulnerabilities & Exposure – CVE - Database). This essentially tries to identify "exploit signatures" that may be present in a software, prior to its deployment, and can be manipulated for launching more devastating attacks. Outdated, unpatched, or vulnerable software are sources of potential threats that must be compared with the final application prior its placement in the ASTRID trusted repository (Section 6.3.3).

In addition to this mechanism and in order to be able to verify the trustworthiness of the software to be deployed against a wider vector of vulnerabilities, ASTRID will also integrate more advanced dynamic analysis systems including "fuzzers" and concolic execution engines. Dynamic analysis systems [24], such as "fuzzers", monitor the native execution of an application to identify flaws. When flaws are detected, these systems can provide actionable inputs to trigger them. They require the generation of adequate "test cases" to drive the execution of the vulnerability analysis process. Concolic execution engines [25][26] utilize program interpretation and constraint solving techniques to generate inputs to explore the state space of the binary, in an attempt to reach and trigger vulnerabilities.

Despite the efficiency of such approaches, they are mainly used for software vulnerability assessment during the pre-deployment phase. However, a more challenging step is the ability to be able to also detect any vulnerabilities/attacks performed against the deployed functions and services during run-time. Towards this direction, ASTRID will investigate the provision of more advanced control-flow property-based attestation (CFPA) mechanisms (Section 6.4.5) aiming to perform an in-depth analysis of the running software behaviour; software that is running as expected by verifying the integrity of the entire control-flow of the service. This is useful per se to identify dangerous or unexpected behaviours, but it also feeds the run-time processes that monitor the execution and should verify the legitimacy of the responses to external stimuli.

In a nutshell, CFPA provides the ability to check any deployed function or service for any deviations from their normal execution flow; this allows for a deeper investigation of any abnormalities (that can be a result of an attack) without limiting the attack vector to only specific types of vulnerabilities. For this, however, what is needed is the identification and definition of adequate **Control-flow Properties & Policies** against which the CFPA will check the correctness of the execution flow of an application. We define a set of properties to be attested as the minimal required set of execution related attributes that will allow the verifier to confidently detect any control-flow deviations. These properties directly map to specific security-critical subsets of the application control-flow graph that need to be identified beforehand. They might include but not limited to: the current firmware version it is running, the version of its configuration file or presence of certain hardware properties, execution paths to specific memory regions, ports and network interfaces, etc. The aim of this approach is to reduce the size of the attested binaries, thus, reducing the strain and the computational overhead on the system, while sustaining a high level of assurance.

The output of this block is expected as additional metadata that reports:

a) An assessment of the appropriateness of the service for applications with different levels of criticality. The criticality of an application for the Smart Grid is very different from that of a web blog application.
b) The outcomes of static software analysis and the relationship between inputs and expected outputs.
c) The set of Control-flow Properties & Policies that will be used for attesting the integrity and correctness of the deployed application during run-time. Essentially these will include the set of: (i) the expected Control-flow Paths (CFPs) for the application in the sense that any deviation out of these CFPs will be flagged as suspicious, and (ii) specific properties to be measured during run-time for the successful completion of the attestation.

6.3.2 Service enrichment

Run-time monitoring, analysis, and protection of the virtual service relies on the availability of security enablers in the deployed graph. Service enrichment adds these components to the original graph, so that they can be automatically deployed by the service orchestrator.

The ASTRID architecture envisions the addition of the following types of components:

i. Log collectors.
ii. Execution hooks.
iii. Legal validators.
iv. Standalone cyber-security appliances.
v. Run-time subsystem.

Log collectors are agents that gather logs from applications, daemons, and the kernel, aggregate them, and fill a remote database. They hide the heterogeneity of formats and interfaces, can filter and aggregate data, and sometimes provide data fusion capabilities. They are primarily co-located with VFs (and depicted as orange ellipses in Figure 10), but some components for data aggregation might be deployed as independent VFs. There is no need to explicitly specify the remote database in the graph, since this will be part of the run-time subsystem. D1.1 reported many frameworks already available for this purpose; from an architectural perspective, the main guideline is to select agents that support remote programmability at run-time.

Execution hooks are triggered upon invocation of internal functions and access to system resources. One primary interest is the analysis of network traffic, which represents one of the main threat vectors. In this respect, among the variety of data plane technologies listed in D1.1, the eBPF emerged as the best technology for ASTRID, given its uncommon characteristic to be linked to any kernel function, including system calls and network I/O. The eBPF is fully programmable, and this represents another priority requisite for dynamic adaptation of monitoring and inspection tasks. In the same way of log collectors, the eBPF must be co-located with VFs as well (again, the orange ellipses in Figure 10, this time limited to kernel scope). The eBPF is implemented in the kernel, so from the service description perspective this turns into a deployment constraint (for instance, the selection of the right bootable image). Control of the eBPF is done through external agents, which must therefore be included alongside the VF.

Legal validators are responsible for timestamping, encryption, integrity, digital signing and any other operation intended to give legal validity to data and events. The validation process may introduce latency and overhead in the collection process; in this case, it should only be limited to relevant aspects to fulfil legal obligations (e.g., on privacy) and to be used as evidence in court.

Standalone cyber-security appliances are the virtual versions of legacy firewalls, IPS/IDS, antivirus, etc. The deployment of this appliances consumes more resources than the embedded elements, and thus should be avoided as much as possible. Indeed, the provisioning of one additional VM with its own kernel and network interface would require additional CPU, RAM, disk, and network resources; instead, embedded components as log collectors and eBPF are expected to have negligible resource consumption compared with VFs[9]. However, they might be necessary to mitigate or investigate in depth particular attacks, so we do not completely preclude their (maybe limited and temporary) usage.

Finally, the **run-time subsystem** may be deployed at run-time as well, to dynamically instantiate a fully working ASTRID environment (orange circle), as further elaborated in Section 6.6. Clearly, the deployment of the run-time subsystem as part of the service graph must be subject to sever deployment constraints on the trustworthiness, dependability, and security of the underlying virtualization infrastructure.

6.3.3 ASTRID trusted repository

This repository collects virtual functions and services compliant with the ASTRID framework. It is equivalent to any other VF/service marketplace, so any service provider that wants to use the ASTRID security framework has only to select the ASTRID repository.

According to best practice for software distribution, the ASTRID trusted repository should provide integrity mechanisms as checksum computation and digital signs. The certification of services and VFs is already envisioned by some specifications (for instance, see Clause 6.2.5.2 of [7]).

6.4 Run-time subsystem

The run-time subsystem is the "smart" core of the ASTRID framework, bringing autonomicity and dynamicity to more traditional detection frameworks. In this respect, it represents the implementation of a security orchestrator. The run-time subsystem controls and programs the monitoring and

[9] The verification of this assumption will be part of validation and performance analysis of the ASTRID framework.

inspection agents in the service graph, collects data and measurements, feeds detection algorithms, and supports human staff with semi-autonomous decisions. The interaction among the different components in the run-time subsystem is driven by a common representation of the available data and capabilities, which are collectively denoted as the "security context model." The security context model may include the following data:

- *security logs* generated by the virtual functions;
- *security events* generated by the virtual functions;
- *encryption capability* to protect both user data as well as control data exchanged within the ASTRID security framework;
- *trust and privacy requirements* for connecting and exchanging data with other functions;
- *identity and access control capability*, that can be used to authenticate external entities, set access rights, trace commands, etc.;
- *certification, timestamping, and digital signature capability* that may be used to guarantee the origin and integrity of security data generated by the function;
- *traffic mirroring* and other replication capabilities that can be used for legal interception of data flows;
- *retained data* that may be used for criminal investigation;
  subsets of the application's *control-flow execution graphs* when the attestation of the software correctness and integrity fails; as a result of a potential software compromise.

6.4.1 Execution environment

Figure 10 shows how the enriched service description results in the deployment of additional components at run-time in the execution environment (indicated as orange ellipses in the pictorial representation of the service topology in the Cloud/NFV Infrastructure). We remind that the installation and configuration of the execution environment remains a prerogative of the service orchestrator, which at deployment time takes care of installing all the libraries, proxies, and agents required by the security enrichment.

ASTRID components in the execution environments are in charge of monitoring and inspection tasks at different layers of the software stack. They include:

- Kernel hooks (i.e., eBPF programs), that work in the kernel and monitor network traffic and system calls[10]. As previously noted, the eBPF is implemented in the kernel, but needs a userland control agent for remote control and exportation of the data. The kernel component is an execution virtual machine to run eBPF programs. The latter are injected through the control agent.

- Log agents that work in user-space and collect logs from libraries, daemons, and even the kernel, so their scope extends to the whole stack.

- VF-defined monitoring information is conceived to drive orchestration actions[11], but might also be used for detection purposes. They are built-in in the software of the VF, not part of the ASTRID enrichment process. Subscription and configuration to these sources are specific for each orchestration model and can be managed by a set of plug-ins.

Additional components or functions in the execution environment are expected to give legal validity to the collected information. They could be deployed as standalone agent and inserted in a local processing pipelines, or directly integrated in other agents (for example, they may be programs part of

[10] eBPF programs can be attached to *any* kernel function, not limiting the scope to system calls and packet handling. However, such scope is considered enough for the objectives of the Project.
[11] As an example, we can mention the VnfIndicator and MonitoringParameter information elements in the ETSI VNF descriptors (see Clause 7.1.11 of [10]).

an eBPF chain). The exact nature of these components and their integration with the other local agents will be specified in the updated version of the ASTRID architecture, based on the outcomes from Task 2.5.

The interaction with the execution environment also requires the setup of communication channels, for both control and data messages. These communication buses between virtual functions and the ASTRID run-time are secure channels deployed by the service orchestrator. There are multiple implementation alternatives envisaged at this stage:

I. *In-band*. The simplest approach would be to use virtual network(s) in the IaaS already used for communication between the VMs. This does not require any specific effort on the orchestration side, but the traffic for data collection may overwhelm the network and should be recognizable so as to not alter traffic statistics for detection purposes. In addition, though the usage of encryption is taken for grant, there are still security concerns in mixing the management and service data traffic.

II. *Out-of-band in the service graph*. The instantiation of a dedicated virtual network for the control and data channel looks a better approach with minimal overhead on service management. In this case, the enrichment process must envision an additional virtual NIC for each VM and a service-wide flat network connecting all VMs and the run-time subsystem. The need for separate virtual networks for the control and data channels will be considered in the technical activity and reported in the updated architecture.

III. *Out-of-band in management interfaces*. The underpinning assumption for automatic life-cycle management through service orchestration is i) the access to monitoring and context information about the execution of VFs and the network service, and ii) the possibility to interact with VFs to trigger management scripts. That means a management channel must be available outside the execution environment. An example of management channel is the Ve-Vnfm interface in the ETSI MANO architecture [8], which may correspond to the control network of OpenStack or a physical network for Docker containers.

6.4.2 ASTRID Security Controller

The Security Controller represents the most valuable part of the run-time subsystem, conceived to automate as much as possible the behaviour of the whole framework. It positions between the reaction and mitigation policies and the context, and orchestrates security functionalities. We intentionally avoid the term "orchestrator" because it works at the control layer, while service orchestration involves typical management operations. Overall, we consider the whole ASTRID run-time subsystem as security orchestrator. Moreover, this nomenclature is consistent with other initiatives in this field (see Section 8).

Overall, the Security Controller will work according to an Event-Condition-Action pattern, where events are triggered by detection or management entities, conditions come from the current context (graph topology, security configurations, data and events), and actions entails modifications of the security hooks (monitored data, frequency, granularity, filtering, marking, etc.), re-configuration of the detection algorithms, changes in the service graph. ECA rules are expressed by policies, which represent the real "smartness" of the Security Controller and encompass both reaction and prevention actions. According to this description, the role of the Security Controller is comparable to an SDN controller, which mediates between network applications (reaction and prevention policies, in our cases) and the underlying data plane.

As briefly mentioned above, the control scope of the Security Controller is rather broad and encompasses both the management and data planes. With respect to the management plane, the Security Controller works in tight cooperation with the service orchestrator. It expects the description of the grounded graph (including the actual number of instantiated virtual functions and networks), as well as notification of relevant events, including initialization, start, scale, topology change, stop, removal. The Security Controller also sends indication about VF that got compromised, vulnerabilities found in some components or their configurations, topology changes (insertion/removal/replacement of virtual functions), configuration changes (insertion/removal/replacement of security functions), and so on. The Security Controller may send commands directly to the service orchestrator or to the Security Dashboard, depending on the operational mode (autonomous, supervised, manual).

With respect to the data plane, the Security Controller instantiates and configures detection and analysis algorithms. For instance, if a service only requests protection against remote access, the Security Controller will instantiate a firewall module. If the service requests protection against DoS, the Security Controller will instantiate both a DoS detection algorithm and a DoS mitigation module. If the service requests protection against misuse, the Security Controller will instantiate an IPS algorithm, and so on. The Security Controller also re-program the execution environment, leveraging the abstraction offered by the Context Broker. The main technical challenge here is the definition of common actions at the policy level and their translation into configurations and code for the heterogeneous set of security hooks. In the current architecture, this will be realized by selecting pre-defined programs and configuration files from an internal library, but the long-term ambition would be the definition of dynamic code generation and run-time compiling.

We can give a concrete example of how the Security Controller is expected to behave in case of DoS service. Detection of volumetric DoS is typically based on analytics on the network traffic. Since deep inspection of the traffic leads to high computational loads and latency, an initialization policy only requires statistics about on the aggregate network traffic that enters the service, which may be collected by standard measurements reported by the kernel. The same policies also initialize an algorithm for network analytics and sets the alert thresholds. Upon detection of an anomaly in the traffic profile, an event is triggered and the Security Controller invokes the corresponding DoS policy. The policy now requires finer-grained statistics, and the Security Controller selects an eBPF filter for packet classification, installs and configures it. The policy also requires the detection algorithms to work with the broader context information available. As soon as the analysis comes to a new detection, it triggers a new alert, this time including the relevant context (i.e., identification of suspicious flows, origins, etc.). Before taking the decision about how to react, the mitigation policy may evaluate some conditions to check if the suspicious flow comes from an expected user of the service, has been previously blacklisted or whitelisted, is acceptable based on previously recorded time series. The actions to be implemented (e.g., dropping all packets, dropping selected packets, redirecting suspicious flows towards external DoS mitigation hardware/software, stop the service, move part or the whole service to a different infrastructure) is therefore notified to the Security Controller, which again translates them in a set of commands for the external service orchestrator and/or configurations and programs to be installed in the execution environment. Notifications about the detected attack and the implemented actions are also sent to the Security Dashboard. In this respect, we recall that the Security Controller can work in fully-automated, semi-automated, or supervised mode.

6.4.3 Context Broker

One of the main distinctive characteristics for the ASTRID framework is *programmability*, that is the capability to shape the depth of inspection according to the current need, in both spatial and temporal dimensions, so to effectively balance granularity of information with overhead. As already noted, this goes beyond mere re-configuration of individual components and their virtualization environments. This would change the reporting behaviour by tuning parameters that are characteristics of each app (logs, events), network traffic, system calls (e.g., disk read/write, memory allocation/deallocation), RPC toward remote applications (e.g., remote DB). Indeed, programmability also includes the capability to offload lightweight aggregation and processing tasks to each virtual environment, hence reducing bandwidth requirements and latency.

The first task for the Context Broker is to manage the heterogeneity of sources and protocols, which is reflected in different data and control interfaces. The Context hides this heterogeneity and exposes a common **data model** to the other components in the run-time subsystem, both in the data and control planes, for discovering, configuring, and accessing the security context available from the execution environment. The Context Broker implements most of the data plane functions in the run-time subsystem (see Figure 9): data collection, data storage, data abstraction.

