An Assurance Framework and Process for Hybrid Systems*

Marco Anisetti[1][0000−0002−5438−9467], Claudio A. Ardagna[1][0000−0001−7426−4795], Nicola Bena[1], and Ernesto Damiani[2][0000−0002−9557−6496]

[1] Dipartimento di Informatica, Università degli Studi di Milano, Milan, Italy
{marco.anisetti, claudio.ardagna, nicola.bena}@unimi.it
[2] Center for Cyber-Physical Systems (C2PS), Khalifa University, Abu Dhabi, UAE
ernesto.damiani@ku.ac.ae

Abstract. Security assurance is a discipline aiming to demonstrate that a target system holds some non-functional properties and behaves as expected. These techniques have been recently applied to the cloud, facing some critical issues especially when integrated within existing security processes and executed in a programmatic way. Furthermore, they pose significant costs when hybrid systems, mixing public and private infrastructures, are considered. In this paper, we a present an assurance framework that implements an assurance process evaluating the trustworthiness of hybrid systems. The framework builds on a standard API-based interface supporting full and programmatic access to the functionalities of the framework. The process provides a transparent, non-invasive and automatic solution that does not interfere with the working of the target system. It builds on a Virtual Private Network (VPN)-based solution, to provide a smooth integration with target systems, in particular those mixing public and private clouds and corporate networks. A detailed walkthrough of the process along with a performance evaluation of the framework in a simulated scenario are presented.

Keywords: Assurance · Hybrid System · Security · Virtual Private Network

1 Introduction

In today digital and connected society, users and enterprises interact with smart services and devices to carry out day-to-day activities and business processes. Distributed systems are rapidly and continuously evolving, from service-based systems to cloud and microservices-based architectures and, more recently, towards Internet of Things (IoT) and edge infrastructures. At the same time, traditional private infrastructures are still widely used, resulting in hybrid systems mixing public and private endpoints and introducing new concerns undermining the users' perceived trust (e.g., [25]).

* Research supported, in parts, by EC H2020 Project CONCORDIA GA 830927 and Università degli Studi di Milano under the program "Piano sostegno alla ricerca".

In the last couple of decades, the research community has extensively produced new solutions to increase the trustworthiness of such systems. Security verification and protection have been increasingly important, and should be fully integrated within systems' lifecycle and executed in a automated way. Security assurance, defined as way to gain justifiable confidence that IT systems will consistently demonstrate a (set of) security property and operationally behave as expected [6], is gaining the momentum. Assurance solutions in fact have been applied to service-based systems, cloud, and IoT [7,9], addressing novel and peculiar requirements such as multi-layer evaluation and continuous monitoring, as well as evidence-based verification. Notwithstanding their huge benefits, little focus has been put in defining assurance frameworks that can be easily integrated into existing hybrid systems, complementing other security processes and providing a programmatic way to execute assurance evaluations. Many of the existing assurance techniques and frameworks (e.g., [10,14]) are ad hoc and cannot handle a modern IT system as a whole. They require some effort for being integrated with the target system, interfering with its normal operation (e.g., performance) and introducing not-negligible (monetary and business) costs. Also, they fall short in providing some form of automation.

In this paper, we extend our assurance framework in [4] enabling a centralized security assurance, complementing existing security processes and systems, and providing automation of assurance activities. The framework implements an API-based approach facilitating integration and automation, and relies on Virtual Private Networks (VPNs) to target both public and private infrastructures. Our contribution is threefold. We first define the requirements a security assurance framework and corresponding process have to fulfill in our hybrid scenario (Section 2). We then propose a novel API-based assurance framework addressing these requirements and targeting hybrid systems (Section 3). To this aim, the assurance process implemented by the framework relies on an enhanced REST interface (Section 4) and on several modifications to a standard VPN configuration (Section 5). We finally present a detailed walkthrough of such a process (Section 6.1), an experimental evaluation of the framework performance (Section 6.2), and a comparison with the state of the art according to the identified requirements (Section 7).

2 Assurance Requirements

The advent and success of cloud computing and Internet of Things (IoT) are radically changing the shape of distributed systems. Hybrid systems, building on both private and public technologies, introduce new requirements and challenges on security assurance techniques, which must take a step forward for being applicable to modern architectures. In particular, the definition of new assurance processes is crucial to fill in the *lack of trustworthiness* that is one of the main hurdles against the widespread diffusion of such systems.

Despite targeting complex systems, a security assurance process should be lightweight and not interfere with the normal operation of the system under verification. The need of a lightweight process is strictly connected to its *psychological acceptability* [22], meaning that final users are more willing to *accept* to perform assurance activities that preserve the behavior of the system and do not increase overall costs. In fact, although the undebatable advantages given by a continuous evaluation of system security, users are recalcitrant with respect to a process perceived as heavy and costly [27].

Cost management and optimization are the foundation of assurance adoption. Costs refer to *monetary costs* in terms of additional human and IT resources, as well as *performance and business costs* in terms of overhead, latency, and reliability. *Monetary costs* include the need of highly specialized personnel, on one side, and resources allocated and paid on demand on the other side, which are spent to manage non-functional aspects of the system often considered as superfluous. *Performance costs* include the need of continuously verifying the security status of a system. They intrinsically introduce a not-negligible overhead and latency, an assurance process has to cope with. Assessment activities are only viable if they take resource demands under control, avoiding scenarios where they become a source of attack. *Business costs* are partially overlapped with performance costs and model how much assurance activities interfere with the normal operations of a business process. On one side, the changes required to connect an assurance process to the system under evaluation should be reduced to the minimum, and mostly work at the interface level. On the other side, an assurance process cannot threat itself the system. For example, run-time verification of a system security status cannot increase the risk of system unavailability by performing penetration testing on the production system. A good balance between active and passive testing/monitoring should be provided. Finally, security assurance is just one of the security activities that should be performed. An assurance process must complement and integrate with traditional detection and prevention security, by means of an assurance framework implementing a (semi-)automatic process that is easy to integrate with existing security solutions.

We identify the main requirements an *assurance process* has to satisfy (MUST/ SHOULD) to address the peculiarities of modern systems, as follows.

Transparency: it MUST not interfere with the normal operation of the business process, being transparent to the final user of the system where the assurance process is performed.

Non-invasivess: it MUST require the least possible set of changes to the target system.

Safety: it MUST not introduce (or at least minimize) new risks on the target system.

Continuity: it SHOULD provide a continuous process, verifying the status of security while the system is operating and evolving.

Lightness: it SHOULD be lightweight and cope with systems having limited resources.

Adaptivity: it SHOULD be dynamic and incremental to adapt to changes in the system under verification and its environment.

Complementarity: it SHOULD complement and integrate with existing security processes.

Such requirements should be supported by a centralized framework tuning each aspect of the assurance evaluation. The *framework* itself has its own requirements [5], which are summarized in the following.

Evidence-based verification: it SHOULD implement a verification built on evidence collected on the target system, to get the real picture of its security status.
Extensibility: it MUST inspect hybrid targets, from traditional private networks to public clouds, as well as hybrid clouds and IoT.
Multi-layer: it SHOULD assess system security at different layers, from network protocols to application-level services.
Scalability: it SHOULD support a scalable process, able to manage an increasing number of assurance processes and evaluations.
Automation: it SHOULD be an automatic or semi-automatic process, whose actions can be triggered either manually or by external events.

