SeCloud: Computer-Aided Support for Selecting Security Measures for Cloud Architectures

Yuri Gil Dantas and Ulrich Schöpp
Fortiss GmbH, Munich, Germany

Keywords: Securing Cloud Architectures, Security Architecture Patterns, Automation.

Abstract: The adoption of cloud infrastructures requires the deployment of security measures to protect assets against threats (e.g., tampering). Several security measures/technologies are available for securing cloud infrastructures, such as Service Mesh Istio and OpenID Connect. In the current state of practice, the selection of security measures is manually done by an expert (e.g., a security engineer). It becomes challenging for experts to make these selections due to the complexity of cloud infrastructures and the vast number of available security measures and technologies. This article proposes a tool for automating the recommendation of security measures for cloud architectures. Our tool expects as input information both the cloud architecture and assets identified during a threat analysis, and recommends security measures for protecting such assets against threats. We validate our tool in a case study that provides cloud services for unmanned air vehicles (UAVs).

1 INTRODUCTION

Cloud infrastructures offer runtime environments with sophisticated mechanisms for reliability, observability, manageability and security. These infrastructures provide several benefits for business and IT, including lower implementation and maintenance costs.

Security is one of the biggest concerns about cloud infrastructures, especially because the data is no longer controlled by the client who purchased the cloud service. Indeed, a literature review conducted by (Carroll et al., 2011) confirms that security is the main risk for businesses using cloud infrastructures.

Two attack surfaces against cloud systems have brought the attention of security researchers and engineers: 1) External attackers may carry out attacks against cloud services (Eliseev et al., 2021). For example, without suitable security measures, an external attacker may carry out spoofing attacks to impersonate a legitimate client in accessing critical data. 2) An internal service may also be the source of attacks due to, e.g., misconfiguration, compromise, or being intentionally malicious (Oleshchuk and Køien, 2011). To implement an effective defense-in-depth strategy, it is important to consider also threats that originate from internal services. Internal services may, e.g., carry out elevation-of-privilege attacks to access data from critical services without authorization. Potential attacks through these attack surfaces shall be mitigated by selecting and deploying suitable security measures for cloud infrastructures.

Before selecting security measures, security engineers perform a threat analysis to identify assets, threat scenarios, and attack paths leading to threat scenarios. Threat analysis is recommended for the early stages of the system development, i.e., during the design of the cloud architecture to avoid expensive changes later in the cloud system lifecycle.

The identification of potential security measures to mitigate the identified threat scenarios is a main challenge for cloud architectures. Examples of technologies offering security measures are Service Mesh Istio, the authentication protocol OpenID Connect, and the Kubernetes network plugin Cilium. In the current state of practice, the selection of security measures is manually done by an expert (e.g., a security engineer).

There are many cloud technologies and platforms to choose from, and it is hardly feasible to evaluate the implications of many possible choices in detail, e.g., due to time constraints. Even when the technologies are understood in principle, it is not easy to keep track of the consequences of selecting a combination of them. One would also like to understand the trade-offs between different choices regarding system development and operation, e.g., in the form of additional requirements for certificate management, or regarding resource overhead.

The cloud native landscape is vast, and it's

Dantas, Y. and Schöpp, U.
SeCloud: Computer-Aided Support for Selecting Security Measures for Cloud Architectures.
DOI: 10.5220/0011901900003405
In *Proceedings of the 9th International Conference on Information Systems Security and Privacy (ICISSP 2023)*, pages 264-275
ISBN: 978-989-758-624-8; ISSN: 2184-4356

easy to become overwhelmed by its growing number of competing and overlapping platforms and technologies.[1]

Figure 1 may give an impression of the vastness.

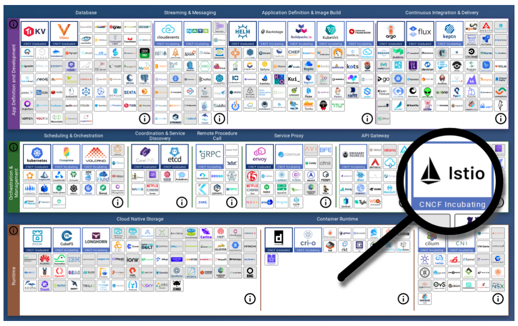


Figure 1: Cloud Native Technology Landscape (excerpt).

Contribution: This article proposes SECLOUD, a tool for automating the recommendation of security measures for cloud architectures. (i) SECLOUD computes threat scenarios for assets provided as input information by the user. (ii) SECLOUD computes attack paths based on an intruder model, and the cloud architecture received as input information. (iii) SECLOUD recommends security measures for mitigating threat scenarios (i.e., addressing all attack paths leading to threat scenarios). The target user for SECLOUD is security engineers responsible for assessing the security of cloud architectures and hardening such architectures with security measures. SECLOUD has been implemented in the logic programming engine clingo (Potassco, 2022).

SECLOUD is built on the work by (Dantas and Nigam, 2022), who proposed a tool for automating the recommendation of security architecture patterns for autonomous vehicle architectures. The original aspects of our work are: (i) extension of the domain-specific language for cloud architectures, (ii) specification of security measures suitable for cloud architectures, (iii) more specific reasoning principles for recommending security measures. The work by (Dantas and Nigam, 2022) only considers the cybersecurity property satisfied by the pattern as the main condition to recommend patterns, and (iv) specification of constraints to reduce the number of recommended solutions with security measures to deal with scalability and usability issues.

SECLOUD is available online at (SeCloud, 2022).

Structure of this Article: The remainder of this article is structured as follows. Section 2 describes a running example to help us introduce the contributions of the article. Section 3 describes an intruder model for cloud infrastructures. Section 4 describes the workflow of SECLOUD, including its inputs and output artifacts. Section 5 describes the domain-specific language of SECLOUD, including how SECLOUD specifies security measures to enable their automation through the reasoning rules described in Section 6. Section 7 illustrates the benefits of using constraints and the increased automation enabled by SECLOUD. We conclude the article by discussing related and future work in Sections 8 and 9.

