Pattern-driven Security, Privacy, Dependability and Interoperability management of IoT environments

Othonas Soultatos*†, Manos Papoutsakis*†, Konstantinos Fysarakis§, George Hatzivasilis†,
Manos Michalodimitrakis†,George Spanoudakis§, Sotiris Ioannidis†
* Department of Computer Science, City, University of London, London, UK
† Institute of Computer Science, Foundation for Research and Technology Hellas, Heraklion, Greece
§ Sphynx Technology Solutions AG, Zug, Switzerland
othonas.soultatos@city.ac.uk, {fysarakis,spanoudakis}@sphynx.ch, {paputsak,hatzivas,manmix,sotiris}@ics.forth.gr

***Abstract*—Achieving Security, Privacy, Dependability and Interoperability (SPDI) is of paramount importance for the ubiquitous deployment and impact maximization of Internet of Things (IoT) applications. Nevertheless, said requirements are not only difficult to achieve at system initialization, but also hard to prove and maintain at run-time. This paper highlights an approach to tackling the above challenges, through the definition of pattern language and a framework that can guarantee SPDI in IoT orchestrations. By integrating pattern reasoning engines at the various layers of the IoT infrastructure, and a machine-processable representation of said pattern through Drools rules, the proposed framework can provide ways to fulfill SPDI requirements at design time, and also provide the means to guarantee those SPDI properties and manage the orchestrations accordingly. Moreover, an application example of the framework is presented in an Industrial IoT monitoring environment.**

***Index Terms*—Pattern-driven engineering, Internet of Things, Security, Privacy, Dependability, Interoperability.**

I. Introduction

The ubiquitous presence of smart computing devices, referred to as the IoT, has the potential to significantly enhance everyday lives, as it could be the enabler of anything from smart home and industrial automation to fully autonomous systems. Nevertheless, IoT applications and their enabling platforms increasingly interact directly with the physical world, often in critical applications, while they often generate and use sensitive personal data. This leads to significant security and privacy concerns, which are exacerbated by the heterogeneity of IoT devices and the intrinsic, often strict, quality of service requirements of the various application domains [1]–[3].

Motivated by the above, this work proposes a holistic approach to the management of IoT environments based on the definition of architectural SPDI patterns and a framework built on top of that, able to process and reason on said patterns in an automated manner, across all layers of an IoT deployment (backend, network, field). Through this approach, the high level concept of which has been presented in [4], the proposed framework enables the design-time and run-time guarantee of the SPDI properties of IoT and Industrial IoT (IIoT) applications, also triggering adaptations when the desired properties are violated, to revert the system to the desired state.

This paper is organized as follows: Section II presents a background to the underlying technologies and concepts used; Section III presents our pattern language definition model and some implementation aspects; Section IV details an application used in an wind-park environment; Section V presents some related works, and; Section VI features the concluding remarks and pointers to future work.

II. Background

A. SPDI requirements in IoT environments

Considering the dynamicity, scalability and heterogeneity of IoT and IIoT environments, as well as the intrinsic security and privacy requirements of the various areas of application, the proposed pattern-driven approach aims to provide a holistic approach covering the Security, Privacy, Dependability and Interoperability properties of IoT/IIoT systems. The subsections below provide more details on each of the key properties.

1) Security: Security is generally composed of the three properties of confidentiality, integrity, and availability, sometimes also abbreviated as CIA. In more detail:

- *Confidentiality*: the disclosure of information happens only in an authorised manner; i.e. non-authorised access to information should not be possible.
- *Integrity*: maintenance and assurance of the accuracy and consistency of data.
- *Availability*: the invocation of an operation to access some information or use a resource leads to a correct response to the request.

Therefore, a holistic approach will need to cover these three aspects, at the component as well as at the end-to-end level.

2) Privacy: There have been plenty attempts to define privacy over the years but, so far, no universal definition could be created. Despite the fact that the claim for privacy is universal, its concrete form differs according to the current era and context (technical landscape) [6]. In any case, IoT devices generate, process, exchange and store vast amounts of security add safety-critical data as well as privacy-sensitive information hence careful handling is needed, both from an ethical as well as a regulatory perspective (esp. in cases where medical data is involved). It is important to understand that information collected in a system becomes personal if identity can be correlated with an activity [7]. Such identification can be direct or indirect.This is why data protect law does not

apply to anonymous data (i.e., data in which the data subjects are no longer identifiable).