Generally speaking, an assurance framework MUST *at least* implement a process that has the lowest possible impact on the target resources and normal system activities (*transparency*), do not modify the current ICT infrastructure or at least require very few modifications (*non-invasiveness*), do not affect security by introducing new risks (*safety*) or hindering existing security processes (*complementarity*), while being generic enough to address peculiarities of hybrid systems (*extensibility*).

3 Assurance Framework

We present a framework that provides a lightweight assurance solution addressing the peculiarities of modern distributed systems, mixing public endpoints on the cloud, microservices, and *private* deployments not directly reachable from the outside (e.g., *traditional* private corporate networks and private clouds). The framework has been first defined in [4] and here extended to address requirements *complementarity* and *automation*. The original framework in [4], offering only a graphical dashboard, constrained the ability to integrate framework's functionalities with existing security processes, and to trigger such functionalities in a automated way. To address all requirements in Section 2, it adopts a layer-3 VPN that connects the framework with the private deployments under verification (i.e., the *target networks*), and offers a REST API providing full and programmatic access to framework's functionalities.

The architecture of the assurance framework is presented in Figure 1 and aims to address two main scenarios: *i)* support from programmatic integration of the framework within existing (possibly legacy) systems, *ii)* support for the verification of modern systems mixing public and private endpoints. Concerning scenario *i)*, the framework can be used either *manually* by users interacting with

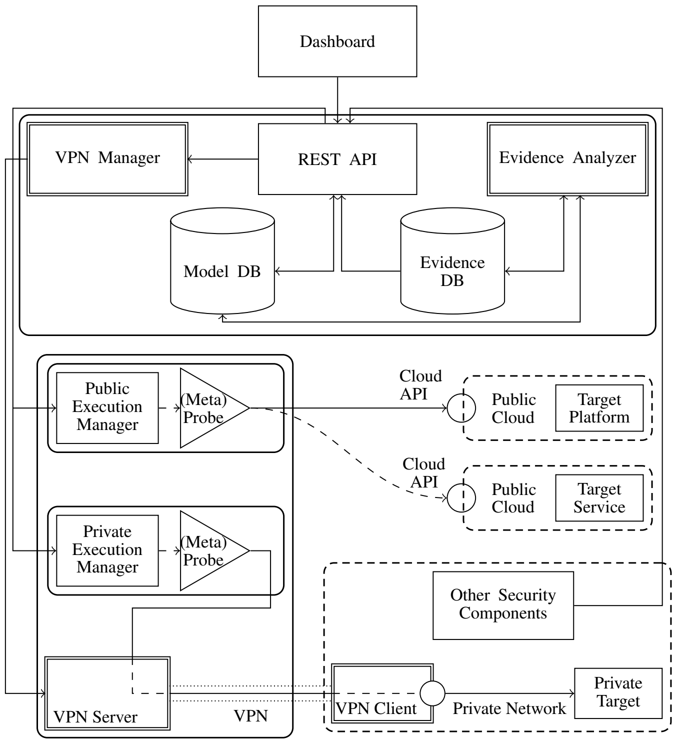


Fig. 1: Our framework architecture. Double line rectangles highlight new components.

the dashboard, or *programmatically*, by exploiting standard *REST APIs*, thus satisfying requirements *complementarity* and *automation*. Concerning scenario *ii)*, the framework implements a VPN-based approach to seamlessly integrate with and verify private corporate networks and private clouds, thus satisfying requirements *transparency*, *non-invasiveness*, and *extensibility*. To this aim, different *VPN Servers* are installed within the framework, each one responsible to handle isolated VPN tunnels with client devices placed in the target networks. A single VPN connection consists of a *VPN Client* directly connected to the target network, and a *VPN Server* installed in the framework.

The framework manages an assurance process (Section 5) that consists of a (set of) evaluation rule (evaluation in the following). Each evaluation is a Boolean expression of test cases, which are evaluated on the basis of the evidence collected by *probes* and *meta-probes*. *Probes* are self-contained test scripts that assess the status of the given target by collecting relevant evidence on its behavior. They return as output a Boolean result modeling the success or failure of the test case. *Meta probes* are defined as *probes* collecting *meta-information*, such as the response time of a service. The framework components are summarized in the following.

Dashboard is the graphical user interface used to configure new evaluations and access their results. It works by connecting to the APIs provided by component *REST API*.

REST API manages the overall assurance process by the means of a REST interface. Upon receiving an evaluation request, either from the *Dashboard* or directly from the APIs, it creates the necessary objects in the database and orchestrates their execution.

Execution Manager is in charge of the evaluation process. It selects and executes the relevant *probes*. There are two types of *Execution Managers*: one targeting public clouds (*Public Execution Manager*), and one targeting private deployments (*Private Execution Manager*). The only difference between them is the way in which traffic is routed to the destination.

Model Database is the main database. It stores most of the information needed by the framework, including evaluation configurations and target details.

Evidence Analyzer produces the overall result of an evaluation by collecting the results of the single test cases and validating them against the Boolean expression of the evaluation.

Evidence Database stores the results of *probe* execution, including both the collected evidence and the Boolean results.

VPN Server is a dedicated VM running the VPN software. It handles several VPN tunnels, one for each private network, which are strictly isolated. It acts as a default gateway for multiple *Private Execution Managers*.

VPN Client is physically located into the target network. It establishes a VPN connection with the *VPN Server* in the framework, traversing the firewall protecting the private network.

VPN Manager is a REST API service that manages the automatic configuration of the VPN. It automatically generates configuration files and handles all activities needed to manage VPN connections.

VPN Client and *VPN Server* are the stubs mediating the communication between the target system and the framework, respectively. They act as intermediaries supporting protocol translation and VPN working, and interacting with the *VPN Manager* for the channel configuration.

The framework supports *scalability* by scaling horizontally the low-level execution components, such as the *Execution Manager* and *VPN Server*.

Example 1. Let us consider an assurance evaluation targeting a public website composed of two test cases chained with a logic AND: *i)* a test case evaluating compliance against Mozilla best practices for websites and *ii)* a test case evaluating the proper configuration of HTTPS. The *Execution Manager* executes the assurance process as follows. Two *probes* are executed to collect the evidence needed to evaluate the two test cases, producing two Boolean results. Those results are then evaluated by the *Evidence Analyzer* according to the evaluation formula, a conjunction (AND) of test cases *i)* and *ii)*. As such, the overall evaluation is successful if and only if both test cases succeed.

4 REST Interface

REST is a popular paradigm for developing backend applications, which is based on the concepts of *resources* and *operations* performed on such resources, in terms of HTTP paths (resources) and HTTP methods (operations). A REST interface can be described by using the *OpenAPI* standard, the standard for documenting REST applications [19]. The standard itself is referred to as the *OpenAPI specification*, and defines the format of the application's documentation, which is referred to as *OpenAPI document*. A valid OpenAPI document is a JSON object and can be represented either in JSON or YAML.

Our assurance framework builds on an API-based approach, where component *REST API* provides a REST interface complemented by an OpenAPI document. Together, they facilitate the integration of assurance activities with existing security solutions, addressing requirement *complementarity*, and the automation of such activities, addressing requirement *automation*.

4.1 OpenAPI Document

An OpenAPI document is composed of different parts describing all the aspects of a REST service: resources, operations on such resources, valid requests and responses, possibly with examples, as well as non-functional aspects, such as how authentication is handled. They are briefly described in the following, along with short excerpts of the OpenAPI document of our component *REST API*.

