

Connecting Security Requirements Analysis and Secure Design Using Patterns and UMLsec

Holger Schmidt¹ and Jan Jürjens^{1,2}

¹ Software Engineering, Department of Computer Science, TU Dortmund, Germany
² Fraunhofer ISST, Germany

{holger.schmidt,jan.jurjens}@cs.tu-dortmund.de

Abstract. Existing approaches only provide informal guidelines for the transition from security requirements to secure design. Carrying out this transition is highly non-trivial and error-prone, leaving the risk of introducing vulnerabilities.

This paper presents a *pattern-oriented* approach to connect *security requirements analysis* and *secure architectural design*. Following the divide & conquer principle, a software development problem is divided into simpler subproblems based on *security requirements analysis patterns*. We complement each of these patterns with *architectural security patterns* tailored to solve classes of security subproblems. We use *UMLsec* together with the advanced modeling possibilities for software architectures of UML 2.3 to equip the architectural security patterns with security properties, and to allow *tool-supported* analysis and composition of instances of these patterns. We validate our approach using two case studies and illustrate its support for Common Criteria certifications.

Keywords: security requirement, secure design, architectural pattern.

1 Introduction

When building *secure systems*, it is instrumental to take *security requirements* into account right from the beginning of the development process to reach the best possible match between the expressed requirements and the developed software product, and to eliminate any source of error as early as possible. Knowing that building secure systems is a highly sensitive process, it is important to *reuse* the experience of commonly encountered challenges in this field. This idea of using *patterns* has proved to be of value in software engineering, and it is also a promising approach in *secure software engineering*. Moreover, *tool support* greatly increases the practical applicability of secure software engineering approaches. Tools not only guide software developers in their daily activities, they also help to make the construction of complex secure systems feasible and less error-prone.

In fact, there already exist a number of approaches to security requirements analysis and secure design. Although this can be considered a positive development, the different approaches are mostly not integrated with each other. In particular, relatively little work has been done on bridging the gap between security

requirements analysis and design, and existing approaches only provide informal guidelines for the transition from security requirements to design. Carrying out the transition manually at the hand of these guidelines is highly non-trivial and error-prone, which leaves the risk of inadvertently introducing vulnerabilities in the process. Ultimately, this would lead to the security requirements not to be enforced in the system design (and later its implementation).

This paper presents an integrated and pattern-oriented approach connecting security requirements analysis and secure architectural design. We use a security requirement analysis method [14] that makes extensive use of different kinds of patterns for structuring, characterizing, analyzing, and finally realizing security requirements. We extend this approach by *architectural security patterns* to construct *platform-independent secure software architectures* that realize previously specified security requirements. We specify structural and behavioral views of these architectural security patterns using *UML*¹ (*Unified Modeling Language*) *class diagrams*, *composite structure diagrams*, and *sequence diagrams*. We annotate these diagrams based on an improved version of the security extension *UMLsec* [8] named *UMLsec4UML2* [15] to represent results from security requirements analysis in the architectural security patterns. More specifically, we apply the advanced modeling possibilities of UML2.3 and UMLsec4UML2 to architectural design to construct the architectural security patterns presented in this paper. Moreover, our approach allows the tool-supported analysis of instances of these patterns with respect to security.

The rest of the paper is organized as follows: we present background about the patterns for security requirements engineering in Sect. 2. In Sect. 3, we first give an overview of the UMLsec4UML2-profile that adopts UMLsec to support UML2.3. Then, we use this profile to specify security patterns for software components and architectures. Furthermore, we generally discuss the application of these patterns yielding global secure software architectures. In Sect. 4, we validate our approach using two case studies and illustrate its support for Common Criteria certifications. We consider related work in Sect. 5. In Sect. 6, we give a summary and directions for future research.

2 Pattern-Oriented Security Requirements Analysis

SEPP (*Security Engineering Process using Patterns*) (see [14] for a comprehensive overview) is a pattern-based approach to construct secure software systems that especially deals with the early software development phases. SEPP makes use of *security problem frames* (SPF) and *concretized security problem frames* (CSPF), which constitute *patterns* for security requirements analysis. (C)SPFs are inspired by *problem frames* invented by Jackson [7] for functional requirements. SPFs are patterns for structuring, characterizing, and analyzing problems that occur frequently in secure software engineering. Following the divide & conquer principle, SPFs are used to decompose an initially large software development problem into smaller subproblems. Then, for each instantiated SPF,

¹ <http://www.omg.org/spec/UML/2.3/Superstructure/PDF/>

a CSPF is selected and instantiated. CSPFs involve first solution approaches for the problems described by SPFs. For example, there exists an SPF for the problem class of confidential transmission of data over an insecure network, and a CSPF that represents the corresponding solution class of using cryptographic key-based symmetric encryption to protect such data transmissions.

