Cybersecurity

RESEARCH

Open Access

Abstract security patterns and the design of secure systems

Eduardo B. Fernandez[1*], Nobukazu Yoshioka[2], Hironori Washizaki[3] and Joseph Yoder[4]

Abstract

During the initial stages of software development, the primary goal is to define precise and detailed requirements without concern for software realizations. Security constraints should be introduced then and must be based on the semantic aspects of applications, not on their software architectures, as it is the case in most secure development methodologies. In these stages, we need to identify threats as attacker goals and indicate what conceptual security defenses are needed to thwart these goals, without consideration of implementation details. We can consider the effects of threats on the application assets and try to find ways to stop them. These threats should be controlled with abstract security mechanisms that can be realized by *abstract security patterns (ASPs)*, that include only the core functions of these mechanisms, which must be present in every implementation of them. An abstract security pattern describes a conceptual security mechanism that includes functions able to stop or mitigate a threat or comply with a regulation or institutional policy. We describe here the properties of ASPs and present a detailed example. We relate ASPs to each other and to Security Solution Frames, which describe families of related patterns. We show how to include ASPs to secure an application, as well as how to derive concrete patterns from them. Finally, we discuss their practical value, including their use in "security by design" and IoT systems design.

Keywords: Security patterns, Secure software development, Security requirements, Secure software architecture, IoT systems design

Introduction

When solving a problem, we should try to produce first an abstract, conceptual solution, before we get concerned with implementation details. Those details may obscure the objectives of the solution adding unnecessary complexity and might make us overlook the possibility of solving a similar problem in the same way. We need to understand the problem before we can solve it; according to Polya: "Visualize the problem as a whole as clearly and as vividly as you can. Do not concern yourself with details for the moment" (Polya 1957). Building a software application is solving a specific type of problem, where the abstract solution must be realized using software. In the requirements and analysis stages of software development we are trying to understand the problem, it is too early to introduce implementation aspects that just would make the problem harder. This is also true for security. Security is a quality aspect that constrains the behavior of applications by imposing access and use restrictions on the data and other assets, which means that the requirements stage is the appropriate stage to start addressing security. We only want to indicate at this stage which security controls are needed, not their most efficient or convenient implementation. If we consider a financial application example, we only want to specify the business rules of accounts, customers, and transactions with their corresponding restrictions. These restrictions may include: "customers are the only ones who can perform transactions on their own accounts", "an account owner can close his/her account", and similar type of constraints. The constraints come from the semantics of the

*Correspondence: fernande@fau.edu
[1] Department of Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL, USA
Full list of author information is available at the end of the article

application—they may reflect business rules and regulations, and from the necessity to defend against possible threats. At this stage, we can add patterns (or other artifacts), to define abstract security mechanisms to express these restrictions. From our concern for security, by pattern we mean a *security pattern*, an encapsulated solution to a security problem (Fernandez 2013; Steel et al. 2005). These abstract patterns would include only the fundamental characteristics of the security mechanism, not including implementation aspects. In Avgeriou (2003), we introduced[1] the idea of abstract security pattern (ASP), that describes a conceptual security mechanism that realizes one or more security policies able to control (stop or mitigate) a threat or comply with a security regulation or policy. Most works using security patterns (Blakeley and Heath 2004; Fernandez 2013; Schumacher et al. 2006; Steel et al. 2005) apply *concrete patterns*, which provide protection at specific architectural levels or components, e.g., secure virtual address space (VAS) in operating systems (Fernandez 2013). In fact, an examination of the literature did not find any work on patterns (security or other types), where abstraction is explicitly considered. While this concept is mentioned in the patterns of the GOF (Gamma et al. 1994), they did not develop their possibilities. We develop here the concept of ASP based on the definition above and we show its possibilities. The common context of all Abstract Security Patterns is the problem space of the corresponding applications, that can be expressed using domain models for specific knowledge areas. ASPs can be related to each other using pattern diagrams or more precisely through Security Solution Frames. Pattern diagrams indicate how patterns relate to each other, showing the contribution a pattern brings to another. Security solution frames (SSFs) are sets of patterns that correspond to specific concerns of a solution, e.g., authentication (Uzunov et al. 2015a, b; Uzunov and Fernandez 2021).

Some of the ASPs correspond to standard security mechanisms, e.g., Access control (Authorization and Reference Monitor), Security Logger/Auditor, and Authenticator. Others may specify more detailed aspects, e.g., Access Control/Authorization models including the Access Matrix, role-based access control (RBAC), and Multilevel models (Fernandez 2013); although those may not be strictly abstract models, they correspond to institution policies. Starting from ASPs, when developing the lifecycle steps of a complete application we can use a hierarchy of patterns going from abstract security patterns to platform-oriented versions of these patterns and their code realizations.

ASPs are different from principles of good security design, e.g., Single-Point-of-Access (Yoder and Barcalow 2000) or Need to Know (Fernandez et al. 2011), even if these can be represented as patterns. ASPs correspond to application defenses, intended to be realized later by specific computational mechanisms, they do not describe principles, that may have many realizations. In each use case of the requirements stage we can specify what security controls we need and in the conceptual model of the analysis stage we add the corresponding ASPs after we have enumerated the expected threats.

Our contributions include:

- Development and characterization of the concept of ASP by means of analysis and examples.
- Description of the relationships of ASPs to other ASPs and to SSFs to structure the defenses needed for an application.
- Demonstrating the value of ASPs by showing how they can be used to build conceptual models, to derive new patterns, and other possible uses for secure application development, including security by design and IoT systems design.

"Security patterns and security solution frames" section presents some background, while "Abstract security patterns (ASPs)" section discusses the nature of ASPs and presents a complete example of one of them. "ASP-based hierarchies" section considers the properties of ASP-based hierarchies, while "Relationships between ASPs and security solution frames" section relates ASPs to SSFs. "ASPs in secure conceptual models" section shows how to use ASPs to build secure conceptual models. "Deriving concrete patterns from ASPs" section illustrates the derivation of concrete patterns starting from ASPs. "Formalization of ASPs" section formalizes ASPS, while "Evaluation of effectiveness" section evaluates their effectiveness. "Related work and discussion" section discusses related work, while "Conclusions and future work" section presents some conclusions and possible future work.

Security patterns and security solution frames

A *security pattern* is a solution to a security problem, intended to control (stop or mitigate) a specific type of threat by defining a security mechanism, or a way to realize a security policy or regulation, applicable in a given context (Fernandez 2013; Schumacher et al. 2006). The problem solved by the pattern is briefly described in its "Intent" section and elaborated in a "Problem"

[1] We introduced the idea in a two-page paper (Fernandez et al. 2008), but did not develop its properties. We further developed the idea in (Fernandez et al. 2014). This paper considerably expands these previous works. We have also published four complete ASPs (Fernandez et al. 2016, 2018;, 2019, 2020).

section. A set of forces define constraints for the solution, e.g., "the solution must accommodate a variety of users". The solution is typically expressed using UML class, sequence, state, and activity diagrams (although we usually need only one or two of these models). A set of consequences indicate how well the solution satisfied the forces; in particular, how well the attacks were controlled, or a regulation was fulfilled. An implementation section provides guidelines on how to use the pattern in an application, indicating what steps are needed, their possible realizations, and variants. A "related patterns" section enumerates other patterns that complement the pattern or that provide alternative solutions.

Security patterns are classified as architecture patterns because they describe global architecture concepts, e.g., "what type of authentication is needed to control access for the users of a system?" A few of them can also (or instead) be considered as design patterns because they handle aspects of the security code of a component. ASPs are in effect a variety of analysis patterns. An *analysis pattern* describes a semantic aspect of an application, e.g., the description of types of accounts in a financial institution (Fowler 1997). Some security patterns are more useful by looking at them in more than one perspective. For example, the Security Logger in (Steel et al.2005) concentrates on the software implementation of this pattern, so this is a design pattern; the Security Logger in Fernandez (2013) is an ASP because it emphasizes the core functions of this pattern. If we combine both perspectives, this pattern can be useful to software architects and developers. We call the patterns derived from an ASP *concrete patterns,* because they refer to some specific software environment, e.g., a distributed system. There are different degrees of concreteness depending on how specific they are. It is possible to even define patterns with contexts using specific technologies or architectural styles, e.g., IoT patterns (van Heesch et al. 2011; Washizaki et al. 2021). In the development of a software system, we need to use patterns for several architectural levels.