3) Dependability: Dependability is the ability of a system to deliver its intended level of service to its users [8]. The main attributes which constitute dependability are reliability, availability, safety and maintainability. Dependable systems impose the necessity to provide higher fault and intrusion tolerance. The satisfaction of these attributes can avoid threats such as faults, errors and failures offering fault prevention, fault tolerance and fault detection.

4) Interoperability: Desired interoperability characteristic imposes special requirements on the designed framework. Interoperability gives an ability to a system or a product to connect and work with other systems or products. Interoperability is defined as a characteristic of a product or system, whose interfaces are completely understood, to work with other products or systems, present or future, in either implementation or access, without any restrictions [9].

The following types of interoperability can be defined: *Technological interoperability* enables seamless operation and cooperation on heterogeneous devices that utilize different communication protocols; *Syntactic interoperability* establishes clearly defined formats for data, interfaces and encoding; *Semantic interoperability* settles commonly agreed information models and ontologies for the used terms that are processed by the interfaces or are included in exchanged data; *Organizational interoperability* cross-domain service integration and orchestration through common semantic and programming interfaces.

B. SPDI patterns

Patterns are re-usable solutions to common problems and building blocks to architectures [10], [11]. In this work, SPDI patterns encode proven dependencies between security, privacy, dependability and interoperability properties of individual components of IoT applications and corresponding properties of orchestrations of such components. More specifically, a pattern encodes relationships of the form $P1 and P2 and and Pn \rightarrow Pn + 1$ where $Pi(i = 1, , n)$ are properties of individual components and Pn+1 is a property of the orchestration of these components. The relation encoded by a pattern is an entailment relation. The runtime adaptations that can be enabled by SPDI patterns may take three forms: (1) to replace particular components of an orchestration; (2) to change the structure of an orchestration, and; (3) a combination of (1) and (2).

C. Automated pattern processing

An important requirement for implementing a usable SPDI pattern-driven management and adaptation of the IoT infrastructure is to support the automated processing of developed patterns. To achieve this, the SPDI patterns can be expressed as *Drools* [12] business production rules, and the associated rule engine, by applying and extending the Rete algorithm [13]. Drools is a business-rule management system with a forward-chaining and backward-chaining inference-based rules engine, allowing fast and reliable evaluation of business rules and complex event processing. A rules engine is also a fundamental building block to create an expert system which, in artificial intelligence, is a computer system that emulates the decision-making ability of a human expert.

In this work Drools are used to encode the relationship between orchestration-wide (end-to-end) and component properties in SPDI patterns in ways that allow the inference of the component-level and orchestration-level SPDI properties required.

III. Implementation Approach

A. Pattern language definition

The first step in implementing the envisioned framework is to define a language through which patterns can be expressed; said language must enable: the design of IoT applications that satisfy required SPDI properties; the verification that existing IoT applications satisfy required SPDI properties at design time, prior to the deployment of the application, and; the adaptation of IoT applications or partial orchestrations of components within them at runtime in a manner that guarantees the satisfaction of required SPDI properties.

Considering the above, the defined language needs to: provide constructs for expressing/encoding dependencies between SPDI properties; be structural; support static and dynamic verification of SPDI properties; be machine process-able.

1) IoT orchestration system model: An IoT orchestrations' system model was defined for the specification of IoT application components and their interactions, as well as the SPDI properties which may be required of such components and their orchestrations. Once defined, this model (referred to as IoT system model in the rest of this document) can be used in conjunction with patterns to enable the reasoning required for determining the applicability of particular SPDI patterns in specific IoT applications and subsequently reason based on them to enable different types of adaptation. The IoT system model advocates an orchestration-based approach. In this approach, the interactions between the different types of components of such applications (e.g., software components, software services, sensors, actuators) interact with each other as specified as orchestration(s) within the IoT application. Such orchestrations are modelled by an $Orchestration$ entity. An orchestration of activities may be of different types depending on the order in which the different activities involved in it must be executed; i.e, an orchestration may be defined as a $Sequential, Parallel, Merge, Choice\ or\ Iterate$ orchestration.

Moreover, an orchestration involves orchestration activities. The types of IoT application activity implementers are grouped under the general concept of $placeholder$. The language introduces also subclasses of the general class Placeholder to represent $Orchestration$ and $OrchestrationActivity$. A placeholder is accessible through a set of $interfaces$. An interface is a named set of operations through which the functions and the data of the placeholder can be accessed from any element outside it.