Each CSPF contains a *machine domain*, which represents the software to be developed in order to fulfill the *requirement*. The environment, in which the software development problem is located, is described by *problem domains*. According to Jackson [7], we distinguish *causal* domains that comply with some physical laws, *lexical* domains that are data representations, and *biddable* domains that are usually people. Each domain has at least one *interface*. Interfaces consist of *shared phenomena*, which may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain. Since requirements refer to the environment, *requirement references* between the domains and the requirement exist. At least one of these references is a *constraining* reference. That is, the domain this constraining references points to is influenced by the machine so that the requirement can be fulfilled. We developed a comprehensive set of SPFs for confidential and integrity-preserving data transmission and data storage, and authentication problems and the corresponding CSPFs that use symmetric and asymmetric encryption, keyed and non-keyed hashing, digital signatures, password-based and cryptographic key-based mechanisms (see [14] for details).

3 Pattern-Oriented Transition to Secure Architectural Design

This section contains the main scientific contributions of this paper. To proceed after security requirements analysis following SEPP to the development of secure software architectures that realize the security requirements, we develop in this section *architectural security patterns*. We specify these patterns using UML and an improved version of the security extension UMLsec, which is introduced in Sect. 3.1. We describe patterns for security components in Sect. 3.2, and we present patterns for secure software architectures related to CSPFs in Sect. 3.3. In Sect. 3.4, we briefly explain the process of instantiating GSAs. Then, we discuss the composition of different instances of GSAs yielding global secure software architectures in Sect. 3.5. Finally, we outline an approach to verify global secure software architectures based on the UMLsec4UML2-profile and the UMLsec tool suite in Sect 3.6.

3.1 UMLsec4UML2

In this section, we present an overview of a *notation* for the specification of structural as well as behavioral views of architectural security patterns based on UML. As explained in [11], UML includes special support for modeling software architectures since version 2.0. For example, the current UML version 2.3 supports typical architectural concepts such as *parts*, i.e., black-box components,

connectors, and *required* and *provided interfaces* (see Sects. 3.2 and 3.3 for details). For this reason, we specify our architectural security patterns based on different kinds of UML2.3 diagram types, i.e., class diagrams, composite structure diagrams, and sequence diagrams. Moreover, we use *UMLsec* [8] to pick up results from security requirements analysis, and to annotate the different UML diagrams representing the structural and behavioral views of architectural security patterns accordingly. Since UMLsec is a profile for UML1.5², we developed a UML2.3-compatible profile called *UMLsec4UML2* that adopts the UML1.5-compatible profile UMLsec. The UMLsec4UML2-profile, all examples shown in this paper, as well as additional material are published in a technical report [15].

We constructed the UMLsec4UML2-profile using the *Papyrus UML*³ editing tool. as a UML2.3 profile diagram. It defines several *stereotypes* and *tags*. Stereotypes give a specific meaning to the elements of a UML diagram they are attached to, and they are represented by labels surrounded by double angle brackets. A tag or tagged value is a name-value pair in curly brackets associating data with elements in a UML diagram.

The original version of UMLsec for UML1.5 is complemented by a tool suite⁴ that supports static checks for stereotypes that restrict structural design models, a permission analyzer for access control mechanisms, and checks integrated with external verification tools to verify stereotypes that restrict behavioral design models. Basically, models created based on the UMLsec4UML2-profile can be verified using this tool suite. However, the UMLsec4UML2-profile introduces a novel way to verify models directly within the UML editing tool. For this purpose, the UMLsec4UML2-profile is enriched with constraints denoted in the *Object Constraint Language* (OCL)⁵, which is part of UML2.3. OCL is a formal notation to describe *constraints* on object-oriented modeling artifacts. The static checks available in the tool suite of the original version of UMLsec are covered by the OCL constraints that are integrated into the UMLsec4UML2-profile.

We use the UMLsec4UML2-profile in the subsequent sections to specify structural as well as behavioral views of architectural security patterns. There, we also explain details about the profile where necessary.

3.2 Generic Security Components

The *generic security components* (GSC) discussed in this section constitute patterns for software components that realize concretized security requirements. We call them “generic”, because they are a kind of conceptual pattern for concrete software components. They are *platform-independent*⁶. An example for a GSC is an encryption component defined neither referring to a specific encryption algorithm nor cryptographic keys with a certain structure and length. In addition

² <http://www.omg.org/cgi-bin/doc?formal/03-03-01>

³ <http://www.papyrusuml.org/>

⁴ <http://www.umlsec.de/>

⁵ <http://www.omg.org/docs/formal/06-05-01.pdf>

⁶ The term *platform-independent* is defined according to the Model-Driven Architecture (MDA) approach (<http://www.omg.org/mda/>).

to GSCs, *generic non-security components* (GNC) are necessary, which do not realize any security requirements. Instead, they represent auxiliary components for GSCs. Typical examples for GNCs are user interface, driver, and storage management components.