2 RUNNING EXAMPLE

SECLOUD is intended to assist security engineers with security architecture decisions for cloud infrastructures. We present it using a real application developed in a research project as a running example.

The example application provides services by unmanned air vehicles (UAVs), such as transportation services or search and rescue services. It optimizes the usage of UAV and implements planning, optimization and prognostic health management functions. While the particular details of the use-case are not important to describe the application of SECLOUD, the application provides a realistic use-case.

The role of SECLOUD is to support the selection of technologies to implement security functions at an early stage in the system development, where the main components and their interfaces have been identified, but where the security architecture of the system is still under consideration.

Figure 2 gives an overview of the logical architecture of the system. The main system components are Service Broker (sb), Multi Resource Manager (mrm), Cognitive Assistant (ca), Operations Manager (om), Fleet Manager (fm). These components are intended to be deployed on a cloud platform. They provide public interfaces where clients may access the services.

The applications communicate with each other through two mechanisms. First, the components offer a Rest/HTTP API for access to resources. Second, they use Kafka[2] for event-based communication. Kafka implements topic-based pub/sub communication. It provides named topics where the application components may publish messages about certain aspects, e.g. the topic `av-updates` is intended for messages pertaining to the status of air vehicles. These messages are delivered to all components that subscribe to them. For example, the Operations Manager receives telemetry

[1] https://www.cncf.io/blog/2020/09/15/top-7-challenges-to-becoming-cloud-native

[2] https://kafka.apache.org/

data from the ground control station and publishes this to `av-updates`. The Fleet Manager subscribes to this topic and thus receives telemetry updates. Pub/-sub communication is convenient for decoupling components, but also introduces challenges for security, e.g. for limiting the impact of the compromise of one internal component.

The main application components are each realized by a set of Docker containers. Figure 3 shows a container architecture of the system. It contains the application containers and support containers, such as for ingress. While, in realistic deployments, Kafka will likely be deployed redundantly, it suffices to consider it as a single container for our purposes.

Figure 3 shows assumptions about the deployment and the security environment of the system. We consider client a public, external component, to which attackers have full access. Most of the components are intended to be hosted on a cloud platform, where ingress serves as a gateway component. While ground control station is part of the system, it needs to be hosted on-site at an airport rather than in the cloud.

The container architecture of Figure 3 represents and early stage of system development. It does not contain any security features. It could be deployed directly using Docker, but the components would communicate over plain http without authentication or authorization. The purpose of SECLOUD is to support the design of a security architecture for the application.

3 INTRUDER MODEL

This section describes an intruder model for cloud infrastructures. The intruder model is taken into account by SECLOUD when computing attack paths. We consider both external and internal intruders based on the Dolev-Yao intruder (Dolev and Yao, 1983). Both intruder models are inspired by related work, e.g., (Oleshchuk and Køien, 2011; Eliseev et al., 2021).

External Intruder. The external intruder assumes that public interfaces may be exploited by attackers, as in (Eliseev et al., 2021). As a result, the external intruder may inject malicious data into the cloud infrastructure through public interfaces (e.g., cloud consumers) to violate the cybersecurity property of assets. Consider, e.g., the architecture described in Section 2, where the interface from client to sb represents a public interface of the system, and the container sb is an asset. A malicious attacker may carry out spoofing attacks impersonating a legitimate client to write unauthorized data to sb, thus violating the authenticity properties of data arriving at sb. We assume that an attack from a public interface to an asset may be possible if there exists an information flow from the public interface to the asset. This is the case for the example above, as shown in the cloud architecture illustrated Figure 3.

Internal Intruder. The internal intruder assumes that internal containers may not be trustworthy, as in (Oleshchuk and Køien, 2011). As a result, an internal container may inject malicious data into other containers (possibly assets) in the cloud system. For example, sb may not be trustworthy and carry out elevation of privilege attacks to access data from other assets like mrm without authorization. We assume that an attack from an internal container to another asset container may be possible if there exists a communication channel from the internal container and to the asset. This is the case for sb and mrm, as illustrated in Figure 3.

4 SeCloud: OVERVIEW

Our goal is to provide automated methods for the selection of security measures for cloud architectures. To this end, we build on SecPat proposed by (Dantas and Nigam, 2022). We propose the use of Knowledge Representation and Reasoning (KRR) (Baral, 2010) for representing cloud architectures, security artifacts, and security measures as knowledge bases. The security measures are recommended in an automated fashion through the specification of reasoning rules. The representation of the knowledge bases are realizable through a domain-specific language (DSL). We specify rules to reason about the security of the cloud architecture, including rules to enable the automated recommendation of security measures. Our tool – SECLOUD – implements both the DSL and reasoning rules for securing cloud architectures. SECLOUD has been developed in the logic programming engine clingo (Potassco, 2022).

Figure 4 illustrates the workflow of SECLOUD. The gray boxes represent either artifacts received as input or generated for output. SECLOUD receives two main artifacts as input, namely cloud architecture and security artifacts.

Architecture Model. This input artifact consists of a Logical Architecture, a Container Architecture, and an Allocation Table. An allocation table denotes the mapping of logical components to containers. Selected containers may be annotated as public (i.e., they are external components that are not under the control of the system) or gateways. Annotating containers as public is relevant for security, especially for identifying potential attack paths.