The individual operations that constitute the interface of a placeholder are represented by the class $Operation$. An operation has a set of parameters: i) name, ii) input and iii) output. Name is used as an identifier for the Operation and the input and output are a set of Parameters.

Placeholders (of all different types) may also be characterised by their SPDI and QoS properties. A property of a placeholder is specified according to the class $Property$. According to it, a Property has a name, a type, a verification, a category and a dataState. The attribute type refers to the state of the property, which can be required or confirmed. A required property is a property that a placeholder must hold in order to be included (considered for) the orchestration. For example, if the required property of an orchestration defining a secure logging process is Confidentiality, then all placeholder activities involved in the orchestration and the links between them may be required to have the Confidentiality property. On the other hand, a confirmed property is a property that is verified at runtime, through a specific means as defined in the Verification.

$Verification$ is a class that describes the way a Property of a Placeholder is verified. The verification process can be done through monitoring, testing, a certificate or via a pattern. In case of a pattern the Mean of verification is the pattern itself; in all the other cases we need an interface to a corresponding monitoring tool, testing service or certificate repository through which the verification can take place.

A Property can belong to *confidentiality, integrity, availability, privacy, dependability, interoperability* or *QoS*. In this way a classification of the properties is achieved.

The final attribute, *dataState*, is referred to state of the data of a Placeholder. If the Placeholder is an Orchestration, then the state of the data will be in_transit. If we have to do with an OrchestrationActivity and the OrchestrationActivity is bound to a storage service for example, then dataState could also be at_rest. If the OrchestrationActivity is bound to a service or device that transforms data, then dataSate could be in_processing.

Finally, the set of all the SPDI properties that are inferred for the different placeholders of an orchestrator by a pattern are aggregated into PropertyPlan object.

Based on the IoT system model presented above we created a corresponding language the constructs of which constitute an EBNF grammar. Due to the lack of space only a snippet of this grammar is presented in Figure 1.

B. Translation to machine process-able format

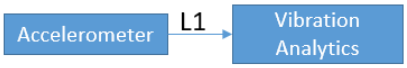
To enable the automated verification of SPDI properties, the IoT deployments described using the above language need to be translated to Drools; to achieve this they are used as input to an ANTLR4 [14] lexer, parser and listener. These programs create a Drools fact for every orchestration activity, control flow operation and property. The Drools facts are then inserted in the Knowledge base of Drools, a repository of all the application's knowledge definitions. Sessions are created from the KnowledgeBase in which data can be inserted and

```
model:   modelelement (COMMA modelelement)*
    ;

modelelement
       :     propertysubject
       |     property
       |     patternrule
       ;

propertysubject
       :     placeholder
       |     operation
       |     parameter
       |     link
       ;

placeholder
    :   placeholdertitle OPEN_PAREN placeholderid CLOSE_PAREN
    |  placeholdertitle OPEN_PAREN placeholderid COMMA interfacename (COMMA
interfacename)* COMMA propertyname (COMMA propertyname)* CLOSE_PAREN
    |  orchestration
    |  orchestrationactivity
    ;
placeholdertitle: 'Placeholder';
placeholderid: STRING;
```

Fig. 1. EBNF of pattern language (snippet)

process instances started. A knowledge session is the way to interact with Drools and the core component to fire Drools rules. Rules themselves are also hold in a knowledge session. The information that is stored in the KnowledgeBase is used for reasoning.

For example, Figure 2 shows a simple orchestration along with its description using the IoT application language. As we can see, the orchestration consists of two Placeholders, Accelerometer and VibrationAnalytics, and a Link between them, named L1. Moreover, they are in sequence (Sequence1), which means that the output of the former is consumed as input by the latter.




```
Placeholder ( "Accelerometer" ),
Placeholder ( "VibrationAnalytics" ),
Link ( "L1", " Accelerometer ", " VibrationAnalytics " ),
Sequence ( "Sequence1", " Accelerometer ", " VibrationAnalytics " )
```

Fig. 2. Simple orchestration

During the first step of the translation of an IoT application orchestration to Drools facts the ANTLR4 lexer recognizes keywords and transforms them in tokens. The created tokens are used by the ANTLR4 parser for creating the logical structure, i.e. the parse tree. Next, the ANTLR4 listener allows us to communicate with Drools every time a node in the parse tree is entered. The listener takes information from the tokens and sends it to Drools. Drools then creates instances from the corresponding Java classes and stores the received information at the class attributes. During the last step, the created java instances are inserted as facts into the knowledge session. These Drools facts are used by Drools rules, which are fired when a condition is met.