According to [16], the architecture of software is multifaceted: there exists a structural view, a process-oriented view, a function-oriented view, an object-oriented view with classes and relations, and a data flow view on a given software architecture. Hence, we specify each GSC and GNC based on a structural view using UML2.3 class and composite structure diagrams, and control and data flow views using UML2.3 sequence diagrams. We make required and provided interfaces of GSCs and GNCs explicit using sockets, lollipops, and interface classes. After GSCs are instantiated, the process-oriented and object-oriented views can be integrated seamlessly into the structural view. Semantic descriptions of the operations provided and used by the components' interfaces can be expressed as OCL pre- and postconditions.

We use GSCs and GNCs to structure the machine domain of a CSPF. The GSCs and GNCs describe the machine's interfaces to its environment and the machine-internal interfaces, i.e., the interfaces between the GSCs and GNCs. Each CSPF is related to a set of GSCs and GNCs.

Given a CSPF, the following procedure can be applied to construct GSCs and GNCs that help to realize the concretized security requirement of the CSPF:

1. Each interface of the machine with the environment must coincide with an interface of some GSCs and GNCs.
2. GSCs and GNCs that serve the same purpose can be represented by one such component, e.g., several storage management components can be represented by one storage management component.
3. For each interface between the machine and a biddable or display domain a user interface component should be used. If the same CSPF contains different interfaces between the machine and a biddable or display domain, user interface components represented by GNCs must be kept separate from user interface components represented by GSCs. For example, a generic non-security user interface component to edit some text should be kept separate from a generic security user interface component to enter a password.
4. For each interface from the machine to a lexical domain, a storage management component should be used. Symbolic phenomena correspond to return values of operations or to getter/setter operations.
5. For each interface of the machine domain with a causal domain, a driver component should be used. Causal phenomena correspond to operations provided by driver components.
6. GSCs adequate to realize the concretized security requirement should be used, such as components for symmetric / asymmetric encryption / decryption, cryptographic key handling, hash calculation, etc.

We enrich GSCs with UMLsec4UML2 language elements to express security properties based on the CSPFs the GSCs are related to. Since each CSPF considers at least one asset to be protected against the malicious environment, these

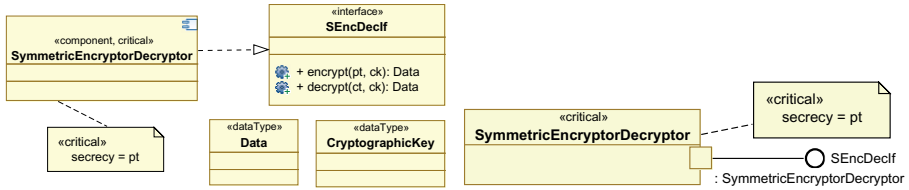


Fig. 1. Structural View of GSC SymmetricEncryptorDecryptor

assets should be considered by the GSCs associated to the CSPF. Consequently, we equip the GSCs dealing with the assets with the stereotype **«critical»**, and we assign values (e.g., in terms of attributes, parameters, return values, etc.) to the tags of this stereotype accordingly. In case of an asset to be kept *confidential*, we assign this asset to the **{secrecy}** tag, and in case of preserving the *integrity* of an asset, we assign this asset to the **{integrity}** tag.

According to the described procedure, we have developed catalogs (see [14, pp. 150 ff.] for details) of GSCs and GNCs for each available CSPF. For instance, there exist GSCs for keyed and non-keyed hash processing, calculation of random numbers, digital signature processing, etc. In the following, we present the GSC SymmetricEncryptorDecryptor as an example.

GSC SymmetricEncryptorDecryptor. The SymmetricEncryptorDecryptor is a conceptual pattern for a component that provides symmetric encryption and decryption services (see [12, pp. 59 ff.] for details). Concrete implementations of symmetric encryption and decryption algorithms are, e.g., the *javax.crypto.Cipher* class provided by *SUN's Java 6 Standard Edition*⁷, the *encryption.pbe.StandardPBEStringEncryptor* class provided by *Jasypt*⁸, and the *crypto.engines* class provided by *Bouncycastle*⁹.

Figure 1 shows the structural view of this GSC using a class diagram and a composite structure diagram. The first diagram defines the type of the port used in the second diagram. Moreover, the first diagram explicates the provided interface of the GSC. For reasons of simplicity, we do not present the GSC's behavioral view here.