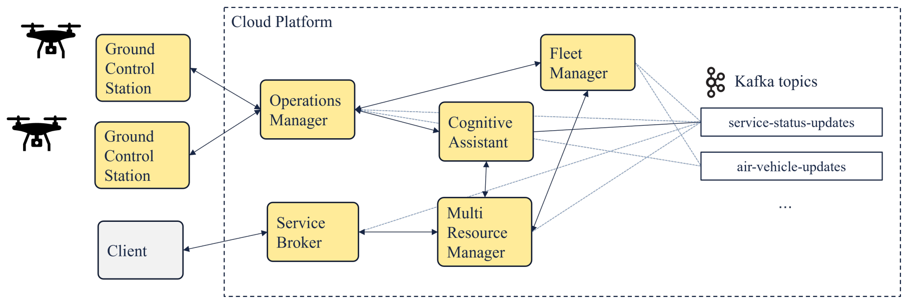


Figure 2: High-level architecture of running example.

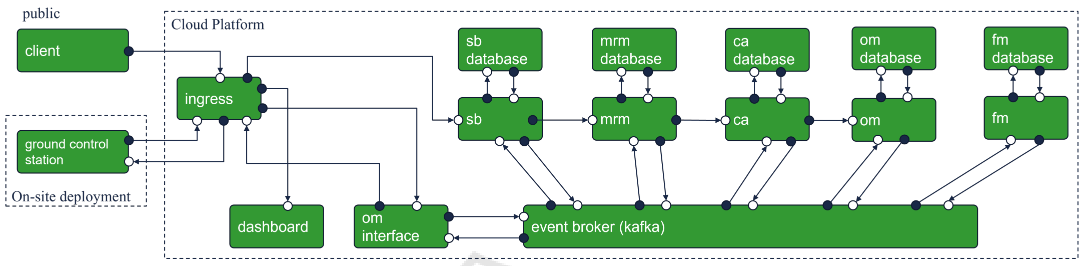


Figure 3: Container architecture of running example.

Security Artifacts. This input artifact consists of artifacts that may be identified in a Threat Analysis and Risk Assessment (TARA) analysis. Examples of security artifacts identified in a TARA analysis are assets and threat condition. SECLOUD expects that at least assets are provided as input. Assets are objects (e.g., software elements) that need protection. SECLOUD expects that an asset is associated with a container.

Both the cloud architecture and security artifacts are specified in the DSL of SECLOUD. This specification enables SECLOUD to reason about the security of the cloud architecture through reasoning principles. The reasoning principles enables the automated computation of threat scenarios and attack paths. SECLOUD attempts to deploy security measures wherever they are applicable to mitigate threat scenarios. SECLOUD outputs assumptions that need to be valid to ensure that the recommended security measure works as intended. The assumptions are turned into requirements that are output together with the recommended measures. These requirements shall be implemented and validated during the development of the system. In summary, SECLOUD provides as output artifacts **threat scenarios** and **attack paths**, as well as **security measures alongside assumptions/requirements**.

4.1 Clingo

Clingo is an engine to implement logic programs based on Answer-Set Programming (ASP) semantics (Gelfond and Lifschitz, 1990).

A logic program is a set of rules. Each rule is of the form $\mathsf{a}_0 \leftarrow \mathsf{a}_1,\ldots,\mathsf{a}_n$, where the literal a_0 is the head of the rule, and the literals $\mathsf{a}_1,\ldots,\mathsf{a}_n$ are the body of the rule. A literal is an atom (a_m) or a negated atom ($\neg\mathsf{a}_m$). A rule with an empty head is a constraint. A model (often referred by this article as solution) is an interpretation satisfying a set of rules.

This article uses the clingo notation, where :- denotes $\leftarrow$, and not denotes $\neg$. Identifiers beginning with capital letters (e.g., A, B) denote variables that during clingo's execution are instantiated by appropriate terms. Identifiers beginning with a lower-case letter (e.g., a, b) are constants. The _ (underscore) character specifies that the argument can be ignored in the current rule.

5 DOMAIN-SPECIFIC LANGUAGE

SECLOUD provides a domain-specific language (DSL) for specifying (a) cloud architectures, (b) security artifacts, and (c) security measures. Table 1 provides the predicates/facts for specifying selected architectural elements and security artifacts. With the exception of threat conditions, threat scenarios and attack paths, all these facts shall be provided as input. The threat conditions denote the adverse consequences if the cybersecurity property of an asset is violated. Following the STRIDE methodology (Shostack, 2014), this article considers the authenticity, integrity, and authorization properties. These properties may be violated by, respectively, spoofing, tampering, and elevation of privilege attacks.

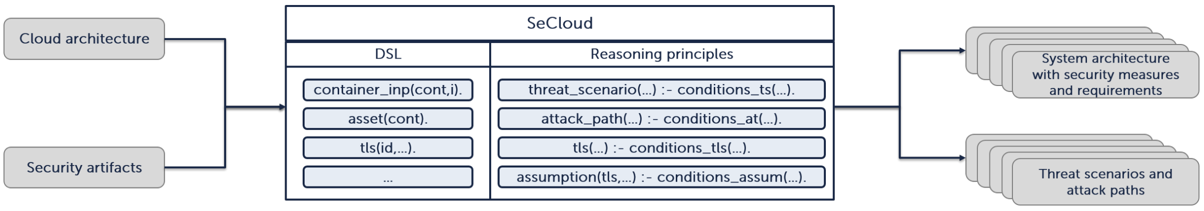

Cloud architecture
Security artifacts
SeCloud
DSL
Reasoning principles
container_inp(cont,i).
asset(cont).
tls(id,...).
...
threat_scenario(...) :- conditions_ts(...).
attack_path(...) :- conditions_at(...).
tls(...) :- conditions_tls(...).
assumption(tls,...) :- conditions_assum(...).
System architecture with security measures and requirements
Threat scenarios and attack paths

Figure 4: SeCloud's workflow. Gray boxes are artifacts either received as input or generated for output. The light blue boxes under SeCloud are for illustrative purposes only.