Data collection. The Context Broker collects data from monitoring and inspection processes deployed in the execution environment (*data channel* in Figure 10). The Context Broker hides the heterogeneity and asynchrony of the sources, organizes historical data, and provides simple querying and fusion capabilities in data access. The Context Broker may also separate data with legal validity from the main flow, if such information must be kept in a distinct repository so as to fulfil normative requirements (Task 2.5 will provide a definitive answer to this issue). The Context Broker also offers a homogeneous control interface for configuring and programming different data sources, by implementing the specific protocols (*control channel* in Figure 10). Since the implementation of all possible protocols clearly falls outside of the Project scope, the Context Broker must follow a modular architecture, where different sources can be supported by developing simple extensions (i.e., plug-ins).

Data storage. Given the very different semantics of the context data, the obvious choice is non-relation databases (NoSQL). This allows to define different records for different sources, but also poses the challenge to identify a limited set of formats, otherwise part of the data might not be usable by some detection algorithms. The validity and volume of data affect the size of the database and the need for scalability. Local installations are suitable when the data are kept for days or months, but cloud storage services may be necessary for longer persistence or larger systems. On the other hand, remote cloud storage is not suitable for real-time or even batch analysis. Another design issue is the possibility to scale-out horizontally and/or inborn support for parallel processing and big data analytics, if data volume becomes large.

Data abstraction, fusion, and querying. The flexibility in programming the execution environment is expected to potentially lead to a large heterogeneity in the kind and verbosity of data collected. For example, some virtual functions may report detailed packet statistics (i.e., those at the external boundary of the service), whereas other functions might only report application logs. In addition, the frequency and granularity of reporting may differ for each VF. The definition of a (security) context model is therefore necessary for detection algorithms to know what could be retrieved (i.e., *capabilities*) and what is currently available, how often, with each granularity (i.e., *configuration*). Correlation of data in the time and space dimensions will naturally lead to concurrent requests of the same kind of information for different time instants and functions. In this respect, searching, exploring and analysing data in graph databases should be considered as implementation requirement. Indeed, unlike tabular databases, they support fast traversal and improve look up performance and data fusion capabilities. Finally, the last implementation requirement is the ability to perform quick look-ups and queries, also including some forms of data fusion. That would allow clients to define the structure of the data required, and exactly the same structure of the data is returned from the server, therefore preventing excessively large amounts of data from being returned. This could turn very useful during investigation, when the ability to understand the evolving situation and to identify the attack requires to retrieve and correlate data beyond typical query patterns.

The security context retrieved by the Context Broker contains a lot of information about service usage patterns, users, exchanged data, and so on. Access to this data should therefore be limited to authorized roles and algorithms. In addition, configuration of the remote data plane must remain a prerogative of the security controller and trusted policies, so it is important to track the issuer of such commands. The Context Broker is therefore expected to enforce access policies settled by the Idm module.

6.4.4 Programs library

The ASTRID programs library is a collection of software that can be injected into the programmable hooks present in the execution environment. Different languages can be used by different hooks: ELF binaries, java bytecode, python scripts, P4/eBPF programs. Such programs are written and compiled offline, and then pushed in the repository by the Security Dashboard. They also include metadata for identification and description, so to be easily referred by the ASTRID Security Controller. The scope of ASTRID programs includes monitoring, inspection, and enforcement actions; it is clearly limited by the instruction set of the execution virtual machine (if any).

From a security perspective, the current architecture assumes that the ASTRID programs are safe. This is implicitly guarantee, for example, for the eBPF, where the code is executed within an execution sandbox. In case of general-purpose languages, the correctness and safety of the source code might be verified by static source-code tools, similarly to what happens for virtual functions[12].

6.4.5 Detection algorithms

Detection algorithms process, analyse and correlate security-related data and events. They can be mapped to existing security functions (DoS detection, IPS, IDS, antivirus, etc.). They are fed by the Context Broker, rather than implementing their own monitoring and inspection hooks. The availability of heterogeneous data from multiple sources theoretically allows the detection of any kind of threats and attacks, including the typical scope of host-based, network-based, and hybrid IDSes, and antiviruses. From a practical perspective, however, the real range of algorithms will be limited by the possibility to find an acceptable trade-off between the complexity to implement local inspection and the communication overhead. This aspect will be explicitly considered in the demonstration and validation phase of the Project.

The ASTRID architecture supports both existing detection tools as well as the design of innovative algorithms, leveraging the availability of tailored data and information from the whole graph. From an architectural perspective, each algorithm will only be required to implement the interfaces towards the Context Broker and the Security Controller. For existing tools, this could be achieved by developing plug-ins or adapters. The interface to the Context Broker will be used to retrieve relevant information, including both real-time and historical data. This interface will allow selective queries to return aggregate of data, with respect to virtual functions and time periods. The interface to the Security Controller is used to notify security events like threats and attacks, that may trigger some forms of reaction. The description of the event may include an estimation of the accuracy of the detection, so to trigger the collection of more detailed information; alternatively, this information could be retrieved by evaluating specific conditions on the current security context.

According to the proposal, at least two algorithms will be proposed to show the effectiveness and to evaluate the ASTRID framework. One algorithm will analyse the run-time traces of system calls for detection of unexpected execution flows (*Control-flow Property-based Attestation*), while one algorithm will look for anomalies in the network traffic flows.

Control-flow property-based attestation (CFPA) proposes another view towards verifying the integrity of only those critical software functionalities. The contributions of such a CFPA-based architecture are twofold: first of all, ASTRID will use advanced tracing mechanisms for extracting the control-flow graph of a service (i.e., extended Berkeley Packet Filters (eBPFs)) that provides high performance with minimal overhead to the execution times of the system. Second, ASTRID will redefine the attestation process by checking only the security-wise critical functions (instead of the whole

[12] This last option does not fall under the Project objectives, so it is not explicitly further discussed in the ASTRID architecture. Anyway, its implementation does not significantly affect the architecture, since it only requires the presence of additional tools for software verification.

application), resulting in a novel, scalable by design, lightweight, control flow attestation scheme (more detailed description of the proposed scheme will be provided in the context of WP3).

The insight behind this approach is that we do not need the attestation of the entire system but only the execution properties of the security sensitive functionalities that are running on the system. The identification of which functionalities should be monitored is implementation dependent and depends on the entire service graph. The properties to be attested will be identified during the pre-deployment phase and will be included in the Control-flow Properties and Policies that will accompany the entire service graph. The aim of this procedure is to check both behavioural properties and low-level concrete properties about the entity's configuration and execution, such as the current firmware version it is running, the version of its configuration file or presence of certain hardware properties, integrity of measurements, execution paths to specific memory regions, ports and network interfaces, etc. Furthermore, some of these properties might need to be attested individually while others might require to be approached as a system of systems that need to be attested by the involved devices as a group. Overall, CFPA will enable ASTRID to provide an efficient mechanism to verify software- and system-integrity (during run-time) in order to detect any attempt to modify the execution of the loaded application; even in the case of attacks that were not anticipated (or have not been seen) before.

6.4.6 Reaction and Prevention Policies

Policies define the behaviour of the system. Conceptually, reaction and prevention policies do not implement inspection, detection or enforcement tasks, so they do not correspond to any existing security function (IDS/IPS, antivirus, VPN). Instead, they represent an additional upper layer for control of security services. Policies are therefore used to automate the response to expected events, avoiding whenever possible repetitive, manual, and error-prone operations done by humans. Based on the Project scope and objectives, the usage of ECA patterns for expressing the logic behind a policy looks an effective choice to achieve tangible results in the short period and to cover a broad range of interesting cases. The definition of an ECA policy requires at least 3 elements:

- an Event that defines when the policy is evaluated; the event may be triggered by the data plane (i.e., detection algorithms), the management plane (i.e., manual indications from the dashboard, notifications from the service orchestrator), or the control plane (i.e., a timer);
- a Condition that selects one among the possible execution paths; the condition typically considers context information as data source, date/time, user, past events, etc.;
- a list of Actions that respond, mitigate, or prevent attacks. Actions might not be limited to simple commands, but can implement complex logics, also including some form of processing on the run-time context (e.g., to derive firewall configuration for the running instance). They can be described by imperative languages, in the forms of scripts or programs.

The range of possible operations performed by policies include enforcement actions, but also re-configuration and re-programming of the monitoring/inspection components in the execution environment. Enforcement and mitigation actions are mostly expected when the attack and/or threat and their sources are clearly identified and can be fought. Instead, re-configuration is necessary when there are only generic indications, and more detailed analysis could be useful to better focus the response. A typical example is a volumetric DoS attack. To keep the processing and communication load minimal, the monitoring process may only compute rough network usage statistics every few minutes. This is enough to detect anomalies in the volume of traffic but does not give precise indication about the source and identification of malicious flows to stop. Re-configuring the local probes to compute per-flow statistics or more sophisticated analysis helps implement traffic *scrubbing*[13].

A special case of prevention is the **initialization** of the programmable components, that should be triggered once the service orchestrator has deployed the graph and prior to activation and connection to the public Internet. Initialization creates the initial configuration of the execution environment, including monitoring and enforcement tasks, by applying the security policies to the current run-time instantiation of the service template. For example, a typical initialization task is the generation of firewall rules based on connectivity requirements that can be inferred by the graph description and explicit rules from the service designer. This configuration shall be applied after the graph is deployed but before it is started, so it is important to receive the corresponding notifications from the service orchestrator.

The adoption of advanced reasoning models, even based on machine learning and other forms of artificial intelligence, is clearly a very promising yet challenging target to automate the system behaviour. This would open the opportunity for dynamically adapting the response to new threat vectors. In this respect, the historical analysis and correlation of the events and conditions with the effects of the corresponding actions from existing policies or humans would provide useful hints to assess the effectiveness of the latter, so to identify and improve the best control strategies.

6.4.7 Legal repository

The legal repository keeps records of activities from end users, as required by the normative framework. This may include the logs of calls for a VoIP or videoconferencing service, the list of uploads/downloads for a file sharing service, and so on. The legal repository is depicted as a separate component to underline the fact that some constraints may apply to its implementation. For instance, the legislation may force a specific technology, physical location and restricted access, etc.

According to preliminary outcomes from Task 2.5, the legal responsibility for the activities carried out within a given service is assigned to the Service Provider. He is in charge of collecting records that can unambiguously identify the source and time of relevant events. For example, for a VoIP application the Service Provider may record the time, duration, caller, and callee of each call; upload times, uploader, and access times may be kept for a file sharing service. One important aspect is that not all information can be collected by the ASTRID framework. The analysis of binary files to detect viruses and other malware for a file sharing service can likely be performed through some ASTRID agent; however, registration of other events and properties more related to the internal business logic (e.g., user identity and start/end time of a call) are expected to come from the same virtual functions. In this respect, it is important to harmonize all sources in a common framework. The topic will be investigated in detail in T2.5, and the revised architecture will reflect the main outcomes from that task.

6.5 Security Dashboard

The Security Dashboard is the main management tool of the ASTRID framework. It is used to interact with both the pre-deployment and run-time subsystems. For example, it can be used to edit the service templates for enrichment, to select specific software analysis, to visualize anomalies and security events and to pinpoint them in the graph topology, to set run time security policies, and to perform manual reaction. With respect to the last two options, we point out that security policies are the best way to respond to well-known threats, for which there are already established practice and consolidated methodologies for mitigation or protection. However, the identification of new threats and the

[13] Scrubbing is a technical term used to indicate a cleansing operation that analyses network packets and removes malicious traffic (ddos, known vulnerabilities and exploits). It is usually implemented in dedicated devices or infrastructures, able to sustain high volumetric floods at the network and application layers, low and slow attacks, RFC Compliance checks, known vulnerabilities and zero day anomalies.

elaboration of novel countermeasures require direct step-by-step control over the on-going system behaviour. The dashboard interacts with the orchestration system to give security provider full control over the graph in case of need.

The Security Dashboard is not a mere GUI; indeed, it implements a common and homogeneous interface to all manageable system components. As a matter of fact, we can better define the Security Dashboard as an API, which may be invoked by multiple roles; control access will limit the functionalities available to each user. This is necessary to support multiple business models and to provide awareness to different stakeholders. For example, some API may be devoted to set high-level security policies by the Service Provider. He may ask for "firewalling" services, intrusion detection, malware protection, and so on. Another example is the notification of attacks or anomalies to the Service Provider or end user, so they can be aware of a potential threat to their data or to service continuity. When the Security Provider and the Service Provider roles are implemented by different departments of the same organization, it may be convenient to use a common interface with different usage rights. This facilitates the extensions of existing orchestration tools. For example, the Security Dashboard may be implemented in an additional tab of the existing service orchestrator GUI, already including service design, selection of virtual functions, service deployment and management.

We expect a GUI to be available at least for the Security Provider, but some partners may wish to integrate the Security Dashboard APIs in their own orchestration GUI, depending on their exploitation plans.

6.6 Deployment and instantiation options

The creation of a working ASTRID environment requires the deployment and instantiation of a number of complementary components: the pre-deployment subsystem, the run-time subsystem, and the Security Dashboard. Once more we remark that the two operational scopes might not be present in all business cases, so the deployment options for the above components can be analysed separately.

The **pre-deployment subsystem** is expected to be installed manually. Based on the following considerations, we will not define specific interfaces or integration requirements for this subsystem, which will be developed by each interested partner according to its exploitation and business plans:

- multiple implementation options are possible, with tight, loose, or no integration with other software development and orchestration tools;
- it may also be (at least partially) integrated in the security dashboard, when the Security Provider covers both scopes;
- specific drivers are required to edit different formats of service templates;
- standard service descriptions are not available for all orchestration models (see Annex A);
- it may consist of a mix of both automatic and manual operations for the different phases; for instance, the selection of VFs from the marketplace may require direct intervention from the Supply Chain and Service Acceptance specialists.

The **run-time subsystem** is conceived to be the core asset of the ASTRID project. According to the design described in Section 6.4, it is a modular subsystem, where additional detection algorithms and reaction policies can be deployed dynamically. The implementation of a demonstration prototype is therefore mandatory for the project, to show the feasibility of the proposed solution and to carry out performance evaluation. From the perspective of a Security Provider, there are a few deployment and instantiation options for the ASTRID run-time subsystem, as described in the following.

The run-time subsystem may be pushed as **additional component in the service graph** as part of the enrichment process, for automatic deployment and instantiation as a sort of virtual security function over a virtualization infrastructure. In this case, the service orchestrator deploys both the original service and the ASTRID virtual function.

Alternatively, the run-time subsystem may be **deployed apart in a dedicated infrastructure**, also including big data techniques for efficient implementation of advanced analysis algorithms. In this case, the same instance can be shared by multiple services, so to improve resource utilization and the base for analysis and correlation (e.g., trigger an event when similar anomalies occur in more than one service, apply prevention actions to all services upon detection and identification of an attack to one service). The service orchestrator still remains in charge of instantiating the security hooks embedded in the components of the service graph. It also needs a mechanism to automatically connect such local agents to the external framework, including the configuration of IP addresses, secure communication channels, authentication and control access, and so on. The most straightforward solution would be the insertion in the service graph of a sort of meta-component for the run-time subsystem, which only gives information about how to connect to an external function, i.e., parameters to set up a virtual private network (which may be an overlay over the Internet or a network slice), network addresses and ports of the Context Broker, secret materials for encryption and authentication. The same solution could also be applied to connect the run-time environment with the external Security Dashboard.

Independently of the deployment choice, dependability of the whole system relies upon integrity, availability, and trustworthiness of the ASTRID components. The selection of trusted infrastructures and the usage of remote attestation procedures based on TPM/TEE should therefore be enforced by deployment constraints, so to ensure the integrity of the boot process and the confidentiality of the overall system operation. Though attestation of the service topology and the run-time subsystem is not explicitly envisioned by the project workplan, it is a highly desirable feature for commercial exploitation by partners.

The **Security Dashboard**, similar to the pre-deployment subsystem, must exist before and independently of the deployment of the service graph, and is expected to be tailored to different business models. As a matter of fact, it may be a standalone tool for Security Providers, a component of the service orchestrator for the Service Provider, or an intermediary solution with components in both domains (i.e., to collect high-level policies from the Service Provider and give control to the Security Provider). It will not be part of the enrichment process and will not be deployed automatically by the service orchestrator. Instead, tailored integration with existing or complementary tools is expected on a case-by-case base.