C. Pattern-driven property verification - Privacy Example

As mentioned in section III-A1 a mean to verify a SPDI property is by using the patterns. In this subsection a privacy-

focused example is analyzed.

In order to guarantee privacy not only components that form the service should be checked for privacy but also their composition. At each layer of composition, the data union that the layer produces should be evaluated. As an example, consider the composition of a service C of two components A, B. Let us assume that for each $xinA, B, C$

- OUT^x are the sets of outputs of x
- IN^x are the sets of inputs of x
- $E^x = IN^x \cup OUT^x$
- V^x and C^x are two disjoint subsets of EX which partition it into public parts V^x and confidential parts C^x
- L is a corpus of sets that are predefined that expose privacy

Then in order the composition to satisfy the privacy requirements, the following properties must hold:

1) $V^A \cap L = \emptyset$
2) $V^B \cap L = \emptyset$
3) $V^C \cap L = \emptyset$

Moreover, when data are at rest (i.e. in storage) we should take precautions that:

4) $(V^A \cup V^B \cup V^C) \cap L = \emptyset$

Still, the following properties should also hold:

5) $(VA \cup VB) \subseteq V^C$
6) $(V^A \cup V^B) \cap C^C = \emptyset$

The machine process-able format of the above patterns in the form of Drools is shown in Fig: 3. On this pattern we make the appropriate checks that the pattern can be applied on lines 2 till 9 then we make the appropriate definitions and we call at line 14 a predefined function that is created to make the above checks at the sets mentioned above.

```
1. rule "Identifiability"
2. when
3.   $A: Placeholder($output_A: Activity.output)
4.   $B: Placeholder($output_B: Activity.output)
5.   $ORCH: Merge($A, $B)
6.   $OP: Property( propertyName == "Identifiability",
7.     subject == $ORCH, satisfied == false)
8.   $SP: PropertyPlan(propeties contains $OP)
9. then
10.  PropertyPlan newPropertyPlan = new PropertyPlan($SP);
11.  newPropertyPlan.removeProperty($OP);
12.  Property NP_A = new Property($OP, "Identifiability", $A);
13.  Property NP_B = new Property($OP, "Identifiability", $B);
14.  IdentifiabilitycheckResult=checkIdentifiability(NP_A,
   NP_B,$ORCH)
15.  insert(NP_A)
16.  insert(NP_B)
17.  insert(newPropertyPlan);
18. end
```

Fig. 3. Identifiability

D. Key modules

The implementation of the SPDI pattern-driven approach of this work relies on the development of some key components and their integration at the various layers of an IoT deployment. These are:

- *(Backend) Pattern Orchestrator:* Module featuring an underlying semantic reasoner able to understand instantiated Patterns, as received from the Admin and transform them into composition structures (orchestrations) to be used by architectural patterns to guarantee the required properties. The Pattern Orchestrator (PO) is then responsible to pass said patterns to the corresponding Pattern Modules (as defined in the Backend, Network and Field layers), selecting for each of them the subset of these that refer to components under their control (e.g. passing field-specific patterns to the IoT gateway).
- *Backend Pattern Module:* Features the pattern engine for the framework backend, along with associated subcomponents (knowledge base, reasoning engine). It enables the capability to insert, modify, execute and retract patterns at design or at runtime in the backend; these interactions will happen through the interfacing with the Pattern Orchestrator (see above). It is able to reason on the SPDI properties of aspects pertaining to the operation of the framework backend. Moreover, at runtime the backend Pattern module may receive fact updates from the individual Pattern Modules present at the lower layers (Network & Field), allowing it to have an up-to-date view of the SPDI state of said layers and the corresponding components.
- *Network Pattern Module:* Integrated in the network (typically on a Software Defined Network controller) to enable the capability to insert, modify, execute and retract network-level patterns at design or at runtime.
- *Field Layer Pattern Module:* Typically deployed on the IoT/IIoT gateway, able to host design patterns as provided by the Pattern Orchestrator. Since the compute capabilities of the gateway can be limited, the module is able to host patterns in an executable form compared to the pattern rules as provided in the other layers. The executable patterns are able to guarantee SPDI properties locally based on the data retrieved and processed by the monitoring module.