Example. Consider the container architecture illustrated in Figure 3 and the predicates described in Table 1. Assume that both ground control station and sb have been identified as assets in the initial threat analysis. The user of SeCloud specifies the container architecture and the identified assets as follows.

```
% Containers (excerpt)
 container_out(client,o1).
 container_out(gcs,o2).
 container_inp(gcs,i1).
 container_inp(ingress,i2).
 container_inp(ingress,i3).
 container_onp(ingress,o3).
 container_onp(ingress,o4).
 container_inp(sb,i5).
 container_out(sb,o6).

% Connections (excerpt)
 conn(sg1,o1,client,i2,ingress).
 conn(sg2,o2,gcs,i3,ingress).
 conn(sg3,o3,ingress,i1,gcs).
 conn(sg3,o4,ingress,i5,sb).
 conn(sg3,o6,sb,i6,sb_database).

% Assets (excerpt)
 asset(gcs).
 asset(sb).
```

Listing 1: Specification of container architecture and assets.

5.1 Specifying the Security Content of Cloud Technologies

We refer to security measures at the architectural level as security architecture patterns (Cheng et al., 2019). Security architecture patterns are architectural solutions for mitigating threat scenarios. This section describes by example how SeCloud provides semantically-rich description of patterns that will enable the automated reasoning described in Section 6.

Before introducing the security architecture patterns, we describe the DSL of SeCloud for specifying security architecture patterns. The bottom of Table 1 describes the predicates/facts for specifying (a) the pattern instantiation, (b) the pattern attributes. The former denotes all relevant architecture elements for instantiating the pattern. The relevant architecture elements are the pattern components, and the pattern channels. The pattern attributes denote the intent and problem addressed by the pattern. That is, the pattern attributes consist of the desired cybersecurity property achieved by the pattern, the threat addressed by the pattern, and the attack surface suitable to instantiate the pattern. You may read this as the pattern mitigates the particular threats (e.g., tampering) at the attack surface against the cybersecurity property (e.g., integrity).

We consider two kinds of attack surfaces, namely external and internal interface. The former denotes any attacks carried out by with external entities, such as entities that are public or entities that send data to the system through a gateway. The latter denotes any attacks carried out within the system. Considering the attack surface is relevant because some patterns may only be applied inside the system (internal interface) or outside the system (external interface). The traditional firewall is an example of such patterns that may only be applied at the external (a.k.a. network) interface.

At present, SeCloud formulates five security architecture patterns for the recommendation of different kinds of technologies for cloud applications: *Cilium*[3] is a Kubernetes network plugin with advanced functionality for providing, securing, and observing network connectivity between containers. Cilium uses IPsec to transparently encrypt data in transit between applications to avoid data tampering attacks. Cilium is also able to enforce policies for satisfying both authenticity and authorization properties. *TLS* (Transport Layer Security) is a protocol to provide secure communication over a network communication channel. In the context of cloud computing, TLS can be used to enforce secure communication between containers to prevent attackers to tamper with the data exchanged between applications. Mutual Authentication is described in Section 5.1.1. *OpenID Connect*[4] is an authentication protocol built on top of OAuth 2.0. OpenID Connect allows a client to authenticate itself when accessing services. An OpenID Connect

[3]https://cilium.io
[4]https://openid.net/connect

Table 1: DSL for (selected) architecture elements, security artifacts, and security architecture patterns.

Fact	Description
	Architectural elements
container_inp(id,$\mathsf{id_i}$)	id is a container and it has an input port $\mathsf{id_i}$.
container_out(id,$\mathsf{id_o}$)	id is a container and it has an output port $\mathsf{id_o}$.
conn (id,$\mathsf{id_{1o}}$,$\mathsf{id_1}$,$\mathsf{id_{2i}}$,$\mathsf{id_2}$)	id is a signal connecting an output port $\mathsf{id_{1o}}$ of container $\mathsf{id_1}$ to an input port $\mathsf{id_{2i}}$ of container $\mathsf{id_2}$.
gateway(id)	id is a gateway container.
public(id)	id is a public container that may be accessible by external users.
	Security artifacts
asset(id)	denotes that container id is an asset.
threat_condition (id,secp,$\mathsf{id_{ast}}$)	id is the adverse consequence if the cybersecurity property secp of asset $\mathsf{id_{ast}}$ is violated, where secp $\in$ {authenticity, integrity, authorization}.
threat_scenario (id,$\mathsf{ts_{des}}$,as,ts,$\mathsf{id_{ast}}$)	id is a threat scenario, with description $\mathsf{ts_{des}}$, originating from attack surface as, to violate the cybersecurity property of asset $\mathsf{id_{ast}}$ with threat ts, where ts $\in$ {spoofing, tampering, elevation_of_privilege}.
attack_path (id,el,$\mathsf{id_{ast}}$)	id is an attack path denoting that malicious data may be injected from element el to target asset $\mathsf{id_{ast}}$.
	Security architecture patterns
security_pattern (id,pat,cp,inp,int,out)	id is the unique identification of security architecture pattern pat. This pattern consists of a list of components cp. The last three parameters inp, int and out denote, respectively, the input, the internal, and the output channels related to the pattern.
security_attributes (pat,as,secp,ts)	pat is a security architecture pattern suitable for satisfying the cybersecurity property secp by mitigating threat ts at the attack surface as. as $\in$ {external_interface, internal_interface}, secp $\in$ {authenticity, integrity, authorization}, ts $\in$ {spoofing, tampering, elevation_of_privilege}.

server verifies user credentials (e.g., username and password) and issues identity tokens. Client may use such identity token to authenticate themselves when accessing application services. *Service Mesh Istio*[5] is an application-level infrastructure for managing services in a cloud environment. Security-wise, Istio can enforce secure communication between applications by encrypting traffic and providing mutual authentication (i.e., to ensure integrity and authenticity). It provides support functions, such as key distribution and rotation. Istio provides the functionality to enforce policies to authorize resource access in the mesh.