The GSC SymmetricEncryptorDecryptor abstracts from algorithm-specific details such as cryptographic key lengths, the stream and block modes of the algorithms, and so on. Instead, the SymmetricEncryptorDecryptor component is designed to represent the essence of symmetric encryption and decryption services, i.e., the usage of the same cryptographic key for encryption and decryption. The SymmetricEncryptorDecryptor provides the interface SEncDecIf, which contains an operation *encrypt()* to symmetrically encrypt a plaintext *pt* using a cryptographic key *ck*. The result is a ciphertext *ct*. Additionally, it provides an inverse

⁷ <http://java.sun.com/javase/6/docs/api/>

⁸ <http://www.jasypt.org/>

⁹ <http://www.bouncycastle.org/>

operation `decrypt()` that calculates the plaintext `pt` given the ciphertext `ct` and the cryptographic key `ck`, which has to be equal to the cryptographic key used for the encryption process. This relation between the encryption and decryption functions can be formally expressed as follows: $\forall pt : Data; ck : CryptographicKey \mid decrypt(encrypt(pt, ck), ck) = pt$. We equip the GSC `SymmetricEncryptorDecryptor` with the stereotype `«critical»`, and we assign the plaintext `pt` to the tag `{secrecy}`, since the goal of this GSC is to keep the plaintext confidential.

In the next section, we explain how different GSCs and GNCs are combined to obtain patterns for secure software architectures related to CSPFs.

3.3 Generic Security Architectures

We combine the GSCs and GNCs constructed or selected for a given CSPF to obtain *generic security architectures* (GSA). Such a GSA represents the structure of the machine domain of the CSPF. Since GSCs and GNCs are platform-independent, so are GSAs. Based on the connection of CSPFs and GSAs, traceability links are introduced. Hence, our approach allows to understand which security requirements are realized by the different parts of a software architecture. This improves the maintainability of the software. Similar to GSCs and GNCs, we specify GSAs based on structural views using UML2.3 composite structure diagrams, and control and data flow views using UML2.3 sequence diagrams. The structural as well as the behavioral views of GSAs comprise the composed views of the GSCs and GNCs they consist of. We construct GSAs according to the following procedure:

1. An adequate basic software architecture to connect the GSCs and GNCs has to be selected. The GSA presented in the following and the ones introduced in [14, pp. 160 ff.] organize the components in a layered architecture. For this purpose, each of the GSAs contains an *application* component, which coordinates the behavior of all other components and provides a simplified interface (compared to directly using the interfaces of the components it connects) to the environment.
2. If components can be connected directly, one connects these components.
3. If components cannot be connected directly (e.g., because a component produces incompatible output for another component), additional *adapter* components to connect them must be introduced.
4. Interfaces between the machine and its environment must be designed in the GSA according to the interfaces of the machine domain of the corresponding CSPF.

We enrich GSAs with UMLsec4UML2 language elements to express security properties based on the CSPFs the GSAs are related to. We apply the stereotype `«secure dependency»` to the specification of the structural views of GSAs according to the following procedure:

1. The structural view of a GSA should be organized in a package stereotyped `«secure dependency»`, which contains a class diagram to define port types

for the composite structure diagram that is also contained in this package. Connections between components contained in the composite structure diagram are expressed using either simple connectors or lollipop notation.

2. As described in Sect. 3.2, GSCs refer to assets, and they are already equipped with the `«critical»` stereotype and corresponding tagged values. GSCs connected in the structural view's composite structure diagram with other GSCs or / and GNCs might allow the transmission of assets to these components. According to the `«secure dependency»` stereotype, these GSCs or / and GNCs should be stereotyped `«critical»`, too. Moreover, the tagged values of these components should be equal to the tagged values of GSCs that are connected to them.
3. `«use»` dependencies between components stereotyped `«critical»` in the structural view's composite structure diagram should be stereotyped according to the tagged values of the `«critical»` stereotype. That is, if the tag `{secrecy}` is assigned a value, then the corresponding `«use»` dependency between the components should be stereotyped `«secrecy»`, and if the tag `{integrity}` is assigned a value, then the corresponding `«use»` dependency between the components should be stereotyped `«integrity»`. The dependencies stereotyped `«use»` between components and interfaces of components labeled `«critical»` in the structural view's class diagram should be stereotyped analogously.

Moreover, the behavioral views of GSAs are equipped with the `«data security»` stereotype. Given a package stereotyped `«data security»` containing a structure and a behavior diagram, the requirements defined in the structure diagram using the `«critical»` stereotype should be fulfilled with respect to the behavior diagram and environment description (especially the malicious environment and the value of the tag `{adversary}`). We apply the stereotype `«data security»` to the specification of the behavioral views of GSAs according to the following procedure:

1. The behavioral view should be organized in a package stereotyped `«data security»`.
2. The structural view previously discussed should be reused by importing the corresponding package into the one of the behavioral view.
3. A specification in terms of a set of sequence diagrams should be included in the behavioral view to describe the collaboration between the different GSCs and GNCs contained in the GSA at hand.
4. The attacker model, i.e., especially the `{adversary}` tag, is not assigned a value on the level of patterns. Instead, the attacker model is fixed when instantiating GSAs (see Sect. 3.4 for details).

