Towards Developing Secure Software using Problem-oriented Security Patterns

Azadeh Alebrahim, Maritta Heisel

paluno - The Ruhr Institute for Software Technology,
University of Duisburg-Essen, Germany
{firstname.lastname}@paluno.uni-due.de

Abstract. Security as one essential quality requirement has to be addressed during the software development process. Quality requirements such as security drive the architecture of a software, while design decisions such as security patterns on the architecture level in turn might constrain the achievement of quality requirements significantly. Thus, to obtain sound architectures and correct requirements, knowledge which is gained in the solution space, for example from security patterns, should be reflected in the requirements engineering. In this paper, we propose an iterative method that takes into account the concurrent development of requirements and architecture descriptions systematically. It reuses security patterns for refining and restructuring the requirement models by applying problem-oriented security patterns. Problem-oriented security patterns adapt existing security patterns in a way that they can be used in the problem-oriented requirements engineering. The proposed method bridges the gap between security problems and security architectural solutions.

Keywords: Requirements engineering, security requirements, problem frames, security patterns, digital signature

1 Introduction

Many software systems fail to achieve their quality objectives due to neglecting quality requirements at the beginning of the software development life cycle [1]. In order to obtain a software that achieves not only its required functionality but also the desired quality properties, it is necessary to consider both types of requirements, functional and quality ones, early enough in the software development life cycle. Security is one essential quality requirement to be considered during the software development process.

Architecture solutions provide a means to achieve quality requirements. While requirements are supposed to be the architectural drivers, architecture solutions represent design decisions on the architecture level that in turn affect the achievement of quality requirements significantly. Decisions made in the design phase could constrain the achievement of initial requirements, and thus could change them. Hence, requirements cannot be considered in isolation. They should be co-developed with the software architecture iteratively known as *Twin Peaks* as proposed by Nuseibeh [2] to support the creation of sound architectures and correct requirements [3]. The sooner architectural knowledge becomes involved in the process of the requirements analysis, the less costly are the changes to be made. Hence, there is a need for a method that systematically takes into account the development of both artifacts, concurrently.

There exist solutions to security problems represented as patterns [4] that can be applied during the design and implementation phases of software development processes.

In order to reuse the knowledge which is gained in the solution space such as security patterns, we make use of *problem-oriented security patterns* [5]. Problem-oriented security patterns reuse existing security patterns and mechanisms and adapt them in a way that they can be used in the problem-oriented requirements engineering.

In this paper, we follow the concept of the twin peaks [2] that advocates the concurrent and iterative development of requirements and software architecture descriptions, as requirements and software architecture cannot be developed in isolation. We propose a method on how to exploit the knowledge gained in the solution domain systematically in the problem space considering the intertwining nature of requirements engineering and architectural design. Our method provides an instantiation of the twin-peaks model for developing secure software systems. We make use of the problem-oriented security patterns to refine and restructure the requirement models.

The proposed method relies on problem frames [6] as a requirements engineering approach. It is important that the results of the requirements analysis with problem frames can be easily re-used in later phases of the development process. Since UML [7] is a widely used notation to express analysis and design artifacts, we make use of a specific UML profile [8] for problem frames that carries over problem frames to UML. We use the problem frames approach, because 1) it allows decomposing the overall software problem into subproblems, thus reducing the complexity of the problem, 2) it makes it possible to annotate problem diagrams with quality requirements, such as security requirements, 3) it enables various model checking techniques, such as requirements interaction analysis and reconciliation [9] due to its semi-formal structure, and 4) it supports a seamless transition from requirements analysis to architectural design (e.g. [10]).

The benefit of the proposed method is manifold. First, it provides guidance for refining security problems located in the problem space using problem-oriented security patterns. Second, the elaborated security requirement models can easily be transformed into a particular solution at the design level. Thus, it bridges the gap between security problems and security solutions. Third, it supports the concurrent and iterative development of requirements and architectural descriptions systematically. Fourth, it supports less experienced software engineers in applying solution approaches early in the requirements engineering phase in a systematic manner.

The remainder of the paper is organized as follows. We present the background on which our approach builds in Sect. 2, namely problem frames and problem-oriented security patterns. In Sect. 3, we describe our method. Section 4 is devoted to illustrating the applicability of our method by taking into account digital signature as a specific problem-oriented security pattern. Related work is discussed in Sect. 5, and conclusions and future work are given in Sect. 6.

2 Background

In this section, we give a brief overview on basics our approach relies on. We first describe the concepts of the problem frames approach in Sect. 2.1. Then, we introduce problem-oriented security patterns in Sect. 2.2.

2.1 Problem Frames

Requirements analysis with problem frames [6] proceeds as follows: to understand the problem, the environment in which the *machine* (i.e., software to be built) will operate must be described first. To this end, we set up a *context diagram* consisting of *machines*, *domains* and *interfaces*. Domains represent parts of the environment which are relevant for the problem at hand. Then, the problem is decomposed into simple subproblems that fit to a *problem frame*. Problem frames are patterns used to understand, describe, and analyze software development problems. An instantiated problem frame is a *problem diagram* which basically consists of one *submachine* of the machine given in the context diagram, relevant domains, interfaces between them, and a *requirement* referring to and constraining problem domains. The task is to construct a *(sub-)machine* that improves the behavior of the environment (in which it is integrated) in accordance with the requirement.

We describe problem frames using UML class diagrams [7], extended by a specific UML profile for problem frames (UML4PF) proposed by Hatebur and Heisel [11]. A class with the stereotype ≪`machine`≫ represents the software to be developed. Jackson distinguishes the domain types biddable domains (represented by the stereotype ≪`BiddableDomain`≫) that are usually people, causal domains (≪`CausalDomain`≫) that comply with some physical laws, and lexical domains (≪`LexicalDomain`≫) that are data representations. To describe the problem context, a *connection domain* (≪`ConnectionDomain`≫) between two other domains may be necessary. Connection domains establish a connection between other domains by means of technical devices. Figure 5 shows a problem diagram in the context of our case study *smart grids* (see Sect. 4.1). It describes that smart meter gateway submits meter data to an authorized external entity. The submachine *SubmitMD* is one part of the smart meter gateway. It sends the *MeterData* through the causal domain *WAN* to the biddable domain *AuthorizedExternalEntity*.