We describe how SECLOUD captures cloud technologies in the form of semantically-rich description of security architecture patterns by example using Mutual Authentication. Table 2 describes the pattern attributes for all the patterns supported by SECLOUD.

[5]https://istio.io

5.1.1 Mutual Authentication

Mutual Authentication (mTLS) (Vasudev et al., 2020) is a security measure for verifying the authenticity of two entities that wish to exchange data over a communication channel. Assume a client and a server connected through a communication channel. This pattern ensures that the server authenticates with the client, and the client authenticates with the server before the actual communication occurs. To ensure the authenticity of these entities over a communication channel, both the client and the server shall provide their digital certificates to prove their identities to each other. This is in contrast to TLS, where only the server presents a certificate. The Mutual Authentication pattern additionally guarantees the integrity of data exchanged between the client and server given that the identities of the entities have been correctly verified. Finally, for satisfying the authorization property, this pattern assumes that both client and server implement policies

Table 2: Security architecture patterns currently supported by SECLOUD. This table describes that a pattern may be used to mitigate threat (e.g., spoofing) at the attack surface (e.g., external interface) to satisfy the security property (e.g., authenticity).

Pattern	**Threat → Security Property**	**Attack Surface**
Cilium	spoofing → authenticity tampering → integrity elevation of privilege → authorization	internal interface
OpenID Connect	spoofing → authenticity	internal/external interface
TLS	tampering → integrity	external interface
mTLS	spoofing → authenticity tampering → integrity elevation of privilege → authorization	internal/external interface
Service Mesh Istio	spoofing → authenticity tampering → integrity elevation of privilege → authorization	internal/external interface

for data access control. Mutual Authentication may be applied for components communicating over an internal or external interface. The instantiation of the Mutual Authentication pattern is shown in Table 3.

Based on the structure of mTLS shown in Table 3, SECLOUD instantiates this pattern as shown below.

```
security_pattern(id,mTLS,(cp1,cp2),
  (inp1,inp2),(int1,int2),none).
```

Listing 2: Pattern instantiation (mTLS).

```
security_attributes(mTLS,
  attack_surface(external_interface,
    internal_interface),
  property(authenticity,integrity,
    elevation_of_privilege),
  threat(spoofing,tampering,
    elevation_of_privilege)).
```

Listing 3: Security attributes (mTLS).

6 SECURITY REASONING PRINCIPLES

SECLOUD specifies reasoning principles to reason about the security of the cloud architecture, including reasoning principles to (a) compute threat scenarios, (b) enumerate attack paths for the identified threat scenarios, (c) recommend security architecture patterns for mitigating threat scenarios. While the main focus of this section is about the pattern recommendation, we also provide a brief explanation of (a) and (b).

Computing Threat Scenarios. SECLOUD computes threat scenarios that may violate the cybersecurity property of assets. We consider the authenticity, integrity, authorization of assets received as input. Based on STRIDE (Shostack, 2014), we consider that these three properties may be violated, respectively, by spoofing, tampering, and elevation of privilege attacks. For example, for each asset received as input, we consider that the integrity of the asset may be violated by tampering attacks.

While this is a fairly coarse way of computing threat scenarios, SECLOUD provides the possibility to define more fine-grained criteria for threat scenarios that may use all available information. This can be done by defining inference rules of the following form.

```
threat_scenario(ID, THREAT, SOURCE,
                  TYPE, ASSET) :- ...
```

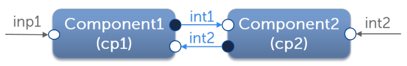
Listing 4: Threat scenario inference rule (snippet).

Computing Attack Paths. SECLOUD computes attack paths for each threat scenario. To this end, SECLOUD implements the intruder model described in Section 3 with capabilities to carry out both external and internal attacks.

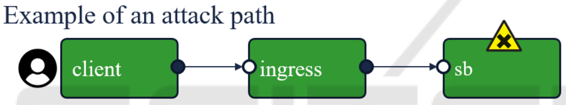
Example. Figure 5 illustrates a potential attack path (based on the architecture from Section 2) for three threat scenarios. The upper part of Figure 5 illustrates an attack path from client (public interface) targeting the asset sb (short for service broker). The lower part Figure 5 describes three threat scenarios for asset sb. The attack surface is obtained from the attack path originated at the external interface (i.e., by client).

We now introduce three predicates for specifying whether a threat scenario is mitigated or not. First, mitigated_by(IDTS,IDPAT) specifies when a threat scenario IDTS is mitigated by a suitable security architecture

Table 3: Instantiation of the Mutual Authentication pattern. The assumptions represent only a selection.

	Description	SECLOUD Specification
Pattern name	Mutual Authentication	NAME=mTLS;
Structure	 inp1 Component1 (cp1) int1 int2 Component2 (cp2) int2	COMPONENTS=[cp1,cp2]; INPUT_CH=[inp1,inp2]; INTERNAL_CH=[int1,int2];
Intent	This pattern is used when two entities over a communication channel verify each other's identity (authentication). Given the correct verification of the entities, this pattern also satisfies the integrity of messages exchanged between the sender and the receiver. Both entities implement authorization policies.	TYPE_SEC_PROPERTY= [authenticity,integrity,authorization]; TYPE_THREAT= [spoofing,tampering,elevation_of_privilege]; TYPE_ATTACK_SURFACE= [external_interface,internal_interface];
Problem addressed	This pattern prevents that data exchanged between two entities in communication channel are spoofed or tampered. This pattern prevents such entities to gain privileges in accessing resources that shall not be authorized.	
Assumptions/ Requirements (selection)	CP1 and CP2 have digital certificate signed by a trusted CA.	TYPE_ASSUMPTION=entity_has_received_certificate_signed_by_trusted_ca; COMPONENTS=[cp1,cp2];

