Ontology-driven model of security patterns.
Common description

Property	*Range and predefined values*	*Description*	*Example*
Class: **Pattern**			*pattern_SecureDistributedPublishSubscribIoT*
Metadata: This section contains properties, used to identify a particular pattern, its authors, the idea of it. Also, it holds links to an original documents. In many cases it is impossible to put a full description of the pattern because of copyright and trademark, so the "textreview*" fields retell a content of the pattern in own words. The most valuable thing is an original pattern's document, so it should be a way to get access to it, locally or remotely.			
textName	**xsd:String**	Pattern's primary name	*Secure Distributed Publish/Subscribe (P/S) pattern for IoT*
textAKAName	**xsd:String**	Pattern's alternative name(s)	
textAuthor	**xsd:String**	Pattern's author(s)	*Eduardo B. Fernandez* *Nobukazu Yoshioka* *Hironori Washizaki*
textURL	**xsd:String**	URL(s) of webpage that describes a pattern	*https://www.researchgate.net/publication/339103887_Secure_Distributed_PublishSubscribe_PS_pattern_for_IoT*
textPDF	**xsd:String**	Downloadable URL(s) of pattern's PDF	
textReference	**xsd:String**	Reference to a paper, describing a pattern	*E.B.Fernandez, N. Yoshioka, H. Washizaki, "Secure Distributed Publish/Subscribe (P/S) for IoT, 2020. Procs. Asian PLoP'20, March 4 6, Taipei, Taiwan. 9 pages.*
textIntent	**xsd:String**	Pattern's Intent (full text)	*In an IoT system, decouple the publishers of events from those interested in the events (subscribers). Subscription and publication are performed securely.*
textReviewContext	**xsd:String**	Brief text description of pattern's context	Something like that: *Information exchange between IoT/IIoT devices (e.g. smart thermostats or sprinkler systems, different sensors) with minimal security control and cloud/fog applications.*

textReviewProblem	**xsd:String**	Brief text description of pattern's problem	Something like that: *Subscribers (S) register and receive messages of their interest sent by a publisher (P). The main concerns are how to organize the interactions between them securely, avoiding rogue participants, insecure communications, unwanted P/S operations.*
textReviewSolution	**xsd:String**	Brief text description of pattern's solution	*In addition of the standard P/S functions it is possible to use secure channels for protected communications, access control for restricting the actions of publishers and subscribers, security logging, and digital signatures.*
Organizational and scope aspects: This section describes organizational aspects of a pattern (type, used template) and its scope, i.e. relation to other patterns. It has to be given only information about a particular pattern, without worrying about dependent patterns, because the automatic reasoning procedures allow to create a full "network" of pattern relations, thanks to the inverse and symmetric properties.			
hasType	**Pattern** Predefined instances: Class: SecurityPattern type_SecurityPattern Class: ThreatPattern type_ThreatPattern Class: MisusePattern type_MisusePattern	Type of a pattern, like security pattern, misuse pattern, or threat pattern. It is possible to define this with the class assertion, e.g. <pattern> is an instance the SecurityPattern class.	The first option: *hasType value type_SecurityPattern* The second option is to tell it is an instance of the *SecurityPattern* class
hasTemplate	**Template** Predefined instances: template_POSA template_GOF template_ESP ???	Template, used to describe a pattern. It can be POSA or GOF. POSA stands from "Pattern Oriented Software Architecture". GOF stands from "Gang of Four".	*hasTemplate value template_POSA*
hasGroup (inverse: isGroupOf)	**Group**	Tells to which group a pattern belongs to. It can be possible to use the class assertion here, e.g. create a hierarchy with abstract	*hasGroup value patterngroup_SecureMiddleware*