Metadata is the header of the document and specifies, among the others, the version of the specification the document adheres to, a high-level description of the APIs, and the version of the APIs the document refers to.

```yaml
openapi: 3.0.1
info:
  contact:
    email: info@moon-cloud.eu
  description: Moon Cloud REST API are the most important component of Moon Cloud,
↪  governing the overall framework.
  license:
    name: BSD License
  termsOfService: https://www.moon-cloud.eu/policies/terms/
  title: Moon Cloud API
  version: v1.9.9-alpha
```

The *metadata* excerpt shows section metadata of our framework. For instance, it shows that the version of the APIs is `v1.9.9-alpha`.

Paths defines the available resources; for each resource, it specifies the HTTP URL and the possible operations, in terms of HTTP methods, that can be performed on it. Each operation contains, among the others, a mnemonic name, valid requests, and corresponding responses.

```yaml
paths:
  /abstract-evaluation-rules/:
    summary: The possible evaluations a user can execute.
    description: This resource represents an evaluation a user can execute, possibly by
↪  composing it with other Abstract Evaluation Rule.
```

```
    get:
      operationId: abstract-evaluation-rules_list
      summary: List all the existing Abstract Evaluation Rule.
```

The *paths* excerpt shows the resource `abstract-evaluation-rule` and one of the possible operations, identified by the HTTP method `GET`. Such an operation lists all the resources of that type.

Request defines a valid request for an operation. It contains the schema detailing the format of such a request.

```
  /evaluation-rules/:
    post:
      operationId: evaluation-rules_create
      summary: Creates a new User Evaluation Rule
      description: Creates a new User Evaluation Rule by composing together one or more
↪  Abstract Evaluation Rule.
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/EvaluationRule'
        required: true
```

The *request* excerpt shows a `POST` operation creating a new resource (`evaluation-rules`). It provides a short and long description, and the request format.

Responses defines the possible responses that can be returned upon an operation on a resource. Different responses are identified by different HTTP status codes (e.g., `200` success, `400` bad request). Each response contains the schema detailing the format of such a response.

```
      responses:
        "201":
          description: User Evaluation Rule created and started successfully.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/EvaluationRule'
```

The *response* excerpt shows the response returned upon a successful creation of a resource of type `evaluation-rule`. It is identified by the status code `201` and contains the format of the response.

Components is a top-level section including the definition of resources, requests, and responses. This way, they are defined only once and referred to in other parts of the document using a specific syntax and increasing reuse along the document.

```
components:
  schemas:
    EvaluationRule:
      type: object
      properties:
        id:
          readOnly: true
          title: ID
          type: integer
        name:
          maxLength: 50
          minLength: 1
          title: Name
          type: string
```

The *components* excerpt shows a portion of the schema of a resource of type `EvaluationRule`.

4.2 Component *REST API*

The component *REST API* contains the framework main business logic and offers a REST interface to use the framework functionalities. The exposed resources can be divided in two main categories: *i) asset management*, allowing users to manage the assets (i.e., targets) registered within the framework, *ii) evaluation management*, allowing users to schedule evaluations and view their results. This interface is used by the *Dashboard*, which is the web-based graphical interface the users interact with. Recalling Section 2, an assurance process should be integrated with existing (security) solutions and processes (*complementarity*), as well as provide some form of *automation*. Both requirements are achieved by means of the exposed REST APIs, and facilitated by the corresponding OpenAPI document. The OpenAPI document is automatically generated from the application code, and served at a REST endpoint itself.

In general, an OpenAPI document serves for three main purposes: *i)* model-driven engineering (MDE), *ii)* documentation, *iii)* analysis.

Model-driven engineering consists of a development process centered around the business models. An OpenAPI-based model-driven engineering defines the first steps of the development process. The OpenAPI document describes the interface the application exposes and then develops the application by adhering to such a document. The coding phase can be partially automated by using code generation tools that, based on the OpenAPI document, generate most of the code boilerplate. Code generation can be also used to generate client libraries interacting with a REST server. These libraries are provided at a higher level of abstraction than plain HTTP calls. In our case, the OpenAPI document is used to generate several clients, supporting the use of our framework through a command-line interface (CLI) and in a continuous integration/continuous delivery (CI/CD) pipeline.

Documentation is another important use case for OpenAPI documents. Being a standard format, applications wishing to consume an API can exploit its OpenAPI document to get a detailed view of how such a service works. Parts of the application can be realized by code generation tools. Furthermore, developers can leverage visual tools, such as *ReDoc*, which display graphically an OpenAPI document. In our case, the OpenAPI document is served at a public endpoint, providing a comprehensive documentation of the framework REST interface, including several examples.

Analysis of an OpenAPI document is a research line that focuses on service validation by automatically generating test cases [18], transformations to other models (e.g., UML [13]), extensions to the specification to improve code generation [23]. In our case, the OpenAPI document is used to automatically generate several test cases, making it easier to perform functional and non-functional testing.

Our framework supports the complementarity and automation of assurance activities bypassing the graphical user interface and making use of the programmatic interface, namely the APIs offered by component *REST API*. These APIs can be, in fact, invoked within automatic or semi-automatic processes, for instance by triggering an evaluation when other events occur. Furthermore, libraries interacting with the framework can be automatically generated by exploiting the published OpenAPI document.

5 Assurance Process

The assurance process implemented by the framework in Figure 1 must assess both public and private targets. To address both scenarios, the framework builds on Virtual Private Network (VPN), addressing the must-have requirements *transparency*, *non-invasiveness*, *safety*, and *extensibility* in Section 2. The goal is to implement an assurance process that can be smoothly integrated with any kind of private target system, by means of a Site-to-Site VPN between the framework and the private targets the framework has to assess.

5.1 Building Blocks

Virtual Private Network (VPN) stands for a set of technologies used to build overlay networks over the public network. It provides hosts with remote access to a corporate network, or connects several geographically-distributed networks as if they are separated by one router [3].

In this paper, we focus on *Site-to-Site VPN*, where several networks are connected using the VPN. In each network connected to the VPN, there is a host acting as a *VPN gateway*, mediating traffic between internal hosts within its network and *other networks*. It routes traffic coming from internal hosts to the other *VPN gateways* and back. *VPN gateways* are called either *VPN clients* or *VPN servers*, where servers can handle connections to multiple clients, while a client establishes a single tunnel with a server.

VPNs usually combine a *virtual network interface card* (virtual NIC) and a socket-like connection. A virtual NIC is a NIC that has no physical correspondence, and is associated with a userspace process – in this case the VPN software. Packets *sent* by such process to its virtual NIC are *received* by the Operating System (OS), and further processed just like a real network packet. At the same time, the OS can *send* packets to it, and the VPN software, through its NIC, acts as the receiver. The socket-like connection is used to transmit packets between *VPN gateways* using a cryptographic protocol. The virtual NIC is used to send and receive packets coming from and whose destination is the host's network.

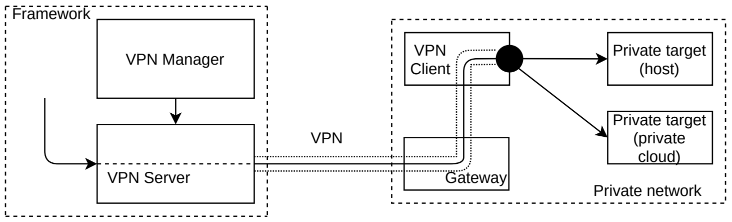


Fig. 2: Architecture of VPN-Based Solution [4].