Example of an attack path



Threat scenarios

Attack Surface	Threat	Asset
external interface	spoofing	sb
external interface	integrity	sb
external interface	elevation of privilege	sb

Figure 5: Illustration of one attack path and three threat scenarios computed by SECLOUD.

pattern IDPAT. We assume that a threat scenario is mitigated if the attack path leading to the threat scenario is addressed. Second, mitigated(IDTS) expresses that the threat scenario IDTS by some pattern. Finally, nmitigated(IDTS) specifies that IDTS is not mitigated. The reasoning rules for three predicates are as follows:

```
mitigated(IDTS)   :- mitigated_by(IDTS,IDPAT).
nmitigated(IDTS) :-
  threat_scenario(IDTS,_,_,_,_),
  not mitigated(IDTS).
```

Listing 5: Mitigation rules.

6.1 Pattern Instantiation

SECLOUD specifies reasoning rules for automating the recommendation of security architecture patterns. These rules specify the conditions for when a particular security architecture pattern can be recommended to mitigate threat scenarios targeting assets, which were identified by the user. Whenever a security architecture pattern is recommended, the rule mitigated_by applies to infer which threat scenarios have been mitigated. SECLOUD only outputs architecture solutions when all threat scenarios have been mitigated. This is ensured through the specification of the following constraint, which expresses that any non-mitigated threat scenario is not allowed (implies empty/false).

```
:- nmitigated(IDTS).
```

Listing 6: Constraint: No non-mitigated threat scenarios.

A security architecture pattern is recommended if there is a match between security artifacts and the pattern attributes. That is, SECLOUD considers information related to a threat scenario, i.e., attack surface, threat, and property violated by the threat, and the pattern attributes. A security architecture pattern is recommended if the following conditions hold:

$$\text{attack\_surface} \in \text{pattern\_attribute}$$
$$\text{threat} \in \text{pattern\_attribute}$$
$$\text{security\_property} \in \text{pattern\_attribute}$$

We illustrate a reasoning rule specified in SECLOUD to recommend security architecture patterns. The following reasoning rule specifies the conditions for recommending security architecture patterns. The rule checks whether there is one instance of security_attributes for the security artifacts specified on the right side of the rule.

```
{ recommended_pattern(PAT,IDTS,IDAP, EL1,EL2)
  : security_attributes(PAT,AS,SECP,TS) } = 1
:- threat_scenario(IDTS,_,AS,TS,AST),
   threat_condition(_,SECP,AST),
   get_ap_from_ts(IDTS,IDAP),
   attack_path(IDAP,EL1,EL2).
```

Listing 7: Pattern recommendation rule.

The above rule derives facts of the form recommended_pattern(PAT,IDTS,IDAP,EL1,EL2), which express that pattern PAT has been recommended to address attack path IDAP of threat scenario IDTS. EL1 and EL2 are architectural elements in the attack path IDAP. These architecture elements may become pattern components. For example, assume that the mTLS pattern has been recommended. The instantiation denotes that mTLS used for the communication from EL1 (i.e., client) to EL2 (i.e., server). SECLOUD may derive the input, internal and output channels from the (baseline) system architecture received as input. Omitted here, we have a rule for mapping recommended_pattern to security_pattern (described in Table 1).

6.2 Combinations and Constraints

Pattern instantiation will produce a large number of possible solutions. The number of solutions depends on the number of (a) assets, (b) attack paths leading to threat scenarios, and (c) security architecture patterns. SECLOUD computes all possible solutions with security architecture patterns as long as there exists a match between security artifacts and pattern attributes.

The purpose of SECLOUD is to assist the selection of architecture options, so it is essential to select reasonable options from the set of possible instantiations. By building on an ASP solver like clingo, we can define sophisticated constraints on the possible combinations of patterns, which can be checked efficiently by the solver. Indeed, the intended usage of ASP solvers is to generate a (possibly very large) set of potential solutions up front and to select suitable ones using constraints (Lifschitz, 2019).

SECLOUD uses the following constraints.

1. **All threats have been mitigated.** SECLOUD specifies a constraint to only computes architecture solutions where all threat scenarios have been mitigated. That is, SECLOUD discards solutions if at least one threat scenario has not been mitigated. This constraint is shown in Listing 6.
2. **Only one pattern for addressing each threat scenario.** SECLOUD may output two security architecture patterns for addressing the same threat scenario. For example, possible solutions for mitigating tampering threats violating the integrity of assets are Mutual Authentication and Service Mesh patterns. SECLOUD specifies a constraint to only recommend one pattern for each threat scenario avoiding that two patterns are recommended for the same threat scenario.
3. **No TLS and mTLS for the same element.** SECLOUD may compute solutions where TLS and mTLS are recommended for the same element. These solutions may be redundant in the sense that the element should have two certificates, one for TLS and one for mTLS. SECLOUD specifies a constraint to avoid TLS and mTLS from being recommended for the same element.
4. **No mTLS for public elements.** Mutual Authentication may be impractical for public elements, as certificates would need to be distributed to all external clients. SECLOUD therefore specifies a constraint to avoid the mTLS pattern from being recommended for public elements.
5. **Only one pattern for addressing equivalent security artifacts.** SECLOUD may compute several combinations of patterns for addressing threat scenarios. This might lead to expensive solutions to be implemented during the development of the system. For example, SECLOUD may output a solution where Service Mesh is recommended for two containers, and Cilium is recommended for the remaining containers in the system. It might be expensive and unnecessary to deploy a Service Mesh for only two containers of the system, especially when Cilium may be deployed for all containers. SECLOUD specifies constraints to recommend the same pattern to address security artifacts with equivalent attributes (i.e., attack surface, threat, and cybersecurity property).