According to the described procedures, we have developed a catalog of GSAs (see [14, pp. 160 ff.] for details) for each available CSPF.

GSA for CSPF Confidential Data Transmission Using Cryptographic Key-Based Symmetric Encryption. In the following, we present as an

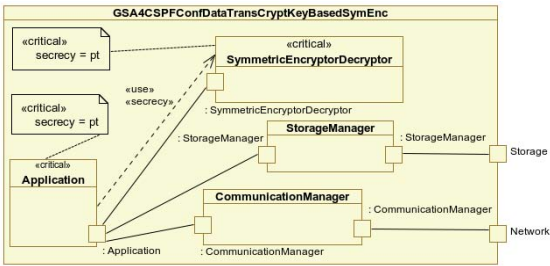


Fig. 2. Structural View of GSA for “CSPF Confidential Data Transmission Using Cryptographic Key-Based Symmetric Encryption”

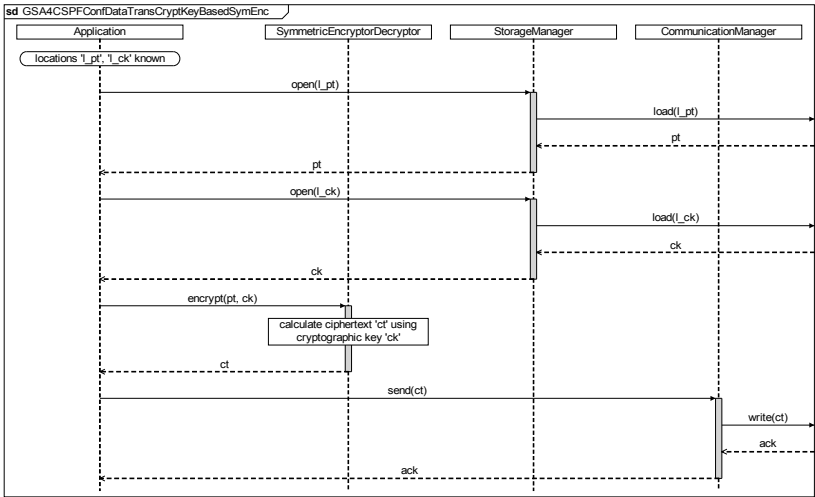


Fig. 3. Behavioral View Complementing Fig. 2

example a GSA for the machine domain of the CSPF confidential data transmission using cryptographic key-based symmetric encryption. Figure 2 shows the structural view of the GSA using a composite structure diagram. The **Sender machine domain** loads the **Sent data** and **Cryptographic key₁** domains from a storage device. Hence, we introduce the GNC **StorageManager** to access a storage device. The **Sent data** domain is encrypted using the **Cryptographic key₁** domain. For this reason, we introduce the GSC **SymmetricEncryptorDecryptor** presented in Sect. 3.2. Furthermore, the **Sender machine domain** sends the encrypted data to the **Communication medium domain**. Hence, we introduce the GNC **CommunicationManager** to access a network.

According to the CSPF this GSA is related to, the plaintext **pt** represented in the CSPF as lexical domain **Sent data** should be kept confidential. Hence, the GSC **SymmetricEncryptorDecryptor** and the GNC **Application** that makes use of this GSC

are stereotyped `<<critical>>`. Furthermore, the `{secret}` tag is assigned the plaintext `pt`, and the `<<use>>` dependency between the `GSC SymmetricEncryptorDecryptor` and the `GNC Application` is stereotyped `<<secret>>`.

The overall behavior of the GSA is depicted in Fig. 3. Initially, the locations of the plaintext `pt` (that corresponds to the `Sent data` domain) and the cryptographic key `ck` (that corresponds to the `Cryptographic key1` domain) are known, and they are retrieved from a storage device using the `GNC StorageManager`. Then, the ciphertext is constructed by the `GSC SymmetricEncryptorDecryptor`. Finally, the `GNC CommunicationManager` sends the ciphertext to a network. Both the structural as well as the behavioral views are contained in a package (not explicitly depicted in this paper) stereotyped `<<data security>>`.

In the next section, we discuss the instantiation of GSAs.

3.4 Instantiation of GSAs

GSAs are instantiated based on the corresponding CSPF instances. The selection process for GSAs is heavily intertwined with the (security) requirements engineering phase and described in detail in [6]. An application layer component should have a name that equals the name of the machine domain of the corresponding CSPF instance. Moreover, the interfaces between a GSA and the environment should be named after the domains of the corresponding CSPF instance. The names of shared phenomena of the CSPFs should be re-used for the instantiation of the messages contained in the sequence diagrams of the behavioral views of the corresponding GSAs. Thus, each component contained in a GSA is instantiated, too.