Virtual NICs of the same VPN have IP addresses belonging to the same subnet, called *VPN subnet*. When the operating system of the *VPN gateway* handles a packet whose destination is a host in the *VPN subnet*, it sends the packet to the local virtual NIC, like a normal routing operation. Two sets of routing rules have to be defined: *i)* on each network, a rule on the default gateway that specifies to route traffic for *other networks* to the local *VPN gateway*; *ii)* on each *VPN gateway*, a rule that specifies to route traffic for *other networks* to the local virtual NIC.

However, a traditional VPN implementation does not permit to address many of the requirements in Section 2. The aforementioned routing rules, in fact, must be installed on both sides of the communication. Setting up these routes on the targets' default gateways requires access to the devices to alter their configurations. This violates properties *non-invasiveness*, *transparency*, and *safety*. We therefore propose a VPN-based approach at the basis of our assurance process that addresses the requirements, by adding several configurations on top of a standard VPN setup. The three logical building blocks of our VPN approach are: *VPN Client*, *VPN Server* and *Conflict-Resolution Protocol* (Figure 2).

VPN Client *VPN Client* establishes a VPN connection with the server, exposing its network to the framework. It realizes a *Client-side NAT* that avoids setting up routing rules on the target network. The issue is that packets generated by the framework and injected by the *VPN Client* into the target network have a source IP address belonging to the framework network. As such, responses to such packets would be routed to the target network default gateway (because they appertain to a different network than the current one) instead of the *VPN Client*. To address this, we propose a lightweight approach based on network address translation (*NAT*), which does not require to configure default gateways. Once packets are received by the *VPN Client* from the framework through the VPN, it translates their source IP address in the *VPN Client* IP address. Since this belongs to the same subnet of the target hosts, no routes need to be configured. Responses can directly reach the *VPN Client*, where the destination IP address of the packets is translated back. We implemented this address translation with *nftables*, available in Linux-based operating systems.

VPN Server *VPN Server* handles VPN tunnels with several clients; each tunnel is isolated to each other. It implements a *Server-side NAT*, to provide higher dynamics. There are two problems behind *Server-side NAT*, both involving routing configuration. On one side, *VPN Clients* need to know the network IP address of the framework (Section 5.1); on the other side, these routes must be known a priori, an assumption not trivial in our scenario. The network IP address of the framework, in fact, can change, for example, if the framework moves to a different cloud provider or for security reasons. We address the aforementioned problems by setting up different NAT rules on the *VPN Server.* They modify packets coming from the framework just before being received by the virtual NIC of the VPN software. These rules change the source IP address of packets by replacing it with the virtual NIC IP address of the server. Thus, packets received by a *VPN Client* have a source IP address belonging to the current *VPN subnet.* Then, corresponding responses generated by the target hosts, after the application of *Client-side NAT*, have a destination IP address appertaining to the *VPN subnet.* Recalling that a *VPN Client* knows how to handle packets generated – or appearing to be generated – directly from the *VPN subnet*, the *VPN Client* OS can route those packets to the local virtual NIC, without additional configurations. They are then received by the VPN software and finally sent to the server. *Server-side NAT* is implemented as a set of *nftables* rules.

Conflict-Resolution Protocol A mandatory requirement for a Site-to-Site VPN is that each participating network must have a non-conflicting net ID. Guaranteeing this assumption is necessary to allow a single VPN server to connect multiple networks together – in our case to allow a single *VPN Server* to handle several target networks. In corporate VPNs, it is trivial to assert this property, since the networks are under the control of the same organization. This assumption is not valid in our scenario, where two target networks could have the same network IP address, or a target network could conflict with the framework network. We propose an approach called *IP Mapping* to solve this issue.

IP Mapping is based on the concept of *mapping* the *original* network to a new one, called *mapped* network and guaranteed to be unique. Each IP address of the *original* network is translated into a new one, belonging to the corresponding *mapped* network. This translation is reversible, and the *mapped* address is specified by the framework as the target when executing a new evaluation. *IP Mapping* is realized through 3 functions whose pseudocode is described in Figure 3. The overall protocol, which is completely transparent to the final user, works as follows.

First, when a new target network is being registered, the function *map_net* is invoked by the framework, to obtain a non-conflicting version of the *original* target network. The pair $\langle$*original*, *mapped*$\rangle$ is saved into the database. Function *map_net* is offered by *VPN Manager* as a REST API.

When a user issues a new evaluation, she enters the *original* target IP address. The framework calls *map_ip* to obtain its *mapped* version, and builds the

```
INPUT
$s \in S$: VPN Server
$n_O$: network to map

OUTPUT
$n_M$: mapped version of $n_O$

MAP_NET
available_nets ← db_query_select($s$);
if length(available_nets) != 0 then
  pair ← ⟨available_nets[0], $n_O$⟩;
  db_query_insert(pair);
else Error();
return pair;
```

```
INPUT
$n_O.j$: $j$-th IP address $\in$ network $n_O$

OUTPUT
$n_M.j$: $j$-th corresponding
        IP address $\in$ network $n_M$

MAP_IP
$n_O$ ← net_id($n_O.j$);
host_id ← host_id($n_O$, $n_O.j$);
$n_M$ ← get_corresponding_net($n_O$);
$n_M.j$ ← build_address($n_M$, host_id);
return $n_M.j$;
```

```
INPUT
$n_M.k$: $k$-th IP address $\in$ network $n_M$

OUTPUT
$n_O.k$: $k$-th corresponding
        IP address $\in$ network $n_O$

REMAP_IP
$n_M$ ← net_id($n_M.k$);
host_id ← host_id($n_M$, $n_M.k$);
$n_O$ ← get_corresponding_net($n_M$);
$n_O.k$ ← build_address($n_O$, host_id);
return $n_O.k$;
```

Fig. 3: IP Mapping: Pseudocode [4].

corresponding test case using this IP address as destination. The test packets are then sent through the VPN. Function *map_ip* is offered by *VPN Manager* as a REST API.

The *VPN Client* receives the packets and calls *remap_ip* to get the *original* version of the destination IP address of the packets. This address is then set as the destination address: packets can now be sent to the target.

When corresponding responses reach back the *VPN Client*, the latter invokes *map_ip* to obtain the *mapped* version of the current IP source address; the result is set as the new IP source address. This second translation is issued to re-apply *IP Mapping* and let packets becoming correct responses to the ones generated by the framework. Finally, they are sent along the VPN and reach the framework.

Functions *map_ip* and *remap_ip* are implemented by a set of NAT rules using *nftables*.

The soundness of the overall VPN setup passes from *IP Mapping*, which, using the terminology in Figure 3, must support the following properties.

1. *Mapping uniqueness*: let $A \subseteq n_M \times S$; $\forall\, a_i, a_j \in A$, $(a_i.s = a_j.s \land a_i \neq a_j) \Rightarrow (a_i.n \neq a_j.n)$
2. *Mapping correctness*: $\forall\, n_O\ \forall$ address $\in n_O$ $remap\_ip(map\_ip(address)) = map\_ip(address)^{-1}$
3. *Implementation correspondence*: $\forall\, n_O$, $\forall$ address $\in n_O$, $map\_ip'(address) = map\_ip''(address)$

The first property expresses that no conflicts can happen, that is, two *mapped* networks with the same network IP address attached to the same *VPN Server* cannot exist. The second property expresses the reversibility of the translation process. It guarantees that a response to *mapped* packets generated by the framework is correct, that is, the source IP address of a response is equal to the destination IP address of a request. The third property expresses the need of having two implementations of *map_ip* (as a REST API or NAT rule) with the same

Table 1: Comparison of a standard layer-3 VPN and a layer-3 VPN with our modifications on top [4].