Security solution frames (SSFs) are sets of patterns that correspond to a specific aspect or concern of a security solution (Uzunov et al. 2015a; Uzunov and Fernandez 2021). Their vertical grouping collects together all the patterns that are related to a security concern, often in a hierarchical structure; an Authentication SSF could look like Fig. 4 (Uzunov and Fernandez 2021). Different levels of abstraction (concreteness) of a security mechanism define a vertical structuring, while we can define a horizontal association by connecting different concerns. Security patterns and SSFs are not very useful in isolation (Fernandez 2013; Uzunov et al. 2015b), they should be applied in the stages of a systematic methodology to build secure systems.

We have developed a systematic way of enumerating threats (Fernandez 2013) which consists of analysis of the flow of events in a use case or a sequence of use cases, in which each activity is analyzed to uncover related threats. This analysis should be performed for all the system uses cases. We use pattern diagrams and unified modeling language (UML) class, sequence, and use case diagrams to describe patterns and architectures. A pattern diagram (Buschmann et al. 1996) shows the relationships between patterns, where patterns are shown as rounded rectangles; each arc from pattern A to pattern B indicates the contribution of pattern A to the pattern it points (B).

A *domain model* (DM) is a model of an area of knowledge, e.g., health systems, using an appropriate language. A DM can be defined using sets of related patterns that represent specific aspects of the domain. The components of domain models are usually described in UML or some ontology language like OWL. A *reference architecture* (RA) is an abstract architecture, with concepts of a particular domain (or set of domains), with no implementation aspects (Avgeriou 2003; Taylor et al. 2010). RAs are reusable, extensible, and configurable; they can be built as a set of related patterns describing some type of architecture and they can be instantiated into a concrete software architecture by adding implementation aspects. *Security reference architectures* (SRAs) include a set of ASPs (or other artifacts) that provide defenses for the threats of a reference architecture (Fernandez 2015).

Abstract security patterns (ASPs)

As indicated earlier, an ASP is a security pattern that describes a conceptual semantic restriction in a domain, which can be a defense to a threat or a way to comply with a regulation, with no implementation aspects. That is, an ASP provides only the necessary core functions for those objectives. In this section, we use the Authenticator ASP as example (developed as a complete pattern in (Fernandez et al. 2018); further examples of ASPs include Secure IaaS (Fernandez et al. 2016), Secure Network Segmentation (Fernandez et al. 2019), and Secure Publish/Subscribe (Fernandez et al. 2020). The Intent section of an Authenticator pattern as described in (Fernandez 2013) is: "A user or system (subject) requesting access to a system identifies itself, how does the system verify that the subject is who it says it is? The subject must present some information that is recognized by the system as identifying this subject. After being authenticated, the requestor is given some proof of this fact."

Authentication restricts access to a system to only registered users; it handles the threat where an intruder enters a system and tries to perform unauthorized access to information or other resources.[2] This definition is what

[2] Systems usually also have another level of control that restricts access to specific resources, this is Authorization.

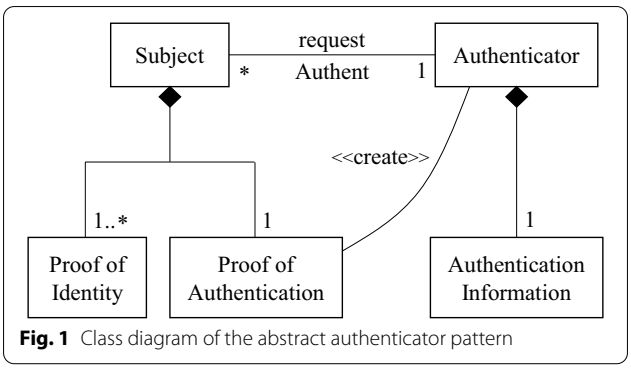

Fig. 1 Class diagram of the abstract authenticator pattern

is usually called *entity authentication* (Song et al. 2003). There are many ways to perform this authentication, that go from manual ways, as done in voting places during elections, to purely automatic ways, as when accessing a restricted web site. Authentication as an abstract function requires a core sequence of activities:

1. The subject requests to enter a system providing its identity and some proof of identity.
2. If the system, using its identity information, recognizes the subject, it grants it entrance to the system and creates for it a proof of authentication (token) for later use. If not, the request is denied.

Concrete realizations of this sequence may implement these steps in different ways, but all must perform these two steps. Figure 1 is the class model of the *Abstract Authenticator*, which shows the classes used to perform the activities above. Class *Subject* indicates the active entity that requests access to the system through some type of interface to class *Authenticator* by providing a *Proof of Identity* (owned by the subject). The Authenticator then searches class *Authentication Information* to decide if the subject is legitimate. Class Authentication Information includes information previously stored by the subject, which is needed to authenticate it, e.g., a list of passwords, a set of fingerprints, a cryptographic protocol, a history of past interactions, or similar. The Authenticator provides the subject with a *Proof of Authentication,* so that the requestor needs not authenticate itself again in later accesses. Dynamic models for this pattern include sequence diagrams for the use cases "Register a subject" and "Request access", which realize the scenario above (Fernandez 2013; Fernandez et al. 2018). Figure 2 shows the successful execution of the use case "Request access". This diagram performs the steps described above: after validating the proof of identity presented by the subject; the Authenticator then creates a proof of authentication which is assigned to the subject.

Even in the absence of any implementation, we can define abstract threats in the ASP. These threats represent violations of the semantic constraints of the application. For the Abstract Authenticator we can have:

- T1. Present fake or stolen proof of identity, to let the attacker impersonate a legitimate subject and get access to the system.
- T2. Steal a proof of authentication for later attempts to enter the system.
- T3. Unauthorized reading of authentication information to obtain a proof of identity.
- T4. Unauthorized modification of authentication information, to produce disruption.

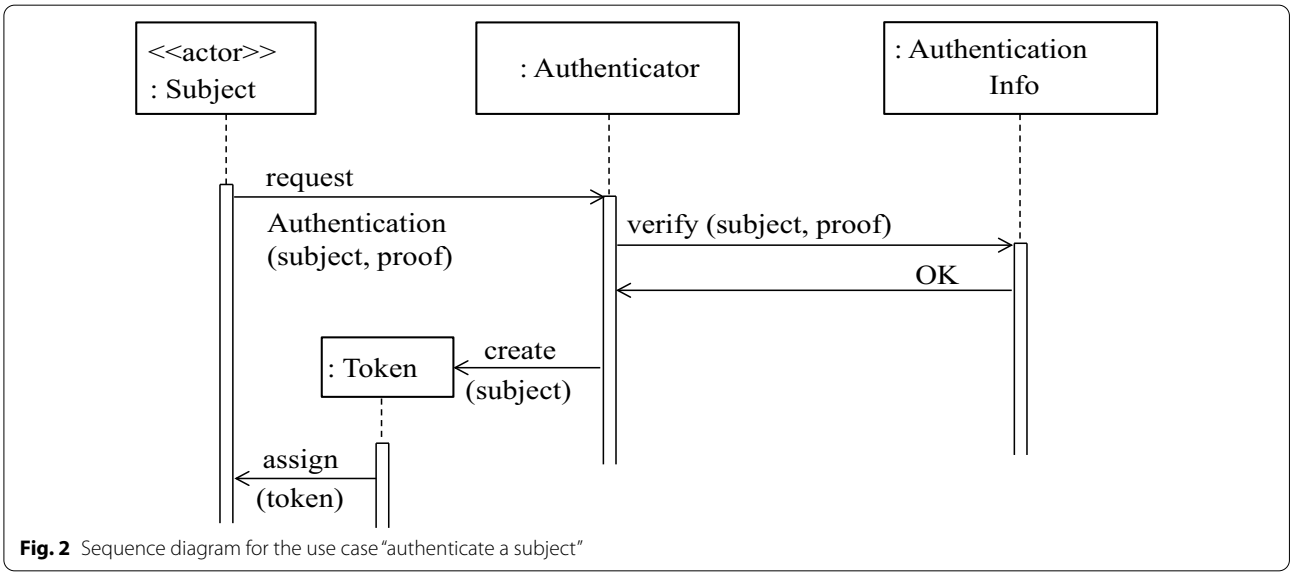

Fig. 2 Sequence diagram for the use case "authenticate a subject"