When we state a requirement we want to change something in the world with the machine to be developed. Therefore, each requirement expressed by the stereotype ≪`requirement`≫ constrains at least one domain. This is expressed by a dependency from the requirement to a domain with the stereotype ≪`constrains`≫. A requirement may refer to several domains in the environment of the machine. This is expressed by a dependency from the requirement to these domains with the stereotype ≪`refersTo`≫. The requirement *RQ4* constrains the domain *WAN*, and it refers to the domains *MeterData* and *AuthorizedExternalEntity*.

2.2 Problem-oriented Security Patterns

We make use of the concept of *problem-oriented security patterns* [5], which are described as problem diagrams. Security patterns and mechanisms have been reused and adapted in a way that they can be used in the problem-oriented requirements engineering as problem-oriented security patterns. A problem-oriented security pattern consists of a three-part graphical pattern and a template. The graphical pattern involves one problem diagram that describes the structure of the generic functional requirement and the involved domains. It is annotated with a specific security requirement, for which we provide a solution approach. The second part of a problem-oriented security pattern is a problem diagram that describes the particular solution approach for the security requirement. Conclusively, we provide a problem diagram that describes how the problem diagram describing the functional problem and the problem diagram describing the security solution can be composed to solve the overall problem.

The proposed template consists of two parts documenting additional information related to the domains in the graphical pattern. Such information is not observable in the

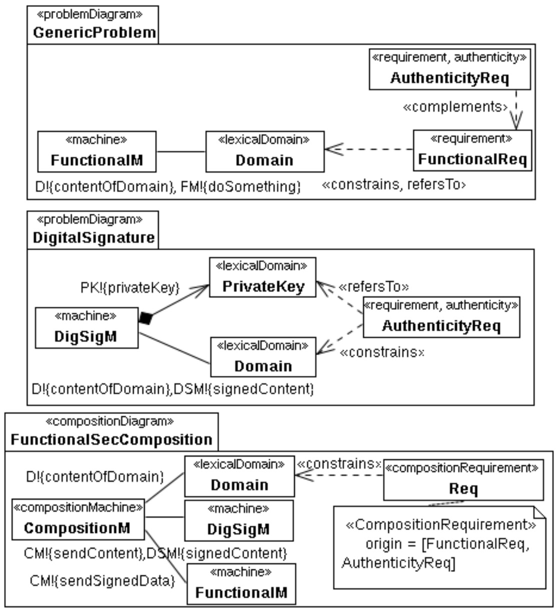


Fig. 1. Problem-oriented digital signature pattern (graphical pattern)

graphical pattern. The first part accommodates information about the security mechanism itself such as name (*Name*), purpose (*Purpose*), description (*Brief Description*), and the quality requirement which will be achieved when applying this pattern (*Quality Requirement to be achieved*). Moreover, a security solution may affect the achievement of other quality requirements. For example, improving the security may result in decreasing the performance. Hence, the impact of each security solution on other quality requirements has to be captured in the first part of the template (*Affected Quality Requirement*). A security pattern not only solves a problem, but also produces new functional and quality problems that have to be addressed either as *Requirements* to be elicited or as *Assumptions* needed to be made in the second part of the template. We elicit new functional and quality problems as requirements if the software to be built shall achieve them. Assumptions have to be satisfied by the environment [12]. They do not guarantee to be true in every case. For the case that we assume the environment (not the machine) takes the responsibility for meeting them, we capture them as assumptions. This should be negotiated with the stakeholders and documented properly.

Digital signature is an important means for achieving integrity and authenticity of data. Using the digital signature, the receiver ensures that the data is created by the known sender. We present the *problem-oriented digital signature pattern* by its corresponding graphical pattern depicted in Fig. 1 and its corresponding template shown in Table 1. The graphical pattern first describes the functional problem expressed as the problem diagram *GenericProblem*. It describes the functional requirement *FunctionalReq* and the involved domains. The functional requirement *FunctionalReq* has to be met by the functional machine *FunctionalM*. The functional requirement is complemented by the security requirement *AuthenticityReq* demanding "the verification of genuineness of data". The first problem diagram in Fig. 1 depicts the functional problem. Depending on the functional requirement, the problem diagram might contain other domains that are not relevant for the security problem. Hence, they are not represented in the pattern.

Table 1. Problem-oriented digital signature pattern (template)

Security Solution	
Name	Digital Signature
Purpose	For *Domain* constrained by the requirement *FunctionalReq*
Brief Description	Sender produces a signature using the private key and the data.
Quality Requirement to be achieved	Security (integrity and authenticity) *IntegrityReq*, *AuthenticityReq*
Affected Quality Requirement	Performance *PerformanceReq*
Necessary Conditions	
Requirement ☐ **Assumption** ☐	Confidentiality of private key during storage shall be/is preserved.
Requirement ☐ **Assumption** ☐	Integrity of private key during storage shall be/is preserved.
Requirement ☐ **Assumption** ☐	Confidentiality of signature machine shall be/is preserved.
Requirement ☐ **Assumption** ☐	Integrity of signature machine shall be/is preserved.

The digital signature as a solution for the authenticity problem is expressed as the problem diagram *DigitalSignature* in the middle of Fig. 1. It consists of all domains that are relevant for the solution. The machine *DigSigM* should achieve the authenticity requirement *AuthenticityReq* by signing the *Domain* using the *PrivateKey* which is part of the machine *DigSigM*. The third part composes the functional machine *FunctionalM* with the security machine *DigSigM* by introducing a new machine *CompositionM* that has to meet the requirement *CompositionReq* composed of the requirements *FuncitonalReq* and *AuthenticityReq*.