	Standard layer-3 VPN	Our approach
Client-side requiring configuration	Yes	No (*Client-side NAT*)
Server network known a priori	Yes	No (*Server-side NAT*)
Conflicting networks	Not allowed	Allowed (*IP Mapping*)
Address conflict resolution	Manual	Automatic (*VPN Manager*)
Plug-and-play integration	No	Yes

behavior. We note that the pseudocode in Figure 3 is a possible implementation of the three functions.

Table 1 summarizes the differences between a standard VPN and the one described in this paper. Our solution does not require any configurations on the target network, thanks to *Client-side NAT*; it also does not require to know the network IP address of the framework, thanks to *Server-side NAT*. Moreover, the networks participating in the VPN can have conflicting IP addresses, which are automatically disambiguated by *IP Mapping* and *VPN Manager*. To conclude, our solution allows a plug-and-play integration between the framework and the target network.

5.2 Assurance Process

The assurance process implemented by our framework is first configured with the registration of a private network and the creation of the *VPN Client*. We note that all actions involving interactions with our framework can be performed either manually by using the *Dashboard* or automatically by using the *REST API*. It then starts its activities with an evaluation request, where the user specifies the (set of) evaluation she wants to execute and the corresponding configurations. *REST API* orchestrates the process by creating the necessary objects within the database (*Model Database*) and by selecting the *Execution Manager* that executes the evaluation. In case of a private target, a *Private Execution Manager* connected to a *VPN Server* is selected. Also, *REST API* transparently obtains the *mapped* version of the target IP address, by invoking REST function *map_ip* exposed by *VPN Manager*.

The (set of) test case, derived from the requested (set of) evaluation, is sent to the selected *Execution Manager*, which executes the necessary (set of) *probe*. In case of a public target, test packets generated by the *probe* are sent directly to the target, and responses reach back the *probe* without involving the VPN. Otherwise, they are sent to the *VPN Server* that applies *Server-side NAT*, and then, passing through the VPN, reach the *VPN Client*. At this point, it applies *i) remap_ip* changing the destination IP address of the packets and *ii) Client-side NAT* changing the source IP address of the packets. Thanks to the last modification, responses to such packets flow back to the *VPN Client*, applying the converse of the previous steps: *i)* the reverse of *Client-side NAT* and *ii)* the reverse of *remap_ip*, that is, *map_ip*. When those packets are received by *VPN Server*, it applies the converse of *Server-side NAT*, by first forwarding them to the *Execution Manager* and then to the *probe*.

The *probe* produces the Boolean result of the test case and stores it into the *Evidence Database*. The overall result of the evaluation is determined by the *Evidence Analyzer*, which evaluates the evaluation's Boolean formula against the Boolean result(s) of the execution.

A concrete example of the process is described in Section 6.

6 Walkthrough and Experiments

We present a walkthrough of our assurance process and its experimental evaluation.

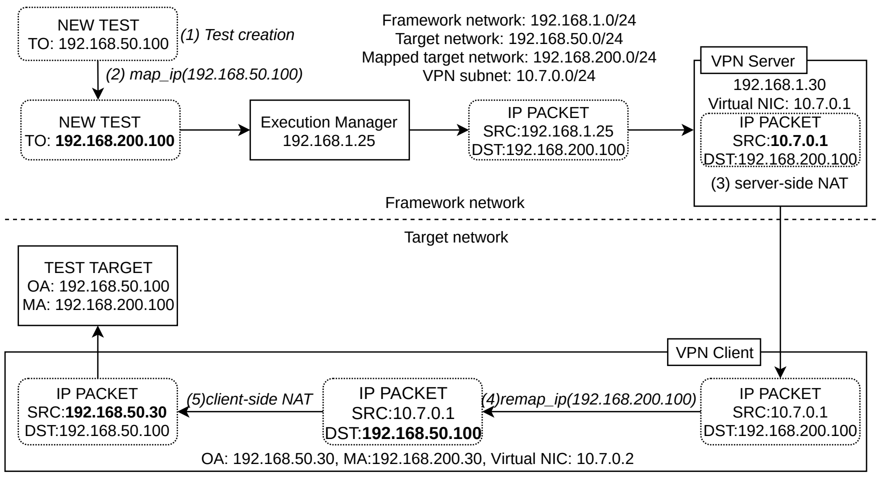
6.1 Process in Execution

Our framework supports the composition of multiple evaluations, tailoring framework's functionalities to match user needs. In the following, for simplicity but no lack of generality, we consider a singleton evaluation, named *Observatory-Compliance*, which checks whether a website has implemented common best practices, such as HTTPS redirection and cross-site-scripting countermeasures. In particular, we present the detailed working of an assurance process whose target is located into a private network, thus involving the use of our VPN-based solution (Section 5). We note that our VPN-based approach is transparent to the users, introducing no differences between private and public targets, except for the *VPN Client*.

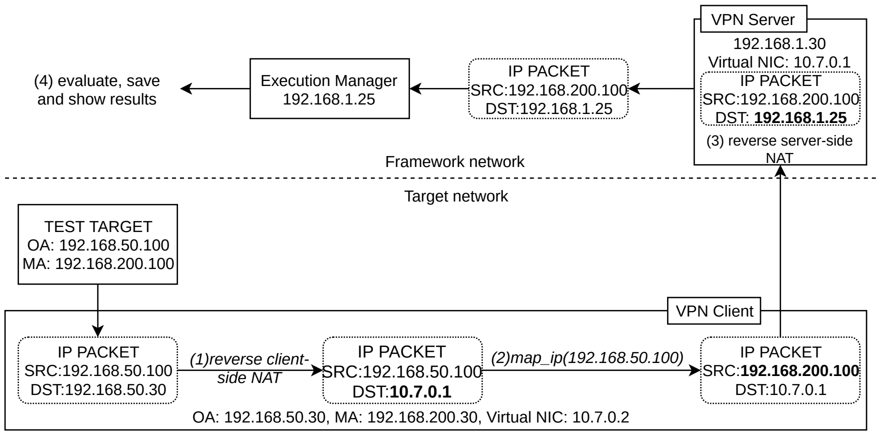
The parameters describing this process are the following: framework net ID `192.168.1.0/24`, target net ID `192.168.50.0/24`, mapped target net ID `192.168.200.0/24`, and VPN subnet net ID `10.7.0.0/24`.

Preparation. A prerequisite for the working of a VPN-based process is to register the private network within the framework. When the user inserts a new net ID (`192.168.50.0/24` in our example), the component *REST API* calls the *map_net* API exposed by *VPN Manager*, obtaining the *mapped* version of the input network (`192.168.200.0/24`). As described in Section 5.1, this mapping is stored in the framework database and triggers the creation of a new *VPN Client*. *VPN Manager* also configures *VPN Server* to support connections from the client. The client device is then moved into the correct location and connected to the server, establishing a VPN tunnel whose net ID (VPN subnet) is `10.7.0.0/24`.

Assurance request. The assurance request in Figure 4(a) starts with the framework receiving an evaluation request (*Step (1)* in Figure 4(a)), in this case for evaluation *Observatory-Compliance*. Such request contains, among the others, the IP address of the target (`192.168.50.100` in our example). Component *REST API* orchestrates the process as follows. First, it creates the necessary



(a) Assurance request



(b) Assurance response

Fig. 4: Packet flow: (a) assurance request, (b) assurance response [4].

objects, for instance, a test case, in the database to manage the evaluation. Being a private target, it calls the *map_ip* API exposed by *VPN Manager*, obtaining the *mapped* version of the target address (`192.168.200.100`) (*Step (2)*). Next, it chooses the *Execution Manager* that is in charge of the evaluation. For the aforementioned reason, a *Private Execution Manager* is selected. Such *Execution Manager* executes the *probe* as specified by *REST API*, targeting the *mapped* address. An excerpt of the input, in JSON format, a *probe* receives from its executor is shown in Listing 1.