IV. Application Example

To demonstrate the use of the concepts and constructs defined in the above sections, a simple use case is analyzed, featuring a Wind Park IIoT deployment. In more detail, we assume that data captured by an IIoT accelerometer sensor on the Wind Turbine is relayed to IIoT gateway for vibration analytics, and the output of the analytics is relayed to the backend for monitoring and alarm purposes. We further assume that the required SPDI property is end-to-end confidentiality, throughout the interactions involved.

When defining the orchestration depicted in Fig. 4, the end-to-end confidentiality property required is broken down to individual properties for the two activities and the link between them. Using the language defined in subsection III-A1, the above workflow can be formally described as depicted in Fig. 5.

In Fig. 6 the steps of the next phase, i.e. the system deployment, are shown, following the generic process detailed in the previous section.

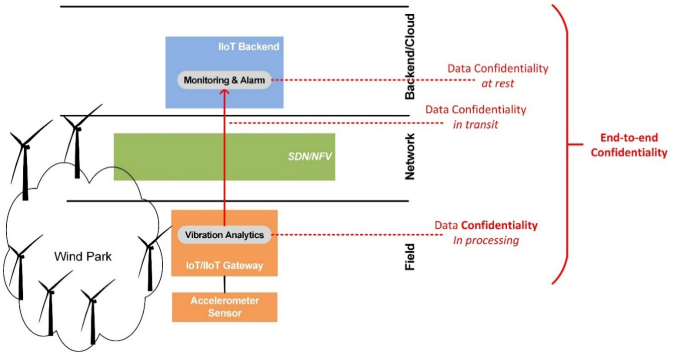


Fig. 4. Sample Application - Windpark IIoT deployment

```
0. ORCH "Seq2"
1. Placeholder (Placeholder1, (Vibration Analysis Activity, Vibration Analysis Description))
2. Placeholder (Placeholder2, (Monitoring Alarm Activity, Monitoring Alarm Description))
3. Sequence (Placeholder1, Placeholder2)
4. Link (Link1, Vibration Analysis, Monitoring Alarm)
5. Property (AP_1, Placeholder1, required, (certificate, interface), confidentiality, in_processing)
6. Property (AP_2, Placeholder2, required, (monitoring, interface), confidentiality, at_rest)
7. Property (OP, "Seq2", required, (pattern-based, "PR1"), confidentiality, end_to_end)
8. Pattern rule: (PR1: AP_1, AP_2→ OP)
```

Fig. 5. Scenario orchestration definition

The rules for the individual properties are stored on the Pattern Global Repository and then relayed by the Pattern Orchestrator to the pertinent layers for monitoring and verification; i.e. AP1, at the IIoT gateway, AP2 to the SDN Controller, while AP3 only stays at the backend. At runtime, the individual SDN pattern engines collect monitoring data from the corresponding interfaces defined for each property at the specific layers components, reason on collected data and trigger adaptation actions if needed. Changes in the system state related to the monitored properties are stored as new facts or trigger updates in the stored facts in the corresponding Pattern repositories; for the network and field pattern engines, these are also transferred to the backend repository, to enable it to have an up-to-date global view of the SPDI state of the whole deployment. This process is shown in Fig. 7.

For brevity reasons, let us analyze only the interactions between Pattern Orchestrator and Network Pattern Engine: As soon as PO receives the instantiated pattern as shown in Fig:5 on line 6 it sees that a Link should be created. So PO contacts the Network pattern Engine and checks if the Link Pattern was previously send. If not PO creates the pattern and sends it to the Network Pattern Engine. An example of the link pattern appears below.

```
package rules
import sdn.PathmanagerCreator;
import backendPatternNotificator
rule "Link"
  when
  $link:Link()
   exists Placeholder(name== $link.endpoint1)
   exists Placeholder(name== $link.endpoint2)
  then
  $link.ID=PathManagerCreator.CreateLink(
      $link.endpoint1,$link.endpoint1);
end
rule "Link DELETION"
```