The CSPF instance that constitutes the basis for a GSA instance includes an environment description in terms of problem domains, interfaces in between, and domain knowledge. This information is used to construct an attacker model based on the `UMLsec4UML2`-stereotype `<<secure links>>`. Given a package stereotyped `<<secure links>>` with the tagged value `{adversary=default}` containing a deployment diagram with a dependency stereotyped `<<secret>>` (or / and `<<integrity>>` or / and `<<high>>`) between two nodes, these nodes should be either connected via a communication path stereotyped `<<LAN>>` or `<<encrypted>>` or `<<wire>>` (but not `<<Internet>>`) or labeled `<<LAN>>`. We now explain how we use the `<<secure links>>` stereotype to construct an attacker model based on the instantiated CSPFs:

1. An environment description in terms of a deployment diagram should be included in a package stereotyped `<<secure links>>`. The diagram should represent the state of the environment before the envisaged software system is in operation. Hence, the constraint associated to the `<<secure links>>` stereotype is *not* fulfilled.
2. Each machine domain, lexical domain, and causal domain should be modeled as a node (3-D box) in the deployment diagram.
3. The physical connections between nodes should be modeled as communication paths (solid line between two nodes) in the deployment diagram, and

each path should be stereotyped according to its type as either `«LAN»` or `«encrypted»` or `«wire»` or `«Internet»`.

4. The tag `{adversary}` of the `«secure links»` stereotype should be assigned a value according to domain knowledge collected during requirements analysis. For example, if shared phenomena exist indicating that data items of a connection can be deleted, read, and inserted arbitrarily, then the `«Internet»` should be applied, and the tag `{adversary}` should be assigned the value `default` (see [8] for details).
5. The tag `{adversary}` of the `«data security»` stereotype of the behavioral views of the instantiated GSAs should be assigned the same value as the value of the equally named tag of the `«secure links»` stereotype.

In the next section, we show how different GSA instances are combined yielding global secure software architectures.

3.5 Composition of Different GSA Instances

Composing different GSA instances means that one must decide whether the components contained in more than one GSA instance should appear only once in the global architecture, i.e., whether they can be merged. Basically, three different categories of components must be considered: application layer components, GNC instances, and GSC instances.

Choppy et al. [2] developed for a set of *functional* subproblem classes a corresponding set of subarchitectures that solve these subproblems. Moreover, the subarchitectures are composed based on dependencies between the subproblems such as parallel, sequential, and alternative dependencies. We adopt the principles by Choppy et al. [2] to merge application layer components and GNC instances.

We now discuss the composition of GSA instances to obtain a global secure software architecture that still fulfills the security requirements realized by the corresponding GSA instances. Especially confidentiality requirements must be treated carefully, since the composition of incompatible components can lead to non-fulfillment of confidentiality requirements.

If two GSA instances contain GSC instances that serve the same purpose, then these components cannot be merged in general. The question to be answered is if the two GSC instances can use the same algorithm-specific configuration, e.g., a specific algorithm, key lengths, salt lengths, etc., to fulfill the different security requirements. For example, a specific encryption algorithm and specific key lengths might be sufficient to solve one security subproblem, while the same configuration would lead to a vulnerable system if applied to another subproblem. The level of abstraction of GSC instances might not allow to decide whether they can be merged or if they have to be kept separately. Then, it is necessary to refine the GSC instances to platform-specific security components to come to this decision. An approach to deal with the composition of GSC instances before their refinement to platform-specific security components is to merge GSC instances of the same type into one *configurable* GSC instance of this type. Here,

configurable means that we equip such a component with a variable mode, i.e., the algorithm a component realizes as well as the used cryptographic key can be changed at runtime.

Choppy et al.[2] considered only the structural composition of subarchitectures. Since our GSA instances are additionally equipped with behavioral views, we also consider the composition of these descriptions. The resulting behavioral specification of the global secure architecture represents the life-cycle of the software to be constructed and its components. The behavior of GSA instances is given as a set of sequence diagrams, which should be composed based on the subproblem dependencies determined for the structural composition of the application layer components: in case of a sequential dependency, the sequence diagrams should be composed according to the order defined by this dependency. In case of a parallel dependency, the sequence diagrams should be composed in such a way that the effects and output realized by the different GSA instances are fulfilled jointly. If GSC instances are merged into one configurable GSC instance, then its re-configuration should be included in the corresponding sequence diagrams. The result is a platform-independent global secure software architecture described by structural and behavioral views.

We briefly discuss an approach to analyze global secure software architecture, i.e., to verify the constraints associated with UMLsec4UML2 stereotypes that are contained in such architectures.