7 CASE STUDY

This section illustrates the use of SECLOUD for the cloud architecture described in Section 2. We evaluate the use of the constraints defined in Section 6.2, and discuss the recommended patterns by SECLOUD.

For the sake of illustration, we assume two assets: ground control station and sb. As described in Section 2, we assume that client is a public component, ingress is a gateway, and ground control station sends and receives data through ingress. This means that all attack paths involving client and ground control station denote potential attacks through an external interface. The remaining attack paths denote potential attacks through an internal interface.

Table 4: Experiments with constraints. Scenario with 2 assets, 31 attack paths, and 5 security architecture patterns. The entry '-' means that SECLOUD has not returned all solutions after 60 minutes. The constraint's ID (e.g., 1, 2,...) refers to the constraints described in Section 6.2.

Constraint	#Solutions	Execution time (s)
none	-	3600
1	-	3600
1, 2	-	3600
1, 2, 3	2916	0.27
1, 2, 3, 4	1458	0.13
1, 2, 3, 4, 5	6	0.06

SECLOUD has computed 93 threat scenarios, and 31 attack paths (28 and 3 attack paths based on, respectively, the external and internal intruder). For each attack path, SECLOUD considers three threat scenarios that may violate the property of the asset through spoofing, tampering, and elevation of privilege. Thus, the number of threat scenario is $31 \times 3 = 93$.

There are many ways of applying the security architecture patterns to mitigate these threat scenarios. With the patterns and constraints defined in Sections 5 and 6, SECLOUD recommends six architecture options almost instantly. These options are shown in Table 5.

To understand how SECLOUD may deal with a larger number of possible solutions, e.g., when extended with more patterns, it may be instructive to consider also its performance when some of the constraints from Section 6.2 are lifted. Table 4 describes the results of our experiments. We ran the experiments on a 1.9GHz Intel Core i7-8665U with 16GB RAM wuth Ubuntu 18.04 LTS and clingo 5.2.2.

SECLOUD could not enumerate all solutions without any constraints and with constraints 1, and 1 & 2. With constraints 1 & 2 & 3, and 1 & 2 & 3 & 4, SECLOUD computed all solutions within 0.27 and 0.13 seconds, respectively. These results illustrate that while the number of pattern instantiations being considered is the same in all cases, imposing constraints to filter solutions is sufficient to improve performance. However, the number of solutions were still high, impacting the usability of SECLOUD (i.e., it is impractical for the user to choose a solution). The breakthrough was the specification of constraint 5 for only recommending one type of pattern to address equivalent security artifacts. Together with the other constraints, SECLOUD computed six solutions within 0.06 seconds.

The six solutions are shown in Table 5. In general, they represent a reasonable selection of architecture options to address the 93 threat scenarios identified by SECLOUD. Solution 6 has previously been selected manually in the development of the system that serves as our running example. One aspect that is perhaps not reasonable is that all solutions use the authorization policies of the Istio service mesh to verify OpenID Connect tokens, even when no components are placed in the mesh. This is an artifact of the current small selection of security architecture patterns. However, the reasons for selecting the patterns are documented by SECLOUD, so users may replace individual patterns in each solution.

Indeed, SECLOUD provides documentation for the identified attack paths, threat scenarios, and new requirements for each solution. The requirements are traced to threat scenarios, which themselves are associated to attack paths. For each threat scenario, SECLOUD documents which security architecture pattern in the selected solution has mitigated the threat.

To aid the user in selecting a solution, SECLOUD computes simple metrics of the solution. We consider the number of new requirements on application components and on the infrastructure deployment as one selection criterion. We distinguish such requirements in the following sense:

- *Application requirements* need to be considered in the development of the applications themselves. For example, Solution 1 adds the new application requirement that sb uses mTLS to communicate with sb database. This needs to be considered by the developers, e.g., by using a suitable library.
- *Infrastructure requirements* need to be implemented when building the infrastructure and deploying the components. For example, Solution 6 adds a requirement that an authorization policy for the communication from sb to mrm has been defined. The configuration of the Istio service mesh is also defined in the form of requirements. For example, Solution 6 has a requirement that sb must be part of the service mesh.

If the aim is to provide security measures in a transparent manner for the application developers, then Solution 6 will likely be preferable over Solution 1. This choice is indeed the case as Solution 6 makes encryption transparent, while Solution 1 requires application developers to use mTLS explicitly.

8 RELATED WORK

SECLOUD is built on SecPat proposed by (Dantas and Nigam, 2022). SecPat enables the recommendation of security architecture patterns for autonomous vehicle architectures, while SECLOUD supports the

Table 5: Solutions recommended by SECLOUD.

