Abstract security patterns for requirements specification and analysis of secure systems

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

[1]Dept. of Computer Science and Engineering, Florida Atlantic University, USA
ed@cse.fau.edu
[2]GRACE Center, National Institute of Informatics, Tokyo, Japan nobukazu@nii.ac.jp
[3]Waseda University, Tokyo, Japan washizaki@waseda.jp
[4]The Refactory, Inc, Urbana, IL, USA joe@joeyoder.com

Abstract. During the requirements and analysis stages of software development, the primary goal is to define precise requirements rather than being concerned with the details of software realizations. Security is a semantic aspect of applications and their constraints on the application should de described at this moment. From a security point of view we only want to indicate which specific security controls are needed, rather than getting involved with low-level design and implementation details. Therefore, at these stages, it is useful to have a set of patterns which define abstract security mechanisms. These patterns should specify only the fundamental characteristics of the security mechanism or service, not specific software aspects. We present the concept of Abstract Security Pattern (ASP), which describes a conceptual security mechanism that realizes one or more security policies able to handle a threat or comply with a security-related regulation or institutional policy. We present a detailed example of an ASP. We relate ASPs to each other using pattern diagrams as well as to Security Solution Frames and tactics. Finally, we discuss their value for defining security requirements and for building secure systems.

1 Introduction

When solving a problem, it can be useful to start from an abstract, conceptual solution, before we get involved with implementation details, which may just confuse us and because of the possibility of solving more than one problem in the same way. Building a software application is solving a problem. In the requirements and analysis stages of software development we are trying to make the problem precise, we are not concerned with software aspects. Security is a quality aspect that constrains the semantic behavior of applications (by indicating restrictions), so the requirements stage is the development stage to address it, but we only want to indicate which specific security controls are needed, not their implementation. For example, in bank applications we only want to specify the semantic aspects of accounts, customers, and transactions with their corresponding restrictions. In the case of the bank, we need to specify that customers are the only ones who can perform transactions on their own ac-

counts and similar type of constraints. At this stage, it appears useful to provide a set of patterns (or other artifacts) which define abstract security mechanisms that can describe these restrictions. These patterns should specify only the fundamental characteristics of the mechanism or service, not specific software aspects. We introduce[1] here the idea of *Abstract Security Pattern (ASP*), which describes a conceptual security mechanism that realizes one or more security policies able to handle (stop or mitigate) a threat or comply with a security-related regulation or institutional policy. Most works on security patterns [1, 17, 18] emphasize concrete patterns which solve security problems at given architectural levels or units, e.g., Secure Virtual Address Space (VAS) in operating systems [7]. In fact, we have not seen any work on patterns where this abstraction level is explicitly considered. While this separation is implied in the original patterns of the GOF [9], they did not develop their possibilities. We develop here the concept of ASP based on the definition above and we show some examples of them. The common context of all Abstract Security Patterns is the problem space of the corresponding applications or domain models for some knowledge areas. We can relate ASPs to each other using pattern diagrams as well as to Security Solution Frames and tactics. Security Solution Frames (SSFs) are sets of patterns (vertically and horizontally related) that correspond to a specific concern of a solution, e.g. authorization [20].

Some of the ASPs correspond to basic security mechanisms, e.g., Access control (Authorization and Reference Monitor), Security Logger/Auditor, and Authenticator. Others specify more detailed aspects, e.g. Access Control/Authorization models include the Access Matrix, Role-Based Access Control (RBAC), and Multilevel models [17]. Starting from ASPs, along the lifecycle of a complete application we apply a hierarchy of patterns going from abstract security patterns to platform-oriented versions of these patterns and their code realizations.

ASPs should not be confused with patterns that describe basic principles of good security design, e.g. Single-Point-of-Access [24]. ASPs correspond to application concepts, to be realized by computational mechanisms, they do not describe principles. However, as we show here, they can be used effectively in conjunction with patterns that describe principles. In the requirements stage we can specify what security controls we need in each use case and in the conceptual model of the analysis stage we add the corresponding ASPs.