6.7 Integrated ASTRID platform

The design and operation of secure cloud and NFV services according to the ASTRID conceptual business chain and workflow, as shown in Figure 5 and Figure 6, respectively, is a mix of technical tools and management procedures that can be implemented in-house or externalized. The role of Security Provider may involve multiple sub-roles or functions in different business domains (i.e., service design, software development, service provider, security consulting), raising the opportunity for multiple business models.

The main goal of the ASTRID project is the development of the enabling technologies envisioned by the overall architecture, which are within the run-time scope. The integrated ASTRID platform will therefore deliver a fully functional run-time environment, including the Security Dashboard. Release options will include source code and compilation instructions for sure, but might also include cloud images for evaluation. The pre-deployment system will not be released as standalone component, though alternative integration options will be investigated in the two Use Cases. The detailed architecture, logical components, and technologies of the integrated ASTRID platform will be reported in D1.3.

Indeed, the realization of the overall ASTRID architecture largely depends on the integration with existing components and tools, primary for service orchestration but also for mitigation and risk assessment. For instance, the Security Dashboard can be a standalone component, but some partners may prefer to integrate it in their existing GUIs. In the same way, the graph enrichment may be integrated in the Security Dashboard or leverage existing tools for editing service templates. Again, user policies may be collected by manual procedures (i.e., contracts, emails) or by existing management tools (e.g., an orchestration GUI, a customer application, the Business Support Systems / Operations Support Systems of network Service Providers). Each partner might therefore provide its own implementation of a pre-deployment system as part of its exploitation and dissemination strategy.

The list of additional components and/or integrated platforms that will be released in addition to the foreseen official ASTRID framework will be reported in a following exploitation deliverable.

6.8 Design and implementation plans

The design and implementation of the discrete components envisioned by the ASTRID architecture will be part of the technical activities in WP2/3. Table 2 lists the specific Tasks that will be in charge of each architectural component.

Table 2. Mapping of discrete architectural components to technical activities.

Subsystem	Component	Task
Pre-deployment	Static sw analysis	3.1
	Service security enrichment	2.4
	ASTRID trusted repository	2.4
Run-time	ASTRID Security Controller	2.4
	Reaction algorithms	2.4
	Context Controller	2.1
	Context Broker	2.1, 2.2
	Local monitoring and inspection	2.1
	Local legal validation	2.5
	Legal repository	2.5
	Communication channels	2.4
	Identity management and access control	2.3
	Detection algorithms	3.2, 3.3
	Data repository	2.2
GUI	Security Dashboard	3.4

7 Requirements

This Section briefly maps the architectural components to the functional requirements identified in D1.1. Table 3 lists all the requirements for the ASTRID framework and indicates how they are fulfilled by the ASTRID architecture. The first three columns report the identifier (ID), priority (P), and name (Name) of the requirement. The fourth column (F) indicates whether the requirement has been completely fulfilled by the architecture (✓), it has been taken into account but requires further consideration at a further stage – for example, implementation or validation (★), cannot be fulfilled by the current architecture (✗), or must be evaluated at a later stage (-). The last column (Details) explains how the architecture fulfil the requirement and what steps are still required during research and implementation. The list will be updated in upcoming deliverables based on the result of research and implementation activities.

Table 3. Fulfilment of function requirements identified for the ASTRID framework.

ID	P	Name	F	Details
R001-CS-DSG	T	Data correlation over the whole graph	✓	The Context Broker collects data and events from all service components and feeds a centralized set of detection algorithms.
R003-CS-DSG	T	Local processing and programmability	✓	The Context Broker implements a control interface to local security agents. The local security agents developed in WP2 must be dynamically programmable/configurable. The eBPF will be used as main technology to run lightweight inspection and enforcement programs locally.
R004-CS-DSG	T	Secure communication	★	Secure communication will be implemented in the control and data channels between the Context Broker and local agents, between the Security Dashboard and the Security Controller, and in the API of the service orchestrator. Investigation on attribute-based encryption is foreseen in WP2.
R007-CS-DSG	T	Access control	★	The methodologies, algorithms, and tools for access control will be defined in WP2.
R008-CS-DSG	T	Repository of trusted security programs	✓	An ASTRID security program repository is present in the run-time subsystem. The deployment of this component in trusted infrastructures and the usage of access control mechanisms will ensure the origin and integrity of such programs.
R009-CS-DSG	T	Collection of detection algorithms	★	The abstraction provided by the Context Broker allows to run multiple detection algorithms in parallel. Some algorithms will be developed in WP3 for evaluation purposes.
R010-SD-DSG	T	Integration with orchestration tools	✓	The ASTRID Security Controller triggers life-cycle management operations on service graphs through APIs of software orchestrators. The ASTRID run-time subsystem has been designed as a standalone component, to avoid the need for complex modifications or re-design of existing software orchestration tools.
R011-CS-FUN	T	Hybrid software vulnerability analysis covering both pre- and after-deployment of software services	★	Static source-code analysis is performed before deployment, so to generate the metadata and information to verify the software execution at run time. Algorithms will be developed in WP2.
R012-CS-FUN	T	Support Vulnerability Analysis Protocol and Algorithms Agility	✓	The Security Controller allows to select one or more detection algorithms among those available. Reaction and prevention policies implement the control logic to choose which algorithm to run and when. They also re-configure the local agent to feed the algorithms with the needed context.
R013-CS-FUN	T	Access to heterogeneous security context	★	The Context Broker hides different control and data interface to the rest of the system. This allows adding new sources without modifying detection algorithms and reaction policies. WP2 will implement local agents for monitoring log files, system calls, network packets.

ID	P	Name	F	Details
R014-CS-FUN	T	Packet inspection	★	The ASTRID architecture allows pushing inspection programs from the internal repository. This enables a great level of flexibility in the design of deep-packet inspection tasks. The adoption of eBPF technology will improve performance with respect to other user-space or in-kernel mechanisms.
R016-CS-FUN	T	Behavioural analysis through attestation techniques	★	Pre-deployment analysis will allow to identify execution paths and to compute integrity checks for verification at run-time. Some algorithms will be developed by WP3.
R020-SD-FUN	T	Recovery from compromised graphs	✓	The presence of reaction and prevention policies and the interface to the service orchestrator allows the ASTRID Security Controller to trigger management actions on the service graph, including re-deployment, replacement of compromised functions, etc.
R022-SD-IFA	T	REST API	★	WP3 will develop a Security Dashboard that can be used both through the GUI and API.
R026-CS-PRF	T	Average time to detect compromised software	-	This requirement will be considered in WP4.
R027-CS-PRF	T	Average time to detect network anomalies	-	This requirement will be considered in WP4.
R028-CS-PRF	T	Average time to respond to attack	-	This requirement will be considered in WP4.
R032-SD-PRF	T	Average time to replace a single function	-	This requirement will be considered in WP4.
R033-SD-PRF	T	Average time to re-deploy the service graph	-	This requirement will be considered in WP4.
R034-SD-PRF	T	Average time to switch between redundant deployments	-	This requirement will be considered in WP4.
R035-SD-PRF	T	Average time to change the forwarding rules in a service chain	-	This requirement will be considered in WP4.
R037-SD-RLB	T	The attack surface does not increase	✓	The current architecture has selected the eBPF technology for implementing programmable local agents. eBPF programs run in an isolated virtual machine within the kernel and are safe. Other local agents for log collection have very small footprint and do not introduce vulnerabilities (WP2).
R040-LAW-FUN	T	Certification and legal validity	★	The architecture envisions a separate repository for data with legal validity. The mechanisms to certify the origin, time, and integrity of such data will be investigated in WP2.

ID	P	Name	F	Details
R002-CS-DSG	M	Data correlation over multiple service graphs	✓	The run-time subsystem is not tightly bound to the service graph. Alternative deployment options are possible (see Section 6.6), where multiple graphs are connected to the same Context Broker. To completely fulfil this requirement, the implementation must be able to distinguish different services and to manage detection over a single or multiple graphs.
R005-CS-DSG	M	Historical data	✓	The Context Broker includes database to keep data for historical analysis. The persistence of data in the Context Broker can be changed according to the specific needs.
R015-CS-FUN	M	Distributed firewalling	★	The internal structure of the run-time subsystem allows he definition of multiple reaction and prevention policies. A policy will be developed to automate the application of filtering rules in each virtual machine/container.
R017-SD-FUN	M	Automatic firewall configuration	★	The service topology will be loaded in the Security Controller. A specific reaction and prevention policy will be developed in WP2 to automatically translate communication requirements at the graph level into filtering rules for all the enforcement agents.
R018-SD-FUN	M	Dynamic modification of the service topology	✓	The interface between the Security Controller and software orchestrator will allow to trigger modifications in the service graph, according to pre-defined life-cycle management actions (e.g., replace virtual function, add security function, isolate virtual function, create honeypot, etc.).
R024-CS-USB	M	Graphical User Interface	★	The ASTRID Security Dashboard will implement a GUI (WP3).
R021-SD-FUN	M	Automatic enhancement of the service graph	★	The Security Dashboard allows users (service provider) to select desired security requirements. These security requirements will be mapped to concrete instantiations and configurations of the run-time subsystem (e.g., the detection algorithm to be run, reaction/protection policies, etc.).
R036-SD-DSG	M	Lightweight operation and small impact on the service graph	✓	Local agent for log collection and eBPF have already proven to be secure, lightweight, and high-performance technologies.
R039-SW-DSG	M	Integration with existing logging facilities	★	WP2 will develop a data collection framework (local agents + Context Broker) largely based on existing tools.
R006-CS-DSG	L	Privacy and sensitive data	✗	Extension left for commercial implementations.
R019-CS-FUN	L	Autonomous operation	✓	The ECA model for reaction and prevention policy allows the ASTRID Security Controller to trigger the execution of some logic. That logic will check the current context and will issue commands, including re-configuration of the local agents, modifications to the service graph, notification to users.
R023-CS-SUP	L	Cyber-Threat Intelligence (CTI)	✗	This is possible by creating reports that relates the context with the detected threats. Left for commercial implementations.
R025-CS-USB	L	Notification of security events	✗	Left for commercial implementation.

ID	P	Name	F	Details
R029-CS-PRF	L	Detection rate	-	This requirement will be considered in WP4.
R030-CS-PRF	L	Precision of detection	-	This requirement will be considered in WP4.
R031-CS-PRF	L	Performance of attestation services	-	This requirement will be considered in WP4.
R038-SW-DSG	L	Digital right management	✗	Not covered by current architecture.
R041-LAW-FUN	L	Automatic encryption of interception channels	-	This requirement will be considered in WP4.

8 Relationship with similar architectures

The current market of cyber-security products is largely fragmented. In addition to legacy firewalls, intrusion prevention/detection systems and antivirus for personal and enterprise usage, many vendors are already tackling new business opportunities by delivering solutions for enterprise's networks and endpoints, pure and hybrid cloud, industrial devices and networks, security analytics, and integrated solutions. The last class of products aims at getting real-time visibility into all activity on systems, networks, databases, and applications; such solutions apply artificial intelligence and machine learning to wide data sets, facilitating response to attacks by quick one-touch remediation actions. Information from vendors are extremely blurry about technical features and architectures, giving the impression that a tailored solution is developed for each customer by composing discrete standalone appliances and components. This make a direct comparison with the ASTRID architecture impossible.

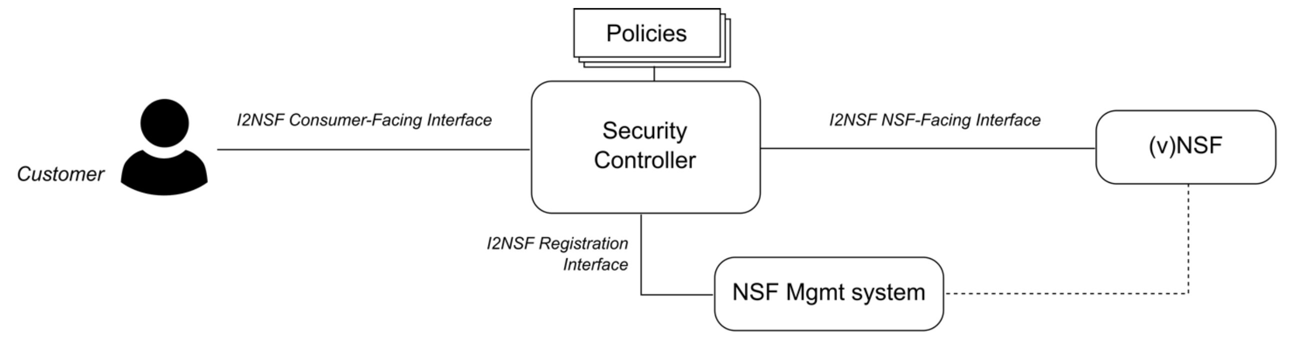
8.1 The Interface to Network Security Functions framework

Looking at the research domain, the I2NSF framework is the only relevant effort that can be compared with ASTRID. As preliminary consideration, we note that the ASTRID architecture is largely compliant with the I2NSF framework, and it may represent a possible implementation for virtualized environments. Here, we briefly compare the two frameworks under several aspects; the interested reader finds a detailed description of I2NSF in Annex B.

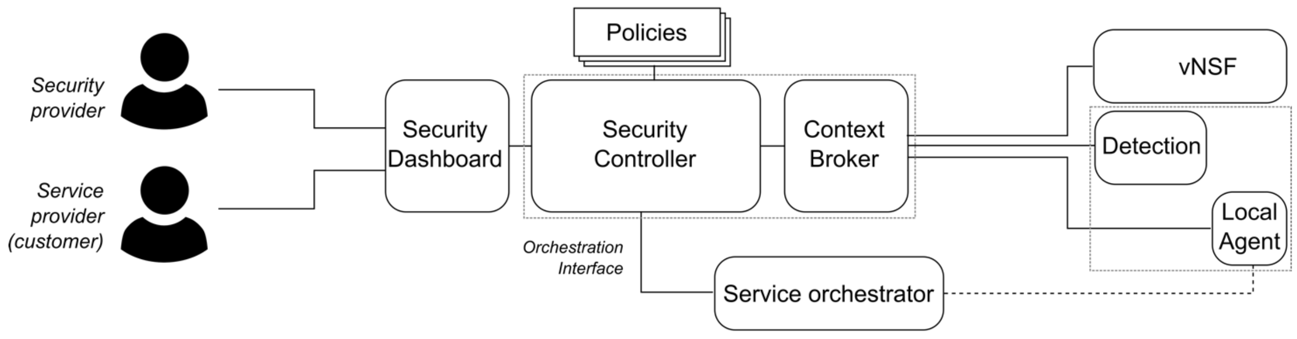
Scope. As the name implies, the I2NSF framework specifically looks at network security functions, i.e., security services that manipulates network packets. The framework targets multiple domains (access networks, enterprise networks, cloud services), and applicability to both physical appliances and virtual instances. It covers the control plane (i.e., selection and configuration of NSFs), but the definition of mechanisms for translating high-level policies into policy rules is currently out of scope for the working group. The data plane and management plane do not fall in the scope of I2NSF; the data plane is totally left to vendors of security functions, and no specific assumptions or requirements are made on deployment, device configuration, life-cycle management, and resource provisioning. ASTRID has a broader scope, also including software analysis in addition to network threats, but it explicitly focuses on virtual NSFs, which can be dynamically created and managed in the context of each single virtual service. However, this does not preclude in principle the usage of hardware acceleration and physical network functions (PNF), which are part of the management capability of orchestration tools, yet this option does not fall under the Project's objectives. ASTRID explicitly covers the data, control, and management planes, targeting better efficiency and programmability, so to effectively tackle the continuous evolving threats landscape. Though, this does not preclude the possibility to use standalone security appliances. As final consideration, we note that both frameworks target monitoring and enforcing policies, even though I2NSF is more leaned to enforcing and ASTRID is currently more focused on detection and situational awareness.