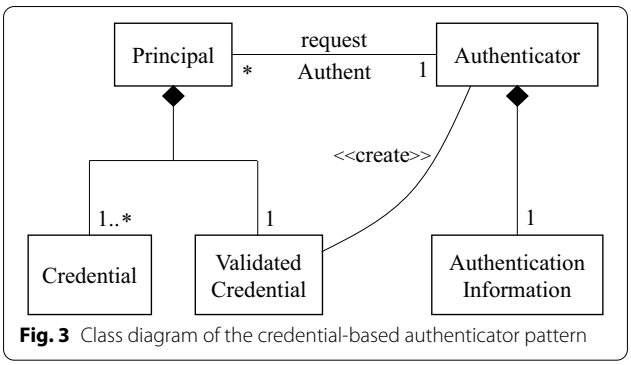

Fig. 3 Class diagram of the credential-based authenticator pattern

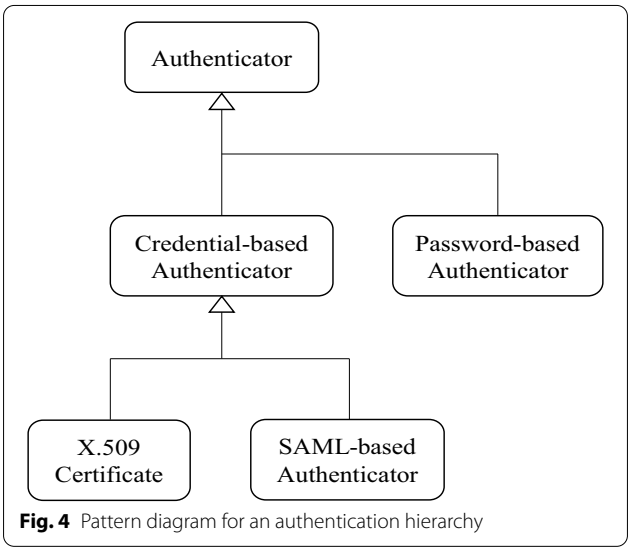

Fig. 4 Pattern diagram for an authentication hierarchy

- T5. Register a subject that has more privileges using false information and then impersonate that subject.

Figure 3 shows the class model of a concrete type of Authenticator. The *Credential-based Authenticator* pattern includes the complete Abstract Authenticator functions where its classes are reinterpreted as: *Principal* corresponds to a responsible subject, Proof of Identity becomes a *Credential* presented by the Principal, Proof of Authentication becomes a *Validated Credential,* and the *Authentication Information* is a procedure to validate credentials (Fernandez 2013). For X.509 Certificates the *Certification Authority* generates credentials, and the Credential includes a set of Attributes that carry the signature of a Certification Authority, authenticate a subject, and maybe include authorization rights and other descriptions of the subject. Authentication is performed by the Authenticator using the certificate and confirming the validity of the Certification Authority that issued the credential.

The threats to the Credential-based Authenticator include implementation-oriented versions of the abstract threats; for example, using a stolen certificate, as well as new threats like using expired credentials (Morrison and Fernandez 2006).

ASP-based hierarchies

We can define hierarchies that show the concrete patterns that can be derived from ASPs. For example, Fig. 4[3] describes a hierarchy of authenticators. ASP-based hierarchies are a way to organize Security Solution Frames, discussed in "Security patterns and security solution frames" section. We show relationships between patterns using pattern diagrams (Buschmann et al. 1996), where rounded rectangles represent patterns and directed arcs indicate the contribution of a pattern to another pattern. The patterns in these diagrams can have relationships such as generalization and aggregation. A generalization hierarchy can relate several patterns with the same concern. The Credential-based Authentication is a concrete security pattern derived from the Abstract Authenticator, as discussed earlier. Credential-based Authenticators use portable proofs of identity and include X.509 Certificates and SAML-based Authentication, among others. The X.509 Authenticator (Fig. 5) requires an extra class to describe the Certification Authority, while SAML-based Authentication (Fig. 6) uses assertions and requires accessing a specific site for validation. The Password-based Authenticator uses a List of Passwords as Authentication Information (Fig. 7). We can deduce some properties, using the Authenticator pattern as an example.

The class models of the concrete patterns derived from an ASP must include all the classes of the ASP from which they were derived as well as classes that handle new aspects required by the specific environment. There may be new or modified attributes and operations in the classes derived from the ASP. If C_i = set of classes in ASPi, Cci = set of classes in a concrete pattern derived from ASPi, and Cnew = new classes in concrete pattern Cci, we have: Cci = Ci ∪ Cnew. By "class" we mean the information in that class; the actual class may be split or merged with another class in the concrete levels but the application's semantic information in the ASP must be preserved.

The context defines the environment where the pattern is valid and any conditions for its application; it is the main determinant of the difference of a pattern with another in a hierarchy. In general, the context of a pattern

[3] This is a partial authentication hierarchy; there are many other ways to perform authentication.

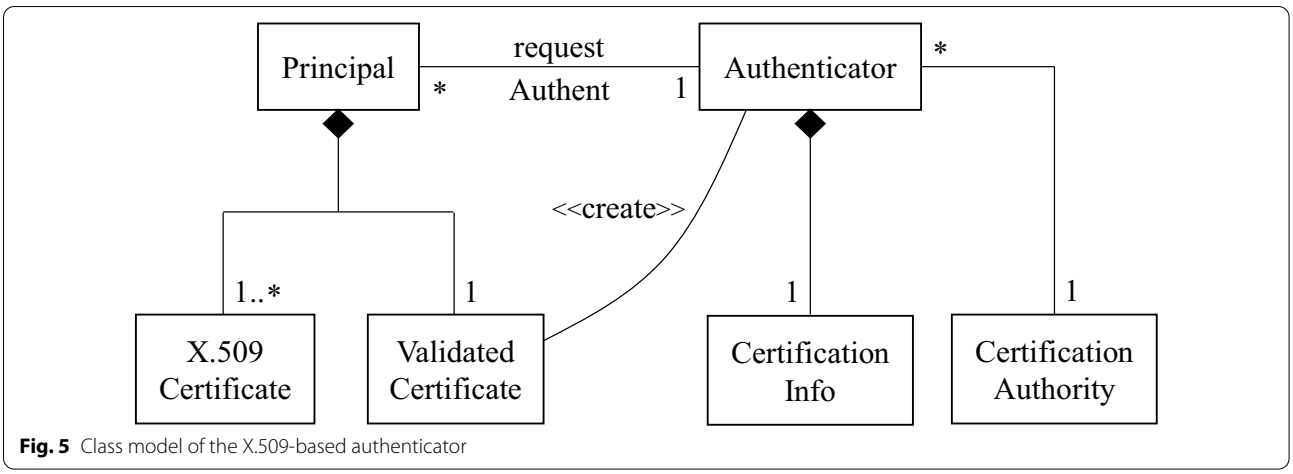

Fig. 5 Class model of the X.509-based authenticator

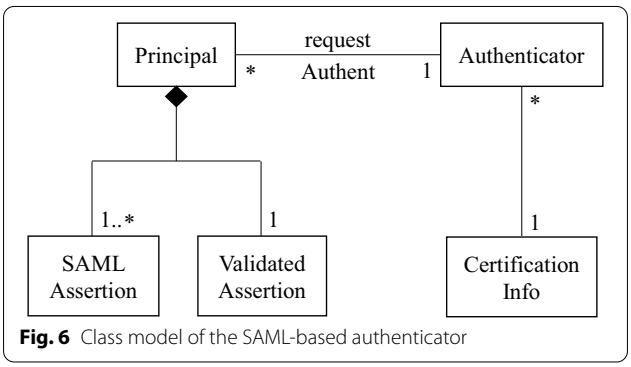

Fig. 6 Class model of the SAML-based authenticator

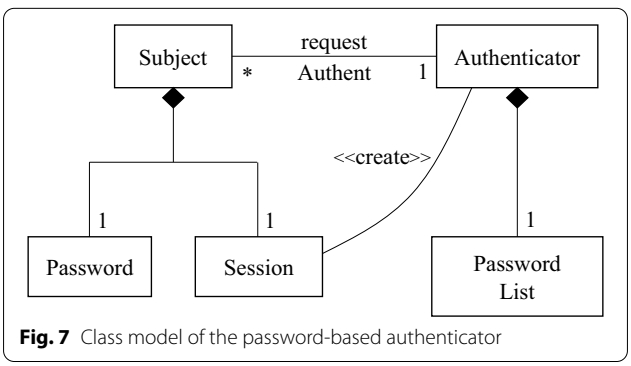

Fig. 7 Class model of the password-based authenticator

(CL) subsumes the context of its descendants: $CL_i \supseteq CL_j$, where i precedes (it is higher) j in the hierarchy. For example, the context of an Abstract Authenticator applies to any domain while the context of a Credential-based Authenticator is valid only for distributed systems, and the context of an X.509 certificate applies only to distributed systems that follow this standard. The threats of the concrete patterns are specific realizations of the abstract pattern's threats using the changed context, or are new threats due to the extra elements in the class diagram (classes or attributes); that is $T_j \supseteq T_i$, where i precedes j in the hierarchy.