In summary, our contributions include:

- Development of the concept of ASP in detail by means of a complete example.
- Description of the relationships of ASPs to other ASPs and to SSFs.
- Showing the value of ASPs in the requirements and analysis stages by enumerating possible uses.

Section 2 presents some background, while Section 3 discusses the nature of ASPs and presents examples of them. Section 4 relates ASPs to SSFs, while Section 5

[1] We actually introduced the idea in a two-page paper [6] but did not develop its properties.

shows how these patterns can be used. Section 6 discusses related work. The paper ends with some conclusions in Section 7.

2 Security patterns and Security Solution Frames

A security pattern is a solution to the problem of controlling (stopping or mitigating) a set of specific threats through some security mechanism or to implement some security policy or regulation, defined in a given context [7, 17]. This solution resolves a set of forces which constrain it and define guidelines for the solution, e.g. "the solution must be transparent to the users". The solution is usually expressed using UML class, sequence, state, and activity diagrams (although we usually don't need all these models). A set of consequences indicate how well the forces were satisfied by the solution; in particular, how well the attacks were handled or a regulation was satisfied. An implementation section provides hints on how to use the pattern in an application, indicating what steps are needed and possible realizations. A section on related patterns indicates other patterns that complement the pattern or that provide alternative solutions.

Security patterns can be considered architectural patterns because they usually describe global software architecture concepts, e.g., do we need authentication between two units in a distributed system? They can also be seen as design patterns because security can sometimes be considered an aspect of a software subsystem. Abstract security patterns are in effect a variety of analysis patterns [8]. An analysis pattern describes a semantic aspect of an application, e.g., the characteristics of accounts [5]. For example, the Security Logger in [18] emphasizes the implementation aspects of this pattern, so this is a design pattern. The patterns in [7] are either ASPs or architecture patterns. The Security Logger/Auditor in [7] is an ASP because it emphasizes the fundamental functions of this pattern and not its implementation.

Security Solution Frames (SSFs) are sets of patterns (vertically and horizontally related) that correspond to a specific aspect of a solution [20]. The idea is to group together all the patterns that consider a type of solution; a Secure Channel SSF would collect patterns that are used to build secure channels such as Symmetric Cryptography, Asymmetric Cryptography, Digital Signature, and similar. Different levels of abstraction define a vertical structuring while different concerns define a horizontal association.

3 Abstract Security Patterns (ASPs)

An ASP is a security pattern that describes a conceptual semantic restriction in a domain which can be a defense to a threat or follow a regulation, with no implementation aspects. An ASP describes the essential functions that must be present to handle a threat or regulation in an implementation-independent way. For example this is the Intent section of an Authenticator pattern as described in [7] (slightly improved): "When a user or system (subject) identifies itself to the system, how do we verify that

the subject intending to access the system is who it says it is? Present some information that is recognized by the system as identifying this subject."

Authentication handles the threat that an intruder could enter a system and access information he should not see. It is clear that there are many ways to perform this authentication, that go from manual identification, as done in voting places, to purely automatic ways, as when accessing a web site. Authentication as an abstract function requires a basic sequence of activities:

1. The subject presents a request to enter a system indicating its identity
2. The system requires some proof of identity
3. The subject provides such proof
4. The system grants the subject entrance to the system and may provide a proof of authentication for further use

Concrete realizations of this sequence implement these steps in different ways but they all must perform these activities. Figure 1 shows the class model of the Abstract Authenticator, obtained from the realization of the activities above (as it would be done for any object-oriented design). In this model class Subject indicates the active entity that can ask for access to the system by issuing a request to class Authenticator. This class consults class Authentication Information to decide if the subject is legitimate. Class Authentication Information includes a set of whatever information is needed to authenticate users, e.g. a list of passwords, a set of fingerprints, or similar. Dynamic models describe the use cases "Register a subject" and "Request access". The class models of the concrete patterns derived from an ASP must include all the classes of the ASP from which they were derived plus classes needed to handle new aspects required by the specific environment. For the same reason, there may be new or modified attributes and operations in the classes derived from the ASP.