The test packets generated by the *probe* are routed by the *Execution Manager* to the *VPN Server*. Upon receiving them, *VPN Server* applies *Server-*

```
{
    "config": {
        "url": "http://192.168.200.100"
    }
}
```

Listing 1: Input of the *probe* for evaluation *Observatory-Compliance*.

side NAT (*Step (3)*), which changes the source IP address of the packets to its virtual NIC address (`10.7.0.1`). Modified packets are then sent through the VPN, finally reaching *VPN Client*. At this point, *VPN Client* executes function *remap_ip* (*Step (4)*), which replaces the destination IP address of the packets with their *original* version. In our example, it changes from `192.168.200.100` to `192.168.50.100`. Then, it applies *Client-side NAT* (*Step (5)*), which changes the source IP address from the *VPN Server* virtual NIC address to its IP address (`192.168.50.30`). Finally, test packets reach their target.

Assurance response. The assurance response Figure 4(b) starts when the test target sends back responses to the *VPN Client*. Such responses can directly reach the *VPN Client*, since their destination, thanks to *Client-side NAT*, is the *VPN Client* itself. This phase applies the assurance request steps in the reverse order, to correctly forward responses to the *probe*. *VPN Client* first executes the reverse of *Client-side NAT*, by replacing the destination IP address of the packets with the *VPN Server* virtual NIC (*Step (1)* in Figure 4(b)). It then applies *map_ip* to change the source IP address with the corresponding *mapped* version, in our example it changes from `192.168.50.100` to `192.168.200.100` (*Step (2)*). Next, packets are forwarded to the *VPN Server*. Upon their reception, *VPN Server* applies the reverse of *Server-side NAT* (*Step (3)*). This step changes the destination address of the packets from the address of the *VPN Server* virtual NIC (`10.7.0.1`) to the address of the *Execution Manager* (`192.168.1.25`). Finally, the packets reach the *probe*, which evaluates and stores test result in the *Evidence Database* (*Step (4)*). In parallel and asynchronously, the component *Evidence Analyzer* produces the evaluation result, by collecting the Boolean results of the test cases forming the evaluation and evaluating them against the formula. In our example, this step is trivial since it involves evaluating a formula composed of either `TRUE` or `FALSE`. The result is finally written in the main database (*Model Database*) and can be accessed by the user.

An excerpt of the evaluation output is shown in Listing 2, showing that two best practices, `x-content-type-options` and `x-frame-options`, have been effectively implemented. As such, the evaluation result is `TRUE`.

6.2 Experiments

Our framework has been implemented as a set of microservices written in Python. Our VPN-based solution has been realized on top of *OpenVPN*, a flexible and open-source VPN solution that permits to tune every aspect of a VPN tunnel. In particular, we configured a layer-3 VPN using TCP as the encapsulating

```
{
    "status": true,
    "data": {
        "grade": "A",
        "x-frame-options": {
            "expectation": "x-frame-options-sameorigin-or-deny",
            "result": "x-frame-options-sameorigin-or-deny",
            "description": "X-Frame-Options (XFO) header set to SAMEORIGIN or DENY",
            "link": "https://infosec.mozilla.org/guidelines/web_security#x-frame-options",
            "hint": "X-Frame-Options controls whether your site can be framed, protecting
↪   against clickjacking attacks. It has been superseded by Content Security Policy's
↪   <code>frame-ancestors</code> directive, but should still be used for now."
        },
        "x-xss-protection": {
            "expectation": "x-xss-protection-1-mode-block",
            "result": "x-xss-protection-enabled-mode-block",
            "description": "X-XSS-Protection header set to \"1; mode=block\"",
            "link": "https://infosec.mozilla.org/guidelines/web_security#x-xss-protection",
            "hint": "X-XSS-Protection protects against reflected cross-site scripting (XSS)
↪   attacks in IE and Chrome, but has been superseded by Content Security Policy. It can
↪   still be used to protect users of older web browsers."
        }
    }
}
```

Listing 2: Sample output of the *probe* for evaluation *Observatory-Compliance*.

protocol, to maximize the probability of traversing firewalls in the path from the framework to the target system. *Client-side NAT*, *Server-side NAT*, and *IP Mapping* have been implemented as NAT rules with *nftables*.

Framework components have been packaged as *Docker containers* executed within Virtual Machines, all using operating system *CentOS 7 x64*. The following components run on single-container dedicated VMs: *REST API* (2 vCPUs, 4 GBs of RAM) *Execution Manager* (6 vCPUs, 4 GBs of RAM), *VPN Server* (1 vCPU, 4 GBs of RAM), running *OpenVPN* version *2.4.6* and *nftables* version *0.8*. All VMs have been deployed on a Dell PowerEdge M360 physical host that features 16 CPUs Intel® Xeon® CPU E5-2620 v4 @ 2.10 GHz and 191 GBs of RAM.

The target system has been deployed on AWS EC2 and was composed of two virtual machines *t2.micro*, both with 1 vCPU and 1 GB of RAM. The first one, *VPN Client*, with operating system *Ubuntu 18.04 x64*, *OpenVPN* version *2.4.7*, and *nftables* version *0.8*. The second one, test target, with operating system *Ubuntu 16.04 x64* offering *WordPress* version *5.2.2*.

We finally setup two experiments with the goal of computing the difference between two possible deployments: *i) public deployment* exposing the target on the public network, *ii) private deployment* using the approach in Section 5. The difference between the two deployments has been expressed in terms of the overhead that the private deployment adds on top of the public deployment, according to the following evaluations.

– *Infowebsite* that extracts as much information as possible from a target website. It is denoted as `E1` in Figure 5.

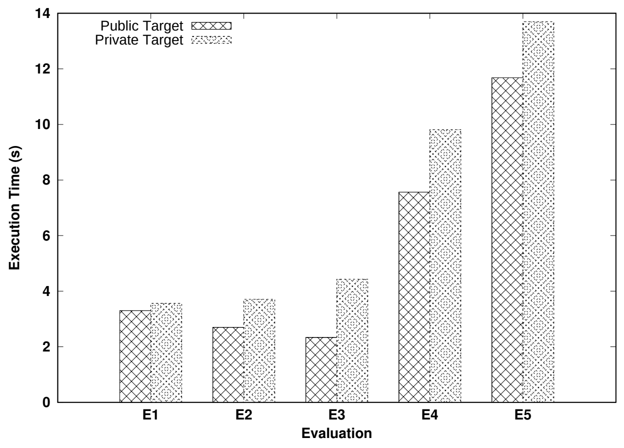


Fig. 5: Execution times of evaluations E1 – *Infowebsite*, E2 – *Observatory-Compliance*, E3 – *SSH-Compliance*, E4 – *TLS-strength*, E5 – *WordPress-scan* [4].

- *Observatory-Compliance* that checks whether a website has implemented common best practices, such as HTTPS redirection and cross-site-scripting countermeasures. It is denoted as E2 in Figure 5.
- *SSH-Compliance* that checks the compliance of a SSH configuration against Mozilla SSH guidelines. It is denoted as E3 in Figure 5.
- *TLS-strength* that evaluates whether the TLS channel has been properly configured, such as avoiding weak ciphers and older versions of the protocol. It is denoted as E4 in Figure 5.
- *WordPress-scan* that scans the target *WordPress*-based website looking for *WordPress*-specific vulnerabilities. It is denoted as E5 in Figure 5.