The forces in a pattern define constraints on its solution indicating a tension that motivates the need for the pattern. Forces are given names indicating which aspect they constrain. The following forces apply to the possible solution of the Abstract Authenticator:

- *Closed system* If the authentication information presented by the user is not recognized, no access is granted (Saltzer and Schroeder 1975). In an open system, all subjects can have access except those who are blacklisted for some reason. A closed system provides a higher degree of security, but some systems are open because of their objectives.
- *Registration* Users must register their identity and provide identity information to let the system recognize them later.
- *Flexibility* Large systems may have a variety of individuals or systems (users) who require access to the system, as well as a variety of system units with different access restrictions. We must be able to handle this heterogeneity appropriately, or we risk security exposures.
- *Dependability* We need to authenticate users in a reliable and secure way. This means a robust protocol and a high degree of availability. Otherwise, users may deceive the authentication process or enter when the system authenticator is down.
- *Protection of authentication information* Users must not be able to read or modify the authentication information. Otherwise, they can give themselves access to the system, simulate to be somebody else, or disrupt the access of legitimate subjects.

- *Simplicity* The authentication process must be relatively simple; otherwise, the users or administrators may be confused. User errors are annoying to them, administrator errors may lead to security exposures.
- *Reach* Successful authentication only gives access to the system, not to any specific resource of the system. Access to these resources must be controlled using other mechanisms, typically authorization.
- *Tamper freedom* It should be very difficult to falsify the proof of identity presented by the user; otherwise we can have impostors.
- *Cost* There should be tradeoffs between security and cost, higher security can be obtained at a higher cost.
- *Performance (response time)* Authentication should not take a long time or users will be annoyed. However, more secure authentication methods may take a longer time.
- *Frequency* Subjects should not have to authenticate often. Frequent authentications waste time and annoy the users.

Note that there are no implementation aspects in these forces, i.e., they describe security requirements for the solution that complement its conceptual class model. In fact, these forces apply to any system where access should be restricted only to specific subjects. Concrete versions of this pattern would add aspects related to their specific context. For example, a Password-based Authenticator would add (among other forces):

- *Strength* A password must be hard to discover, even for an attacker who has access to the password file and high computational power.
- *Protection of Authentication Information* The password file must not be accessible to the users. Otherwise, they could use powerful computers to discover passwords by trial and error.
- *Validity* There should be convenient ways to revoke or invalidate registered passwords.

The forces of the ASP may appear under more specific forms in a concrete pattern, e.g., in the examples above, protection of authentication information takes specific forms. New forces can be introduced to consider a new context; in this example "Strength" is a new force, specific to passwords (although it can be considered a special case of tamper freedom).

The reverse of what happens for contexts is true about forces and consequences, the forces in concrete patterns include (maybe modified) those of the abstract pattern plus new forces (and their consequences) due to their more specific environments. That is: $F_j \supseteq F_i$, where i precedes j in the hierarchy.

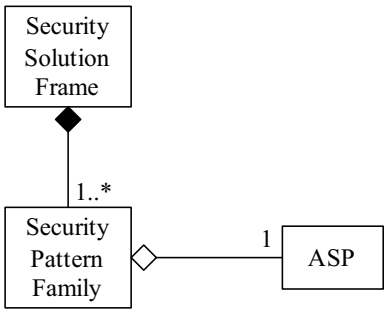

Fig. 8 From SSFs to ASPs

Patterns in general are obtained by abstracting common concepts of implementations found in real systems (best practices); ASPs can be obtained by abstracting the properties of several concrete patterns or directly from the security constraints of several applications. There is no algorithm to produce ASPs, abstraction is a human activity which depends on the experience and ability of the pattern builder. Deciding what is really essential in an ASP is not always clear and judgment is necessary; for example, the first force of the Authenticator (closed system) may be interpreted to be a property (principle) of the system where it is used, and not of the pattern itself.

Relationships between ASPs and security solution frames

Figure 4 shows authenticators related to each other by generalization, e.g. an X.509 certificate pattern "is a" credential pattern. As shown later, there can be directed associations between ASPs that describe peer associations between them. As discussed earlier, patterns can be associated also by aggregation (Rumbaugh et al. 1999), where a pattern is composed of other patterns.

An important use of ASPs is to identify and organize SSFs. As indicated earlier, SSFs are solution structures that encapsulate and organize security patterns; they realize security requirements. SSFs can facilitate the work of designers by collecting together all the relevant patterns to realize some security requirements, guiding the designer from an abstract conceptual level to a concrete implementation-oriented level. SSFs define horizontal and vertical pattern structures. As shown in Fig. 4, vertical structures are hierarchies of patterns specialized going from ASPs to technological implementations. Horizontal structures, *security pattern families* (SPFs), are sets of peer-related patterns that complement each other and define different aspects of a security policy. ASPs act as roots of these hierarchies and can be used to characterize SSFs, where each lower level is a pattern specialized for some specific context (Fig. 8). For example, an

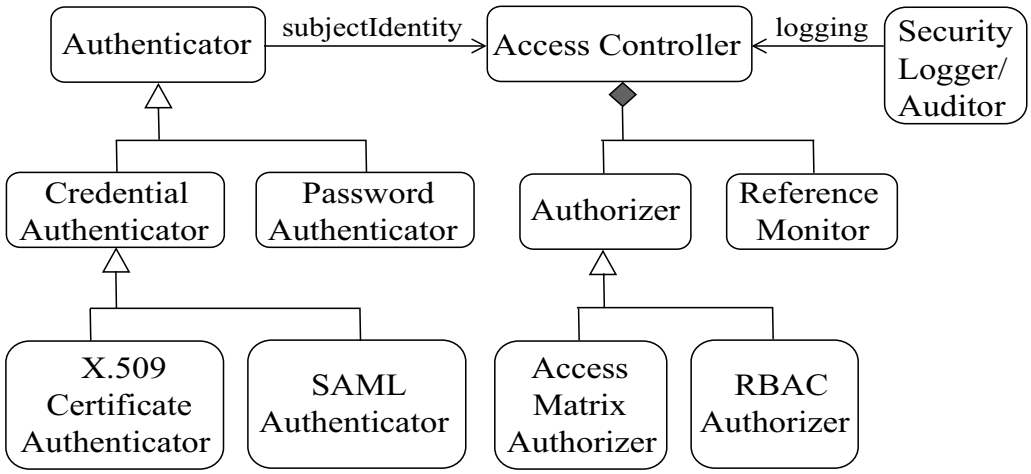

Fig. 9 Combining SSFs

SSF for Authentication includes (among others) a family of Authenticator patterns, which includes Credential-based Authenticator, Password-based Authenticator, and others. If we add an Authorization/Access Control hierarchy (as in Fig. 9), we would have an SSF, although Fig. 4 itself is also an SSF. Figure 9 relates an Authenticator SSF to an Access Controller SSF. The Access Controller includes an Authorizer, which defines authorization rules that may correspond to and Access Matrix or an RBAC model. The Authorizer must be complemented with a Reference Monitor (Fernandez 2013) to enforce the rules.

We can draw an SSF in its own graph or draw separate graphs for each level to correlate patterns from different families. The latter type of diagram is useful when we want to understand a complete system; for example, when building a banking application we can correlate all the security patterns needed to protect accounts (see next section). A given security pattern can belong to more than one family or to more than one SSF. Ref. (Uzunov et al. 2015a) contains complete descriptions of two SSFs: Authorization and Security Information Management. Authorization includes three SPFs: Conceptual Authorization Model, Enforcement Architecture, and Security Process. The Conceptual Authorization Model SPF includes one ASP: Abstract Authorization. The Enforcement Architecture SPF includes the ASP Abstract Authorization Architecture. The Security Information Management SSF contains the Policy Management SPF, which in turn contains the ASP Abstract Policy Manager. Reference (Uzunov and Fernandez 2021) contains SSFs for secure communications.