3.6 Verification of Global Secure Software Architectures

We use the UMLsec4UML2-profile and the UMLsec tool suite to show that the GSC and GNC instances in a global secure software architecture work together in such a way that they fulfill the security requirements corresponding to the different subproblems. Based on the `«secure dependency»` stereotype and the OCL constraints contained in the UMLsec4UML2-profile, it is possible to check whether critical data items might be leaked (`{secrecy}` and `{high}`) or changed (`{integrity}`). These checks can be executed directly within compatible UML editing tools such as Papyrus UML and MagicDraw UML. Based on the stereotypes `«data security»` and `«secure links»` and the OCL constraints contained in the UMLsec4UML2-profile, it is possible to check whether behavior introduced by a GSA might compromise confidentiality (`{secrecy}` and `{high}`) or integrity (`{integrity}`). Such checks can be executed based on the UMLsec tool suite, which makes use of the SPASS theorem prover¹⁰ for the verification of properties of behavioral models.

We now illustrate the previously presented approach in the next section.

4 Validation

We tested the approach presented in the previous sections using two case studies: a *secure text editor* and an *Internet-based password manager*. We performed

¹⁰ <http://www.spass-prover.org/>

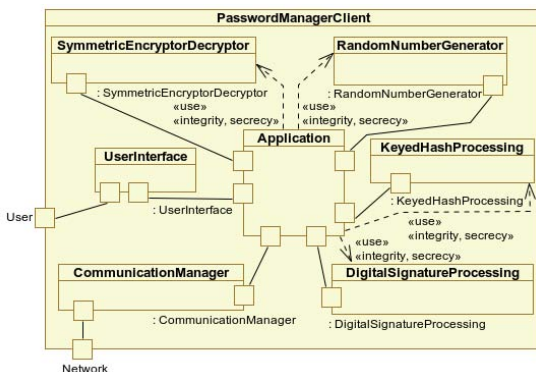


Fig. 4. Structural View of Password Manager Client

the complete development life-cycle for both case studies, i.e., from requirements engineering to architectural and fine-grained design to the implementation and testing of the programs. Moreover, we created all development artifacts as advocated by SEPP. While we present in this paper only a small part of the password manager case study, the complete results are contained in [14] and partly in a technical report about UMLsec4UML2 [15].

Security requirements analysis of the password manager following SEPP as outlined in Sect. 2 leads to the elicitation and analysis of 13 different security requirements, e.g., about the confidentiality and integrity of the different usernames and passwords. Due to partly overlapping security requirements, only 11 different CSPF instances are developed. Consequently, 11 different GSAs are instantiated and combined yielding a global secure software architecture. Figure 4 partly shows the structural view of global secure software architecture of the password manager client expressed using a composite structure diagram. There, instances of GSCs for encryption/ decryption, keyed hash processing, digital signature processing, generation of random numbers, as well as instances of GNCs for the user interface and network communication work together, coordinated by the application layer GNC instance, in order to ensure secure communication between the password manager client and server over the Internet. The global architecture consisting of instances of GSCs and GNCs significantly helped us to proceed with the development phases to follow. We identified adequate frameworks, off-the-shelf components, and API modules based on the generated artifacts, i.e., the components, the explicit interface descriptions, the protocol descriptions, and the UMLsec4UML2 security annotations. Consequently, the programming phase had to cover creating the glue code to connect the existing components and modules only. In summary, the case studies show that using patterns to bridge the gap between security requirements analysis and secure architectural design constitutes a feasible and promising contribution to the field of secure software engineering.

We also evaluated our approach with respect to ISO/IEC 15408:2005 aka Common Criteria (CC) certifications. The usage of the architectural artifacts generated following our approach for a CC certification is possible based on the TOE (Target Of Evaluation) Design Specification (TDS) of the class ADV Development. For instance, EAL (Evaluation Assurance Level) 5 requires a semi-formal modular design, i.e., a representation of the TOE's structure in terms of subsystems and a description of the parts the subsystem consists of in terms of modules. In addition to the TDS requirements for EAL 4, it is necessary to also describe those modules that represent SFR (Security Functional Requirement)-supporting modules in detail, i.e., by describing its SFR-related interfaces, return values from those interfaces, and called interfaces to other modules. These TDS requirements are met by the artifacts that describe the realizations of GSC and GNC instances. Moreover, a semiformal notation for the SFR-enforcing modules should be used. Since our approach makes use of UML2.3 diagrams, this requirement is fulfilled right away. The tool suite developed for the original version of UMLsec and checking the OCL constraints of the UMLsec4UML2-profile supports creating TDS documents. For instance, the stereotype `«secure dependency»` allows to track the occurrence of assets in the complex TDS documents. Using an UML editing tool such as Papyrus UML the OCL constraints representing this stereotype can be verified to ensure that we covered all relevant occurrences of an asset.

5 Related Work

Recently, an approach [10] to connect the security requirements analysis method *Secure Tropos* by Mouratidis et al. [3] and UMLsec [8] is published. Bryl et al. [1] extended the *Secure Tropos* variant by Massacci et al. [9] by an approach to automatically select design alternatives based on results from security requirements analysis. Compared to our work, these approaches are not based on patterns, and they rather focus on the transition to finer-grained secure design.