Note that the *problem-oriented digital signature pattern* can be used to achieve integrity as well. In Fig. 1, we only showed the use of *problem-oriented digital signature pattern* to achieve the integrity requirement *AuthenticityReq* in order to keep the figure clear and readable. One can apply the same pattern and only replace the authenticity requirement with the integrity requirement to achieve integrity.

The template shown in Table 1 represents the additional information corresponding to the graphical part of the problem-oriented pattern *digital signature*. New requirements and assumptions to be considered are represented in the second part of the template.

3 Problem-oriented Secure Software Development

As we mentioned earlier, problem descriptions and architectural descriptions should be considered as intertwining artifacts influencing each other. We therefore take the Twin Peaks model [2] into account and illustrate our problem-oriented software development method embedded in the context of the Twin Peaks model (see Fig. 2). Our problem-oriented software development method consists of five phases. Phases 1 and 2 are introduced in details in [13,14]. Thus, we describe them briefly here. In this paper, our focus is on identification, selection, and application of appropriate security patterns which refine and restructure the requirement models. So refined problem descriptions can easily be transformed into architectural descriptions.

Phase 1- context elicitation & problem decomposition This phase (see 1 in Fig. 2) involves understanding the problem and its context, decomposing the overall problem into subproblems, and annotating security requirements. In order to understand the problem, we elicit all domains related to the problem to be solved, their relations to each other and the software to be constructed. Doing this, we obtain a *context diagram*

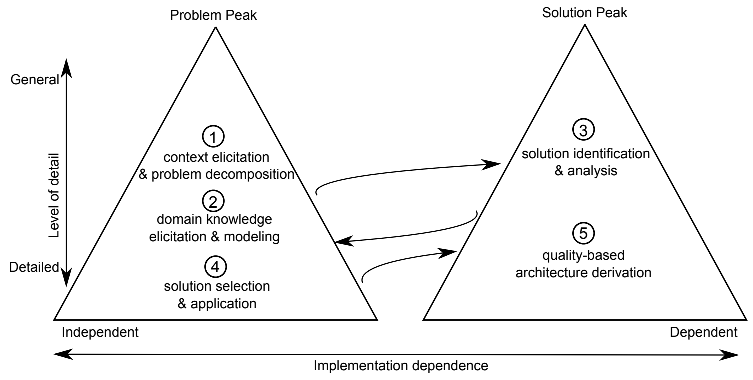

Problem Peak
Solution Peak
General
Level of detail
Detailed
1
context elicitation & problem decomposition
2
domain knowledge elicitation & modeling
4
solution selection & application
3
solution identification & analysis
5
quality-based architecture derivation
Independent
Dependent
Implementation dependence

Fig. 2. Overview of our method embedded in the Twin Peaks Model

consisting of the machine (software-to-be), related domains in the environment, and interfaces between these domains. Then, we decompose the overall problem into subproblems, which describe a certain functionality, as expressed by a set of related functional requirements. We set up *problem diagrams* representing subproblems to model functional requirements.

To analyze and integrate quality requirements in the software development process, quality requirements have to be modeled and integrated as early as possible in the requirement models. Modeling quality requirements and associating them to the functional requirements is achieved by annotating them in problem diagrams. For more information about context elicitation and problem decomposition, see [13].

Phase 2- domain knowledge elicitation & modeling The system-to-be comprises the software to be built and its surrounding environment such as people, devices, and existing software [12]. In requirements engineering, properties of the environment and constraints about it, called *domain knowledge*, need to be captured in addition to exploring the requirements [6,15]. Hence, the second phase (see 2 in Fig. 2) involves *domain knowledge elicitation & modeling*. We elicit the relevant information that has to be collected when dealing with specific software qualities such as security and document them as structured templates called *Domain Knowledge Templates*. To guarantee security, we need domain knowledge about the type of possible attackers that influence the restrictiveness of a security requirement. It depends on the abilities of the attacker how much resources and effort to spend on a requirement, how much influence security has on the behavior of the overall system-to-be, and which solution to fulfill the requirement has to be chosen later on. Different types of attackers can be considered. For example, a software attacker targets at manipulating the software, whereas a network attacker aims at manipulating the network traffic. To describe the attacker we use the properties as described by the Common Methodology for Information Technology Security Evaluation (CEM) [16] for vulnerability assessment of the TOE (target of evaluation i.e., system-to-be). These properties are used to instantiate the domain knowledge templates. For more information regarding eliciting, modeling, and using domain knowledge, we refer to [14].

Phase 3- solution identification & analysis The first two phases are concerned with the activities accommodated in the requirements engineering (known as *problem peak* in the Twin Peaks model). Exploring the solution space for finding security strategies

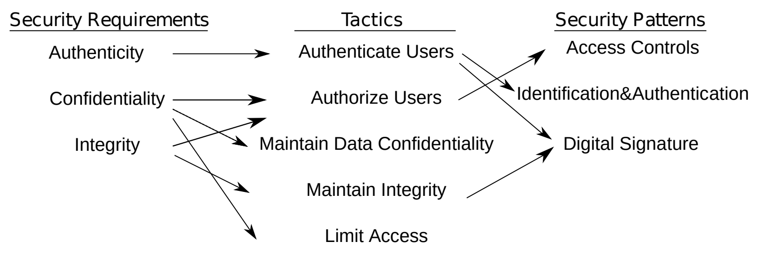


Fig. 3. Relationship between security requirements, tactics, and security patterns

to achieve security requirements is the aim of the third phase (see 3 in Fig. 2). Tactics proposed by Bass et al. [17] provide first basic approaches for the achievement of quality requirements, in particular, security requirements. Such general solutions are proved and recognized as coarse-grained solutions that help the achievement of quality requirements to some extent. Figure 3 shows a synopsis of the relationship between the artifacts in requirements engineering, namely requirements, and the artifacts in architectural design, namely coarse-grained solutions (tactics) and more specific solutions (patterns). It depicts how security requirements are mapped to security tactics (arrows on the left-hand side)[1]. Each security requirement is mapped to one or more tactics. From the tactics, we select the one that can realize the security requirement in a high-level manner. Furthermore, Figure 3 represents the mapping of tactics to more specific ones, namely security patterns. We make use of such mapping to refine the selected tactic further by selecting the appropriate security pattern in the next phase.