A security cluster (SC) is a selection of patterns from different SSFs (Fernandez and Yoshioka 2018). Formally: SCa = {SSFi.pa, SSFj.pb, SSFk.pc,...}, where cluster SCa combines patterns where SSF.pi denotes pattern i in an SSF. SCs can be catalogued by defining a start SSF and using it as index. Figure 10 (a group of pattern diagrams representing SSFs) shows the construction of a Security Cluster, SC1. To define SC1 the designer decided to use credentials as authentication artefact, then selected attribute-based access control (ABAC), for securing its communications she chose the Advanced Encryption Standard, a symmetric encryption algorithm, and finally used Distributed Logging. This specific selection was based on the analysis of the expected threats of this

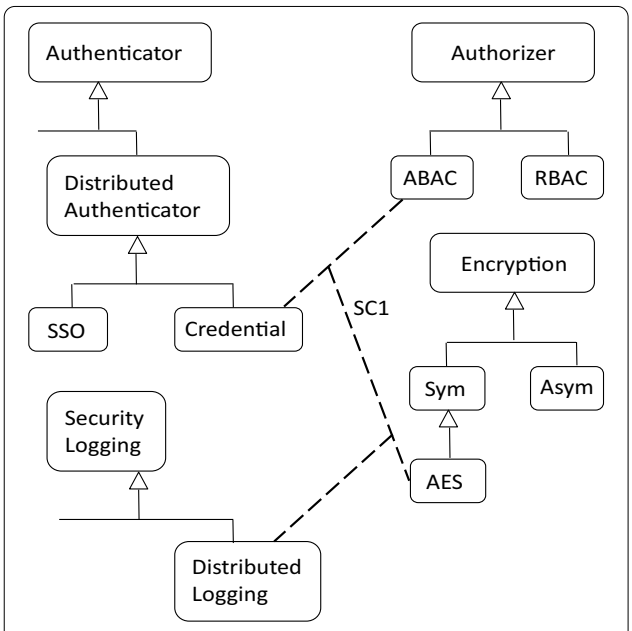

Fig. 10 An SC definition (from 11)

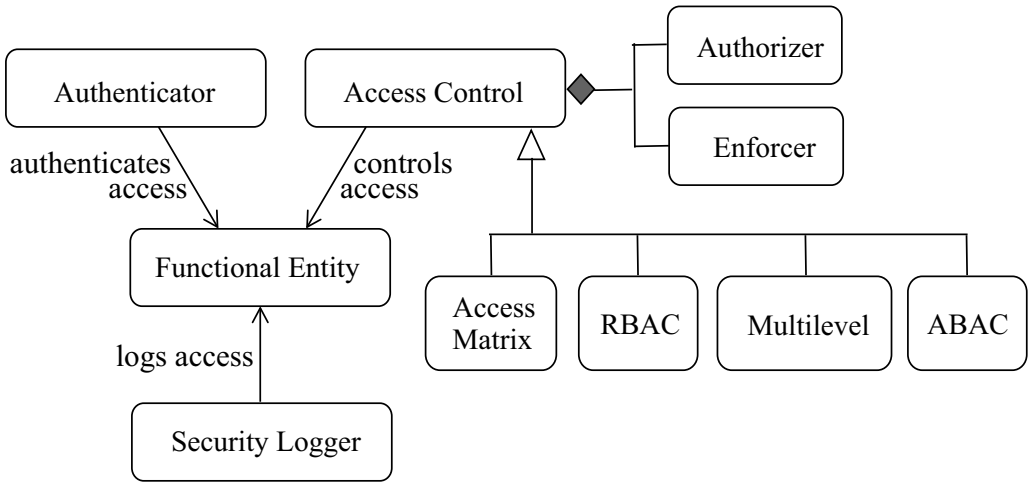

Fig. 11 Entity security services

application, obtained by the method described in "Security patterns and security solution frames" section. As we have shown elsewhere, we can map threats to security patterns that can stop them (Fernandez 2013; Uzunov et al. 2015b). In a catalog, each SC description should include recommended applications or include analysis patterns where it would fit, as shown in Fig. 10.

ASPs in secure conceptual models

We show now how ASPs are useful to secure applications by simplifying the work of the designer, who may not have much experience on security. Figure 11 shows how abstract patterns can provide security controls to a functional entity (Gollmann 2011). Pattern *Functional Entity* represents some functional unit in a conceptual model of an application and its basic security services are described by patterns *Authenticator*, *Access Controller* (showing some common authorization models), and *Security Logger*. These patterns solve the problems described below.

- *Authenticator* (Fernandez 2013; Fernandez et al. 2018). Controls access to the functional entity as a whole unit. We described this pattern in "Abstract security patterns (ASPs)" section.
- *Authorizer* (Fernandez 2013). Describes who is authorized to access specific resources in the functional entioty and how, in an environment in which we have resources whose access needs to be controlled. It indicates for each active entity, which resources a subject can access, what it can do with them, and under what conditions.
- *Enforcer (Reference Monitor, Policy Enforcement Point)* (Fernandez 2013). Enforces authorizations when a process requests access to an object. Done through an abstract process that intercepts all requests for resources from processes and determines if they are authorized by some rule.
- *Access matrix* (Fernandez 2013). Describes authorization rules where subjects are individual users or systems.
- *Role-Based Access Control (RBAC)* (Fernandez 2013). Subjects are assigned rights based on their functions or tasks in an environment in which there is a large number of users or a large variety of resources.
- *Multilevel Security pattern* (Fernandez 2013). Defines how to decide access in an environment with security classifications for subjects and resources.
- *Attribute-based access control (ABAC)* (Priebe et al. 2004). Defines access to resources based on the attributes of the subjects and the properties of the objects.
- *Security Logger* (Fernandez 2013). Logs all security-sensitive actions performed by subjects (who did what to what data and when) and provides controlled access to records for Audit purposes.

It is not necessary to attach these controls to each functional entity; from the threat enumeration process we can determine which services are actually required to stop the threats (Fernandez 2013). Regulations and institution policies may also require additional security mechanisms. In general, we must not add in each entity all possible security mechanisms, which results in systems that:

•Are overly complex, with many redundancies, which bring administrative confusion, a source of possible vulnerabilities.
•Have a high performance overhead, because of redundant checks.
•Are costly, because most security mechanisms are COTS components, and they have to be bought and maintained.

There is also a basic difference between adding design patterns and adding security patterns to an application. Design patterns have no effect on the semantics of the application; adding design patterns is optional and is intended to improve some code aspect such as flexibility, performance, or extensibility. Adding security patterns, on the other hand, can make the application more secure and unless we apply patterns to protect all significant security vulnerabilities the application will not be secure. Security is not based on local transformations as when using design patterns, but requires a global transformation of the whole architecture. By showing the needed security mechanisms and when they are combined with SSFs, ASPs can simplify the job of the designer who has now a guide to decide what security mechanisms are needed according to the possible threats and what specific concrete patterns to use. In this sense, they can effectively complement a secure development methodology (Fernandez 2013; Uzunov et al. 2015b).

Deriving concrete patterns from ASPs

We have written four patterns that can be used to demonstrate the power of the ASP concept by showing the derivation of concrete patterns from some ASPs. One of them (Fernandez et al. 2018) shows derivation of Authenticators in the style of "Security patterns and security solution frames" section; starting from the Authenticator ASP we derive the Credential-based Authenticator. Another (Fernandez et al. 2019), starting from a Network Segmentation ASP derives a pattern for IoT Segmentation, which partitions a network of IoT devices and other entities into subnetworks in order to isolate groups of devices and entities with different security requirements; IoT networks because of their heterogeneity have a large variety of threats different from standard IT threats. Another (Fernandez et al. 2016), describes the derivation of IaaS patterns in a cloud as shown in Fig. 12. The Secure Infrastructure-as-a-Service pattern describes the architecture required for the sharing of distributed virtualized computational resources such as servers, storage, and networks, including a set of security services. The Secure Open IaaS describes the architecture required for the sharing of distributed virtualized computational resources such as servers, storage, and networks, including a set of security services; the implementation of these services is open source and different architectures may have different security services. The Secure Open Stack describes the architecture required for the sharing of distributed virtualized computational resources such as servers, storage, and networks, including a set of security services; the implementation of these services is also open source and different architectures may have different security services (OpenStack defines a standard that contains a set of security services but specific implementations may have additional security services). The Secure Concrete Open Stack corresponds to a generic implementation of the standard. This example shows that from a generalization hierarchy of functional patterns we can derive a corresponding hierarchy of security patterns. Finally, from a Secure Publish/Subscribe ASP we derived an IoT Secure Publish/Subscribe which introduces new defenses to control the new threats present in that context (Fernandez et al. 2020).