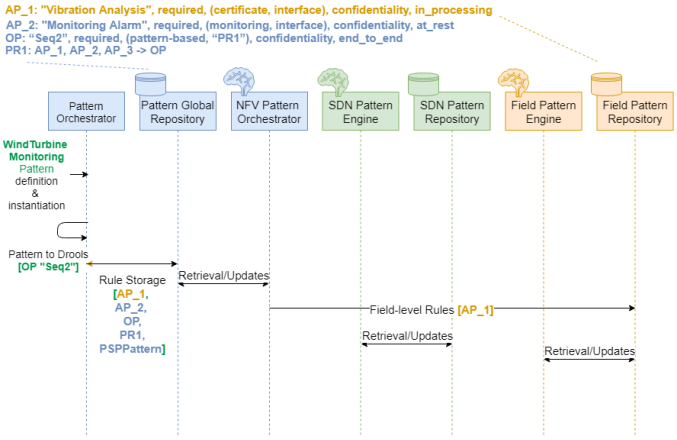


Fig. 6. Sequence - System deployment phase

```
    when
        m: Message( status==Message.
            LINKDELETED)
    then
        backendPatternNotificator.setMessage(
            "link" + m.getLinkId()+"deleted")
            ;
end
```

Listing 1. Link Pattern

The PO sends this pattern to the Network pattern Engine along using its Northbound API and also add some facts to the network pattern engines working memory. Those facts are listed below.

```
Placeholder(ID="ID1",name="vibration analysis
    ",description="Vibration analysis
    description...");
Placeholder(ID="ID2",name="Monitoring Alarm"
    ,description="Monitoring Alarm
    description...");
Link(ID="Link1",entpoint1="vibration analysis
    ",entpoint2="Monitoring Alarm")
```

Listing 2. Facts inserted

As soon as those facts are inserted in the working memory of the drools engine it will fire the Link rule described in Listing 1 which will create the link between the vibration analysis offering and the monitoring alarm offering.

V. Related Works

Researchers have long sought the ability to verify the desired properties of service orchestration as part of the design process, long before the introduction of the IoT concept. They build upon two different approaches to secure SOA applications: model-driven development [15]–[17] and the use of security patterns [18]. In the former, software component and service compositions are modelled using formal languages and the required security properties are expressed as properties on the model [19]. Pattern-based approaches can be found in different research areas such as service-oriented systems [20].

The pattern-driven approach presented herein is inspired from similar pattern-based approaches used in service-oriented

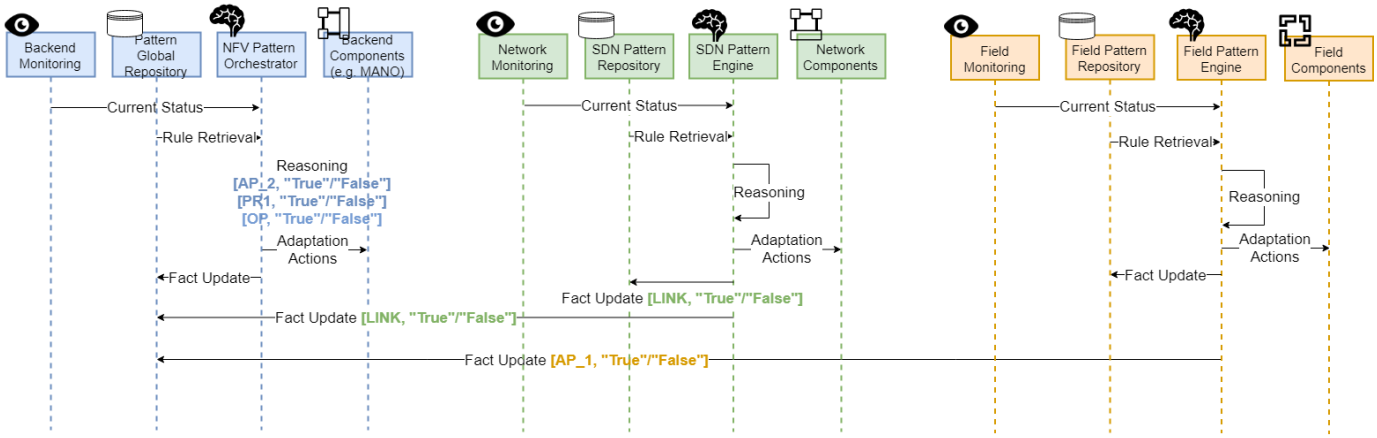


Fig. 7. Sequence - Runtime phase

systems [20], cyber-physical systems [21], and networks [22], while covering the intricacies of IoT deployments and additional properties, while providing guarantees and verification capabilities that span both the service orchestration and deployment perspectives.