		patterns on the top and concrete ones at the bottom.	
usesPattern (inverse: isUsedByPattern)	**Pattern**	Enumerates patterns that are used by this one. For the POSA template it should be taken from "Description" and "Class Diagram".	*pattern_RoleBasedAccessControl* *pattern_Authenticator* *pattern_SecurityLoggerAuditor* *pattern_SecureChannel*
relatesToPattern (symmetric)	**Pattern**	Enumerates patterns that are related to this one. For the POSA template it should be taken from “Related patterns”.	*pattern_SecurePS* *pattern_Broker* *pattern_SecureChannel* *pattern_EnterpriseServiceBus* *pattern_Authorizer* *pattern_IoTSegmentation*
isChildOf (inverse: isParentOf)	**Pattern**	For a concrete pattern shows from which abstract pattern it has been made. It can be possible to use the class assertion here, e.g. create a hierarchy with abstract patterns on the top and concrete ones at the bottom.	*IsChildOf value* *pattern_SecurePublishSubscirbe*
Common characteristics: This section contains common labels, used to characterize a pattern.			
hasDomain (inverse: isDomainOf)	**Domain** Predefined instances: domain_FogComputing domain_EdgeComputing domain_InternetOfThings domain_SCADA domain_Military domain_Ecommerce domain_GridComputing Class: CloudComputingDomain domain_CloudComputing domain_IaaS domain_SaaS domain_PaaS	Tells to which domain(s) a pattern belongs to. Domain is a large functional field of Information Technologies (IT), like Cloud Computing, Internet of Things. It might be less gigantic, like IaaS or NVF.	*hasDomain value domain_InternetOfThings*

	domain_NVF		
hasArchitecturalLayer	**ArchitecturalLayer** Predefined instances: Class: ApplicationArchitecturalLayer al_ApplicationLayer al_ClientLayer al_LogicLayer al_DataLayer al_PlatformAndOperatingSystemLayer Class: CommunicationArchitecturalLayer al_CommunicationLayer al_DistributionLayer al_TransportLayer al_NetworkLayer	"Architectural layer provides another useful dimension, since problems and their solutions in different layers of the architecture differ, yet all are important. Roughly the same architecture continuum has been divided in different ways for communication protocols, business systems, and execution environments, but always with an ordering from low to high level of abstraction, and from network to platform to application. " [VanHilst, 2009] Instances are taken from [VanHilst, 2009]	*hasArchitecturalLayer value al_ApplicationLayer* *hasArchitecturalLayer value al_ClientLayer* *hasArchitecturalLayer value al_CommunicationLayer*
hasConstraintLevel	**ConstraintLevel** Predefined instances: cl_RegulatoryLevel cl_OrganizationalLevel cl_HumanLevel cl_MechanismLevel	"Leveson defines four levels of constraint: mechanism, human (operator or developer), organizational, and regulatory. In Leveson's work on system safety (Leveson, 2004), each level of constraint plays an important role in safety failures and their prevention. By extension, we use the same levels for security with an axis with levels from thing to society. While most security patterns describe mechanisms, the National Training Standard for Information Systems Security (INFOSEC) Professionals (National Security Agency, 1994) is mostly concerned with practices, policies, and regulations. The Common Criteria has functional requirements that apply at the level of mechanisms (Common Criteria Sponsoring Organizations, 2006). But it also has assurance requirements that concern organizational processes to document actions taken. The development of a configuration management plan is a Common Criteria assurance requirement that applies at the	*hasConstrainLevel value cl_MechanismLevel*

		organizational level in the lifecycle stage of domain analysis. The Common Criteria and other standards such as SOX and SSE-CMM (Systems Security Engineering - Capability Maturity Model, 2003) themselves belong at the regulatory level." [VanHilst, 2009] Instances are taken from [VanHilst, 2009]	
hasResponseType	**ResponseType** Predefined instances: rt_Avoidance rt_Deterrence rt_Prevention rt_Detection rt_Mitigation rt_Recovery rt_Forensics	"The response axis based on whether or not and attack happens and the extent, from not happening at all (avoidance), to completely happened and in the past (forensics)." [VanHilst, 2009] Instances are taken from [VanHilst, 2009]	*hasResponseType value rt_Avoidance*
hasLifecycleStage	**LifecycleStage** Predefined instances: lc_ArchitectureAndDesign lc_BuildAndCompilation lc_Implementation lc_Installation lc_Operations lc_Requirements lc_SystemConfiguration lc_Deployment	Tells which which system's lifecycle stage a pattern is applicable. Frankly, most of the patterns are applicable on the Design (Architecture) stage, but it might be possible to have a few exceptions. Instances are taken from [CAPEC].	*hasLifecycleStage value lc_ArchitectureAndDesign*
hasSecurityLevel	**SecurityLevel** Predefined instances: sl_PhysicalSecurity sl_PersonnelSecurity sl_CommunicationAndDataSecurity sl_OperationalSecurity	Tells to which field of security a pattern belongs to. https://en.wikipedia.org/wiki/Physical_security https://en.wikipedia.org/wiki/Secure_communication https://en.wikipedia.org/wiki/Communicatio	*hasSecurtyLevel value sl_CommunicationAndDataSecurity*