We chose these evaluations to maximize test coverage and diversity, from the evaluation of web resources (*Infowebsite*, *Observatory-Compliance*), to the evaluation of protocol configurations (*SSH-Compliance*, *TLS-strength*) and specific applications (*WordPress-scan*). Each evaluation was executed 10 times and the average time was computed. In particular, the execution time measurement started when the *Execution Manager* received the evaluation request, and finished when the executed *probe* terminated.

Performance and Discussion Figure 5 presents the average execution time of evaluations E1–E5. It shows that, as expected, the execution time in the private scenario is higher than the same in the public scenario, with an overhead varying between ≈0.3s and 2s.

More in detail, evaluation E1 (*Infowebsite*) experienced a very low overhead, less than a second. Evaluation E2 (*Observatory-Compliance*) experienced an overhead of approximately 1 second. Evaluations E3, E4 and E5 (*SSH-Compliance*, *TLS-strength*, *WordPress-scan*, resp.) experienced a higher overhead, approximately 2 seconds, increasing execution time from ≈2s to ≈4s for E3, from

≈8s to ≈10s for `E4` and from ≈12s to ≈14s. Overall, the increase in the execution time was globally under control, never exceeding 2 seconds. This overhead can be tolerated in all scenarios supporting requirements in Section 2.

To conclude, there is a subtlety to consider when an assurance process for hybrid systems is concerned: the accuracy of the retrieved results. There could be some cases in which the evidence collected by a *probe* on a public endpoint is different from the one collected by the same *probe* on a private endpoint. For instance, evaluation `E1` (*Infowebsite*), in the private scenario, failed to discover the version of the target *WordPress* website. This was due to a partial incompatibility between the *probe* implementation and our VPN-based solution. Being our approach *probe*-independent, this issue can be solved by refining the *probe* associated with *Infowebsite*. In our experiments, evaluation `E1` was the only experiencing such problem, while the other evaluations were able to collect the same evidence in both private and public deployments.

7 Comparison with Existing Solutions

Many security assurance approaches have been presented in literature, targeting software-based systems [16] and service-based environments [7], and providing certification, compliance, and audit solutions based on testing and monitoring. We analyzed the main assurance frameworks and processes, which can be classified according to the following categories: *monitoring-based*, *test-based* and *domain-specific*. Table 2 provides a comparison of these frameworks, including the one in this paper, with respect to requirements in Section 2.

Monitoring-based frameworks. Aceto et al. [1] provided a comprehensive survey of assurance solutions based on monitoring. They first considered requirement *intrusiveness*, which is similar to our requirements *transparency* and *non-invasiveness*, and found that many commercial monitoring tools do not address such requirement. They then considered requirement *lightness*, because monitoring tends to be expensive in term of resource consumption. Two monitoring frameworks have been presented in [2,12], both building on monitoring tool *Nagios*, thus satisfying, partially, *complementarity* and *automation*. Due to the intrinsic nature of monitoring, these frameworks can easily satisfy the requirement *continuity*. Moreover, the work in [2] can achieve a very good *adaptivity* and offers a monitoring platform both for cloud providers and users. Nevertheless, they require a significant effort in terms of setting up the monitoring infrastructure and resources for maintaining it, thus violating requirements *non-invasiveness* and *lightness*. Framework in [12] has also proven to suffer of *extensibility* and *scalability* issues [24]. [21] described a monitoring framework called *DARGOS*, built with scalability and flexibility in mind. Being fully distributed, it supports *scalability* and can be enriched with more sensors. However, being specifically tailored for the cloud, it cannot be easily adapted to other scenarios. Ciuffoletti [11] presented a novel approach, where a simple, cloud-independent API-based solution has been used to configure monitoring activities. Being based on APIs, it

Table 2: Comparison of state-of-the-art frameworks with the one in this paper [4].

References	Transparency	Non Invasiveness	Safety	Continuity	Lightness	Adaptivity	Complementarity
[2]	✓	~	~	✓	✗	✓	~
[8]	✓	✗	~	~	~	~	~
[10]	~	✗	~	✗	✗	✗	~
[11]	✓	~	✓	✓	✓	~	✓
[12]	~	~	✓	✓	✗	~	~
[26]	~	~	✓	~	~	~	✗
[14]	~	✓	~	✓	✗	~	✓
[17]	✗	✗	~	✓	✗	✓	✗
[20]	✗	✗	✗	✓	✗	✗	✗
[21]	✓	~	✗	~	~	✗	✓
[28]	✗	✗	✗	✗	✓	✓	✗
[4]	✓	✓	~	✓	~	✓	✗
This paper	✓	✓	~	✓	~	✓	✓

(a) Process requirements

Reference	Evidence-based verification	Extensibility	Multi-layer	Scalability	Automation
[2]	~	✗	✓	~	~
[8]	~	✗	~	~	~
[10]	✓	✗	~	✗	✗
[11]	~	✗	~	✓	✓
[12]	~	~	~	~	~
[26]	~	✗	✗	~	✗
[14]	~	✗	~	✓	✓
[17]	✗	✗	✗	✗	✓
[20]	✗	~	~	~	~
[21]	~	✗	✓	✓	✓
[28]	✗	~	~	✗	✗
[4]	✓	~	✓	~	~
This paper	✓	~	✓	✓	✓

(b) Framework requirements

easily satisfies requirements *complementarity* and *automation*. Its cloud-agnosticism is realized through an OCCI (*Open Cloud Computing Interface*) extension, designed towards *Monitoring-as-a-Service*.

Test-based frameworks. Wu and Marotta [28] presented a work-in-progress testing-based framework that instruments client binaries to perform cloud testing. The main issue is that binaries instrumentation may not be always feasible, and might also introduce undesired behavior in modified programs. As such, the framework fails to satisfy requirements *transparency*, *non-invasiveness*, and *safety*. Ouedraogo et al. [20] presented a framework that uses agents to perform security assurance, although agents themselves need to be properly secured. Greenberg et al. [15] claimed that, to protect hosts from agent misuse or attacks, several techniques need to be properly employed. Agents also pose a maintenance problem: they have to be kept updated and things can only become worse as the number of agents increases. Also, they introduce substantial costs since they need to be physically installed on each host/device to be assessed and coordinated, introducing not-negligible network traffic. For these reasons, the agent-based framework in [20] does not satisfy requirements *transparency*, *non-invasiveness* and *safety*. Jahan et al. [17] discussed *MAPE-SAC*, a conceptual approach for security assurance of self-adaptive systems, where the system itself changes, and security requirements must adapt to these changes. While it is not possible to completely evaluate our requirements due to the lack of a real, implemented framework, *MAPE-SAC* fulfills requirements *adaptivity*, *continuity* and *automation*. A different solution has been given in our work in [4], which has served as the basis for the framework described in Section 3. As already discussed, the proposed approach is based on *probes* and *meta-probes*, and fails to address mainly requirements *complementarity* and, partially, *automation*. Also, it addresses only partially requirements *safety*, *lightness*, *extensibility* and *scalability*.