Phase 4- solution selection & application This phase (see 4 in Fig. 2) is concerned with selecting the appropriate security pattern such as *encryption* and *digital signature* that supports the achievement of security requirements. To this end, we make use of the relationship between tactics and security patterns depicted on the right-hand side of Fig. 3. Each tactic is mapped to one or more security patterns. The selected security pattern implements the corresponding tactic. After selecting the appropriate security pattern, we look for the counterpart *problem-oriented security pattern* such as *problem-oriented digital signature pattern*, which can be applied to refine the requirement models.

Phase 5- quality-based architecture derivation In the previous phases, we refined and restructured the problem descriptions by applying problem-oriented security patterns. The so enhanced requirement models contain first security solution approaches. This facilitates deriving a high-level architecture description from the requirement models, which can be further refined by applying appropriate security and design patterns.

In this phase (see 5 in Fig. 2), we define an architecture description that is oriented on the decomposition of the overall software development problem into subproblems represented as problem diagrams. The architecture consists of one component for the overall machine, with the stereotype ≪`machine`≫. For a distributed architecture, we add the stereotype ≪`distributed`≫ to the architecture component. For client-server architectures, there are two components representing client and server, respectively, inside the overall machine. Each component has to be annotated with the stereotype ≪`local`≫. Then we make use of the problem diagrams. The derivation

[1]Note that the figure does not provide a complete list of security requirements, security solutions, and security patterns. It only serves as an example to show how these artifacts are related to each other.

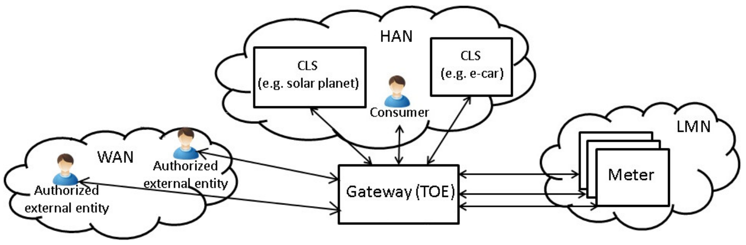


Fig. 4. The context of a smart grid system based on [18]

of the software architecture is achieved by mapping domains from problem diagrams to components in the software architecture. Each submachine in the problem diagrams becomes a component in the architecture. We annotate components with the stereotype ≪`component`≫. This provides support for a seamless integration of security patterns into software architecture.

4 Application Example

To illustrate the application of our method, we use the real-life example of smart grids. As sources for real functional and quality requirements we consider diverse documents such as "Application Case Study: Smart Grid" and "Smart Grid Concrete Scenario" provided by the industrial partners of the EU project NESSoS[2] and the "Protection Profile for the Gateway of a Smart Metering System" [18] provided by the German Federal Office for Information Security[3] and "Requirements of AMI" [19] provided by the EU project OPEN meter[4].

4.1 Introduction to the Case Study "Smart Grid"

To use energy in an optimal way, smart grids make it possible to couple the generation, distribution, storage, and consumption of energy. Smart grids use Information and Communication Technologies (ICT), which allow for financial, informational, and electrical transactions. Figure 4 shows the simplified context of a smart grid system based on [18].

To achieve the goals of the system, 20 use cases are necessary, from which we consider only the use case *Meter Reading for Billing*, as treating all 20 use cases would go beyond the scope of this paper. This use case is concerned with gathering, processing, and storing meter readings from smart meters for the billing process. Table 4.1 shows an excerpt of terms specific to the smart grid domain taken from the protection profile that are relevant to understand the requirements. Detailed description of the example smart grid is described in [9].

The protection profile states that "the Gateway is responsible for handling Meter Data. It receives the Meter Data from the Meter(s), processes it, stores it and submits it to external parties". Therefore, we define the requirements *RQ1-RQ3* to receive, process, and store meter data from smart meters. The requirement *RQ4* is concerned with submitting meter data to authorized external entities.

[2]http://www.nessos-project.eu/
[3]www.bsi.bund.de
[4]http://www.openmeter.com/

Table 2. An excerpt of relevant terms for the smart grid

Gateway	represents the central communication unit in a *smart metering system*. It is responsible for collecting, processing, storing, and communicating *meter data*.
Meter data	refers to meter readings measured by the meter regarding consumption or production of a certain commodity.
Smart meter	represents the device that measures the consumption or production of a certain commodity and sends it to the gateway.
Authorized external entity	could be a human or IT unit that communicates with the gateway from outside the gateway boundaries through a *Wide Area Network (WAN)*.
WAN	WAN provides the communication network that interconnects the gateway with the outside world.

Table 3. Requirements RQ1-RQ4 and security requirements related to RQ4

Requirement	Description	Related functional requirement
RQ1	Smart meter gateway shall receive meter data from smart meters	-
RQ2	Smart meter gateway shall process meter data from smart meters	-
RQ3	Smart meter gateway shall store meter data from smart meters	-
RQ4	Smart meter gateway shall submit processed meter data to authorized external Entities	-
...	...	...
RQ10	Integrity of data transferred in the WAN shall be protected	RQ4
RQ11	Confidentiality of data transferred in the WAN shall be protected	RQ4
RQ12	Authenticity of data transferred in the WAN shall be protected	RQ4
...	...	...