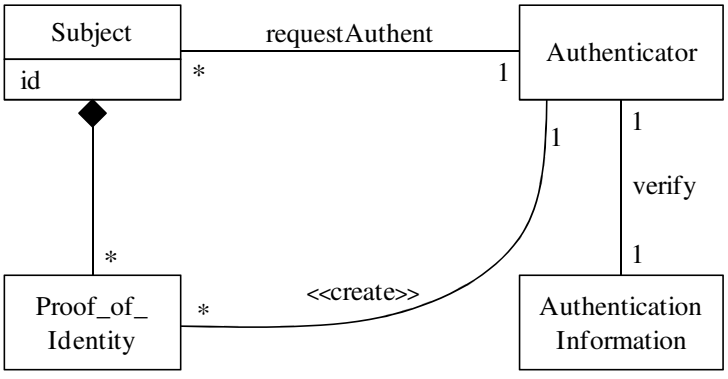

Fig. 1. Class diagram of the Abstract Authenticator pattern (from [7])

Figure 2 shows the class model of the Credential pattern. The Credential pattern includes the complete Abstract Authenticator plus several other classes that define its

environment, the Principal corresponds to the Subject, the Certification Authority creates credentials, and the Credential includes a set of Attributes. The verification of the validity of the Credential is performed differently that in the corresponding ASP. In general, if C_i= set of classes in the ASP_i, C_{ci}= set of classes in a concrete pattern derived from ASP_i, and C_{new}= new classes in concrete pattern C_{ci}, we have: $C_{ci}= C_i \cup C_{new}$

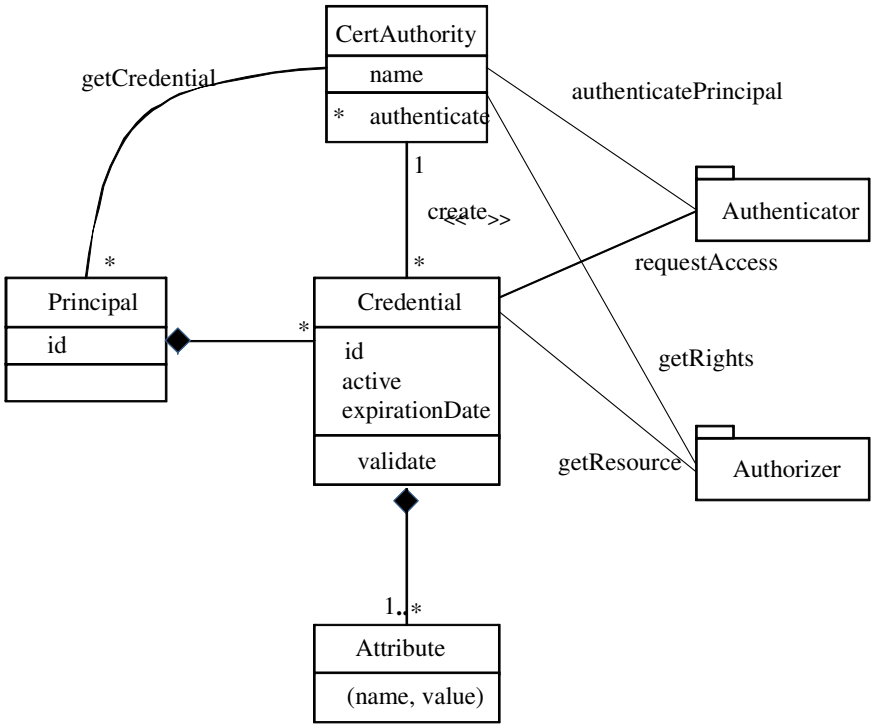

Fig. 2. Class diagram of the Credential pattern (from [7])

We show relationships between patterns using pattern diagrams [3] where rounded rectangles indicate patterns and the directed arcs indicate the contribution of a pattern to the other pattern, e.g. the Authenticator pattern authenticates access to the entity in Figure 5. We can draw pattern generalization hierarchies showing several levels in one diagram as in Figure 3. The class diagram of Figure 1 must be included in all the more specific patterns derived from it. The Distributed Authenticator only applies to distributed environments and its solution includes additional classes to consider the peculiarities of this environment. Credentials (Figure 2) are special types of authenticators used in distributed systems [7], and their varieties include X.509 Certificates, SAML Assertions, and Tokens. The package Authenticator includes the classes Authenticator and Authentication Information from Figure 1. Class Principal corresponds to the Subject of Figure 1. Credential corresponds to the Proof_Of_Identity of

Figure 1 which in this case is created by the Certification Authority and validated by the Authenticator. Authorizer is a pattern used to control access to resources by the principal [7]. In other words, this diagram combines elements of authentication and authorization.