Business model. Both ASTRID and I2NSF considers a similar business model, where Security Providers take care of security services on behalf of their customers. In both cases, customers are allowed to describe their security requirements in terms of high-level policies, whereas the translation to configuration rules is managed by the framework (with automatic or manual procedures). ASTRID also envisions tight coordination between the Security Provider and the Service Provider, which is not present in I2NSF because of its more general scope. For the same reason, all tasks in the ASTRID pre-deployment subsystem are not present in I2NSF. Overall, the ASTRID business model could be viewed as an enhanced version of that of I2NSF for virtual services.

Architecture. The I2NSF reference model and the ASTRID architecture are compared in Figure 11. In a bird's eye view, the ASTRID architecture is more complete, since it is rapidly evolving not being slowed down by a standardization process. They both rely on coordination from a Security Controller, which applies security policies. In both cases, internal policies are defined as flow policies (for instance, in ECA form) that are translated by the Security Controller in specific policy rules for each NSF. Such policies are instantiated through the Security Dashboard in ASTRID; the translation may either be performed automatically or managed by the Security Provider. In the I2NSF model, the translation is directly managed by the Security Controller; however, it is currently suggested to express user policies in a ECA form that can be mapped to policy rules in a straightforward way (Sec. 7.1 of [15]). The I2NSF Consumer-Facing Interface can be mapped to the API/interface of the ASTRID Security Dashboard. On the right side of the picture, the I2NSF framework control physical and virtual instance of security appliances (NSF), but it dictates that they implement the NSF-Facing interface. ASTRID can control vNSF as well, but for the sake of efficiency and programmability it mainly focuses on the separation between inspection tasks and the detection logic. The combination of detection algorithms and local inspection agents are totally equivalent to a NSF, but the substantial difference is the far greater flexibility in composing bespoke security functions to the specific service implementations. ASTRID does not mandate a standard interface for all security functions (vNSF, detection algorithms, local inspection agents), which is not a feasible approach in the short/medium period, but leverages a Context Broker to harmonize the different dialects spoken by the set of heterogeneous controlled elements. The I2NSF NSF-Facing interface may be implemented between the ASTRID Security Controller and the Context Broker, progressively divesting the latter of its functions as more elements adopt the standard interface. In both frameworks the Security Controller needs knowledge about the security functions available. In I2NSF, there is a dedicated Registration interface, and a unspecified "Management System" that depends on the specific environment. It may be part of existing management tools in large infrastructures (as the networks of telecom operators or the data centres of cloud operators). In ASTRID, given the more restricted scope, the management component is the service orchestrator, which deploys and manages vNSF and local agents in the service graph. It knows the capabilities and interfaces of all elements in the data plane, that are made available to the Security Controller through the orchestration APIs. As part of the management framework, the I2NSF is already addressing remote attestation of both the Security Controller and NSFs. This aspect is not currently listed among the ASTRID objectives, but it represents a meaningful extension for concrete exploitation beyond the Project lifespan. Finally, though not explicitly shown in Figure 11, an AAA framework is necessary in both cases to establish secure relationships and channels between the involved elements. ASTRID will explicitly address this aspect, whereas concrete proposals are still missing in the I2NSF framework.


Policies
Customer
I2NSF Consumer-Facing Interface
Security Controller
I2NSF NSF-Facing Interface
(v)NSF
I2NSF Registration Interface
NSF Mgmt system

a) I2NSF reference model



b) ASTRID architecture

Figure 11. Comparison between the conceptual architectures for I2NSF and ASTRID.

The ASTRID architecture can be considered a first implementation of the I2NSF framework, targeting virtualized systems. The I2NSF does not strictly mandate a specific architecture, but it is mostly concerned with interoperability interfaces. In this respect, the ASTRID architecture will continuously look at the I2NSF while defining the corresponding interfaces; ASTRID will also consider the possibility to contribute to the standard with its own architecture and the enhanced data plane.

References

[1] B. Karakostas, "Towards Autonomic Cloud Configuration and Deployment Environments," Intl. Conf. on Cloud and Autonomic Computing (ICCAC), London, UK, pp.93–96, Sept. 2014.

[2] J. Wettinger, U. Breitenbücher, and F. Leymann, "Standards-Based DevOps Automation and Integration Using TOSCA," IEEE/ACM 7th Intl. Conf. on Utility and Cloud Computing (UCC), London, UK, pp.59–68, Sept. 2014.

[3] P. Bellavista, L. Foschini, R. Venanzi, and G. Carella, "Extensible Orchestration of Elastic IP Multimedia Subsystem as a Service Using Open Baton," 5th IEEE Intl. Conf. on Mobile Cloud Comp., Services, and Engineering (MobileCloud). San Francisco, California (USA), pp88–95, Apr. 2017.

[4] M. Ali, S. U. Khan, and A. V. Vasilakos, "Security in cloud computing: Opportunities and challenges," Information Sciences, vol.305, no. 1, pp.357–383, June 2015. DOI: 10.1016/j.ins.2015.01.025

[5] Topology and Orchestration Specification for Cloud Applications. OASIS Standard. Retrieved March 20th, 2017 from http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf Version 1.0.

[6] TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0, OASIS Draft 04, 11 May 2017. [Online] Available: http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/csd04/tosca-nfv-v1.0-csd04.pdf.

[7] Network Functions Virtualisation (NFV) Release 3; Management and Orchestration; Network Service Templates Specification. ETSI GS NFV-IFA 014 V3.1.1, August 2018. Retrieved 13th February, 2019 from https://www.etsi.org/deliver/etsi_gs/NFV-IFA/001_099/014/03.01.01_60/gs_NFV-IFA014v030101p.pdf.

[8] Network Functions Virtualisation (NFV); Management and Orchestration. ETSI GS NFV-MAN 001 V1.1.1, December 2014. Retrieved 13th February, 2018 from http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf.

[9] J. Halpern and C. Pignataro, Service Function Chaining (SFC) Architecture. RFC7665. Retrieved May 12th, 2017 from https://tools.ietf.org/rfc/rfc7665.txt.

[10] Network Functions Virtualisation (NFV) Release 2; Management and Orchestration; VNF Descriptor and Packaging Specification. ETSI GS NFV-IFA 011 V2.4.1, February 2018. Retrieved 24th April, 2019 from https://www.etsi.org/deliver/etsi_gs/nfv-ifa/001_099/011/02.04.01_60/gs_nfv-ifa011v020401p.pdf.

[11] G. Pék, L. Buttyán, B. Bencsáth, A survey of security issues in hardware virtualization, ACM Computing Surveys 45 (3) (2013) 40:2–40:34.

[12] A. M. Medhat, T. Taleb, A. Elmangoush, G. A. Carella, S. Covaci, and T. Magedanz, "Service function chaining in next generation networks: State of the art and research challenges," IEEE Communications Magazine, vol. 55, no. 2, pp. 216–223, February 2017.

[13] J. Garay, J. Matias, J. Unzilla, and E. Jacob, "Service description in the NFV revolution: Trends, challenges and a way forward," IEEE Communications Magazine, vol. 54, no. 3, pp. 68–74, March 2016.

[14] S. Hares, D. Lopez, M. Zarny, C. Jacquenet, R. Kumar, J. Jeong. Interface to Network Security Functions (I2NSF): Problem Statement and Use Cases. IETF RFC 8192, July 2017. [Online] Available: https://www.rfc-editor.org/rfc/pdfrfc/rfc8192.txt.pdf.

[15] D. Lopez, E. Lopez, L. Dunbar, J. Strassner, R. Kumar. Framework for Interface to Network Security Functions. IETF RFC 8329, February 2018. [Online] Available: https://tools.ietf.org/pdf/rfc8329.

[16] S. Hares, J. Strassner, D. Lopez, L. Xia, H. Birkholz. Interface to Network Security Functions (I2NSF) Terminology. IETF Internet Draft, v. 07, January 2019. [Online] Available: https://tools.ietf.org/pdf/draft-ietf-i2nsf-terminology-07.pdf.

[17] A. Pastor, D. Lopez, A. Shaw. Remote Attestation Procedures for Network Security Functions (NSFs), IETF Internet-Draft, v. 07, February 2019. [Online] Available: https://tools.ietf.org/pdf/draft-pastor-i2nsf-nsf-remote-attestation-07.pdf.

[18] Radware. On-Demand, Always-on, or Hybrid? Choosing an Optimal Solution for DDoS Protection. Whitepaper, 2016. [Online] Available: https://www.radware.com/WorkArea/DownloadAsset.aspx?id=6442457904.

[19] Radware. Cloud DDoS Protection Service: attack lifecycle under the hood. Technology Overview Whitepaper, 2016. [Online] Available at: https://www.radware.com/assets/0/314/6442477977/2c6454b4-403b-45b1-ac58-dc628bc210b3.pdf.

[20] F5 Silverline. Protect your business and stay online during a DDoS attack. F5 Product Datasheet, 2016. [Online] Available at: https://www.f5.com/pdf/products/silverline-ddos-datasheet.pdf.

[21] TPM 1.2 Main Specification. URL: https://trustedcomputinggroup.org/resource/tpm-main-specification/.

[22] Global Platform – Technology Document Library, Specification Library. URL: https://globalplatform.org/specs-library/.

[23] N. Borhan, R. Mahmod. Platform Property Certificate for Property-based Attestation Model. International Journal of Computer Applications, Vol. 65, No.13, March 2013, pp. 28-37.

[24] Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. “Dowsing for overflows: A guided fuzzer to find buffer boundary violations”, USENIX 2013.

[25] I. Chipounov, V. Kuznetsov, and G. Candea, “S2E: A platform for in-vivo multi-path analysis of software systems”, ASPLOS 2011.

[26] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing Mayhem on binary code”, IEEE Symposium on Security and Privacy”, 2012.

Annex A Orchestration models

The term "orchestration" is by now a buzzword in the IT word, though it has been used with slightly different meanings in the last years. Intuitively, it recalls the idea of coordinating, arranging, or organizing something together. Indeed, recent definitions in this field agree that orchestration describes automated arrangement, coordination, and management of complex computing systems, middleware, and services.

Orchestration is often discussed in the context of service oriented architectures, virtualization, resource provisioning, hyper-convergence and software-defined data centres. Its purpose is to automate as much as possible multiple complementary workflows, with the main goal of aligning business requirements with applications, data, and infrastructures. While "automation" makes the execution of a single task possible without direct human operations (e.g., launching a web server, stopping a service), "orchestration" broadens the scope to multiple tasks, which collectively execute a larger workflow or process and could involve multiple systems. Differently from mere automation, where the steps are rather deterministic and control is mostly limited to the selection of alternative execution paths based on the dynamic context, orchestration is mostly concerned with optimization of the processes and workflows. This implies an inherent intelligence that considers intents, goals, aspirations rather than technical conditional statements.

In the extended cloud domain (hence including network virtualization), the primary responsibility for orchestration is the selection of resources and deployment of the overall workflow according to multiple requirements and constraints: cost, performance, reliability, efficiency, locality, redundancy, security, scalability, resiliency, and so on. Therefore, the problem is not limited to the definition of the sequence of commands to be called, but includes the elaboration of life-cycle management strategies that are able to autonomously react to changes in the workload, failures, cyber-attacks, anomalies, and unexpected conditions.

From a more technical perspective, cloud service orchestration entails the following tasks:

- dynamic provisioning of virtual resources for computing, networking, and storage;
- deployment, configuration, and update of the software;
- connection and automation of workflows to deliver the defined service.

Orchestration is mainly responsible for:

- provisioning the correct amount of resources, by taking into account performance requirements for the overall service (e.g., latency in processing requests) and computation requirements of its components (e.g., CPU/RAM/network bandwidth requested by the software to process a request);
- supervising the execution of the service, by monitoring resource utilization and service performance;
- performing life-cycle management events (e.g., start/stop the service, scale the service, recover from failure), upon specific conditions.

Many tools are indicated as orchestration solutions, but not all of them can be properly considered as such. For example, configuration management tools as Puppet, Chef, Ansible and SaltStack are designed to reduce the complexity of configuring distributed infrastructure resources, but do not cover the full scope of orchestration. Cloud management software, as OpenStack and Kubernetes, allows dynamic provisioning of resources, boot of software images, and configuration of IP addresses and other properties; however, it cannot perform life-cycle management actions. In this project, we strictly follow the general definition for orchestration, hence configuration management tools and cloud management software are considered part of the a broader orchestration framework.

As previously discussed, orchestration is a far more complex process than mere automation. While automation is largely based on the execution of scripts in common languages, orchestration needs a more complex framework, which describes the overall service in terms of abstract components (i.e., virtual functions) and their logical relationships (i.e., configuration dependencies). This abstraction is used both to compute the virtual resources that are needed to deploy the service and to configure its discrete components. There are currently multiple alternative solutions to describe virtual services, which largely reflects the specific needs of different domains. We arbitrarily indicate these solutions as "orchestration models," though this term is not used in the technical literature, due to the lack of more specific terminology. Our analysis is mostly devoted to provide the interested reader with better knowledge of what orchestration implies and how this could affect the ASTRID framework, without the ambition to give a full review of all available orchestration models. In this respect, we only focus on the most representative models coming from standardization bodies, one for cloud applications and two for NFV.

A.1 OASIS TOSCA

The main goal of the Topology and Orchestration Specification for Cloud Applications (TOSCA) [5] is to enhance the portability and management of cloud applications across alternative environments. TOSCA is basically a language that uses a metamodel to describe an IT service, which defines both its structure as well as how to manage it (see Figure 12). The TOSCA metamodel uses the concept of Service Templates to describe cloud workloads as a topology, which is a graph of Node Templates modelling the components a workload is made up of, and as Relationship Templates modelling the relations between those components. TOSCA further provides a type system of Node Types to describe the possible building blocks for constructing a Service Template, as well as Relationship Type to describe possible kinds of relations. Both Node and Relationship Types may define lifecycle operations to implement the behaviour an orchestration engine can invoke when instantiating a Service Template. For example, a Node Type for some software product might provide a 'create' operation to handle the creation of an instance of a component at runtime, or a 'start' or 'stop' operation to handle a start or stop event triggered by an orchestration engine. Those lifecycle operations are backed by implementation artefacts such as scripts or Chef recipes that implement the actual behaviour.

TOSCA describes *Service Templates* by the following elements:

1. service components and their logical topology, by mean of an XML schema definition;
2. management procedures to create or modify services, by means of artefacts.

As such, TOSCA does not mandate any specific architecture for service orchestration, leaving freedom to alternative implementations. An orchestration engine processing a TOSCA Service Template uses the mentioned lifecycle operations to instantiate single components at runtime, and it uses the relationship between components to derive the order of component instantiation. For example, during the instantiation of a two-tier application that includes a web application that depends on a database, an orchestration engine would first invoke the 'create' operation on the database component to install and configure the database, and it would then invoke the 'create' operation of the web application to install and configure the application (which includes configuration of the database connection).

Service Templates are deployed in TOSCA containers, which are runtime environments that enable the automated provisioning of TOSCA applications. An artefact represents something that can be executed. The representation of the executable can be direct or indirect. A direct representation is something that is the executable itself, like a script, an EJB, a BPMN file, and so on. An indirect executable is a reference to an executable, like a URL of a resource that can be downloaded, an endpoint reference to a port (i.e. an implementation of a WSDL port type), and so on. Typically, descriptive metadata will also be provided along with the artefact. This metadata might be needed to properly process the artefact, for example by describing the appropriate execution environment. TOSCA artefacts include:

- *implementation* artefacts that implement the interfaces (i.e., management operations) defined by Node Types. For example, one management operation could be the instantiation of a Node Type. They must therefore be present in the management environment before any operation can be started. Implementation Artefacts can be implemented using arbitrary technologies, such as shell scripts, Java/JEE or any further programming language.
- *deployment* artefacts that implement the business functionality of a Node Template. These are executables materializing instances of a node (e.g., disk images, installables, zip archives, WAR files). They are deployed in the managed environment during the instantiation of Node Types.