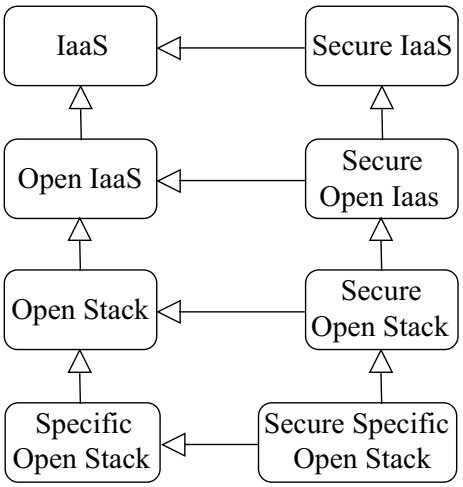

Fig. 12 Deriving secure IaaS patterns

Formalization of ASPs

Many pattern authors include UML models, that are semi-formal models, in the solution section of a pattern. However, patterns are suggestions and this is not a requirement, they are given as examples, not strict guidelines that the designer must follow. Requiring to follow the formalization of a pattern solution would restrict the freedom of the designer when using the pattern in her applications; a formal description may constrain possible implementations or make

additional assumptions that may modify the intended meaning of the pattern. Also, many developers do not have enough background to understand formal models; in fact, some authors avoid even UML models to make their patterns more usable. However, there are several drawbacks to the informal representation of patterns (Warmer and Kleppe 2003; Dong et al. 2007): informal specifications may be ambiguous, and pattern solutions may not be able to be expressed precisely in an informal language; a pattern may have some particular properties that characterize it, and the application of a pattern should maintain these properties. More important, formal specifications allow the use of automated tools to check some properties; for example, object constraint language (OCL) (Warmer and Kleppe 2003), allows querying the model to find out new information. Formal specifications of patterns can be used to discover new patterns or detect the use of patterns in large software systems. Even if a pattern requires tailoring, starting from a precise description facilitates its selection, application, and implementation.

A common option is to add formal constraints to the UML diagram of its solution. As indicated, OCL can be used to define constraints on the data and to query this data. For example, the following could be a postcondition that describes that a proof of authentication is created if the subject ID and its authentication information are found in the Authentication Information class:

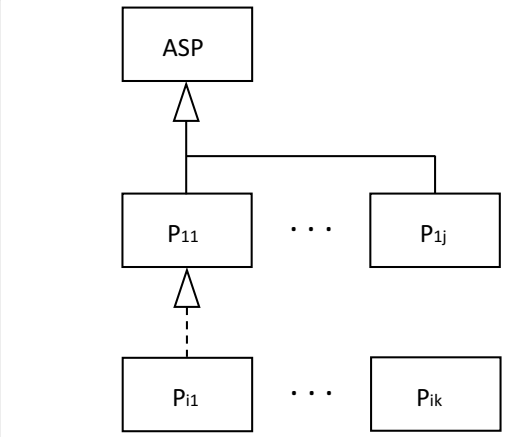

Fig. 13 Metamodel for an ASP and its derived patterns

Another type of formalization is the use of ontologies; an ontology for security patterns is shown in Pereira-Vale and Fernandez (2019). That ontology applies to ASPs, since they are security patterns. In that model, OWL allows queries like: "Obtain the list of concerns of the security patterns used in the lifecycle design stage for operating systems contexts."

Some researchers have formalized structural or syntactic properties of design patterns. Ref. (Le Guennec et al. 2000), proposed a set of modifications to the UML 1.3 meta-model to make it possible to model design patterns and represent their occurrences in UML, with the objective of facilitating automatic processing of applications using patterns. Ref. (Hamid et al. 2016) provided a formal representation and its associated validation mechanisms for the verification of security properties of security patterns. Ref. (Dong et al. 2007) presented a formal specification for design patterns based on first-order logic, temporal logic of action, and Prolog.

```
postcondition
context ProofOfAuthentication
        self.registeredUsers → collect (AuthenticationInformation)
if registeredUsers includes (subject and proof): true
```

Table 1 Summary of ASPs structural formalization in OCL

The class models of the concrete patterns derived from an ASP must include all the classes of the ASP from which they were derived as well as classes that handle new aspects. If Ci = set of classes in ASPi, Cci = set of classes in a concrete pattern derived from ASPi, and Cnew = new classes in concrete pattern Cci, we have: Cci = Ci ∪ Cnew. In OCL: context Cci Cci::=Ci->union(Cnew)
The context of a pattern subsumes the context of its descendants: $CL_i \supseteq CL_j$, where i precedes (it is higher) j in the hierarchy. A CL defines a domain of application (it includes a set of contextual attributes). However, the pattern context is not shown in the class model and OCL expressions are not applicable
The threats of the concrete patterns are specific realizations of the ASP's threats using the changed context, or are new threats due to the extra elements in the class diagram (classes or attributes); that is, $T_j \supseteq T_i$, where i precedes j in the hierarchy and Ti is a list of the threats of pattern Pi. Again, OCL constraints are not applicable because the threats are not shown in the class model of the pattern
The forces in concrete patterns include (maybe modified) those of the abstract pattern plus new forces due to their more specific environments. If Fi is the list of forces of pattern Pi we have that: $F_j \supseteq F_i$, where i precedes j in the hierarchy. This relationship is also valid for the consequences of ASP-based hierarchies.; that is, if CSi is the list of consequences of pattern Pi, we have: $CS_j \supseteq CS_i$ if Pi precedes Pj in the hierarchy. OCL expressions are not applicable
The related patterns in the derived patterns, RDj, include the related patterns of the ASP and those of the patterns above them in the hierarchy; that is $RD_j \supseteq RD_i$
An invariant I in an ASP must be propagated to all its derived patterns, adjusting the variable names in the derived pattern classes. Each ASP has its own invariants (see example above)

To formalize the structure of ASPs and their derived patterns we can use a semiformal approach, combining a metamodel with formal annotations (Washizaki et al. 2009b). Figure 13 shows a metamodel for ASPs and their derived patterns (subpatterns). We first need to define what the UML operation of generalization means for patterns. The UML security model in the pattern solution is a set of classes and associations; we can then define constraints among these classes. A subpattern must preserve those constraints and may possibly add more. Considering Fig. 3, where we have a concrete pattern derived from the pattern of Fig. 1, we can see that the derived pattern includes all the classes of Fig. 1, renamed to indicate their different context. If we go down one more step in the class hierarchy, we get Fig. 5 where there is a new class. Note that the hierarchies starting from ASPs are generalization trees.

We summarize the formal aspects of ASPs in Table 1, where the following definitions apply: C is the set of classes of a class model, CL is a set of contextual attributes, T is a set of threats, F is a set of forces, CSi is a set of consequences, and RDi is the set of related patterns. Then, we may define a pattern Pi as follows

$$P_i = (CL_i, T_i, F_i, C_i, CS_i, RD_i)$$

Pi is an abstract or concrete pattern that provides a solution composed by classes Ci in order to mitigate the corresponding security problem consisting of threats Ti, when considering specific forces Fi in the context of CLi, resulting in consequences CSi. Alternative or complementary solutions are defined by RDi.

Also, a pattern is more than its solution; the texts in its sections are very important guidelines for its correct application. Ref. (Maña et al. 2013) did a formalization of complete security patterns (not just their solutions), with the intention of enabling their automatic handling. We try below to formalize all the sections of the patterns; a set of assertions describe the sections of each pattern. A formal language like OCL, Z, or Alloy could be used to describe each section more precisely, but we only use here standard set notation, where a tuple is indicated as (...), a set as {}, optional elements as [], ID is a unique identifier assigned in a catalog for each pattern, SUD refers to System Under Development. We first show the assertions for ASPs and then assertions for their derived patterns. A derived pattern (DPattern) shows only its changes with respect to the ASP.