The context, which defines the environment where the pattern applies and any conditions for its application, is one of the main determinants of the difference of a pattern with another in a hierarchy. In general, the context of a pattern includes the context of its descendants: Ci $\supseteq$ Cj, where i precedes j in the hierarchy. For example, the context of an Abstract Authenticator applies to any domain while the context of a Distributed Authenticator applies only to distributed systems, and the context of an X.509 certificate applies only to distributed systems that follow this standard. Their threats are specific realizations of the abstract pattern's threats or are new threats due to the extra elements in the class diagram (classes or attributes).

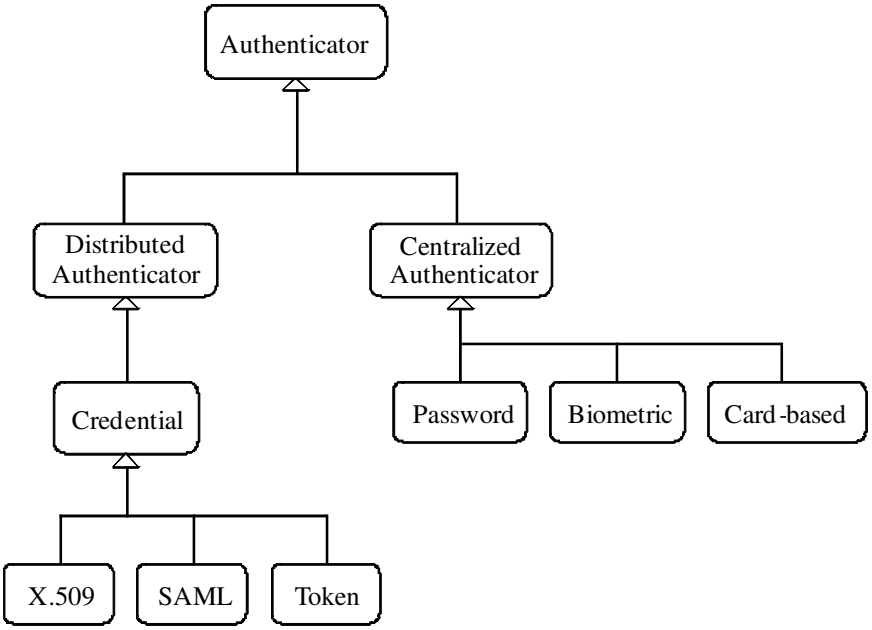

Fig. 3. Pattern diagram for the authentication hierarchy

The Problem section of the Abstract Authenticator could be stated as: How can we prevent unauthorized users from accessing our system? By system we mean an information system, a cyber-physical system, or even a physical system. A malicious attacker could try to impersonate a legitimate user to have access to her resources in the

system. This could be particularly serious if the impersonated user has a high level of privilege. How do we verify that the user intending to access the system is legitimate?

The following forces apply to its possible solution:

- *Closed system*. If the authentication information presented by the user is not recognized, there is no access.
- *Registration.* Users must register their identity information so that the system can recognize them later.
- *Flexibility*. There may be a variety of individuals (users) who require access to the system and a variety of system units with different access restrictions. We need to be able to handle all this variety 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 fool authentication or enter when the system authentication is down.
- *Protection of authentication information*. Users should not be able to read or modify the authentication information. Otherwise, they can give themselves access to the system.
- *Simplicity*. The authentication process must be relatively simple or the users or administrators may be confused. User errors are annoying to them but administrator errors may lead to security exposures.
- *Reach*. Successful authentication only gives access to the system, not to any specific resource in 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.
- *Cost.* There are always tradeoffs between security and cost, more security can be obtained at a higher cost.
- *Performance.* Authentication should not take a long time or users will be annoyed.
- *Frequency.* We should not make users authenticate frequently. Frequent authentications waste time and annoy the users.