A.1.1 TOSCA model

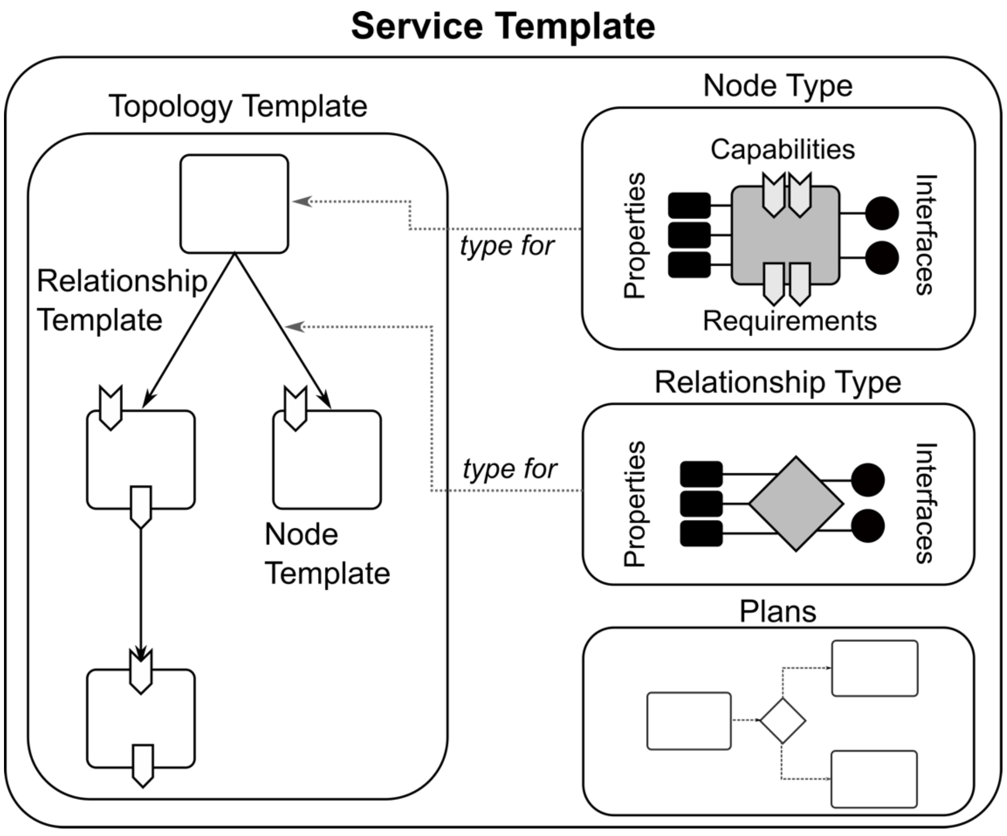


Figure 12. Structure of Service Templates in TOSCA and constituent components.

Figure 12 shows the logical structure of a Service Template, which is made of a Topology Template, Node Types, Relationship Types, and Plans.

A **Topology Template** (also referred to as the topology model[14] of a service) defines the structure of a service. A topology template can be used to instantiate and orchestrate the model as a reusable pattern and includes all details necessary to accomplish it. Topology Templates are defined as (not necessarily connected) directed graphs consisting of nodes and relationships. A node can be an infrastructure component, like a subnet, a network, a server (it can even represent a cluster of servers), or it can be a software component, like a service or a runtime environment. Meanwhile, a relationship describes how nodes are connected to one another.

[14] The term Topology Model is often used synonymously with the term Topology Template with the use of "model" being prevalent when considering a Service Template's topology definition as an abstract representation of an application or service to facilitate understanding of its functional components and by eliminating unnecessary details.

A node in this graph is represented by a **Node Template**. A Node Template specifies the occurrence of a software component node (i.e., an instance) as part of a Topology Template. Each Node Template refers to a **Node Type** that defines a component in terms of its properties, interfaces, capabilities, and requirements. Node Types are defined separately for reuse purposes; Node Templates can provide customized properties, constraints or operations which override the defaults provided by their Node Types.

Node requirements indicate what is needed to use the component in a Service Template. This includes dependencies on features provided by other components, or constraints on the execution environment, such as the allocation of certain resources or the enablement of a specific mode of operation. Similarly, capabilities express features provided by the component. Requirements and capabilities are modelled using specific data types (Requirement Type and Capability Type), so that the same definitions can be used for multiple components. For instance, an application may include a "*Database Connection Requirement*" to describe the need for a database connection; a Java application node may expose a "*Java Servlet Runtime Requirement*", whereas the Apache Tomcat node provides a matching "*Java Servlet Runtime Capability*." While Node Types only provide descriptive metadata for requirements and capabilities, Node Templates will provide concrete values for these properties. In particular, requirements can be fulfilled in two ways: 1) requirements of a Node Template can be matched by capabilities of another Node Template in the same Service Template by connecting the respective requirement-capability pairs via Relationship Templates (e.g., the Java application as source node and Apache Tomcat server as target node in the previous example); 2) requirements of a Node Template can be matched by the hosting environment, for example by allocating needed resources for a Node Template during instantiation (e.g., IP addresses taken from a valid subnet).

A **Relationship Template** indicates some sort of dependency between nodes. Similar to the nodes, each Relationship Template refers to a **Relationship Type**, which defines its semantics, properties, and interfaces. The Relationship Template explicitly identifies the direction of the relationship. There may be different kinds of relationships. For example, a relationship may be established between a web application and a web server, meaning something like "hosted by," or between an Apache web server and a MySQL server, meaning that the database requirement of the former is satisfied by the capability of the latter.

The last component of a Service Template are **Plans**. Management Plans are automatically executable workflow models that imperatively specify the management operations to be executed for a certain management functionality. They describe the management aspects of service instances, especially their creation and termination. These plans are defined as process models, i.e. a workflow of one or more steps. Process models can be either directly included in the plan or referred to. A process model can contain tasks that refer to operations exposed by the interfaces of Node Templates, Relationship Templates, or any other interface (e.g. the invocation of an external service for licensing); in doing so, a Plan can directly manipulate nodes of the topology of a service or interact with external systems. All Plans are expected to be adapted to the concrete execution environment by Service Providers. Instead of providing another language for defining process models, the specification relies on existing languages like BPMN or BPEL, which facilitates portability and interoperability.

Both the nodes and relationships types include properties and interfaces. Interfaces are operations allowed to control the specific component. Interfaces are used by the plan models for lifecycle operations (install, start, stop, etc.), but further management interfaces may be defined for component-specific operations (e.g., *backup_database* and *restore_database* for a MySQL server). All these operations are typically implemented by configuration management tools, as Chef recipes or Unix shell scripts. Operations in the context of a relationship are distinguished in *source* operations and *target* operations, based on the direction of the relationship: source operations are executed on the source node, target operations on the target node. Properties can be defined as arbitrary data structures to make the components configurable. They are similar to "variables" that must be instantiated at run-time (e.g., IP addresses, location of log files). All properties are exposed to interfaces, so that they can be considered during execution.

Service Templates can also be nested, to facilitate the design of complex applications and to reuse common components. For example, the environment to run web applications is usually composed of a web server, operating system, and virtual machine. Hence, a Service Template for an application server may be designed as shown in the left side of Figure 13 by a vendor specialized in deploying and managing application servers. This Template can be viewed as a Node Template, once it exposes the same boundaries (requirements, capabilities, properties, interfaces). The Service Template for the web application can therefore be designed with two Node Templates only, one for the web application and the other one for the application server, as depicted in the right side of Figure 13. During deployment, the Node Template for the application server is substituted with the corresponding Service Template, and this operation is possible because the two elements use the same boundary definition.

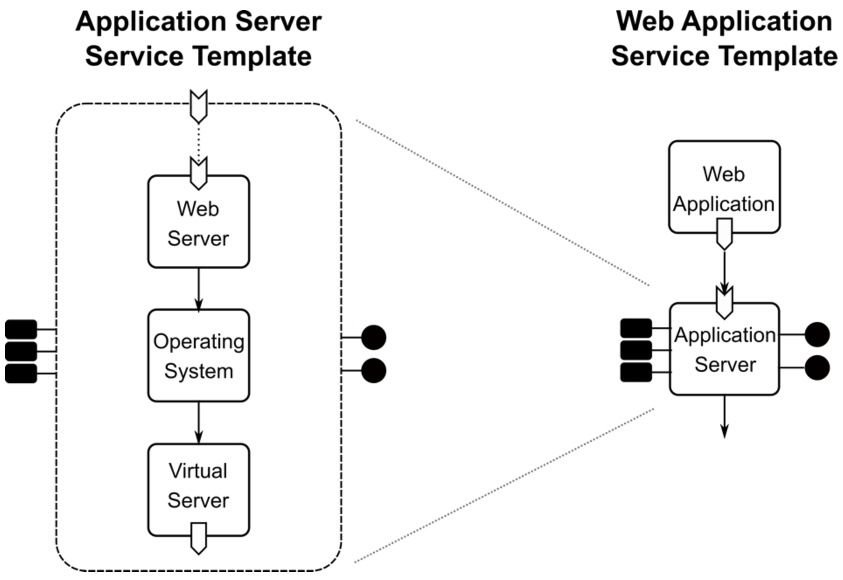


Figure 13. Nested Service Template.

TOSCA specifies *Cloud Service Archives* (CSAR) as packaging format for cloud applications. A CSAR is a portable and self-contained archive that contains all TOSCA model files as well as further software required to enable the automated provisioning and management of the modelled applications: the definitions of types, templates, artefacts, plans, and any additional referenced file. It enables a TOSCA runtime (e.g., OpenTOSCA[15]) to traverse the topology template to create application instances.

A.1.2 Deployment and orchestration

Deploying a TOSCA application implies the creation of an **instance** of its Service Template. The instance is indeed derived by the Topology Template, which can be instantiated either by *imperative* or *declarative* processing. Imperative processing makes use of a special plan, often referred to as "build plan," that orchestrates the management operations provided by nodes and relationships, usually by running the implementation artefacts attached to their interfaces. The build plan will provide actual values for the various properties of Node Templates and Relationship Templates. These values can come from input passed in by users as triggered by human interactions defined within the build plan, by automated operations defined within the build plan (such as a directory lookup), or the templates can

[15] https://www.opentosca.org/.

specify default values for some properties. A declarative TOSCA runtime infers the corresponding logic automatically by interpreting the topology template and deriving the actions to be executed on its own.

Figure 14 shows an indicative example for a OnlineBookstore web application. It is composed by multiple node components, with indication of relationships: the web application runs in the WebSever, which runs on Linux, which is hosted by a virtual machine. The TOSCA runtime (i.e., orchestrator) will first instantiate the VirtualServer template, by invoking the corresponding EC2 API for provisioning, according to the implementation artefacts; this operation may also associate a static IP address. Next the OperatingSystem is booted through the deployment artefacts, and the installation scripts are executed to install the WebServer. Management operations of the WebServer will allow to install the java application that implements the OnlineBookstore application. Finally, any specific implementation artefact for the application may be called to finalize the instantiation. It is worth noting that, according to the concrete definitions of Node Types and Templates, the runtime system will also take care of configuration of all components (e.g., by setting the IP listening addresses for the WebServer).

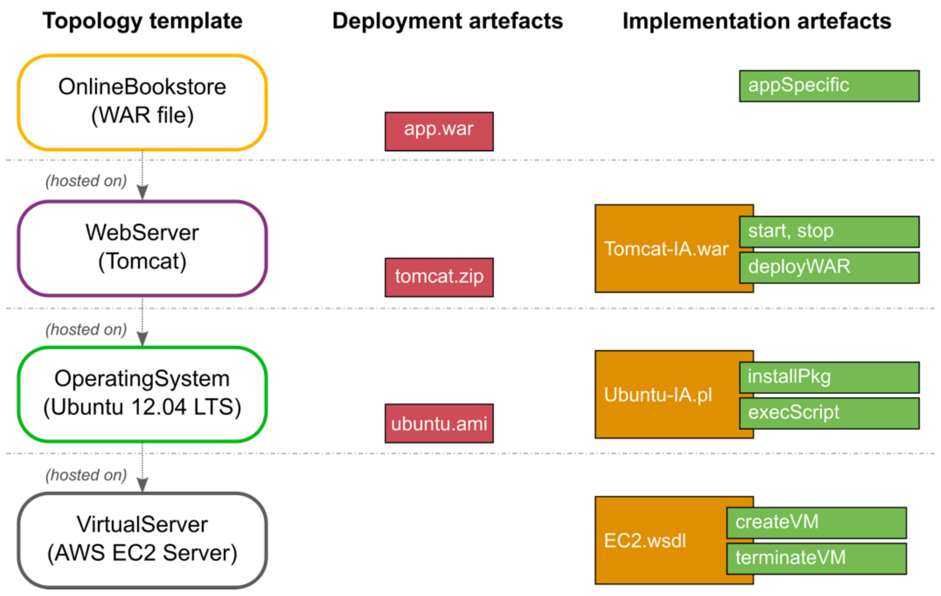


Figure 14. Example of TOSCA topology template for a web application.

The behaviour of a TOSCA application can be defined by policies. In TOSCA, a **Policy Type** defines a type of requirement that affects or governs an application or service's topology at some stage of its lifecycle but is not explicitly part of the topology itself (i.e., it does not prevent the application or service from being deployed or run if it did not exist). TOSCA policies are a type of requirement that govern use or access to resources which can be expressed independently from specific applications (or their resources) and whose fulfilment is not discretely expressed in the application's topology (i.e., via TOSCA Capabilities). A Policy can express such diverse things like monitoring behaviour, payment conditions, scalability, or continuous availability, for example. The definition of policies includes three main elements:

- its *properties*, which define the type of configuration parameters that the policy takes;
- its *targets*, which define the node types to which the policy type applies;
- its triggers, which specify the conditions in which policies of this type are fired.

A Trigger defines an event, condition, and action (ECA) that is used to initiate execution of a policy associated with it. The definition of the Trigger allows specification of the type of events to trigger on, the filters on those events, conditions and constraints for trigger firing, the action to perform on triggering, and various other parameters.

Policies typically address the following concerns: access control (rules to access the service based on organization role, time of day, geographic location, etc), placement (assures affinity or anti-affinity of deployed applications and their resources for performance, availability, reliability, and other matters), quality of service (assures performance and continuity of software components along with considerations for scaling and failover).

A.2 ETSI Network Function Virtualization/IETF Service Function Chaining

Traditional network service deployments are based on complex and rigid configurations, highly dependent on the physical topology and outdated control protocols. The deep effort towards increased agility in service deployment and management has driven a progressive transition from hardware to software- defined paradigms, leading to the recent concepts of service function chaining and network functions virtualization [12]. The common denominator is the freedom to steer packet streams across a set of network functions without the need to change the physical topology or complex configurations. Indeed, these models can be indicated as *data-driven* orchestration and are quite different from the abstraction discussed in Section A.1. Among alternative models and standards that have been proposed [13], the SFC and NFV architectures are currently the main drivers towards future intelligent networks.

A.2.1 ETSI NFV

ETSI has produced a comprehensive framework for Network Function Virtualization, including the definition of the management architecture and the abstraction models for network services. The unique characteristic of the ETSI framework is the definition of a standard Management and Orchestration (MANO) architecture, which should ensure compatibility and interoperability of components from different vendors. This effort is not present in TOSCA and SFC, which focus on the definition of the abstraction of the service but do not dictate any specific architecture for implementing the orchestrator.

NVF Management and Orchestration architecture (MANO)

The MANO architecture [8] relates virtualization infrastructure with management components, as shown in Figure 15. Different colours are used to group elements at the infrastructure, VNF, and orchestration/management layer.

The NFVI includes all hardware and software components which build up the environment where VNFs are deployed, managed, and executed. The NVFI is typically composed of multiple Point-of-Presences (NFVI-PoPs), which represent discrete installations of virtualization technologies in different locations; they may include data centres, central offices, edge installations. Wide-area networks interconnect the different NFVI-PoPs and are also part of the NFVI. From the perspective of VNFs, the NFVI is seen as a single aggregation of computing, networking, and storage resources, which are abstracted by a virtualization layer (i.e., hypervisor or similar technology) in order to provision virtual resources for the execution environment. There is no specific indication about the virtualization layer, leaving the freedom to choose among hypervisors, containers, and hardware acceleration. From a network perspective, there is typically a distinction between NVFI-PoP networks, which are expected to be managed with the same or similar technologies as data centres, and transport networks, where SDN, slicing or more traditional wide-area networks approaches could be used.