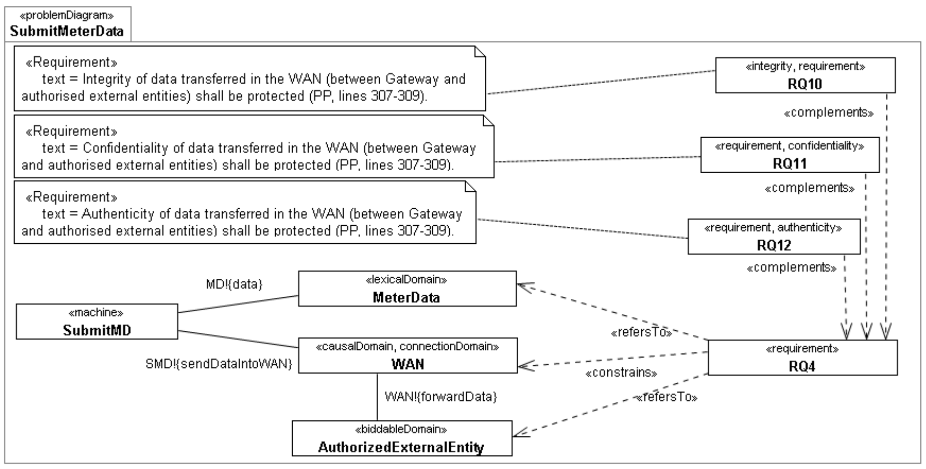
In the smart grids case study, security requirements have to be taken into account. A smart grid involves a wide range of data that should be treated in a secure way. Protection profile defines security objectives and requirements for the central communication unit in a smart metering system. To ensure security of meter data, the protection profile [18, pp. 18, 20] demands protection of data from unauthorized disclosure while transmitted to the corresponding external entity via the WAN (*RQ11*). The gateway shall provide the protection of authenticity and integrity when sending processed meter data to an external entity, to enable the external entity to verify that the processed meter data have been sent from an authentic gateway and have not been changed during transmission (*RQ10*, *RQ12*). Requirements with their descriptions are listed in Table 3.

4.2 Application of our Method to the Smart Grid

Applying our method to the case study smart grids implies the following activities and working results.

Phase 1- context elicitation & problem decomposition In this phase, we set up a *context diagram* to understand the problem and its context and *problem diagrams* to decompose the overall problem into subproblems, and annotate them with required security requirements. Due to the lack of space, we do not show the context diagram here. Figure 5 illustrates the problem diagram for describing the functional requirement *RQ4* annotated by the security requirements *RQ10*, *RQ11*, and *RQ12*.

In the original problem frames approach, the focus is on functional requirements. We extended the UML-based problem frames approach by providing a way to attach quality


«Requirement»
text = Integrity of data transferred in the WAN (between Gateway and authorised external entities) shall be protected (PP, lines 307-309).

«Requirement»
text = Confidentiality of data transferred in the WAN (between Gateway and authorised external entities) shall be protected (PP, lines 307-309).

«Requirement»
text = Authenticity of data transferred in the WAN (between Gateway and authorised external entities) shall be protected (PP, lines 307-309).

Fig. 5. Problem diagram for the requirement RQ4 and its related security requirements

requirements to problem diagrams [13]. We represent quality requirements as annotations in problem diagrams. Since UML lacks notations to specify and model quality requirements, we use specific UML profiles to add annotations to the UML models. We use a UML profile for dependability [11] to annotate problem diagrams with security requirements. For example, we apply the stereotypes ≪`integrity`≫, ≪`confidentiality`≫, and ≪`authenticity`≫ to represent integrity, confidentiality, and authenticity requirements as it is illustrated in Figure 5. In the problem diagram for submitting meter readings (Figure 5), the functional requirement *RQ4* is complemented by the following quality requirements: *RQ10* (integrity), *RQ11* (confidentiality), and *RQ12* (authenticity). For further phases of our method, we consider the security requirements *RQ10* and *RQ12*.

Phase 2- domain knowledge elicitation & modeling The achievement of security requirements requires additional resources such as computational power, which might affect the achievement of other quality requirements such as performance requirements negatively. The resource usage is affected by the strength of the security mechanism to be selected to achieve the corresponding security requirement. The kind of attacker and its characteristics determine the strength of the security mechanism. Therefore, we elicit and model the attacker and its characteristics as domain knowledge. For eliciting security-relevant domain knowledge, we have to instantiate the *Domain Knowledge Template* for each identified attacker once. We identify two network attackers for the two security requirements *RQ10* and *RQ12*. The reason is that the meter data to be transmitted through the network WAN can be manipulated by a network attacker. There is no information in the Protection Profile [18] about the attacker that the system must be protected against. Therefore, we assume that the system must be protected against the strongest attacker. Hence, we select for each property in the domain knowledge template for security the strongest one to obtain values for the column "Value". Table 4 shows the instantiated domain knowledge template for the integrity requirement *RQ10*. We model the network attacker explicitly as a biddable domain. Then, we apply the stereotype ≪`attacker`≫. We assign the attributes of the stereotype ≪`attacker`≫ using mapping provided by Table 4.

Table 4. Instantiated domain knowledge template for security

Quality: Security, Requirement: RQ10			
Elicitation Template			**Mapping to profile**
Domain Knowledge Description	**Possible Values**	**Value**	**Property (Dependability profile)**
Preparation time	one day, one week, two weeks, …	more than six months	Attacker.preparationTime
Attack time	one day, one week, two weeks, …	more than six months	Attacker.attackTime
Specialist expertise	laymen, proficient, expert, …	multiple experts	Attacker.specialistExpertise
Knowledge of the TOE	public, restricted, sensitive, critical	public	Attacker.knowledge
Window of opportunity	unnecessary/unlimited, easy, …	difficult	Attacker.opportunity
IT hardware/software or other equipment	standard, specialized, bespoke, …	multiple bespoke	Attacker.equipment

Phase 3- solution identification & analysis We introduced Fig. 3 in the previous section to show how security requirements, tactics, and security patterns are related to each other. This relationship allows us to identify the possible tactics and patterns for satisfying the security requirements *RQ10* and *RQ12*. For example, one tactic for supporting the achievement of the authenticity requirement is *Authenticate Users* which can be realized by applying the security patterns *Identification & Authentication* and *Digital Signature*. To achieve the integrity requirement, the tactics *Authorize Users* and *Maintain Integrity* can be selected. The tactic *Authorize Users* can be implemented by applying the security pattern *Access Controls*, whereas the tactic *Maintain Integrity* can be achieved by applying the security pattern *Digital Signature*. We decide for the security pattern *Digital Signature* as it can achieve both integrity and authenticity requirements. In the next phase, we apply the counterpart to the *Digital Signature* pattern for the requirements engineering phase, namely the *problem-oriented digital signature pattern*.