VI. Conclusion

In this work we presented a pattern language, and a framework built to run this language, that can be used to guarantee Security, Privacy, Dependability and Interoperability in an IoT infrastructure. Moreover an proof of concept example is presented that ensures privacy between sensors communication in an Windpark IoT application. Further work can be done to improve the usability of the framework by creating a Graphical User Interface for the creation and instantiation of the patterns proposed.

Acknowledgment

This work has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No. 780315 (SEMIoTICS), as well as the Marie Skodowska-Curie Actions (MSCA) Research and Innovation Staff Exchange (RISE), H2020-MSCA-RISE-2017, under grant agreements No. 777855 (CE-IoT)

References

[1] Rantos K. et al., "Secure policy-based management solutions in heterogeneous embedded systems networks", 2012 International Conference on Telecommunications and Multimedia (TEMU), pp 227-232, 2012, DOI:10.1109/TEMU.2012.6294723

[2] Fysarakis K. et al., "Policy-based access control for DPWS-enabled ubiquitous devices" Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), 2014, DOI:10.1109/ETFA.2014.7005233

[3] Fysarakis K. et al., "RT-SPDM: Real-Time Security, Privacy and Dependability Management of Heterogeneous Systems". In: Tryfonas T., Askoxylakis I. (eds) Human Aspects of Information Security, Privacy, and Trust. HAS 2015. Lecture Notes in Computer Science, vol 9190, Springer, Cham, 2015.

[4] Fysarakis K. et al., "Architectural Patterns for Secure IoT Orchestrations", Global IoT Summit 2019 (GIoTS'19), Aarhus, Denmark, June 17-21, 2019.

[5] Hatzivasilis G. et al., "The industrial internet of things as an enabler for a circular economy Hy-LP: a Novel IIoT protocol, evaluated on a wind parks SDN/NFV-enabled 5G industrial network", Computer Communications, 119, 127-137, 2018.

[6] Lukacs A., "What Is Privacy? the History and Definition of Privacy," p. 256265, 2017.

[7] Dennedy M. et al., "The Privacy Engineers Manifesto: Getting from Policy to Code to QA to Value," Apress, p. 400, 2014.

[8] Laprie J., "Dependable computing and fault-tolerance," in Digest of Papers FTCS-15, 1985.

[9] Hatzivasilis, G. et al., "The Interoperability of Things: Interoperable solutions as an enabler for IoT and Web 3.0." 10.1109/CAMAD.2018.8514952.

[10] Abowd G. D., Allen R., and Garlan D., "Formalizing style to understand descriptions of software architecture", ACM Transactions on Software Engineering and Methodology (TOSEM), 4(4), 319-364, 1995.

[11] Schumacher M., "Security engineering with patterns: origins, theoretical models, and new applications", Vol. 2754, Springer Science & Business Media, 2003.

[12] Business Rules Management System (BRMS), https://www.drools.org

[13] Forgy C., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," Artificial Intelligence, vol. 19, p. 1737, 1982

[14] ANother Tool for Language Recognition, https://www.antlr.org

[15] Bartoletti M, et al. "Semantics-based design for secure web services." Software Engineering, IEEE Trans. on, 2008.

[16] Deubler M., et al. "Sound development of secure service-based systems." In Proc. of the 2nd Int. Conf. on Service oriented computing. ACM, 2004.

[17] Geor G., et al. "Verification and trade-off analysis of security properties in UML system models." IEEE Trans. on Software Engineering, 36(3): 338-356, 2010.

[18] N. A. Delessy and E. B. Fernandez, "A Pattern-Driven Security Process for SOA Applications," 2008 Third International Conference on Availability, Reliability and Security, Barcelona, 2008, pp. 416-421.

[19] Dong, J., et al, Automated verification of security pattern compositions. Inf. Softw. Technol., vol. 52, no. 3, 2010.

[20] Pino L., et al. Pattern Based Design and Verication of Secure Service Compositions. IEEE Transactions on Services Computing (2017)

[21] Maa A. et al. Extensions to Pattern Formats for Cyber Physical Systems. Proceedings of the 31st Conference on Pattern Languages of Programs (PLoP14. Monticello, IL, USA. Sept. 2014.

[22] Petroulakis N. et al., "Fault Tolerance Using an SDN Pattern Framework", 2017 IEEE Global Communications Conference (GLOBECOM), 2017