The VIM manages NFVI resources in a single domain, hence there will be multiple VIMs in an NFV architecture. A VIM basically plays the same role as CMS: it creates, maintains and tears down virtual machines, keeps inventory of VMs associated with physical resources, deals with performance and fault management, exposes northbound APIs to upper management systems.

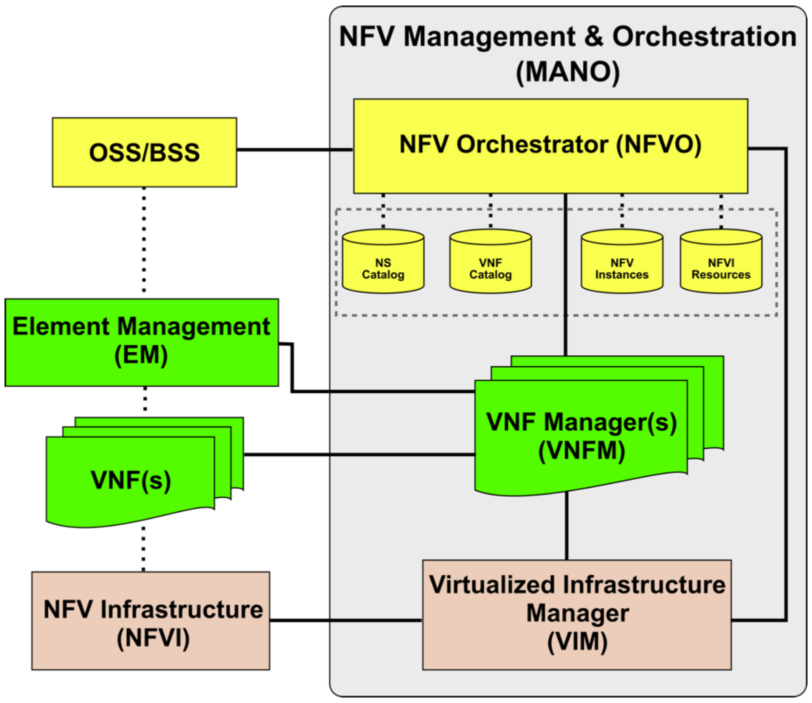


Figure 15. ETSI MANO architecture.

A VNF is a software function that can be orchestrated (i.e., automatically instantiated, started, stopped, terminated, scaled). From a functional perspective, there is no difference between a VNF and its physical counterparts. Examples of possible VNFs are EPC, MME, SWG, PWG, RGW, firewalls, DHCP servers, etc. A VNF may be deployed in a single VM or may be decomposed in multiple components which are split in more VMs.

The management of a VNF is split into two parts, heading to functional and virtual aspects. In both cases, management includes Fault, Configuration, Accounting, Performance and Security (FCAPS). The EM is responsible for FCAPS management of the business logic of the VNF. It is essentially the same as for physical instances of the VNF, because it does not cover any issue related to the virtualization framework. On the other hand, the VNFM manages all aspects related to virtualization: it creates, maintains, scales, and terminates VNF instances in the NVFI, and addresses FCAPS issues related to virtualization (e.g., faults in the NFVI, lack of resources). There may be multiple VNFMs managing separate VNFs or there may be one VNFM managing multiple VNFs. To be more concrete with an example, if there is any issue with the spinning up of a VNF, it will be reported by the VNFM but if the fault is related to a function (for example, some signalling issue in mobile core) it will be dispatched to the EM. VNFM exposes its interface to the EM in case an operator wishes to use single GUI for all kind of FCAPS (virtual + functional).

The NVFO orchestrates the different management elements (VIMs, VNFMs) present in the MANO framework, leading to both resource and service orchestration. A single service may require resources from multiple NFVIs, so the NVFO coordinates, authorizes, releases, and engages such resources through the different VIMs (Resource Orchestration). On the other hand, the NFVO coordinates the VNFMs of the VNFs that are needed to create an end-to-end network service (Service Orchestration), including the instantiation of the same VNFMs, if necessary. An example would be creating a service between the base station VNF's of one vendor and core node VNF's of another vendor. In both cases, the NFVO only interfaces to the management entities (VIM and VNFM) instead of the infrastructures or software (NFVI and VNFs, respectively). The NVFO is the "entry point" for the service description (i.e., VNF Forwarding Graph), which will be described in the next Subsection.

The NFVO operation relies on 4 main catalogues of information about the NFV system. The VNF Catalogue is the repository of all usable VNFs. Each VNF is described by its VNF Descriptor, which is a deployment templates that contains its deployment and operational behavioural requirements. It is primarily used by VNFM in the process of VNF instantiation and lifecycle, but it can also be used by the NFVO to manage and orchestrate Network Services and virtualized resources on NFVI. The NS Catalogue contains all usable Network Services. A NS is described by a deployment template in terms of VNFs and description of their connectivity through virtual links. NFV Instances list holds all details about Network Services instances and related VNF Instances. Finally, the NFVI resources is a repository of resources used by instantiated NSs.

OSS/BSS includes collection of systems/applications that a service provider uses to operate its business. In principle it would be possible to extend the functionalities of existing OSS/BSS to manage VNFs and NFVI directly, but that would result in a proprietary implementation of a vendor, because the interfaces with EM and VNFs are not yet defined by ETSI (dotted lines). The existing OSS/BBS, however, adds value to NFV MANO by offering additional business functions (billing, accounting) which might not be supported by implementations of MANO.

Network service description

The description of a Network Service in ETSI NFV is based on the definition of a Network Service Descriptor (NSD). An NSD contains all the elements that are required by the NFVO for management operations, including the functional and behavioural specifications. Therefore, an NSD includes the reference to involved VNFs, Physical Network Functions (PNFs), and Virtual Links (VLs), as well as the definition of VNF Forwarding Graphs (VNFFGs) and Network Forwarding Paths (NFPs). An NS can also include another NS, hence creating a nested structure. The definition of a NS is based on templates, which define alternative deployment flavours, which represent possible variants for different operational scenarios. Each flavour defines allowed profiles for VNFs, VLs, PNFs, as well as scaling and affinity rules. Thanks to the availability of different profiles, it is possible to choose among different VNF implementations, the number of their instances, and the constraints on the underlying infrastructures (which affect both performance and cost). Figure 16 shows the main logical components of a network service, together with their relationship with the MANO framework.

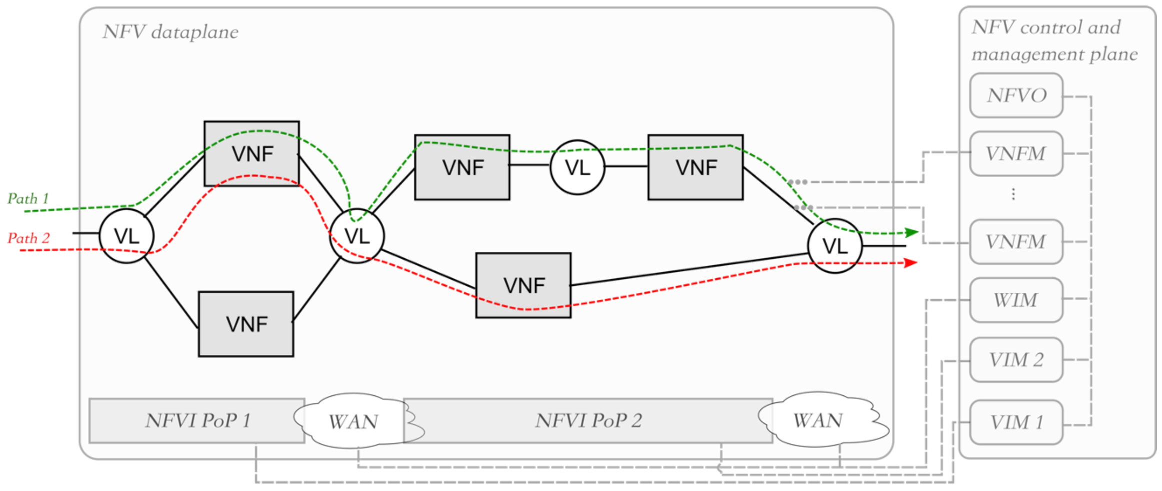


Figure 16. ETSI VNFFG for service description.

VNFs are in turn described by VNF Descriptors (VNFDs) [10]. The VNFD contains the software that implements the business logic for the VNF (delivered as bootable image) and all metadata and properties for its correct instantiation. Such metadata includes the format of the software image, its external connection points to be chained with other VNF to create end-to-end NSs, requirements and constraints on the execution environment, dependencies needed to install, start and terminate the VNF, and life-cycle management scripts. The description permits the specification of multiple deployment flavours (number of instances, affinity rules, monitoring parameters, scaling profiles, etc.), so to match different operational conditions. A VNF can be a standalone component or be composed by sub-components. Each subcomponent corresponds to a different VM or container, interconnected by virtual links. For each component, the description of the deployment and operational behaviour is provided.

PNFs may be identified and included in the NS. In this case, the definition mainly focuses on connectivity requirements, as a PNF by definitions covers its own resource requirements and cannot be deployed in locations other than its own.

VLs describe the connectivity between V/PNFs. Differently from VNFs, VLs does not implement any business logic, so their definition is mostly concerned to requirements on protocols and performance. Shortly, a VL is characterized by the connectivity type, which includes protocols (e.g., Ethernet, IPv4, pseudo-wire, MPLS, etc.), flow patterns (e.g., line, tree, mesh), and QoS parameters (latency, jitter, packet loss, priority). This information is used by the NFVO to configure correct paths in the underlying infrastructure. VL may be used to describe both intra- and inter-NFVIPoP connectivity.

The VNFFG defines the topology of the service, i.e., how VNFs, PNFs, and VLs are connected together. Multiple VNFFG may coexist in the same network service. Each VNFFG selects a possible instance of the NS, opting for one among available profiles. The definition of a VNFFG may also include NFPs that describe a traffic flow in the NS based on policy decisions. An NFP is a sort of collection of classifiers and forwarding rules for the VNFFG so to steer the packets across the set of VNFs. Based on abstract service descriptors, the NFVO generates the sequence of VNF to be traversed and set the appropriate configurations on VNFs and VLs.

A.3 IETF Service Function Chaining

The IETF SFC framework [9] defines a new approach to service delivery and operation, built around the idea of an abstract view of the required service functions and the order in which they are to be applied. It fundamentally represents the latest evolution of software- defined networking, by adding unlimited processing capabilities to basic routing, forwarding, and filtering functions. The SFC architecture envisions both a control plan and a data plane; the main elements of the data plane are the SFC classifier, Service Function Forwarder (SFF), Service Function (SF), and SFC proxy. The traffic is steered across the SFs by adding a Network Service Header (NSH), hence creating a sort of overlay over an unspecified networking infrastructure. Figure 17 shows a schematic view of the framework.

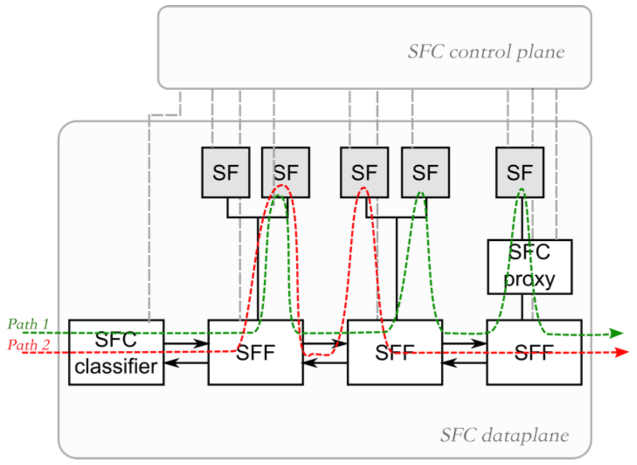


Figure 17. IETF SFC architecture.

The SFC classifier assigns each packet to the proper flow, based on the classification rules set by the control plane. The identification of different flows typically reflects the target application. Each flow is identified by a service function path (SFP) ID, which is inserted in the NSH, and identifies the list of SFs to be traversed. The NSH also contains metadata that will be used to process the packet along the chain and to store the processing state. The SFP ID may change as the result of the processing of a SF, or because of incorrect or partial initial classification; the SFP is the actual sequence of SFs traversed by the packet when it exits the chain.

An SFF forwards packets to SFs and/or other SFFs, according to the information present in the NSH (mainly, SFP ID and list of already traversed SFs). SFs process packets, by implementing network functions (e.g., firewall, DPI). They process packets belonging to several SFPs and can be present with multiple, distributed instances for scalability and resiliency. SFs may take different decisions based on the SFP ID and may update the NSF with an updated context, to be used by the next SFs. The architecture also envisions a SFC proxy to interface legacy SFs that do not understand the NSH.

In the SFC architecture, forwarding rules are installed in SFF components by the control plane. Such rules are set according to high level policies that define a processing pipeline as an ordered (and conditional) sequence of SFs. It is worth pointing out that the SFC architecture does not specify a management plane for deployment and life-cycle operations of SF/SFF. Indeed, the ETSI NFV architecture is often referenced as possible management plane.

A.4 Comparison and considerations

TOSCA is a very flexible abstraction for cloud applications, which can easily represent software, scripts, virtual machines, containers, relationships, and resources for their deployment. Though explicitly designed with cloud applications in mind, the model is very expressive, so that it can also be used for network services [6]. TOSCA shares with OpenStack its cloud-oriented origins, so it mostly focuses on the description of deployment and operation rather than the architecture of the runtime system. A TOSCA model describes the logical structure of the execution environment, which can be built based on the mutual dependencies between software components and resources (virtual machines, operating systems, libraries, application servers, applications). TOSCA supports both declarative and imperative processing; all TOSCA models include artefacts for their deployment, configuration, and management. The derivation of alternative deployment flavours in TOSCA requires the definition of different implementation artefacts for Node and Relationship Templates, but basically does not affect the original Topology Template.

ETSI NFV is more tailored to the specific environment, made of VNFs and network links. While TOSCA prefers a declarative approach, leaving the runtimes the burden (and freedom) to find the better deployment strategy, ETSI NFV relies on a binding orchestration framework to do most of the work in a standard way. As a matter of fact, the ETSI framework delivers predefined execution environments for VNFs (in the form of virtual disk images) instead of describing software components, resources, and their dependencies. Indeed, ETSI MANO orchestrates the composition of network services, but it is not interested in dynamically creation of VNFs from their software model.

The description of NSs and VNFs in the ETSI framework is quite complex, since it aims at covering multiple heterogeneous deployment scenarios. ETSI NFV MANO details resource requirements in the NS but does not extend to the resource model. Instead of defining high-level policies and relying on an optimization engine, the ETSI model enables the definition of many flavours, which differ for the VNF implementation, the number of instances, affinity rules, scaling rules, performance requirements. This simplifies the implementation of the orchestrator (NFVO), which is mostly requested to select one option among those defined by the vendor, and the certification of performance and reliability indexes, since the number of alternative deployments are known. However, it also reduces the flexibility, since it may be difficult to extend the service with additional configuration options that were not envisioned at design time.

IETF SFC focuses mainly on the operation of the service, not how it is described, and treats the SFs as black boxes, considering the chaining of the SFs and the criteria to invoke them as specific to each domain. Independence from the underlying forwarding topology is one explicit target for the IETF framework, which for this reason does not consider resource description.

Enriching a TOSCA application with the ASTRID framework would require some modifications to the Topology Template:

- the ASTRID runtime subsystem should be inserted as Node Template;
- cloud applications should be enriched with additional Node Templates for the ASTRID monitoring and inspection agents.

The interconnection between local agents and the runtime subsystem can be done by the implementation artefacts of the above Templates, hence requiring no modifications to other standard components. Indeed, some additional management plans in the Service Template may be required to include security-related reaction operations (as removal or replacement of compromised software).

