A Pattern-Based Method to Develop Secure Software

Holger Schmidt[1], Denis Hatebur[1,2], Maritta Heisel[1]

[1]*University Duisburg-Essen, Faculty of Engineering, Department of Computer Science and Applied Cognitive Science, Workgroup Software Engineering, Germany*

[2]*ITESYS Institut für technische Systeme GmbH, Dortmund, Germany*

ABSTRACT

We present a security engineering process based on *security problem frames* and *concretized security problem frames*. Both kinds of frames constitute patterns for analyzing security problems and associated solution approaches. They are arranged in a pattern system that makes dependencies between them explicit. We describe step-by-step how the pattern system can be used to analyze a given security problem and how solution approaches can be found.

Afterwards, the security problems and the solution approaches are formally modeled in detail. The *formal models* serve to prove that the solution approaches are correct solutions to the security problems. Furthermore, the formal models of the solution approaches constitute a formal specification of the software to be developed.

Then, the specification is implemented by *generic security components* and *generic security architectures*, which constitute architectural patterns. Finally, the generic security components and the generic security architecture that composes them are refined and the result is a secure software product built from existing and/or tailor-made security components.

KEYWORDS

security requirements engineering, patterns, problem frames, security components, security architecture, formal specification

1. INTRODUCTION

It is acknowledged that a thorough requirements engineering phase is essential to develop a software product that matches the specified requirements. This is especially true for *security requirements*.

We introduce a security engineering process that focuses on the early phases of software development. The process covers engineering of security requirements, security specifications, and security architectures. The basic idea is to make use of special *patterns* for security requirements analysis and development of security architectures.

Security requirements analysis makes use of patterns defined for structuring, characterizing, and analyzing *problems* that occur frequently in security engineering. Similar patterns for functional requirements have been proposed by Jackson (2001). They are called *problem frames*. Accordingly, our patterns are named *security problem frames*. Furthermore, for each of these frames, we have defined a set of *concretized security problem frames* that take into account

generic security mechanisms to prepare the ground for solving a given security problem. Both kinds of patterns are arranged in a pattern system that makes dependencies between them explicit. We describe how the pattern system can be used to analyze a given security problem, how solution approaches can be found, and how dependent security requirements can be identified.

Security specifications are constructed using the formal specification language CSP (Communicating Sequential Processes) by Hoare (1986). We present a procedural approach to construct formal CSP models for instances of security problem frames and concretized security problem frames. These models serve to formally express security requirements. Afterwards they are used to formally prove a *refinement* between the CSP model of a security problem frame instance and a corresponding CSP model of a concretized security problem frame instance. This refinement must *preserve the security requirements* to ensure that the constructed specification realizes the security requirements.

Once we have shown that the selected generic security mechanisms solve the security problems, we develop a corresponding security architecture based on platform-independent *generic security components* and *generic security architectures.* Each concretized security problem frame is equipped with a set of generic security architectures that represent the internal structure of the software to be built by means of a set of generic security components. After a generic security architecture and generic security components are selected, the latter must be refined to platform-specific security components. For example, existing component frameworks can be used to construct a platform-specific security architecture that realizes the initial security requirements.

The rest of the chapter is organized as follows: First, we introduce problem frames and present a literature review. Second, we give an overview of our security engineering process. Then we present the different development phases of the process in detail. Each phase of our process is demonstrated using the example of a secure text editor application. Finally, we outline future research directions and give a summary and a discussion of our work.

2. BACKGROUND

In the following, we first present problem frames and second, we discuss our work in the context of other approaches to security engineering.

2.1 Problem Frames

Patterns are a means to reuse software development knowledge on different levels of abstraction. They classify sets of software development problems or solutions that share the same structure. Patterns are defined for different activities at different stages of the software life-cycle. *Problem frames* by Jackson (2001) are a means to analyze and classify software development problems. *Architectural styles* are patterns that characterize software architectures (for details see (Bass & Clements & Kazman, 1998) and (Shaw & Garlan (1996)). *Design patterns* by Gamma, Helm, Johnson, and Vlissides (1995) are used for finer-grained software design, while *idioms* by Coplien (1992) are low-level patterns related to specific programming languages.

Using patterns, we can hope to construct software in a systematic way, making use of a body of accumulated knowledge, instead of starting from scratch each time. The problem frames defined by Jackson (2001) cover a large number of software development problems, because they are quite general in nature. Their support is of great value in the area of software engineering for years. Jackson (2001) describes them as follows: „A problem frame is a kind of pattern. It defines

an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces, and requirement." (p. 76). Jackson introduces five basic problem frames named *required behaviour*, *commanded behaviour*, *information display*, *simple workpieces*, and *transformation*.