ASPReference= (Name, ID, Version, ShortDescription)

ASPIntent= (Classification, [Preconditions], [{SecurityProperty}])

SecurityProperty= (Name, [Restrictions], Metrics, Forces, [AssetList])

AssetList= {AssetReference}

ASPContext= (ContextDescription, [SUDInterfaceElementList], [Assumptions])

SUDInterfaceElementList={SUDInterfaceElementReference}

ASPForces= {ForceElementList}

ASPThreats= {ThreatList}

ASPSolution= (Description, Model, [Assets], [{ContextElements}])

Assets={Asset}

Asset= (Name,Description,ID)

ContextElements= (Name, Description, ID)

ASPConsequences= ([Postconditions], [Advantages], [Disadvantages], [Scope])

Scope={ScopeSpecification}

ASPRelatedPatterns= {RelatedPattern}

RelatedPattern= (Relation, [RelatedPatternReference])

Relation="uses"|"requires"|"includes"|"extends"|"excludes"|"overlaps"|"complements"|"alternative"

RelatedPatternReference= (PatternID, [version], [validation])

DPatternContext= Subset (ASPContext)

DPatternForces= ASPForces U newForces

DPatternSolution= ASPSolution U newSolution

DPatternThreats= ASPThreats U newThreats

DPatternConsequences= ASPConsequences U newConsequences

DPatternRelatedPatterns= ASPRelatedPatterns U newRelatedPatterns

Evaluation of effectiveness

ASPs are abstract entities that can be implemented in many ways; this means they cannot be evaluated with respect to security or performance through experimentation or testing. Evaluating a specific implementation would not say anything about the model if its security failed. As indicated above, patterns are suggestions for designers, they do not require to be built exactly as described; a designer can cut or add classes, rename classes and attributes, or split classes, as far as their semantics are respected. The evaluation of ASPs must be based on how well they represent the relevant concepts of the systems they describe, how well they handle abstract threats, how complete they are, how precise they are, how they can be applied to the design or evaluation of systems, and how useful they are for other relevant functions.

From the cases of ""ASP-based hierarchies"– "Deriving concrete patterns from ASPs" sections, and our experience, we have found the following uses of ASPs:

- *ASPs can be combined with other ASPs to cover all the security concerns of an application*, including all their architectural levels if we use SSFs. In this way, they can be applied during the stages of a secure systems development methodology such as ASE (Uzunov et al. 2015b) or similar. ASPs can also be combined with patterns describing security principles or good general design principles. For example, the Abstract Authorizer can be combined with Need-to-Know (Fernandez et al. 2011) to assign rights according to the needs of the subjects; Single Point of Access (Yoder and Barcalow 2000) can be combined with Firewall (Saltzer and Schroeder 1975), to restrict the placement of firewalls in a network.
- *Can be used to check for security coverage in a complete design* One of the problems with protecting complex systems is that it is hard for the designers to see if all the high-level security threats have been covered with the applied defenses. This is much easier when we work at the application level, we can enumerate all threats and find the corresponding security patterns to defend against them; using SSFs we can propagate the defenses to the lower levels. This explicit handling of threats also allows evaluation of the security degree reached by the design (Villagran-Velasco et al. 2020), a security measure is the number of threats covered by the security patterns present in the design.
- *Can guide the search for new patterns* (pattern mining) An abstract pattern defines a range of patterns and one can see if corresponding patterns exist at all the lower levels, including different environments, e.g., web services or cloud computing. This was illustrated in "Deriving concrete patterns from ASPs" section.
- *Can serve as abstract prototypes for existing concrete patterns* and to verify they are complete with respect to functions and threat coverage. Starting from an abstract pattern it is easy to see what security constraints (forces, threats) must at least be applied at a specific architectural level. For example, from an ASP for VPNs we can derive TLS and IPSec VPNs (Fernandez 2013). Other examples were shown in "Relationships between ASPs and security solution frames" section. The formalization section can help building those patterns.
- *Can serve as ways to connect and relate different families of patterns.* For example, a Communication Channel can use Intrusion Detection (see "ASP-based hierarchies" and "Relationships between ASPs and security solution frames" sections). If we build a fairly extensive catalog of Security Clusters we can simplify the work of developers.
- *We can build Domain models and/or Security Reference Architectures using ASPs*. As indicated, DMs and SRAs are also abstract architectures.
- ASPs are *a good basis to separate and classify distinct patterns* (Washizaki et al. 2009a). Pattern catalogs usually include several varieties of the same security pattern, perhaps with different names; ASPs can help recognize similar patterns.
- IoT patterns are often variations of more general patterns. When classifying IoT patterns, *ASPs help to establish the difference between ASPs and IoT patterns,* and we can concentrate in finding the changes needed in the pattern description due to the specific environment without having to redefine its core structure, as we have shown in some examples (Fernandez et al. 2020). This helps also with the problem of the variety of IoT pattern descriptions found in the literature (Washizaki et al. 2021).
- Several methodologies that claim to apply "security by design", e.g., Microsoft Security Development Lifecycle (Howard 2006), start from the system architecture, ignoring threats to applications. *ASPs emphasize the need to start earlier to consider the needs of applications;* this will result in more secure systems because methods that start from lower levels ignore higher-level threats.
- The forces in ASPs can become security policies for the derived patterns, thus contributing to make them more secure.
- ASPs can simplify the job of the designer who has now a guide to decide what security mechanisms are needed according to the possible threats and what

specific concrete patterns to use. In this sense, they can effectively complement a secure development methodology (Fernandez 2013; Uzunov et al. 2015b).
- An advantage of standard patterns, and also of ASPs, is that they are seamless with respect to lifecycle approaches such as the rational unified process (RUP) (Rumbaugh et al. 1999), and use similar notation and concepts, which facilitate their use in practice.
- Security usability patterns can complement ASPs by defining interface requirements to make patterns clearer and easier to use; ASPs can be combined with interface patterns (Brambilla et al. 2017).
- We have used ASPs in our own research. In (Fernandez et al. 2021, Washizaki et al. 2021), ASPs provided a convenient way to classify IoT patterns; in (Uzunov et al. 2015b), they are used to embody a subset of the early security requirements in a secure software development methodology; in (Villagran-Velasco et al. 2020) they were used to evaluate the degree of security reached by a secure software methodology using patterns.

Related work and discussion

As indicated earlier, the concept of abstract pattern is present in the original patterns of the GOF (Gamma et al. 1994), but they did not develop their possibilities because they were not concerned with the analysis stage, they were trying to improve code.

Other varieties of security patterns intended to emphasize abstract properties include:

- Jackson's *Problem Frames* (Jackson 2001), have been used to define patterns for security requirements (Hatebur et al. 2007).
- Mouratidis uses Secure Tropos, an approach to support multiple views of security, including organizational and external aspects (Mouratidis et al. 2006).

The approaches based on problem frames have in common with ASPs that they emphasize the core security requirements of the system. However, their patterns use a totally different style, they do not follow the standard pattern structure and use different concepts and notation. As indicated, an important advantage of standard patterns and also of ASPs is that they are seamless with respect to lifecycle approaches such as the rational unified process (RUP) (Rumbaugh et al. 1999), and use similar notation and concepts, which facilitate their use in practice.

The patterns in (Moral-García et al. 2014) describe enterprise activities and their security constraints and thus they can also express application constraints. ASPs are more detailed than those patterns and are very close in style to standard patterns as those in (Fernandez 2013, Gamma et al. 1994); after defining them in the analysis stage their transition to design is straightforward: the design stage just needs to refine them and express them in terms of software artifacts, something not easy to do with patterns using different styles.

Conclusions and future work

We have elaborated and characterized the concept of ASP, introduced by us in earlier work (Fernandez et al. 2008, 2014). We have shown their possible uses to prove that they have several potential advantages, including providing insight into the nature of security patterns, helping define the early stages of a methodology to build secure systems, for pattern mining, and to build SRAs, among others (Fernandez 2015). To realize their advantages, we need a good catalog of ASPs that can be used by designers to define secure conceptual models that address crosscutting concerns. We already have a good number of ASPs, but we need to collect them in a specialized catalog of SSFs organized according to ASPs; this catalog would be useful to help designers satisfy security requirements. This is an important future work.

As indicated earlier, there is no automatic way to build ASPs. It takes experience and abstraction ability to build them. Pattern builders build catalogs and designers use the catalogs to build systems. We have shown two formalizations of ASPs, but specific applications and extensions of this formalization are left for future work. Combining ASPs with SSFs and with a systematic methodology such as ASE (Uzunov et al. 2015b) we believe we have a powerful tool to build secure applications. We have started exploring their use to build IoT applications (Fernandez et al. 2021).

Acknowledgements

We thank the National Institute of Informatics (NII) of Japan for supporting the visits of E. B. Fernandez and J. Yoder to Tokyo. The CIbSE 2014 and the Cybersecurity referees provided useful comments that helped improve this paper.

Authors' contributions

EF proposed the main idea, wrote the paper, and supervised the structure of the paper. NY provided ideas, made corrections, and organized discussion meetings. HW provided ideas, references, and helped with the formalization. JY proofread the paper. All the authors participated in discussions to define and refine the results of the paper. All the authors have read and approved of this content.

Funding

This work received no external funding, but the National Institute of Informatics of Japan funded the trip of the first and fourth authors to Tokyo to participate in meetings where the idea of this paper was developed.

Availability of data and materials

Not applicable.

Declarations

Competing interests
The authors declare no competing interests.