For ETSI NFV, the enrichment needed by the ASTRID framework should consider the following points:

- creation of software images for VNFs that includes the necessary kernel hooks and user-space agents for collecting events, logs, measurements;
- the definition of connection points to interconnect VNFs with the centralized ASTRID framework;
- the definition of deployment flavours that include the necessary interconnection between VNFs and the ASTRID security orchestrator (run-time environment), and require encryption and integrity mechanisms;
- the preparation of specific lifecycle management scripts for each VNF that implement reaction strategies when triggered by the ASTRID security orchestrator (e.g., termination, isolation, replacement of the VNF instance).

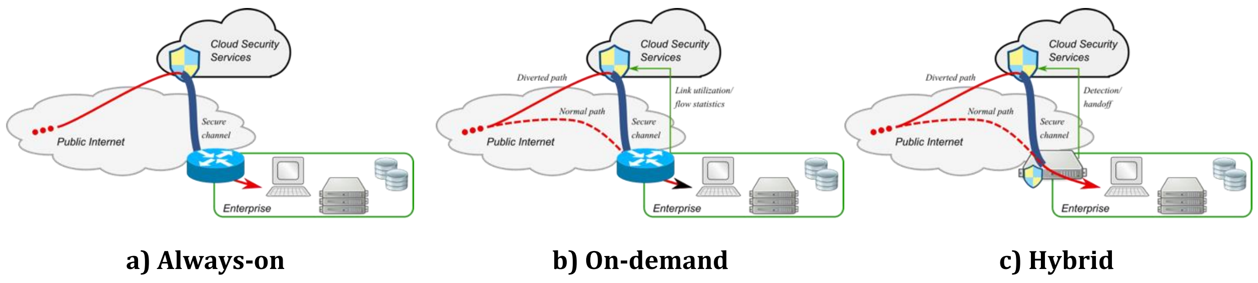
The IETF SFC architecture mainly defines the additional headers to steer packets across multiple SFs but does not consider any standard representation of control policies. This means that it is difficult to define a standard procedure for the enrichment process, since there is no common abstraction of the network service. The lack of a management framework also hinders the definition of common mechanisms for reaction and prevention, since there is no way to change, replace or terminate a SF/SFF. In this case, to fully benefit from the dynamicity of the ASTRID framework, the IETF SFC system should be used in a more general context, also including service orchestration.

Annex B Interface to Network Security Functions (I2NSF)

Fast-evolving threat vectors and ever more complex cyber-attacks are making challenging the design and operation of effective cyber-security infrastructures, not to mention the difficulty to fulfil regulatory requirements. This problem mainly affects small and medium enterprises, which often suffer from a lack of security experts to continuously monitor network, acquire new skills, and propose immediate mitigations. Externalization of security management is emerging as a possible solution.

More and more service providers are today offering cloud-based security services, especially for network protection, by hosting network security functions according to cloud models [18],[19],[20]. Alternative solutions are possible to integrate external security services for attack detection and mitigation into the enterprise (see Figure 18):

- *Always-On*: All traffic is permanently diverted towards the cloud service, which analyses all packet streams and removes any attack. At the customer site, packets are only accepted from the cloud service (through secure channels).
- *On-demand*: Network traffic is normally delivered to the customer site, where traffic statistics are collected by the ingress router and reported to the external cloud service. In case of anomalies in the flow statistics, all traffic is diverted towards the cloud, which now behaves like in the Always-On case.
- *Hybrid*: Protection devices are deployed at the enterprise edge. They analyse and clean network traffic; in case they are not able to mitigate the attack (e.g., large volumetric attacks or unknown attacks) traffic is diverted towards the cloud service.



a) Always-on **b) On-demand** **c) Hybrid**

Figure 18. Cloud protection services.

Always-on and On-demand services better fit the need of small- and medium-sized businesses, which often do not have the budget for expensive security equipment and staff, whereas hybrid solutions represent a smooth transition path for larger enterprises that already have their internal solutions and cannot accept full externalization.

Currently, the usage of cloud-based network security services has some main drawbacks. Every vendor uses different interfaces for its cloud-based services, hence different solutions must be developed by alternative security service providers[16]. Moreover, traffic diversion usually relies on BGP mechanisms, which are very slow and undermine the possibility for quick reaction and seamless operation. Based on these considerations, the I2NSF working group aims at defining a set of software interfaces and data models for controlling and monitoring aspects of physical and virtual NSFs, enabling clients to specify rulesets. As there are many different security vendors or open source technologies

[16] The I2NSF technical documentation uses the more generic term “service provider” to indicate the entity that provides security services. We intentionally avoid this nomenclature, which may be confusing with the main part of this document, and always indicate that entity as “security service provider” or just “security provider.”

supporting different features and functions on their devices, I2NSF only focuses on flow-based NSFs that provide treatment to packets/flows, such as Intrusion Protection/Detection System (IPS/IDS), web filtering, flow filtering, deep packet inspection, or pattern matching and remediation.

B.1 Use Cases for NSFs

RFC 8192 selects several use cases where standard interfaces are required for monitoring and controlling the behaviour of NSFs [14]. These use cases show how NSFs from multiple vendors can be composed together by security providers through their management entities to automate the creation, configuration, and disposal of security services.

Network Service Providers (NSPs) can implement vNSFs as part of the progressive softwarization process of their **access networks**. The implementation of NFV enables NSP to easily create and connect vNSFs close to users, hence representing a valid alternative to physical middleboxes. The range of potential users include residential users, enterprise, mobile users, and also management systems, each with its own access clients. For instance, residential users may request parental control, content management, threat management (web contents, mail, files download). Enterprises may be interested in blocking web sites or social media applications, phishing attacks, malware, botnet, DDoS, and others. Service providers may need policies to securely and reliability deliver contents to their customers, provide isolation between multiple tenants, block malware and DDoS.

In cloud data centre, **virtual firewalls** can be used to add more filtering capacity when bandwidth utilization hits a certain threshold for a specified period of time. The need to deliver on-demand security services motivates the implementation of such appliances in software or virtual forms, rather than hardware appliances. This is also necessary in order to place the firewall instances in the right zone, close to the rack of servers where protected applications are running. Indeed, the deployment of standalone physical appliances for each customer is not technically and financially possible, whereas sharing them complicates their management. The typical requirements is therefore the ability to dynamically deploy and configure virtual firewalls within the tenant partition, according to the position and topology of the user service and its required security policies.

Firewall rules are usually expressed in terms of allowed/blocked addresses, ports, protocols, and a few other parameters. Though the same concept is applied in almost all products, the syntax for rule configuration changes from vendor to vendor, making it difficult for automation. With complex service topologies, the identification of all rules that satisfy all security requirements is usually difficult (and error-prone). Indeed, integrated firewall solutions in cloud management software often provider "security groups" or similar feature to easily identify the set of servers allowed to freely communicate. More **automation in the deployment of firewall policies** would therefore benefit from standard interfaces, which works across multiple vendors and utilize dynamic key management.

Cloud environments provide isolated execution sandboxes, even made by multiple interconnected computing units, but substantially confined to a single data centre or the set of data centres belonging to the same provider. Apart basic firewalling functions, there is currently no standard security services implemented by every provider. The interconnection of cloud services with the enterprise or other external networks must be completely managed by the users. Further, cloud users must trust cloud providers, without any **visibility on security policies** that are applied to protect the infrastructure from external and internal threats. Indeed, no standard interfaces exist to retrieve and manage security policies in a consistent way across different providers.

The **convergence of multiple network services to a common network infrastructure** cuts down CAPEX and OPEX, but also raises more management concerns due to the inter-dependency that are created among the different domains. Examples of service networks include the Void over IP (VoIP), Voice over LTE (VoLTE), Content Delivery Networks (CDNs), Internet of Things (IoT), Information-Centric Networks (ICNs). Attacks to one of these networks may easily jeopardize the operation of other networks, when proper isolation mechanisms are not correctly applied (e.g., network segmentation, network slicing). Preventing DDoS, malware and botnet attacks is not easy, especially when client devices do not confirm to strong security standards (e.g., IoT). The presence of multiple network must not foster the proliferation of bespoke security services and tools. Rather, a standardization effort is required to develop interfaces that are user case independent and technology agnostic, i.e., able to support multiple protocols and data models. This would simplify the application of common security policies across multiple environments, would facilitate the coordination of prevention, reaction, and mitigation measures, and would improve early detection through larger visibility on the execution environment.

The sensitive and critical nature of many digital services requires organization to comply with **regulatory and compliance security policies**, so to isolate various kinds of traffic as well as to be able to show logs and records in case of audit. Examples include the Payment Card Industry – Data Security Standard (PCI-DSS) or the Health Insurance Portability and Accountability Act (HIPAA). Common interfaces to multiple security tools would facilitate the tracking of applied security policies, security events, and security incidents. This could be used in case of audit as proof that traffic was isolated between specific endpoints and all required measures were applied.

B.2 Challenges to provide NSFs

Provisioning of NSF, both in physical or virtual forms, currently faces many challenges, for both service providers and customers [14]:

- **The heterogeneity of NSF in terms of services and features**. As a matter of fact, security functions may be deployed at the network perimeter, in DMZ, on single devices, both in centralized or distributed forms. Possible services range from access control (firewalling, deep-packet inspection, proxies) to detection and mitigation (IPS, IDS), authentication services, endpoint protection, monitoring and correlation (SIEM), encryption and integrity (VPN concentrators and gateways, security gateways). This complicates the definition of common models and interfaces to describe the features and configurations.

- **The heterogeneity of control and management interfaces**. Given the heterogeneity of NSFs and the lack of accepted industry standards, control and management APIs are substantially different for any vendor. This complicates automation of such services, since it is difficult to translate security policies into a different set of control commands.

- **The difficulty in monitoring and tracking the effects of security policies**. This is necessary to know that the intrusion has been stopped, as well as to access the effectiveness of mitigation and response actions. Customers want this monitoring feature in order to plan for the future using "what-if" scenarios with real data. A tight loop between policies and configurations can reduce the time to design and deploy workable security policies that deal with new threats.

- **The increasing usage of dynamic and distributed environments**. With the advent of software-defined technologies for computing and networking, more clients and applications need to dynamically update their security policies, hence to dynamically change the configuration of NSFs. The problem is not limited to the provisioning and allocation of NSFs, but also extends to the translation of security requests to configuration commands. A single NSF may also be shared by multiple tenants, especially when it is implemented with elastic cloud-based technologies, leading to the needs of partitioning resources and avoiding conflicts.

- **Lack of characterization and exchange of capability**. Even the same kind of NSFs may have rather different capability. The knowledge of such capability (either by static description or automatic registration) is required to select and compose NSFs to create specific security services. In addition, (dynamic) negotiation of resources is necessary to size and adapt NSFs to variable workloads.

- **Lack of mechanisms to utilize external databases**. Many security functions depend on signature files, threat intelligence, or other attack description (often referred as "profiles"). There is currently no standard way to build external profiles to be shared by multiple NSF instances; indeed, the effectiveness of the protection heavily relies on the updates supplied by each vendor. As new and more complex threats arise, protection can be improved if enterprises, vendors, and service providers cooperate to develop shared profiles.
- **Lack of automation and integration with software-defined infrastructures**. Effective response to known attack would benefit from more automation. This is possible if a standard mechanism exists to signal anomalies, so that a security controller can re-configure the environment, both NSFs and network policies (routing, forwarding, filtering).
- **Lack of management tools** for dynamic key distribution to NSFs for building security associations.
- **Lack of tools and frameworks for managing high-level policies**. Customers may not have the security skills to select the right set of NSFs and define their configuration. They usually have expectations about their high-level security requirements (protect against external DDoS, guarantee availability of a group of servers, ensure confidentiality and integrity of communications with external sites, etc.). Unfortunately, there is no standard tool today for translating these high-level policies into security functions and their configuration.

B.3 I2NSF framework

I2NSF defines a framework to manage and control external NSFs. The framework assumes the presence of "security users" or "customers", which need security services provided by NSFs. The reference model for I2NSF is based on two functional layers:

- The **Capability Layer** specifies how to control and monitor NSFs. I2NSF will standardize a set of interfaces by which a customer can invoke, operate, and monitor NSFs.
- The **Service Layer** describes how client's security policies may be expressed. This is not limited to enforcement and protection actions, but also includes monitoring to build situational awareness.

Between the two layers, the I2NSF Mgmt System translates policies into commands for NSFs. A client might also interact directly with one or more NSFs through the Capability Layer, but in this case, it will lose the abstraction brought by the Service Layer. From a business perspective, the operation of the I2NSF framework requires a new actor, the Security Service Provider. With respect to the framework users, it can be an internal or external entity. For example, it could be the internal IT security group of a large enterprise; in case of full externalization, it could be a cloud service provider or an independent entity.

Figure 19 depicts the I2NSF Reference Model and identifies the I2NSF interfaces [15]. I2NSF Users define, manage, and monitor security policies for specific network flows within an administrative domain, through the I2NSF Consumer-Facing Interface. The I2NSF NSF-Facing Interface is used to specify and monitor security rules enforced by one or more NSFs. Finally, the I2NSF Registration Interface defines the capabilities of the NSFs, which can be either statically configured or dynamically retrieved.

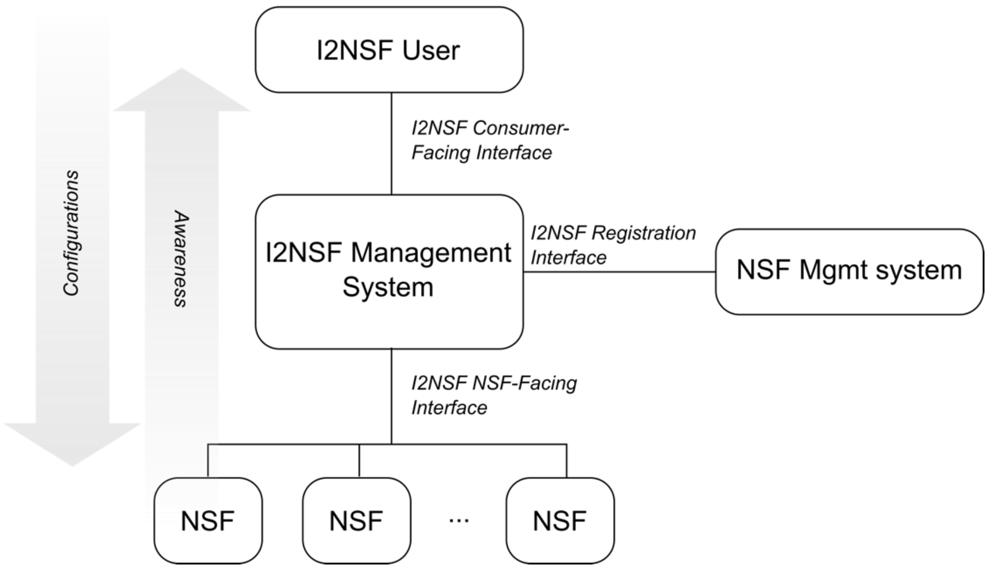

I2NSF User
I2NSF Consumer-Facing Interface
I2NSF Management System
I2NSF Registration Interface
NSF Mgmt system
I2NSF NSF-Facing Interface
NSF
NSF
···
NSF
Configurations
Awareness

Figure 19. I2NSF Reference Model [15].

B.3.1 I2NSF Users and Consumer-Facing Interface

I2NSF Users may be humans (i.e., ICT staff in the enterprise) or their management clients (upstream application, BSS/OSS of network service provider, orchestration software, security portals, etc.). This covers different use cases, where security services are offered to large enterprises, small- or medium-sized businesses, and retail customers [14]. For example, a video-conferencing manager may dynamically inform the underlay network to allow, rate-limit, or deny flows (some of which are encrypted) based on specific fields in the packets for a certain time span. The Consumer-Facing Interface is specifically conceived to decouple security services from NSFs, allowing portability across different administrative domains.