Phase 4- solution selection & application In phase four, we instantiate the *problem-oriented digital signature pattern*. First, we map the problem diagram shown in Fig. 5 to the first part of the graphical pattern described in Sect. 2.2. Figure 5 represents an instance of the first part of the *problem-oriented digital signature pattern*, in which the machine *SubmitMD* represents the machine *FunctionalM*, the domain *MeterData* represents the domain *Domain*, the functional requirement *RQ4* represents the functional requirement *FunctionalReq*, and the authenticity requirement *RQ12* represents the authenticity requirement *AuthenticityReq*. As mentioned in Sect. 2.2, depending on the functional requirement, the problem diagram might contain other domains that are not relevant for the security problem at hand. Hence, they are not represented in the pattern. The causal domain *WAN* and the biddable domain *AuthorizedExternalEntitiy* in Fig. 5 are examples for such a case.

Figure 6 illustrates the solution *digital signature* for the authenticity/integrity problem (second part of the pattern) and the composition problem diagram (third part of the pattern). We instantiate the second part of the pattern. Doing this, we obtain the upper problem diagram, in which the domain *MeterData* represents the domain *Domain*, the domain *PrivateKey* represents the domain *PrivateKey*, the authenticity requirement *RQ10* represents the authenticity requirement *authenticityReq*, and the machine *SigningMachine* represents the machine *DigSigM* in the pattern. As described in Sect. 2.2, the problem-oriented digital signature pattern provides a solution for the integrity problem as well. Thus, we model the authenticity requirement *RQ12* as well as the integrity requirement *RQ10* in Fig. 6. In this problem diagram, the machine *SigningMachine*

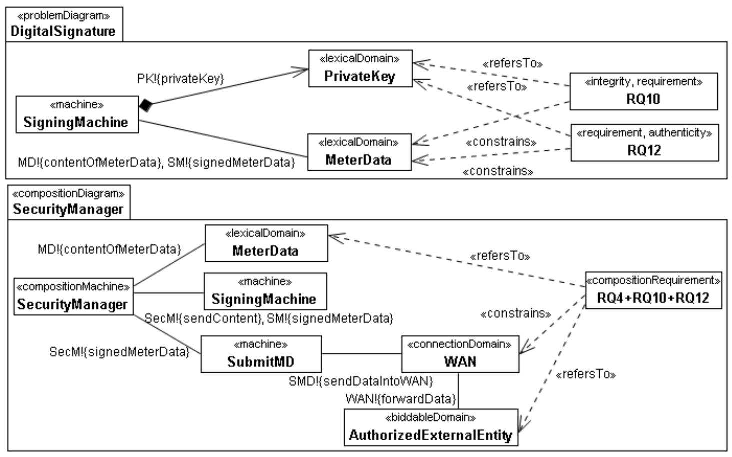


Fig. 6. Instance of the second and third parts of the *problem-oriented digital signature pattern*

receives the key (*privateKEy*) from the domain *PrivateKey* and the data (*contentOfMeterData*) from the domain *MeterData* and signs it by producing the *signedMeterData* using a signing algorithm in the machine *SigningMachine*.

We instantiate the third part of the pattern by mapping the domain *Domain* to the domain *MeterData*, the machine *DigSigM* to the machine *SigningMachine*, the machine *FunctionalM* to the machine *SubmitMD*, the composition requirement *Req* to the composition requirement *RQ4+RQ10+RQ12*, and the composition machine *CompositionM* to the composition machine *SecurityManager*. Doing this, we obtain the lower problem diagram, in which the problem diagram represented in Fig. 5 and the upper problem diagram in Fig. 6 are combined. The machine *SecurityManager* receives the *contentOfMeterData* from the domain *MeterData* and sends it to the machine *SigningMachine*. The machine *SigningMachine* signs the data and sends the *signedMeterData* back to the machine *SecurityManager*. This machine sends the signed data to the machine *SubmitMD*, which sends the data to the *AuthorizedExternalEntity* through the connection domain *WAN*.

In this way, we successfully instantiated and applied the problem-oriented digital signature pattern in phase four of our method in order to achieve the integrity requirement *RQ10* and the authenticity requirement *RQ12* for the functional requirement *RQ4*. We only showed the instantiation of the graphical part of the pattern. The corresponding template can be easily filled out using the information obtained from the solution space and according to the responsibility for the achievement of necessary conditions. As mentioned before, the templates as part of the problem-oriented security patterns describe the effect of each pattern on the requirement models. According to this part of the templates, we have to update the requirement models including domain knowledge (first and second phases of the method) by selecting and applying a problem-oriented security pattern. Therefore, our method has to be conducted as a recursive one to obtain correct requirements and a sound architecture.

Phase 5- quality-based architecture derivation In phase five, we set up an architecture by transforming the machines in the problem diagrams into components in the architecture. The architecture represented as a composite structure diagram in UML is shown in Fig. 7. The component for the overall machine *Gateway* has the stereotype

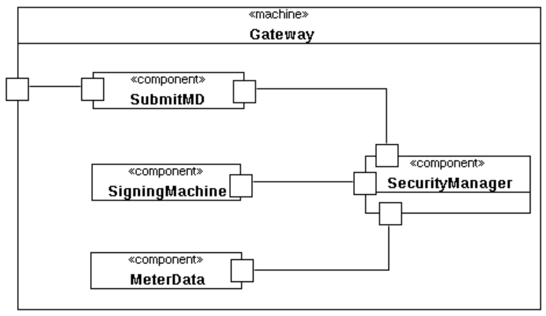


Fig. 7. Part of the architecture for the smart meter gateway

≪machine≫. Within this component, there exist components *SubmitMD*, *Security-Manager*, and *SigningMachine*, which are machines in the problem diagram *Security-Manager* shown in Fig. 6. In addition, we have to transform the lexical domains in the problem diagrams into components in the architecture. The reason is that lexical domains usually are data representations, thus they have to be part of the architecture. The component *MeterData* is such an example. The component *SubmitMD* is connected via *WAN* to outside of the smart meter gateway to *AuthorizedExternalEntity* as it is illustrated in the problem diagram *SecurityManager* shown in Fig. 6. Figure 7 represents only the part of the overall architecture which corresponds to the functional requirement *RQ4* and its related security requirements. For other requirements, we follow the same principle. Doing this, we obtain a full architecture for the smart meter gateway. Note that this architecture is a coarse-grained one. It still needs to be refined by applying security patterns corresponding to the problem-oriented security patterns and design patterns.