		ns_security https://en.wikipedia.org/wiki/Operations_security	
Context characteristics: The “context characteristics” and “context security characteristics” (see below) sections include a set of labels that allow to put a pattern to a context. Here, “context” means a possibility to use the pattern as a part of an architecture of a particular computer system. In particular, this set of labels can be used to support a decision, like “Is the pattern suitable for the system design or not?”. The idea of the context approach is that each computer system can be describe in two ways. Firstly, it is possible to find some unique features, i.e. ***functions*** that build a coherent model of this system. Secondly, most of the computer systems are created from common ***components***, independent from system functions, like hardware servers, operating systems, software services and applications. For example, what functions make a hypervisor (IaaS component) unique? An answer might include “Management of VMs”, “VM migration”, “Virtual networking” etc. Which common components does it consist of? It might be “Hardware server”, “Operating system”, “System service”, “Network service”, “CLI interface”, “API interface”. In many cases security problems of common components are known and described well, and security problems of functions are in focus of research of a new type of computer system. Considering abstract (common) patterns, it is better **to describe more components and less functions**. Considering domain specific patterns, it is better **to describe more functions and less components**. More details are in the ***schema_functions_components.pdf*** file.			
hasAffectedFunction	**Function** see ***schema_functions_components.pdf***	Tells which system function(s) a pattern affects.	*hasAffectedFunction value function_DistributeEventInformation*
hasAffectedComponent	**Component** see ***schema_functions_components.pdf***	Tells which common component(s) a pattern affects.	*hasAffectedComponent value component_IoTApplication* *hasAffectedComponent value component_IIoTApplication* *hasAffectedComponent value component_CloudApplication* *hasAffectedComponent value component_FogApplication*
Context security characteristics: After suitable patterns have been found for a system design (see above), the next step is to correlate them to security challenges. A relevant question, which this set of characteristics allows to answer, is "Does this security pattern solve a particular security problem, valuable for its context?"			

Sure, a final decision is the responsibility of a system architect, but the context security characteristics allow to reduce number of options and show only relevant security solutions.			
hasThreat	**Threat** Predefined instances: see ***schema_threats.pdf*** Class: CommunicationsThreat **threat_ManInTheMiddle** **threat_Interception** **threat_Flooding** **threat_ContentSpoofing** **threat_IdentitySpoofing** **threat_Footprinting** **(=threat_InformationGathering)** **threat_ProtocolAnalysis** Class: SoftwareThreat **threat_SessionManipulation** **threat_BruteForce** **threat_AuthenticationAbuse** **threat_AuthenticationBypass** **threat_PriviledgeAbuse** **threat_PriviledgeEscalation** **threat_Excavation** **threat_CodeInjection** **threat_BufferManipulation** **threat_ExcessiveAllocation** **threat_ManipulationAPI** **threat_InputDataManipulation** **threat_EnvironmentManipulation** **threat_SharedDataManipulation** **threat_Malware**	Tells what threats a pattern describes with connection to component(s) and function(s), figured out on the previous stage. For security patterns defines the possible threats, met by a pattern. For attack pattern defines the possible threats, produced by an implementation of pattern. The idea has taken from [Guan, 2016] Instances are taken from [CAPEC]	“S1: An impostor impersonates a subscriber and subscribes to receive information that will be billed to somebody else or which will give her access to sensitive information.” so *hasThreat value threat_PriviledgeEscalation* "S2: The publisher is an impostor and collects information (and maybe money) from potential subscribers." *hasThreat value threat_IdentitySpoofing* “S3: The subscription messages are intercepted and read or modified by an attacker. The attacker may obtain in this way credit or other personal information from the subscriber.”, so *hasThreat value threat_Interception* "P2: A publisher publishes erroneous information. This action can inject data or commands to a device thus disturbing its operation;" *hasThreat value threat_ContentSpoofing* “P4. An attacker floods subscribers with fake messages thus stopping the subscribers from doing any useful work; this is a Denial of Service attack.”, so *hasThreat value threat_Flooding*
hasSecurityConcern	**SecurityConcern**	Tells which security concern(s) a pattern touches.	*hasSecurityConcern value concern_AccessControl*