Different levels of abstraction could be used to express security policies. In general, customers may not have the skills to define configurations and flow-level policies. For this reason, this interface is expected to model expectations, goals, or intents of the functionality desired by customers. Customers may give indicate which types of destinations are (or are not) allowed for certain users (e.g., enable Internet access for authenticated users, streaming media applications are prohibited on the corporate network during business hours, and so on).

Despite of their simplicity, some user policies may need multiple NSFs located in different places to achieve the desired behaviour. Clearly, the specification of user policies with very similar models than flow policies simplifies the translation.

B.3.2 NSFs and NSF-Facing Interface

Network Security Functions are physical or virtual instances of monitoring and enforcing appliances. They are expected to implement two sub-interfaces:

- *operational and administrative interface*, which is used to change the status and configuration of the NSF, in order to dynamically change their behaviour;
- *monitoring interface*, which can be query- or report-based.

The number of different types and implementations of NSFs is very large, so it may happen that the set of available NSFs are not able to fulfil the customer's request. The I2NSF system must therefore support dynamic discovery of capability, as well as query mechanisms, so that the management system can select the functions that satisfy the customer requirements. Dynamic negotiation will also allow to tune the requirements based on available services/features. The outcome of the negotiation would feed the I2NSF Management System, which would than dynamically allocate appropriate NSFs and configure the set of security services that meet the requirements of the user. RFC 8329 [15] has already identified preliminary characteristics, categories and fields for discovery and registration of capabilities.

B.3.3 I2NSF Security Controller

At the heart of the I2NSF Management Framework, a Security Controller mediates between user's policies and NSF. The Security Controller receives user's policies (in terms of requirements, intents, goals) from customers and translates them into commands that NSFs can understand and execute. The NSF management includes six fundamental operations: create, read, write, delete, start, and stop. They cover both the management of imperative policy rules and management of the NSFs. The Security Controller also gathers monitoring reports (e.g., statistics) from the NSFs and passes back them to the customer. The Security Controller does not only collect individual service information but can also aggregate data suitable for tasks like infrastructure security assessment.

The I2NSF framework assumes the definition of flow-based **policy rules**. Flow level policies may be defined as imperative Event-Condition-Action (ECA) constructs, that define how the system react to given events and conditions. Flow-based NSFs are based on stateful processing, i.e., they consider both the packet content (headers and payload) and context (session state). Currently, the main focus is only on imperative paradigms for policy rules. Even though security functions come in a variety of form factors and have different features, ECA rules would support a wide range of possible behaviours for flow-based NSFs.

Flow-based policy rules should match quite well the NSF-Facing interface, since most types of events, conditions, and actions are common to many security functions, even if different information and data models are used by different vendors. For example, events can include current date/time, notification of state change, user logon/logoff. Conditions may be related to the value of some fields in the packet header (source/destination addresses, ports, protocol type, flags, etc.), packet size, direction of the traffic, geo-location, connection status. Actions may include processing on incoming/outgoing packets (pass, drop, rate limiting, mirroring, encapsulation, forwarding, transformation), loading of functional profiles (IPS profile, signature file, virus definitions, etc.).

On the other hand, the definition of user policies follows more a goal/intent approach, which in general does not reflects the operations of NSFs. In this case, the translation is far more challenging. For this reason, the I2NSF framework is currently limiting the scope to user policies that can be modelled as closely as possible to flow security rules used by individual NSFs. A I2NSF user policy should therefore adopt the ECA structure, but with more user-oriented expression for events, packet context and context, and enforcement actions. In this case, an event could be "user has authenticated", a condition may be "user identifier", and action something like "establish encrypted channel."

Management of NSFs include provisioning and life-cycle operations (start, stop, scale, update), configuration of operational attributes (network addresses, endpoints, number of internal threads, log files, etc.), signalling, and setup of policy rules (this latter already discussed in the previous paragraphs). NSFs could be clustered together and managed by a common system; in this case, the Security Controller interacts with the NSF Manager directly (Figure 20).

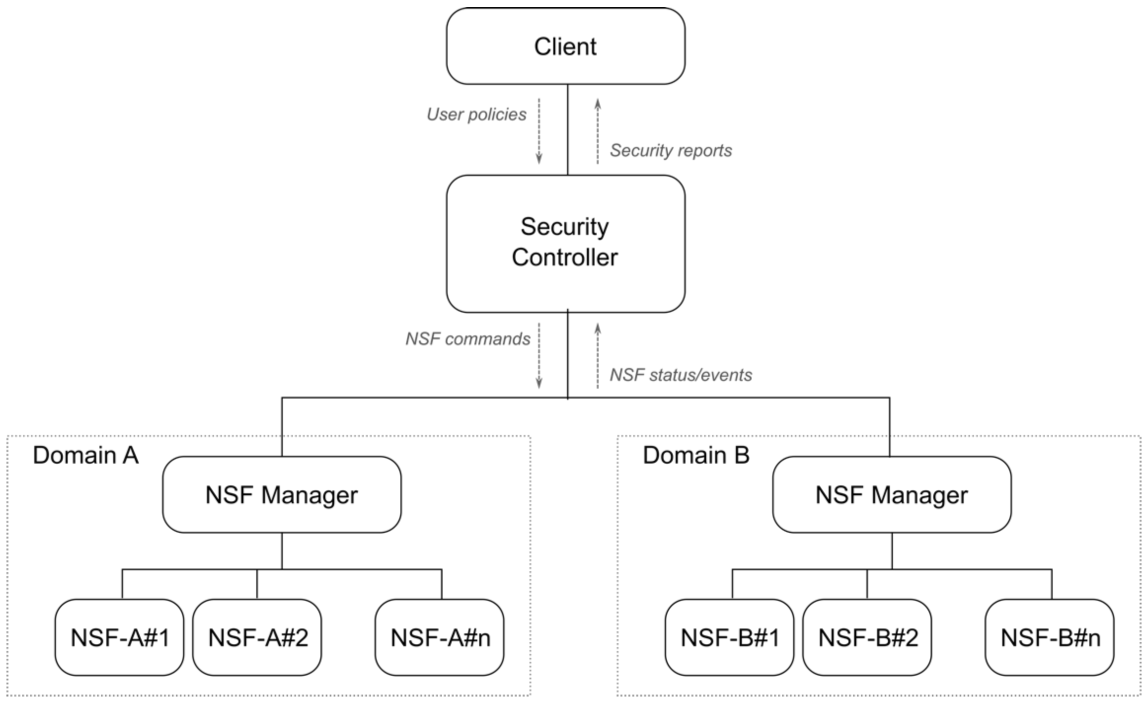


Figure 20. I2NSF management architecture.

The presence of an NSF Manager is a typical case for vNSFs provided by IaaS. In this case, it can dynamically create NSFs for each user and control its whole life-cycle, from instantiation to termination. An NSF Manager can also be used with physical and SaaS instances, which statically deployed by the Security Provider, but could be chained or composed at run-time to create tailored security services. In this case, the NSF Manager hides the number and heterogeneity of the underlying NSFs to the Security Controller and becomes responsible to translate policy rules and ensure their consistency across the set of NSFs.

B.4 Security and trust

The externalization of security services improves the flexibility and resilience, but also brings additional threats due to the lack of security perimeter, multi-tenancy, and third-party infrastructures. The most relevant threats associated with an NSF platform are [15]:

- the control of NSFs by **unauthorized users**, who may change the policy rules so to bypass the intended behaviour or to cause DoS;
- **misuse by authorized users**, who may bypass isolation mechanisms to
  - alter the configurations of other users in the underlying NSFs;
  - take control of an entire NSF or provider platform, by exploiting vulnerabilities in the application or control protocol;
- **loss of integrity** of the NSF software or the underlying platform, which undermines the correct operation and privacy requirements by all users and can be due to:
  - physical attacks to the platform, by console or serial lines, to modify the behaviour of the hardware/software with the same level of privileges as the provider;
  - malicious service providers that modify the software (Operating System or NSF implementation), this represents the most critical case since service providers are the ultimate responsible to detect attacks to the infrastructure.

B.4.1 Secure communication channels

The recommended security mechanisms include:

- authentication, authorization, accounting, and auditing for all users and applications that access the I2NSF environment;
- attestation of NSFs by service providers or third parties, so to detect changes to the I2NSF environment.

According to the reference model shown in Figure 19, the primary vulnerable point for an I2NSF platform is the Customer-Facing interface, which is publicly exposed to all users. This interface may be subject to spoofing, eavesdropping, replication, alteration, privilege escalation, misuse, DoS. A mutual authentication is therefore required between users and the Security Controller, as well as a trusted connection upon successful authentication. The basic requirements for a trusted connection are not limited to confidentiality and integrity, but also entail strong authentication of the peers and attestation of their integrity.

Based on the overall framework scope, the NSFs are not expected to be directly attached to the Security Controller, but to be distributed across the network. There can be a common network for the NSF-Facing interface and data traffic processed by the NSFs or, better, a dedicated network for management purposes only. In either case, packet loss could happen due to failure, congestion, attacks, or other reasons. Therefore, the transport mechanism for the management interface must be secure. The I2NSF framework does not require reliable transport mechanisms; rather, reliability should be directly implemented in the interface protocol by introducing explicit acknowledgement of messages into the communication flow. This would achieve better latency in the delivery of control messages.

Indeed, the selection of security mechanisms for the NSF-Facing interface depends on the specific network segment between the Security Controller and NSFs. When the I2NSF platform is built in a single administrative domain (e.g., it is implemented by a Network Service Provider in its own infrastructure), it can be safely assumed the substantial isolation and protection of the communication environment. In this case, some requirements on authentication and trustworthiness could be partially loosened. Instead, in open environments where the NSFs functions are hosted in multiple external domains, more restrictive security control should be placed over the same interface. The same procedure as for the Customer-Facing interface could be used to establish a trusted connection.

B.4.2 Remote attestation

While it is true that any ICT environment is vulnerable to get compromised by malicious users with physical access, the application of attestation mechanisms improves the trustworthiness of the system by raising the degree of control and physical activity that are needed to perform untraceable modification of the environment.

Within the I2NSF framework, remote attestation is an inescapable requirement to properly address the cooperation between different actors in disjoint administrative domains: users (I2NSF customers), security providers (I2NSF service providers), and security functions (I2NSF Network Security Functions) [17].

From the user perspective, the establishment of trust in a I2NSF platform requires two steps: i) verify the authenticity of the Security Controller, and ii) get proof that NSFs and policy rules are compliant with their security choices.

To set up security services through a NSF platform, the user's client connects and authenticates to the Security Controller. However, before initiating authentication and authorization procedures (and likely accounting and auditing), the client wants to attest that it is connected to a genuine Security Controller. Two properties characterize the genuineness of the Controller: its identity and its integrity. Afterwards, the client can authentication, start a trusted connection with the Controller, and ask for some security services. Before any traffic is actually redirected through the set of NSFs, the client must be sure that it will be processed according to the user policies. The attestation of the selected NSFs and the applied policies must happen at initialization and may be optionally repeated at run-time to detect any following loss of integrity. The attestation of a NSF platform include multiple elements, some of which are present in all possible implementations: firmware, OS, NSF software, in a virtualized environment, the virtualization system (hypervisors, host OS, container frameworks, …).

The attestation of NSF platforms can be enough in case of hardware implementations connected by physical links, which represents a static configuration that can only be modified by the infrastructure provider. However, the dynamicity brought by software-defined networking paradigms, like SDN, NFV and SFC raises additional concerns about the correctness of the network topology (which could bypass the security functions), hence demanding for attestation of the network configuration as well.

The enabling technology for remote attestation is trusted computing. The underlying concept is the presence of hardware which serves as a trust anchor to start a chain of attestations (i.e., *chain of trust*). Currently there are two main trends in this area, driven by two standardization bodies: The Trusted Computing Group (TCP) and the Global Platform (GP). The TCP define the Trusted Platform Module (TPM) [21], a collection of cryptographic functions often implemented by a dedicated hardware chip outside the main processor. The GP specify the Trusted Execution Environment (TEE) [22], this is a secure isolated environment on the same System-on-Chip (SoC). The I2NSF framework is currently considering the TPM solution. The TPM defines an architecture with the following capabilities: performing public key operations, computing hash functions, key management and generation, secret storage of keys and other secret data, random number generation, integrity measurements, attestation… It uses a transitive mechanism: if a user trusts the first execution step (i.e., the hardware *root of trust*, RoT), and each step correctly verifies the integrity of the next executable firmware/software, then the user can trust the whole system. In more concrete terms, every boot stage (BIOS, Bootloader, Security Controller) measures the integrity of the following piece of software and stores it inside a log that reflects the different boot stages, which is then signed with the private key of the RoT. In a TPM, measurements of the software stack are concatenated, so that an unlimited number of measures can be stored in a single on-board Platform Configuration Register (PCR).

The I2NSF also envisions the possibility of a trusted boot for safely storing secrets. In this case, the PCR values could be used as an identity for decrypting confidential information on the server (as encryption keys or sensitive configuration). The basic idea is to encrypt such data by the root of trust, and then subordinate decryption to a particular platform status (i.e., a set of PCR values). This ensures that only trusted software can access keys and other sensitive materials for authentication (for instance, the private keys used to create the secure channel with the client: TLS, IPSec, etc.).

As final consideration, we argue that remote attestation entails the knowledge of the exact firmware/software configuration of both the Security Controller and NSF platform, which might help identify vulnerabilities. The client may delegate a Trusted Third Party (TTP) to compute the integrity of the Controller, hence avoiding the implementation of the whole attestation mechanisms.

B.4.3 Security Controller attestation

The attestation of the Security Controller is a required step for establishing a trusted communication channel. A trusted channel is more secure than an encrypted channel. It entails stronger verification of the identity of the Security Controller, by including its public key or certificate in the measurements of the chain of trust. This prevents spoofing and man-in-the-middle attacks, since a malicious user cannot push its certificate in the chain of measurement of the genuine platform. However, there is still the case where the confidentiality of the private key is lost. This can be solved by a Platform Property Certificate, which binds system properties instead of binary data [23]. Such certificate could connect the platform identity with the Attestation Identity Key (AIK) public key, so hindering the usage of the stolen Security

Controller's key on a different platform (the attacker cannot create a quote with the AIK of the other platform).

The procedure to establish a secure connection with the Security Controller will include therefore the following steps:

1. The client begins the handshake with the Security Controller.
2. The client receives the certificate of the Security Controller.
3. The client asks the Security Controller to generate an integrity report.
4. The Security Controller retrieves the measurements and asks the TPM to sign the PCRs with the AIK. The signature provides evidence that the measurements belong to the Security Controller.
5. The client checks the integrity report, by verifying the quote and the certificate associated to the AIK. The client also checks that the digest of the certificate received at step 2 is present among measurements. These operations can be delegated to a TTP, which may be useful in case of lightweight clients.
6. If the remote attestation is positive, the client continues the handshake and establishes the trusted channel; otherwise, the connection is closed.

B.4.4 NSF platform attestation

The attestation of the NSF platform checks the integrity of the NSFs and their correct behaviour. Platform attestation does not cover the mechanism used to translate user policies into policy rules and NSF configurations, which would be technically too complex (if not unfeasible). The trustworthiness of this process indeed relies on the integrity and attestation of the Security Controller, which the client selects as the reliable intermediary for managing its security services.

The attestation of NSFs can therefore include the integrity of the software, the hosting environment (a physical device, a virtualization platform), and the running configuration. The quotes can be compared with the status information maintained by the trusted Security Controller, with the same procedure that could also be used to SDN (see Section B.4.5).

B.4.5 Topology attestation

Two methods exist to attest the deployment of a topology of a software-defined network topology. The first one is based on typical SDN operation, like in case of OpenFlow, and the second targets the application of SFC.

In the first case, each network node provides measures on its forwarding configuration, which are aggregated and signed by a TPM module. The attestation is then compared with the configuration retrieved from an SDN controller, to verify the correct behaviour of the network.

In the second case, the Proof of Transit can be applied. This mechanism injects specific packets requesting POT and verifies at the egress point of the service path that a correct topology has been enforced, by means of the cryptographic proof provided by POT.