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

- *Strength.* A password must be hard to discover, even for an attacker who has access to the password file and enough computational power.
- *Protection of Authentication Information*. The password file must not be accessible to the users.

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

The reverse of what happens in classes is true about forces and consequences, the forces in a concrete pattern include those of the abstract pattern plus new forces (and their consequences) due to the more specific environment. The forces can be specializations of the abstract forces. That is: $f_j \supseteq f_i$, where i precedes j in the hierarchy.

Because an ASP is a prototype for its concrete realizations, it may be worthwhile to make its description as careful or detailed as possible. We can try to formalize the solution section of the ASP by adding OCL to its UML models [22], or by describing this solution in a formal language (instead of its UML description). For example, [15] considers patterns as parameterized templates or types that are instantiated in applications and provide a formal description of this template. This formalization must be balanced with the need to keep the generality of the pattern and the freedom to implement it in many ways. A formal description may constrain possible implementations or make additional assumptions. Also, a pattern is more than its solution, the forces and consequences for example, are very important for its correct application.

Further examples of ASPs are shown in [7], including: Authorizer, Role-Based Access Control, Reference Monitor, Circle of Trust, Identity Provider, Abstract IDS, and Abstract VPN. We do not present more here for lack of space. Patterns in general are obtained by abstracting concepts of several implementations from real systems (best practices); ASPs are obtained by abstracting the properties of several concrete patterns or directly from the study of real systems. There is no algorithm to produce patterns or ASPs, abstraction is a human activity which depends on the experience and ability of the pattern builder.

4 Relation of ASPs to other ASPs and to Security Solution Frames

The standard associations between classes can be applied to patterns in general, and to ASPs. Figure 3 shows ASPs related to each other by generalization, e.g. an X.509 certificate "is a" credential. Figure 5 shows directed associations between patterns that describe peer associations between ASPs. Patterns can be associated also by aggregation [16], where an ASP is composed of other ASPs.

Security Solution Frames (SSFs), are solution structures that encapsulate and organize security patterns [21]; they realize security requirements. SSFs define horizontal and vertical pattern structures. Horizontal structures correspond to peer-related patterns that complement each other and define different facets of a root security policy, while vertical structures are hierarchies of pattern specializations. SSFs provide guidance for designers in applying security patterns from an abstract conceptual to a concrete design level. ASPs can be used to characterize *Security Pattern Families* which are

collections of related patterns (Figure 4). ASPs define the roots of these hierarchies, where each lower level is a pattern specialized for some specific context. For example, a SSF for Identity Management includes a family of Authentication patterns, which in turn includes Application-driven Authentication, Authentication Server, Password-based Authentication, and others. This family is defined by the Abstract Authenticator. Abstract Authenticator, Application-Driven Authentication, and Password-Based Authentication are in the same vertical family.

Alternatively, we can draw separate graphs for each level to correlate patterns in different families. This type of diagram is useful when we want to understand or explain a complete system; for example, when building a banking application we can correlate all the security patterns needed to protect accounts.

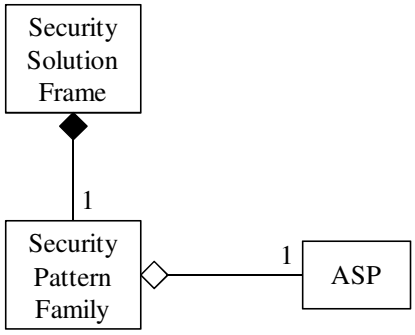

Fig. 4. From SSFs to ASPs

5 Use of ASPs

Figure 5 shows how abstract patterns are used to define security controls for a functional entity. "Functional Entity" represents some functional unit in a conceptual model and the basic security services are described by patterns Authenticator, Access Control (showing its fundamental models), and Security Logger/Auditor. These patterns solve the problems described below.