Domain-specific frameworks. Aslam et al. [8] focused on the assurance of fog computing, discussing a framework based on TPM (Trusted Platform Module) for node audit. By relying on TPMs, it does not satisfy requirement *non-invasiveness*. De la Vara et al. [26] presented an assurance framework targeting cyber-physical systems. Their approach provides several tools supporting the certification process. However, being tailored for model-driven engineering, it requires significant effort and fails to address many of our framework requirements, such as *multi-layer* and *automation*. Elsayed and Zulkernine [14] described a distributed framework for monitoring cloud analytics applications, based on analyzing logs produced by such applications. The proposed approach requires very few configurations at the cloud side, and can be offered through the *Security-as-a-Service* paradigm. Cheah et al. [10] considered the automotive world, where cases are generated after evaluating the severity of threats. Threats are found through threat modeling and confirmed with a penetration testing. The usage of penetration testing violates requirement *non-invasiveness* and, requiring human intervention, requirement *automation*. Often, being tailored for a specific domain, solutions in this category cannot claim requirement *extensibility*.

To conclude, the comparison in Table 2 shows that the existing frameworks (and corresponding processes) do not even come close to addressing the requirements in Section 2. In general, existing solutions mainly target *continuous evaluation* and *multi-layer* infrastructures, as well as *transparency* and *adaptivity*, failing to achieve *non-invasiveness*, *safety*, *lightness*, and *extensibility*. The framework in this paper, instead, provides a first boost in this direction addressing, at least partially all requirements in Table 2. Following the comparison therein, this paper leaves space for future work. We will first aim to extend our framework towards Big Data and IoT environments, further improving *extensibility*, *lightness*, and *scalability*. We will also focus on strengthening the *safety* of the framework and its components, for example the *Execution Manager* that can easily become a single point of failure/attack.

8 Conclusions

Security assurance solutions verify whether a distributed system holds some security properties and behaves as expected, usually complementing traditional security approaches. Existing assurance frameworks and processes however are limited in impact by the fact that they often lack extensibility and interfere with the functioning of the system under verification. In this paper, we extended the VPN-based assurance framework in [4] to provide an assurance process for hybrid systems, from private networks to public clouds, that addresses properties automation and complementarity. The proposed assurance process has limited impact and costs on the target system, while providing a safe and scalable approach that integrates with existing security solutions and support automatic configuration of assurance activities.

References

1. Aceto, G., Botta, A., de Donato, W., Pescapè, A.: Cloud monitoring: A survey. Computer Networks **57**(9), 2093 – 2115 (2013)
2. Alcaraz Calero, J.M., Aguado, J.G.: MonPaaS: An Adaptive Monitoring Platformas a Service for Cloud Computing Infrastructures and Services. IEEE TSC **8**(1), 65–78 (January 2015)
3. Alshalan, A., Pisharody, S., Huang, D.: A Survey of Mobile VPN Technologies. IEEE COMST **18**(2), 1177–1196 (2016)
4. Anisetti, M., Ardagna, C.A., Bena, N., Damiani, E.: Stay Thrifty, Stay Secure: A VPN-based Assurance Framework for Hybrid Systems. In: Proc. of SECRYPT 2020. Paris, France (Virtual) (July 2020)
5. Anisetti, M., Ardagna, C.A., Damiani, E., Gaudenzi, F.: A certification framework for cloud-based services. In: Proc. of ACM SAC 2016. Pisa, Italy (April 2016)
6. Anisetti, M., Ardagna, C.A., Damiani, E., Gaudenzi, F.: A semi-automatic and trustworthy scheme for continuous cloud service certification. IEEE TSC (2017)
7. Ardagna, C., Asal, R., Damiani, E., Vu, Q.: From security to assurance in the cloud: A survey. ACM CSUR **48**(1), 2:1–2:50 (August 2015)
8. Aslam, M., Mohsin, B., Nasir, A., Raza, S.: FoNAC - An automated Fog Node Audit and Certification scheme. Computers & Security **93**, 101759 (2020)
9. Baldini, G., Skarmeta, A., Fourneret, E., Neisse, R., Legeard, B., Le Gall, F.: Security certification and labelling in Internet of Things. In: Proc. of IEEE WF-IoT 2016. Reston, VA, USA (Dec 2016)
10. Cheah, M., Shaikh, S.A., Bryans, J., Wooderson, P.: Building an automotive security assurance case using systematic security evaluations. COSE **77**, 360 – 379 (2018)
11. Ciuffoletti, A.: Application level interface for a cloud monitoring service. CS&I **46**, 15 – 22 (2016)
12. De Chaves, S.A., Uriarte, R.B., Westphall, C.B.: Toward an architecture for monitoring private clouds. IEEE Comm. Mag. **49**(12), 130–137 (December 2011)
13. Ed-douibi, H., Cánovas Izquierdo, J.L., Cabot, J.: OpenAPItoUML: A Tool to Generate UML Models from OpenAPI Definitions. In: Proc. of ICWE 2018. Cáceres, Spain (June 2018)

14. Elsayed, M., Zulkernine, M.: Towards Security Monitoring for Cloud Analytic Applications. In: Proc. of IEEE BigDataSecurity/HPSC/IDS 2018. Omaha, NE, USA (May 2018)
15. Greenberg, M.S., Byington, J.C., Harper, D.G.: Mobile agents and security. IEEE Comm. Mag. **36**(7), 76–85 (July 1998)
16. Herrmann, D.: Using the Common Criteria for IT security evaluation. Auerbach Publications (2002)
17. Jahan, S., Pasco, M., Gamble, R., McKinley, P., Cheng, B.: MAPE-SAC: A Framework to Dynamically Manage Security Assurance Cases. In: Proc. of IEEE FAS*W 2019. Umea, Sweden (June 2019)
18. Karlsson, S., Čaušević, A., Sundmark, D.: QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. In: Proc. of IEEE ICST 2020. Porto, Portugal (March 2020)
19. OpenAPI Initiative: OpenAPI specification (2018)
20. Ouedraogo, M., Mouratidis, H., Khadraoui, D., Dubois, E.: n Agent-Based System to Support Assurance of Security Requirements. In: Proc. of SSIRI 2010. Singapore (June 2010)
21. Povedano-Molina, J., Lopez-Vega, J.M., Lopez-Soler, J.M., Corradi, A., Foschini, L.: DARGOS: A highly adaptable and scalable monitoring architecture for multi-tenant Clouds. FGCS **29**(8), 2041–2056 (2013)
22. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE **63**(9), 1278–1308 (Sep 1975)
23. Sferruzza, D.: Top-Down Model-Driven Engineering of Web Services from Extended OpenAPI Models. In: Proc. of IEEE/ACM ASE 2018. Montpellier, France (September 2018)
24. Taherizadeh, S., Jones, A.C., Taylor, I., Zhao, Z., Stankovski, V.: Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. JSS **136**, 19 – 38 (2018)
25. Teigeler, H., Lins, S., Sunyaev, A.: Chicken and Egg Problem: What Drives Cloud Service Providers and Certification Authorities to Adopt Continuous Service Certification? In: Proc. of WISP 2017. Seoul, South Korea (December 2017)
26. de la Vara, J.L., Ruiz, A., Gallina, B., Blondelle, G., Alaña, E., Herrero, J., Warg, F., Skoglund, M., Bramberger, R.: The AMASS Approach for Assurance and Certification of Critical Systems. In: Proc. of Embedded World 2019. Norimberg, Germany (February 2019)
27. West, R.: The psychology of security. Commun. ACM **51**(4), 34–40 (Apr 2008)
28. Wu, C., Marotta, S.: Framework for Assessing Cloud Trustworthiness. In: Proc. of IEEE CLOUD 2013. Santa Clara, CA, USA (June–July 2013)