Choppy et al. [2] present architectural patterns for Jackson's basic problem frames [7]. The patterns constitute layered architectures described by UML composite structure diagrams. Similar to other approaches considering the connection between problem frames and software architectures such as [13, 4], the work by Choppy et al. does not consider security requirements, behavioral interface descriptions, and operation semantics. Furthermore, only a vague general procedure to derive components for a specific frame diagram is given in [2].

The vast body of patterns for secure software engineering (see [5] for an overview) can be used during the phase that follows the phase presented in this paper, i.e., these patterns are applied in fine-grained design of secure software. Hence, the existing security design patterns and our approach complement each other to such an extent that the existing patterns can be expressed in a unifying way based on SPFs, CSPFs, and GSAs.

6 Conclusions and Future Work

We presented in this paper a novel *pattern-oriented* and *tool-supported* approach to bridge the gap between security requirements analysis and secure architectural design. Its main benefit is that the construction of global secure software architectures based on results from security requirements engineering becomes more feasible, systematic, less error-prone, and a more routine engineering activity.

In the future, we plan to develop new UMLsec4UML2 stereotypes to specify assumptions and facts about the operational environment of the software. Moreover, we intend to develop patterns that support the systematic composition of different GSA instances thereby preserving the associated security requirements.

References

- [1] Bryl, V., Massacci, F., Mylopoulos, J., Zannone, N.: Designing security requirements models through planning. In: Martinez, F.H., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 33–47. Springer, Heidelberg (2006)
- [2] Choppy, C., Hatebur, D., Heisel, M.: Component composition through architectural patterns for problem frames. In: Proceedings of the Asia Pacific Software Engineering Conference (APSEC), pp. 27–34. IEEE Computer Society, Washington, DC, USA (2006)
- [3] Giorgini, P., Mouratidis, H.: Secure tropos: A security-oriented extension of the tropos methodology. *International Journal of Software Engineering and Knowledge Engineering* 17(2), 285–309 (2007)
- [4] Hall, J.G., Jackson, M., Laney, R.C., Nuseibeh, B., Rapanotti, L.: Relating software requirements and architectures using problem frames. In: Proceedings of the IEEE International Requirements Engineering Conference (RE), pp. 137–144. IEEE Computer Society, Los Alamitos (2002)
- [5] Heyman, T., Yskout, K., Scandariato, R., Joosen, W.: An analysis of the security patterns landscape. In: Proceedings of the International Workshop on Software Engineering for Secure Systems (SESS), pp. 3–10. IEEE Computer Society, Los Alamitos (2007)
- [6] Heyman, T., Yskout, K., Scandariato, R., Schmidt, H., Yu, Y.: The security twin peaks. In: Erlingsson, Ú., Wieringa, R., Zannone, N. (eds.) ESSoS 2011. LNCS, vol. 6542, pp. 167–180. Springer, Heidelberg (2011)
- [7] Jackson, M.: Problem Frames. In: Analyzing and structuring software development problems. Addison-Wesley, Reading (2001)
- [8] Jürjens, J.: Principles for Secure Systems Design. PhD thesis, University of Oxford (2002)
- [9] Massacci, F., Mylopoulos, J., Zannone, N.: An Ontology for Secure Socio-Technical Systems. Information Science Reference. In: Ontologies for Business Interaction, pp. 188–207 (2007)
- [10] Mouratidis, H., Jürjens, J.: From goal-driven security requirements engineering to secure design. *International Journal of Intelligent Systems – Special issue on Goal-Driven Requirements Engineering* 25(8), 813–840 (2010)
- [11] Pérez-Martínez, J.E., Sierra-Alonso, A.: UML 1.4 versus UML 2.0 as languages to describe software architectures. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 88–102. Springer, Heidelberg (2004)

- [12] Pfleeger, C.P., Pfleeger, S.L.: Security In Computing, 3rd edn. Prentice Hall PTR, Englewood Cliffs (2003)
- [13] Rapanotti, L., Hall, J.G., Jackson, M., Nuseibeh, B.: Architecture-driven problem decomposition. In: Proceedings of the IEEE International Requirements Engineering Conference (RE), pp. 80–89. IEEE Computer Society, Los Alamitos (2004)
- [14] Schmidt, H.: A Pattern- and Component-Based Method to Develop Secure Software. Deutscher Wissenschafts-Verlag (DWV) Baden-Baden (April 2010)
- [15] Schmidt, H., Jürjens, J.: UMLsec4UML2 - adopting UMLsec to support UML2. Technical Report 838, Technical University of Dortmund (February 2011), <http://hdl.handle.net/2003/27602>
- [16] Shaw, M., Garlan, D.: Software Architecture. Perspectives on an Emerging Discipline. Prentice Hall PTR, Englewood Cliffs (1996)