- **Authenticator** [7]. Described in Section 2.
- **Authorizer (Access Matrix)** [7, 17]. Describe who is authorized to access specific resources in a system, in an environment in which we have resources whose access needs to be controlled. It indicates for each active entity, which resources it can access, and what it can do with them.
- **Reference Monitor** [7, 17]. How do we enforce authorizations when a process requests access to an object? Define an abstract process that intercepts all requests for resources from processes and checks them for compliance with authorizations.

- **Role-Based Access Control (RBAC)** [7]. Describe how to assign rights based on the functions or tasks of people in an environment in which control of access to computing resources is required and where there is a large number of users, information types, or a large variety of resources.
- **Multilevel Security pattern** [7]. How to decide access in an environment with security classifications?
- **Attribute-Based Access Control (ABAC)** [14]. Allow access to resources based on the attributes of the subjects and the properties of the objects.
- **Security Logger /Auditor** [7]. How can we keep track of users' actions in order to determine who did what and when? Log all security-sensitive actions performed by users and provide controlled access to records for audit purposes.

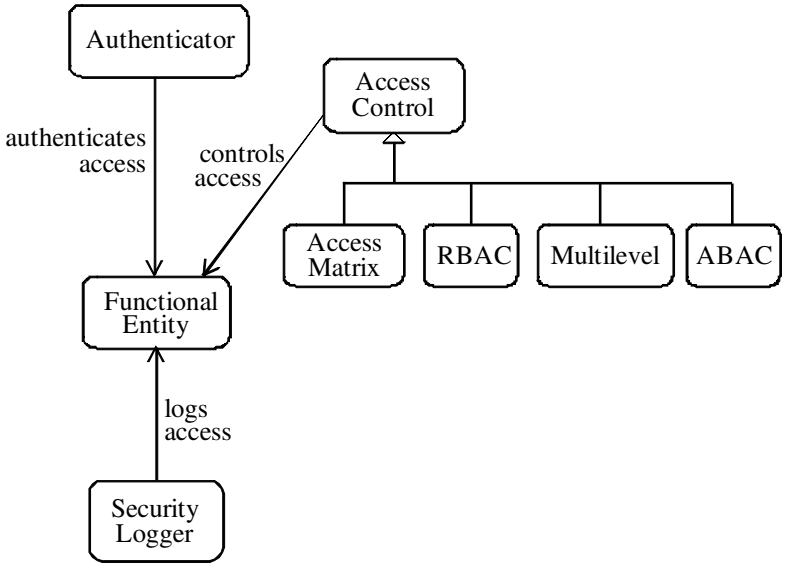

Fig. 5. Basic security services

Not all functional units require these security services, we must determine by enumerating threats which services are actually required [2]. Regulations and institution policies may also dictate the use of additional security mechanisms. However, we should not add in each entity security all possible mechanisms just in case they might be needed. Adding too many security mechanisms results in systems which:

- Are overly complex, with unnecessary redundancies, which bring administrative problems..
- Have a high performance overhead, because of redundant checks.
- Are costly, because most security mechanisms are COTS components and they have to be bought separately.

There is also a fundamental difference between adding design patterns and adding security patterns to an application. Adding design patterns is optional and is intended to improve some aspect such as flexibility or extensibility. Adding security patterns, on the other hand, intends to make the application secure and we must apply patterns to cover all security vulnerabilities or the application will not be secure. Security patterns don't improve code; their purpose is to make a complete architecture secure. Security is not based on local transformations as when using design patterns but requires a global view of the whole architecture. By showing the needed security mechanisms and when 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.