	Predefined instances: concern_AccessControl concern_AttackDetection concern_AttackPrevention concern_Audit concern_Authentication concern_Authorization concern_Containment concern_Coordination concern_EventLogging concern_Identification concern_InformationHiding concern_KeyDistribution concern_MessageAuthentication concern_MessageAuthenticity concern_MessageIntegrity concern_Monitoring concern_ProcessIsolation concern_Sandboxing concern_Realibility concern_ResourceManagement concern_RightsManagement concern_SecureDataStream concern_SecureCommunictations concern_SecureSystemIntegration concern_SecureSystemAdministration concern_SecurityArchitecture concern_SecurityPolicy concern_TrafficMonitoring concern_TrafficFiltration	A security concern represents some security feature(s) [Guan, 2016], xxx Instances are taken from [VanHilst, 2009], [Vale, 2019].	*hasSecurityConcern value concern_EventLogging* *hasSecurityConcern value concern_MessageIntegrity* *hasSecurityConcern value concern_SecureCommunictations*
Inferred characteristics: There is no need to define these properties. Automatic reasoning procedures will assign them from the context properties. STRIDE and security objectives depend of each other. Either of them can be obtain from the hasThreatImpact and hasThreat properties. To create lists of attacks the CAPEC enumeration can be used, to create lists of weaknesses the CWE enumeration can be used. CAPEC and CWE are connected to each other.			
hasThreatImpact	**ThreatImpact** Predefined instances:	Tells which negative impact(s) the threats, described by a pattern, have to component(s) and function(s). It obtains from the CAPEC	

	see ***schema_threats.pdf*** ti_AlterExecutionLogic ti_BypassProtectionMechanism ti_ExecuteUnauthorizedCommands ti_GainPrivileges ti_HideActivities ti_ModifyData ti_ReadData ti_ResourceConsumption ti_UnreliableExecution	attack descriptions (the hasThreat property here). It is not so far from the CAPEC/CWE approach, used to describe consequences of their attack patterns (but with some improvements). This allows to map attacks from CAPEC and weaknesses from CWE to the ThreatImpact items.	
hasSTRIDE	**STRIDE** Predefined instances: STRIDE_Spoofing STRIDE_Tampering STRIDE_Repudiation STRIDE_Information_Disclosure STRIDE_Denial_of_Service STRIDE_Elevation_of_Privilege	Tells which STRIDE item(s) a pattern touches.	
hasSecurityObjective	**SecurityObjective** Predefined instances: SO_AccessControl SO_Accountability SO_Authentication SO_Authorization SO_Availability SO_Confidentiality SO_Integrity SO_NonRepudiation	Tells which security objective(s) a pattern touches.	
hasPossibleAttack	**Attack**	Will be taken from [CAPEC] and other attacks' classifications.	

hasPossibleWeakness	**Weakness**	Will be taken from [CWE] and other weaknesses'/vulnerabilities' classifications.	

References:

[Guan, 2016] H. Guan, H. Yang, and J. Wang, "An ontology-based approach to security pattern selection," Int. J. Autom. Comput., vol. 13, pp. 168–182, Apr. 2016.

[Vale, 2019] A.P. Vale, E. B. Fernández, "An Ontology for Security Patterns". Conference paper. 2019.

[VanHilst, 2009] VanHilst M. et al. A multi-dimensional classification for users of security patterns //Journal of Research and Practice in Information Technology. – 2009. – Т. 41. – №. 2. – С. 87.

[CAPEC] https://capec.mitre.org/

[CWE] https://cwe.mitre.org/