Solution	Recommended Security Patterns	# Additional Application Requirements	# Additional Infrastructure Requirements
1	*Mutual Authentication* for mitigating the threat scenarios with attack paths based on the internal intruder, including attack paths targeting ground control station. TLS for ingress acting as a server for client to mitigate tampering attacks. *OpenID Connect* for client for authentication purposes to address spoofing attacks. The identity tokens of client obtained by OpenID Connect can be checked by an instance of Service Mesh Istio for all containers receiving data from client.	19	34
2	*Cilium* for mitigating selected threat scenarios with attack paths based on the internal intruder, and *Mutual Authentication* for addressing the attack paths targeting ground control station. In this solution, *Mutual Authentication* has been recommended for addressing attacks from outside the system (external interface). TLS for ingress acting as a server for client to mitigate tampering attacks. *OpenID Connect* for client for authentication purposes to address spoofing attacks. The identity tokens of client obtained by OpenID Connect can be checked by an instance of Service Mesh Istio for all containers receiving data from client.	11	22
3	*Cilium* for mitigating selected threat scenarios with attack paths based on the internal intruder. TLS for ingress acting as a server for client and ground control station to mitigate tampering attacks. *OpenID Connect* for both client and ground control station for authentication purposes to address spoofing attacks. The identity tokens of client and ground control station obtained by OpenID Connect can be checked by an instance of Service Mesh Istio for all containers receiving data from client and ground control station.	11	25
4	*Mutual Authentication* for mitigating selected threat scenarios with attack paths based on the internal intruder. TLS for ingress acting as a server for client and ground control station to mitigate tampering attacks. *OpenID Connect* for both client and ground control station for authentication purposes to address spoofing attacks. The identity tokens of client and ground control station obtained by OpenID Connect can be checked by an instance of Service Mesh Istio for all containers receiving data from client and ground control station.	19	37
5	*Service Mesh Istio* for mitigating selected threat scenarios with attack paths based on the internal intruder. *TLS* for ingress acting as a server for client and ground control station to mitigate tampering attacks. *OpenID Connect* for both client and ground control station for authentication purposes to address spoofing attacks. The identity tokens of client and ground control station obtained by OpenID Connect can be checked by the instance of Service Mesh Istio.	11	40
6	*Service Mesh Istio* for mitigating selected threat scenarios with attack paths based on the internal intruder, and *Mutual Authentication* for addressing the attack paths targeting ground control station. TLS for ingress acting as a server for client to mitigate tampering attacks. *OpenID Connect* for client for authentication purposes to address spoofing attacks. The identity tokens of client obtained by OpenID Connect can be checked by the instance of Service Mesh Istio.	11	37

recommendation of patterns suitable for cloud architectures. SecPat provides a DSL and reasoning principles to automate the recommendation of patterns. SecPat only considers one condition to recommend patterns, namely the cybersecurity property satisfied by the pattern. SECLOUD extends SecPat's DSL to include further pattern's attributes, namely the threat mitigated by the pattern, and the attack surface that the pattern may be deployed. SECLOUD follows the STRIDE methodology to map threats to cybersecurity properties. This extension enables a more precise recommendation of patterns through reasoning principle rules. Another extension is the specification and evaluation of constraints to reduce the number of solutions with patterns. This extension deals with scalability issues and improves the usability of SECLOUD.

ThreatGet[6] is a commercial tool for threat analysis. ThreatGet identifies threat scenarios and attack paths in an automated fashion. To deal with such security artifacts, ThreatGet provides a list of potential security measures to be selected by the user. ThreatGet does not instantiate the selected security measures in the system architecture. As a result, it might be unclear for the user to identify which components are relevant to the selected security measures. SECLOUD instantiates the recommended security measures by making explicit which components are part of the security measure (e.g., mTLS for components A and B).

Another commercial tool for STRIDE analysis of cloud architectures is Microsoft's Threat Analysis tool [7]. While SECLOUD is also based on STRIDE, it is able to automatically compute architecture options. Its flexible definition using ASP allows the extension to more fine-grained threat scenario models.

SECLOUD outputs requirements alongside the recommended security measures. Other tools, such as Ansible (Spanaki and Sklavos, 2018), may be used to harden cloud infrastructures by implementing such requirements. Ansible provides the means of, e.g., installing SSL certificates, installing and configuring monitoring tools, and configuring user accounts.

9 CONCLUSION

This article proposed SECLOUD, a tool to assist security engineers with the selection of security measures for cloud architectures. We validated SECLOUD in a case study that provides cloud services for unmanned air vehicles (UAVs). We are currently investigating several future directions, including (i) specification of security measures to address threat scenarios that violate the availability of assets, and (ii) integration of SECLOUD in a model-based system engineering tool that will serve as a frontend to improve its usability.

[6]https://www.threatget.com

[7]https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling

ACKNOWLEDGMENTS

We thank the German Ministry for Economic Affairs and Climate Action of Germany for funding this work through the LuFo V-3 project RTAPHM.

REFERENCES

Baral, C. (2010). *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Carroll, M., Kotzé, P., and van der Merwe, A. (2011). Secure cloud computing: Benefits, risks and controls. In Venter, H. S., Coetzee, M., and Loock, M., editors, *Information Security South Africa Conference 2011, ISSA 2011*. ISSA, Pretoria, South Africa.

Cheng, B. H. C., Doherty, B., Polanco, N., and Pasco, M. (2019). Security Patterns for Automotive Systems. In *MODELS'19*.

Dantas, Y. G. and Nigam, V. (2022). Automating Safety and Security Co-Design through Semantically-Rich Architectural Patterns. *ACM Trans. Cyber Phys. Syst.*

Dolev, D. and Yao, A. C. (1983). On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207.

Eliseev, V., Miliukova, E., and Kolpinskiy, S. (2021). Neural Network Cryptographic Obfuscation for Trusted Cloud Computing. In *Integrated Models and Soft Computing in Artificial Intelligence*, pages 201–207.

Gelfond, M. and Lifschitz, V. (1990). Logic programs with classical negation. In *ICLP*.

Lifschitz, V. (2019). *Answer Set Programming*. Springer.

Oleshchuk, V. A. and Køien, G. M. (2011). Security and privacy in the cloud a long-term view. In *2011 2nd International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE)*.

Potassco (2022). Clingo: A grounder and solver for logic programs https://github.com/potassco/clingo.

SeCloud (2022). https://github.com/ygdantas/SeCloud.

Shostack, A. (2014). *Threat Modeling: Designing for Security*. Wiley.

Spanaki, P. and Sklavos, N. (2018). Cloud Computing: Security Issues and Establishing Virtual Cloud Environment via Vagrant to Secure Cloud Hosts. In *Computer and Network Security Essentials*, pages 539–553. Springer.

Vasudev, H., Deshpande, V., Das, D., and Das, S. K. (2020). A Lightweight Mutual Authentication Protocol for V2V Communication in Internet of Vehicles. *IEEE Trans. Veh. Technol.*, 69(6):6709–6717.