Other possible uses of ASPs include:

- Combine them with patterns describing security principles or good general design principles. For example, the Abstract Authorizer can be combined with Need-to-Know [7]; Single Point of Access [24] can be combined with Firewall [17].
- Check for security coverage in a 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 considered. This is much easier when we work at the application level of abstraction, we can enumerate all threats and find the corresponding security patterns [2].
- 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..
- Serve as abstract prototypes for similar concrete patterns. Starting from an abstract pattern it is easy to see what security constraints must at least be applied at a specific architectural level.
- Serve as ways to connect and relate different families of patterns. For example, a Communication Channel can use Intrusion Detection.
- There are patterns for enterprise models to define global security concerns [17]. These patterns include among others: Asset Valuation, Threat Assessment, Security Needs Identification, and others. ASPs can be used to implement their concerns because they are expressed in terms of application functional activities.
- We can make generalization hierarchies with patterns [Was08], and define patterns which are more and more concrete. For example, starting from a Communication Channel pattern, a Secure Channel denotes a channel where some security measure has been applied, and a Cryptographically-Protected Secure Channel defines a more specific secure communication. We can build SSFs this way.
- We can build Domain models or Reference Architectures using ASPs.

6 Related work and discussion

As indicated earlier, there is a concept of abstract pattern in the original patterns of the GOF [9], but they did not develop their possibilities because they were not concerned with the analysis stage, their work is about good coding practices (design and implementation stages)

Other varieties of security patterns include:

- *Security design patterns*. The Open Group used the style and template of [Gam94] to build security patterns [1].
- A group at CERT took a more literal approach and built *secure design patterns* [4], where they added security to several of the patterns of [9].
- Jackson's Problem Frames [11], have been used as the basis for patterns for security requirements [10].
- Mouratidis uses Secure Tropos, an approach to support multiple views of security, including organizational and external aspects [12].
- *Security usability* patterns. Patterns oriented to build good user interfaces for security [13].

The approach of [10] and [11] has in common with ASPs that they emphasize the basic security requirements of the system. However, their patterns are described in a totally different way, they do not follow the standard pattern structure and use different concepts and notation. The 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) [16] and use similar notation and concepts.

The patterns in [12] describe enterprise activities and their security constraints and thus they also express application constraints. ASPs are more detailed than those patterns and are very close to standard patterns as those in [9] and [3]; after defining them in the analysis stage the 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 the patterns in [10] and [12]. The security usability patterns can complement ASPs by providing interface requirements.

7 Conclusions

From the enumeration of their possible uses in Section 5 we can see that ASPs 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, and for pattern mining [7]. To fulfill 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 [7], but we need to separate clearly ASPs from concrete security patterns and build a specialized catalog.

As indicated earlier, there is no algorithm to build ASPs. It takes experience and abstraction ability to build them. Pattern builders build catalogs and designers use the catalogs to build systems. SSFs are sets of appropriate patterns for some requirement and facilitate the application of patterns by designers who are not expected to be security specialists. Although it is possible to apply some level of formalization to ASPs (to their solutions), they are not formal artifacts and there is no formal validation for them.

ASPs are not implementable, they are abstract models and cannot be evaluated with respect to security or performance through experimentation or testing. A pattern is a paradigm to guide implementation of new systems or evaluation of existing systems. Their evaluation 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. The ultimate validation of ASPs will come from their use by practitioners in actual designs. For now, we need to write a few complete design examples to illustrate their use and make it easier for practitioners to adopt them. The catalog would be organized using SSFs which facilitate their use.

We also need to rediscover some ASPs which have been published in software-oriented descriptions, and rewrite them to emphasize their fundamental abstract properties. In recent work we extended the idea of ASP to define abstract threats [19].

Acknowledgements

We thank the referees for their detailed and useful comments.

References

1. B. Blakeley, C. Heath, and Members of the Open group Security Forum): Technical Guide: *Security Design Patterns*, 2004, http://www.opengroup.org/bookstore/catalog/g031.htm
2. F. Braz, E. B. Fernandez, and M. VanHilst, "Eliciting security requirements through misuse activities" , *Procs. of the 2nd Int. Workshop on Secure Systems Methodologies using Patterns (SPattern'08)*. In conjunction with the 4th International Conference on Trust, Privacy & Security in Digital Business (TrustBus'08), Turin, Italy, September 1-5, 2008. 328-333.
3. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal., *Pattern- oriented software architecture,* Wiley 1996.
4. Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi, *Secure Design Patterns*, Tech. Report CMU/SEI-2009-TR-010, March 2009; Updated October 2009.
5. E. B. Fernandez and Y. Liu, "The Account Analysis Pattern", *Procs. of Euro PLoP (Pattern Languages of Programs).* http://www.hillside.net/patterns/EuroPLoP/submissions-2002.html
6. E. B. Fernandez, H. Washizaki, and N. Yoshioka, "Abstract security patterns", Position paper in *Procs. of the 2nd Workshop on Software Patterns and Quality (SPAQu'08)*, in conjunction with the 15th Conf. on Pattern Languages of Programs (PLoP 2008), October 18-20, Nashville, TN.