Benefits. The proposed method allows software engineers not only to think about security problems as early as possible in the software development life cycle, but also to think about solution approaches solving such security problems. By exploring the solution space, we find appropriate solution mechanisms and patterns, which can be used as *problem-oriented security patterns* for refining security requirement models in the requirement engineering phase. The corresponding templates represent consequences of applying such solution approaches by providing new assumptions and/or requirements to be considered when deciding on a specific pattern. The elaborated security requirement models can easily be transformed into a particular security pattern at the design level. Thus, problem-oriented security patterns support bridging the gap between security problems and security solutions. Note that *problem-oriented security patterns* do not replace the application of "classical" security patterns. *Problem-oriented security patterns* are located in the problem space aiming at restructuring and elaborating security problems while "classical" security patterns are accommodated in the solution space aiming at solving security problems. *Problem-oriented security patterns* in the requirements engineering phase represent the counterpart to "classical" security patterns in the design phase.

5 Related Work

Investigating early phases of software development, namely requirements analysis and architectural design, has been subject to research in related work for many years. This investigation however has been conducted for each phase separately and in isolation.

Little attention has been given to investigating the relationship between requirements phase and architecture phase. Hence, as related work, we discuss approaches that relate requirements and architectures with regard to security.

The security Twin Peaks model [20] provides a framework for developing security in the requirements and the architectural artifacts in parallel. Taking architectural security patterns into account, the model supports the elaboration of the problem and the solution artifacts by considering feedback coming from the counterpart peak. This framework, which is an elaboration of the original Twin Peaks model [2] in the context of security, is similar to our approach regarding the refinement of security requirements in the problem domain. However, the authors make use of the existing methods for each peak separately, namely the goal-oriented requirements engineering approach KAOS [21] to represent security requirements in the problem peak and Attribute Driven Design [17] for the solution peak.

Haley et al. [22] propose an approach to derive security requirements, which are expressed as constraints on the functional requirements. Threat descriptions are treated as crosscutting concerns that impact functional requirements represented as problem frames. Composing threat descriptions with problem diagrams reveal vulnerabilities that can be reduced by adding appropriate security requirements to the corresponding problem diagram. This approach uses problem frames as a basis for identifying vulnerabilities and deriving security requirements. Hence, it can be used complementary to our method as a step prior to our method.

Similar to our work, a method to bridge the gap between security requirements and the design is proposed in [23]. This method introduces new security patterns at the requirements and the design level, in contrast to our approach that reuses the existing security design patterns at the requirements level by applying problem-oriented security patterns. Alebrahim et al. [24] provide an aspect-oriented method for developing secure software. It treats security requirements as crosscutting concerns and refines them by using security patterns. Similar to our approach in this paper, it uses problem frames as a basis for requirements engineering. In contrast, we apply problem-oriented security patterns in this paper, which systematically adopt the notion of security patterns for requirements engineering.

Rapanotti et al. [25] propose an approach similar to our work by introducing *Architectural Frames*. They use architectural styles [26] located in the solution space to support the decomposition and recomposition of functional models within the problem frames framework. The *Pipes and filters* is selected to solve the transformation problems, represented in the problem frame approach by the *transformation frame*. Another architectural style is the Model-View-Controller (MVC) that is considered to solve the control problem, captured by the *commanded behaviour frame*.

Hall et al. [27] present an extension of the problem frames approach allowing design concerns. They suggest to benefit from architectural support in the problem domain by annotating the machine domain with them. However, it does not become clear how the architectural descriptions can be used explicitly in order to structure the machine domain.

These two approaches provide suggestions to use architectural knowledge in the problem domain based on the problem frames concept. However, they do not propose a systematic method on how to find the appropriate architectural solution and how to integrate it in the problem domain explicitly. Further, they only consider functional requirements and not security requirements. From this point of view, these approaches can be considered as complementary to our work.

6 Conclusions and Future Work

We have proposed a systematic method based on the problem frames approach for the iterative development of requirements analysis and software architectures with focus on achieving security requirements. The method provides an instantiation of the Twin Peaks model in the context of security proposing explicit steps that takes into account the intertwining nature of requirements and architectures. The iterative nature of our method as an instance of the Twin peaks model facilitates considering the feedback, which arise in one peak, in the another peak. Our method exploits the knowledge gained in the solution domain in a systematic way to refine the security requirements in the problem space by applying *problem-oriented security patterns*.

To summarize, we have proposed a pattern-based method that

- provides guidance on how requirements analysis and software architectures can be developed incrementally and iteratively to achieve security requirements.
- allows software engineers not only to think about security problems as early as possible in the software development life cycle, but also to think about solution approaches solving such security problems.
- bridges the gap between security problems and security solutions.

In the present work, we applied problem-oriented security patterns, which support the achievement of security requirements as early as possible in the software development life cycle. But not all kinds of security requirements can be achieved by enforcing security patterns. For example, there exist no patterns for *non-repudiation*, *anonymity*, and *privacy* in literature [28]. In the future, we will extend our method to also support the systematic selection of security mechanisms to refine all security requirements in the problem domain. Additionally, we strive for extending our method to support the achievement of not only security requirements in an iterative software development process, but also other quality requirements such as performance by using the architectural knowledge such as generic solutions, patterns, and mechanisms.