Author details
[1]Department of Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL, USA. [2]GRACE Center, National Institute of Informatics, Tokyo, Japan. [3]Waseda University, Tokyo, Japan. [4]The Refactory, Inc, Urbana, IL, USA.

Received: 22 July 2021 Accepted: 6 January 2022
Published online: 01 April 2022

References
Avgeriou P (2003) Describing, instantiating and evaluating a reference architecture: a case study. Enterp Archit J 342:1–24
Blakeley B, Heath C (2004) Members of the open group security forum: technical guide: security design patterns. The Open Group, London http://www.opengroup.org/bookstore/catalog/g031.htm.
Brambilla M et al (2017) "Model-driven development of user interfaces for IoT systems via domain-specific components and patterns. J Internet Serv Appl 8(1):1–21
Buschmann F, Meunier R, Rohnert H, Sommerland P, Stal M (1996) Pattern- oriented software architecture. Wiley, New York
Dong J, Alencar P, Cowan D (2007) Formal specification and verification of design patterns, chapter 5. In: Taibi T (ed.) Design pattern formalization techniques. IGI Publishing, pp 94–108
Fernandez EB (2013) Security patterns in practice: building secure architectures using software patterns. Wiley series on software design patterns. Wiley, New York
Fernandez EB, Yoshioka N (2018) Using a variety of patterns in a secure software development methodology. In: Proceedings 25th Asia-Pacific software engineering conference, Nara, Japan
Fernandez EB, Washizaki H, Yoshioka N (2008) Abstract security patterns. In: Position paper in Proceedings of the 2nd workshop on software patterns and quality (SPAQu'08), in conjunction with the 15th conference on pattern languages of programs (PLoP 2008), October 18–20, Nashville, TN
Fernandez EB, Mujica S, Valenzuela f (2011) Two security patterns: least privilege and security logger/auditor. In: Proceedings of Asian PLoP. http://patterns-wg.fuka.info.waseda.ac.jp/asianplop/proceedings2011/asianplop2011_submission_7.pdf
Fernandez EB, Yoshioka N, Washizaki H, Yoder J (2014) Abstract security patterns for requirements specification and analysis of secure systems. In: Proceedings of the WER 2014 conference, a track of the 17th Ibero-American conference on software engineering (CIbSE 2014), Pucon, Chile
Fernandez EB, Monge R, Hashizume K (2015) Building a security reference architecture for cloud systems. Requir Eng. https://doi.org/10.1007/s00766-014-0218-7
Fernandez EB, Washizaki H, Yoshioka N (2016) Patterns for secure cloud IaaS. In: 5th Asian conference on pattern languages of programs (AsianPLoP)
Fernandez EB, Yoshioka N, Washizaki H (2018) An abstract security pattern for Authentication and a derived concrete pattern, the Credential-based Authentication. In: Asian pattern languages of programs conference (AsianPLoP)
Fernandez EB, Yoshioka N, Washizaki H (2019) Abstract and IoT security patterns for network segmentation. In: Proceedings of the 8th Asian conference on pattern languages of programs (Asian PLoP)
Fernandez EB, Yoshioka N, Washizaki H (2020) Secure distributed publish/subscribe (P/S) pattern for IoT. AsianPLoP
Fernandez EB, Washizaki H, Yoshioka N, Okubo T (2021) The design of secure IoT applications using patterns: State of the art and directions for research. Internet Things 15:100408. https://doi.org/10.1016/j.iot.2021.100408
Fowler M (1997) Analysis patterns—reusable object models. Addison-Wesley, Reading
Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns—elements of reusable object-oriented software. Addison-Wesley, Reading
Gollmann D (2011) Computer security, 3rd edn. Wiley, New York
Hamid B, Gürgens S, Fuchs A (2016) Security patterns modeling and formalization for pattern-based development of secure software systems. Innov Syst Softw Eng 12:109–140. https://doi.org/10.1007/s11334-015-0259-1
Hatebur D, Heisel M, Schmidt H (2007) A pattern system for security requirements engineering. In: Proceedings of ARES, pp 356–365
Howard M (2006) The security development lifecycle: SDL: a process for developing demonstrably more secure software, 1st edn. Microsoft Press, Redmond
Jackson M (2001) Problem frames: analyzing & structuring software development problems. Addison-Wesley, Reading
Le Guennec A, Sunyé G, Jézéquel J-M (2000) Precise modeling of design patterns. In: International conference on the unified modeling language, pp 482–496
Maña A, Fernandez EB, Ruiz J, Rudolph C (2013) Towards computer-based security patterns. In: 20th Conference on pattern languages of programs (PLoP)
Moral-García S, Moral-Rubio S, Rosado DG, Fernández EB, Fernández-Medina E (2014) Enterprise security pattern: a new type of security pattern. Secur Commun Netw (wiley) 7(11):1670–1690. https://doi.org/10.1002/sec.863
Morrison P, Fernandez EB (2006) The credential pattern. In: Proceedings of the conference on pattern languages of programs, PLoP 2006, Portland, OR. http://hillside.net/plop/2006/
Mouratidis H, Weiss M, Georgini P (2006) Modelling secure systems using an agent-oriented approach and security patterns. Int J Soft Eng Knowl Eng 16(3):471–498
Pereira-Vale A, Fernandez EB (2019) An ontology for security patterns. In: 38th International conference of the chilean computer science society (SCCC 2019), Concepción—Chile. November 4–8
Polya G (1957) How to solve it, 2nd edn. Doubleday Anchor Books, New York
Priebe T, Fernandez EB, Mehlau JI, Pernul G (2004) A pattern system for access control. In: Research directions in data and applications security XVIII, Farkas C, Samarati P (Eds.) Proceedings of the 18th annual IFIP WG 11.3 working conference on da-ta and applications security, Sitges, Spain, July 25–28
Rumbaugh J, Jacobson I, Booch G (1999) The unified modeling language reference manual. Addison-Wesley, Boston
Saltzer J, Schroeder M (1975) The protection of information in computer systems. Proc IEEE 63(9):1278–1308
Schumacher M, Fernandez EB, Hybertson D, Buschmann F, Sommerlad P (2006) Security patterns: integrating security and systems engineering. Wiley, New York
Song Z, Li Z, Dou W (2003) Different approaches for the formal definition of authentication property. In: 9th Asia-Pacific conference on communications
Steel C, Nagappan R, Lai R (2005) Core security patterns: best strategies for J2EE, web services, and identity management. Prentice Hall, Upper Saddle River
Taylor RN, Medvidovic N, Dashofy N (2010) Software architecture: foundation, theory, and practice. Wiley, New York
Uzunov AV, Fernandez EB (2021) Cryptography-based security patterns and security solution frames for networked and distributed systems. Submitted for publication (available from the authors)
Uzunov A, Fernandez EB, Falkner K (2015a) Security solution frames and security patterns for authorization in distributed, collaborative systems. Comput Secur 55:193–234. https://doi.org/10.1016/j.cose.2015.08.003
Uzunov A, Fernandez EB, Falkner K (2015b) ASE: a comprehensive pattern-driven security methodology for distributed systems. J Comput Stand Interfaces 41:112–137. https://doi.org/10.1016/j.csi.2015.02
van Heesch U, Hezavehi SM, Avgeriou P (2011) Combining architectural patterns and software technologies in one design language. In: Proceedings of the 16th European conference on pattern languages of programs (EuroPLoP)
Villagran-Velasco O, Fernandez EB, Ortega-Arjona J (2020) Refining the evaluation of the degree of security of a system built using security patterns. In: Proceedings 15th international conference on availability, reliability and security (ARES 2020), Dublin, Ireland
Warmer J, Kleppe A (2003) The object constraint language, 2nd edn. Addison-Wesley, Reading
Washizaki H, Fernandez EB, Maruyama K, Kubo A, Yoshioka N (2009a) Improving the classification of security patterns. In: Proceedings 20th

international workshop on database and expert systems application, pp 165–170
Washizaki H, Fernandez EB, Maruyama K, Kubo A, Yoshioka N (2009b) Improving the classification of security patterns. In: 20th International workshop on database and expert systems application, pp 165–170
Washizaki H, Hazeyama A, Okubo T, Kanuka H, Ogata S, Yoshioka N (2021) Analysis of IoT pattern descriptions. In: SERP4IoT
Yoder J, Barcalow J (2000) Architectural patterns for enabling application security. In: Harrison N, Foote B, Rohnert H (eds.) Proceedings PLOP'97, Also, Chapter 15 in pattern languages of program design, vol 4. Addison-Wesley

Publisher's Note