http://hillside.net/plop/2008/papers/ACMVersions/spaqu/fernandez.pdf (last retrieved October 4, 2011)

7. E. B. Fernandez, *Security patterns in practice: Building secure architectures using software patterns,* Wiley Series on Software Design Patterns, April 2013.
8. M. Fowler, *Analysis patterns -- Reusable object models*, Addison- Wesley, 1997.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns –Elements of reusable object-oriented software,* Addison-Wesley 1994.
10. D. Hatebur, M. Heisel, and H. Schmidt, "A pattern system for security requirements engineering", *Procs. of ARES 2007*, 356-365.
11. M. Jackson, *Problem Frames: Analyzing & structuring software development problems,* Addison-Wesley, 2001.
12. H. Mouratidis, M. Weiss, and P. Georgini, " Modelling secure systems using an agent-oriented approach and security patterns", *Int. Journal of Soft. Eng. and Knowledge Eng*., vol, 16, No 3, 2006, 471-498.
13. Jaime Muñoz Arteaga, Ricardo Mendoza, Miguel Vargas, Jean Vanderdonckt, F. Alvarez, "A methodology for designing information security feedback based on User Interface Patterns". *Advances in Eng. Software*, vol. 40, No 12, 2009, 1231-1241.
14. T. Priebe, E. B. Fernandez, J. I. Mehlau, and G. Pernul, "A pattern system for access control", in Research Directions in Data and Applications Security XVIII, C. Farkas and P. Samarati (Eds.), *Procs of the 18th. Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, Sitges, Spain, July 25-28, 2004.
15. Indrakshi Ray, R.B.France, N. Li, G.Georg, " An aspect-based approach to modeling access control concerns". *Inf. & Soft. Technology,* (9): 575-587 (2004).
16. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Boston, Mass., 1999.
17. M. Schumacher, E. B. Fernandez, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating security and systems engineering*", Wiley 2006.
18. C. Steel, R. Nagappan, and R. Lai, *Core Security Patterns: Best Strategies for J2EE, Web Services, and Identity Management,* Prentice Hall, Upper Saddle River, New Jersey, 2005.
19. A. Uzunov and E. B. Fernandez, "An Extensible Pattern-based Library and Taxonomy of Security Threats for Distributed Systems"- Special Issue on Security in Information Systems of the *Journal of Computer Standards & Interfaces*, 2013 http://dx.doi.org/10.1016/j.csi.2013.12.008
20. A. Uzunov, E. B. Fernandez, K. Falkner, "A software Engineering Approach to Authorization in Distributed, Collaborative Systems using Security Patterns and Security Solution Frames", submitted for publication.
21. A. Uzunov, K. Falkner, and E. B. Fernandez, " A comprehensive pattern-driven security methodology for distributed systems", accepted for the 2[3rd] *Australasian Software Engineering Conference (ASWEC2014*), Sydney, Australia, 2014**.**
22. J. Warmer and A. Kleppe, *The Object Constraint Language* (2[nd] Ed.), Addison-Wesley, 2003.
23. H. Washizaki, E. B. Fernandez, K. Maruyama, A. Kubo, and N. Yoshioka, "Improving the classification of security patterns", *Procs. of the Third Int. Workshop on Secure System Methodologies using Patterns (SPattern 2009).*
24. J. Yoder and J. Barcalow, "Architectural patterns for enabling application security". Procs. PLOP'97, http://jerry.cs.uiuc.edu/~plop/plop97 Also Chapter 15 in *Pattern Languages of Program Design, vol. 4* (N. Harrison, B. Foote, and H. Rohnert, Eds.), Addison-Wesley, 2000.