References

1. L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-functional requirements in software engineering*. Klewer Academic, 2000.
2. B. Nuseibeh, “Weaving together requirements and architectures,” *IEEE Computer*, vol. 34, no. 3, pp. 115–117, 2001.
3. M. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. Heimdahl, and S. Rayadurgam, “Your ”What” Is My ”How”: Iteration and Hierarchy in System Design,” *IEEE Software*, vol. 30, no. 2, pp. 54–60, 2013.
4. M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security patterns: integrating security and systems engineering*. John Wiley & Sons, 2005.
5. A. Alebrahim and M. Heisel, “Problem-oriented Security Patterns for Requirements Engineering,” in *Proc. of the 19th European Conf. on Pattern Languages of Programs (EuroPLoP)*. Universitätsverlag Konstanz, 2014, accepted.
6. M. Jackson, *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.
7. ”UML Revision Task Force”, *OMG Unified Modeling Language (UML), Superstructure*, http://www.omg.org/spec/UML/2.3/Superstructure/PDF.
8. D. Hatebur and M. Heisel, “Making Pattern- and Model-Based Software Development more Rigorous,” in *ICFEM’10*. Springer Verlag, 2010, pp. 253–269.

9. A. Alebrahim, C. Choppy, S. Faßbender, and M. Heisel, "Optimizing functional and quality requirements according to stakeholders' goals," in *System Quality and Software Architecture (SQSA)*. Elsevier, 2014, pp. 75–120.
10. A. Alebrahim, D. Hatebur, and M. Heisel, "A method to derive software architectures from quality requirements," in *Proc. of the 18th Asia-Pacific Software Engineering Conf. (APSEC)*, T. D. Thu and K. Leung, Eds. IEEE Computer Society, 2011, pp. 322–330.
11. D. Hatebur and M. Heisel, "A UML profile for requirements analysis of dependable software," in *Proc. of the Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP)*, ser. LNCS 6351, E. Schoitsch, Ed. Springer, 2010, pp. 317–331.
12. A. Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.
13. A. Alebrahim, D. Hatebur, and M. Heisel, "Towards systematic integration of quality requirements into software architecture," in *Proc. of the 5th Europ. Conf. on Software Architecture (ECSA)*, ser. LNCS 6903, I. Crnkovic and V. Gruhn, Eds. Springer, 2011, pp. 17–25.
14. A. Alebrahim, M. Heisel, and R. Meis, "A structured approach for eliciting, modeling, and using quality-related domain knowledge," in *Proc. of the 14th Int. Conf. on Computational Science and Its Applications (ICCSA)*, ser. LNCS 8583. Springer, 2014, pp. 370–386.
15. A. Lamsweerde, "Reasoning about alternative requirements options," in *Conceptual Modeling: Foundations and Applications*, A. Borgida, V. Chaudhri, P. Giorgini, and E. Yu, Eds. Springer, 2009, vol. LNCS 5600, pp. 380–397.
16. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), "Common Evaluation Methodology 3.1," ISO/IEC 15408, 2009.
17. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, SEI Series in Software Engineering*. Addison Wesley, 2003.
18. H. Kreutzmann, S. Vollmer, N. Tekampe, and A. Abromeit, "Protection profile for the gateway of a smart metering system," BSI, Tech. Rep., 2011.
19. G. Remero, F. Tarruell, G. Mauri, A. Pajot, G. Alberdi, M. Arzberger, R. Denda, P. Giubbini, C. Rodrguez, E. Miranda, I. Galeote, M. Morgaz, I. Larumbe, E. Navarro, R. Lassche, J. Haas, A. Steen, P. Cornelissen, G. Radtke, C. Martnez, A. Orcajada, H. Kneitinger, and T. Wiedemann, "D1.1 Requirements of AMI," OPEN meter project, Tech. Rep., 2009.
20. T. Heyman, K. Yskout, R. Scandariato, H. Schmidt, and Y. Yu, "The security twin peaks," in *Proc. of the Int. Symposium on Engineering Secure Software and Systems (ESSoS)*, ser. LNCS 6542. Springer, 2011, pp. 167–180.
21. A. van Lamsweerde., *Requirements Engineering: From System Goals to UML Models to Software Specifications*. John Wiley Sons, 2009.
22. C. B. Haley, R. C. Laney, and B. Nuseibeh, "Deriving security requirements from crosscutting threat descriptions," in *Proc. of the 3rd int. conf. on Aspect-oriented software development (AOSD)*. USA: ACM, 2004, pp. 112–121.
23. T. Okubo, H. Kaiya, and N. Yoshioka, "Effective Security Impact Analysis with Patterns for Software Enhancement," in *Proc. of the 6th Int. Conf. on Availability, Reliability and Security (ARES)*, 2011, pp. 527–534.
24. A. Alebrahim, T. T. Tun, Y. Yu, M. Heisel, , and B. Nuseibeh, "An aspect-oriented approach to relating security requirements and access control," in *Proc. of the CAiSE Forum*, ser. CEUR Workshop Proceedings, vol. 855. CEUR-WS.org, 2012, pp. 15–22.
25. L. Rapanotti, J. G. Hall, M. Jackson, and B. Nuseibeh, "Architecture-driven problem decomposition," in *Proc. of the 12th IEEE Int. Requirements Engineering Conf. (RE)*, 2004, pp. 80–89.
26. M. Shaw and G. Garlan, *Software Aechitecture: Perspectives on an emerging discipline*. Prentice Hall, 1996.
27. J. Hall, M. Jackson, R. Laney, B. Nuseibeh, and L. Rapanotti, "Relating software requirements and architectures using problem frames," in *Proc. of the IEEE Joint Int. Conf. on Requirements Engineering*, 2002, pp. 137–144.
28. K. Yskout, T. Heyman, R. Scandariato, and W. Joosen, "A system of security patterns," K.U.Leuven, Department of Computer Science, Report CW 469, 2006.