Problem frames are described by frame diagrams, which basically consist of rectangles and links between these. As an example, Figure 1 shows the frame diagram of the problem frame *simple workpieces*.

The task is to construct a *machine* that improves the behavior of the environment it is integrated in.

Plain rectangles denote problem domains (that already exist), and a rectangle with a double vertical stripe denotes the machine to be developed. Requirements are expressed as a dashed oval, which contains an informal description of the requirements. The connecting lines represent interfaces that consist of shared phenomena. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain. For example, if a user types a password to log into an IT-system, this is a phenomenon shared by the user and the IT-system. It is controlled by the user. A dashed line between the dashed oval that contains the requirements description and a domain represents a requirements reference. This means that the requirements description refers to the domain. An arrow at the end of a requirements reference indicates that the requirements constrain the domain. Such a constrained domain is the core of any problem description, because it has to be controlled according to the requirements. Hence, a constrained domain triggers the need for developing a new software (the machine), which provides the desired control.

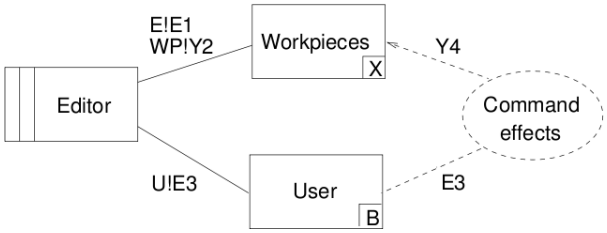

Figure 1: Simple Workpieces Problem Frame

Furthermore, Jackson distinguishes causal domains that comply with some physical laws, lexical domains that are data representations, and biddable domains that are usually people. Côté & Hatebur & Heisel & Schmidt & Wentzlaff (2008) introduced display domains that represent output devices, e.g., video screens.

In the frame diagram depicted in Figure 1, a marker „X" indicates that the corresponding domain is a lexical domain and „B" indicates a biddable domain. A causal domain is indicated by „C" and a display domain is indicated by „D". The notation „E!E1" means that the phenomena of interface „E1" between the domains „Editor" (abbreviated „E") and „Workpieces" are controlled by the „Editor" domain.

Software development with problem frames proceeds as follows: first, the environment in which the machine will operate is represented by a context diagram. Like a frame diagram, a context diagram consists of domains and interfaces. However, a context diagram contains no requirements (see Figure 4 for an example). Then, the problem is decomposed into subproblems. If ever possible, the decomposition is done in such a way that the subproblems fit to given problem frames. To fit a subproblem to a problem frame, one must instantiate its frame diagram,

i.e., provide instances for its domains, phenomena, and interfaces. The instantiated frame diagram is called a *problem diagram*. Furthermore, relevant *domain knowledge* about the domains contained in the frame diagram must be elicited, examined, and documented. Domain knowledge consists of *facts* and *assumptions*. Facts describe fixed properties of the environment irrespective of how the machine is built, e.g., that a network connection is physically secured. Assumptions describe conditions that are needed, so that the requirements are accomplishable, e.g., we assume that a password selected by a user is not revealed by this user to other users.

Successfully fitting a problem to a given problem frame means that the concrete problem indeed exhibits the properties that are characteristic for the problem class defined by the problem frame. A problem can only be fitted to a problem frame if the involved problem domains belong to the domain types specified in the frame diagram. For example, the "User" domain of Figure 1 can only be instantiated by persons, but not for example by some physical equipment like an elevator.

Since the requirements refer to the environment in which the machine must operate, the next step consists in deriving a specification for the machine (see Jackson & Zave (1995) for details). The specification describes the machine and is the starting point for its construction.

2.2 Related Work

In this section, we discuss our work in connection with a selection of other approaches to engineering of security requirements, security specifications, and security architectures.

Security Requirements Engineering

To elicit security requirements, the threats to be considered must be analyzed. Lin & Nuseibeh & Ince & Jackson (2004) use the ideas underlying problem frames to define so-called anti-requirements and the corresponding abuse frames. The purpose of anti-requirements and abuse frames is to analyze security threats and derive security requirements. Hence, abuse frames and security problem frames complement each other.

Gürses & Jahnke & Obry & Onabajo & Santen & Price (2005) present the MSRA (formerly known as CREE) method for multilateral security requirements analysis. Their method concentrates on confidentiality requirements elicitation and employs use cases to represent functional requirements. The MSRA method can be useful to be applied in a phase of the security requirements engineering process that mainly precedes the application of security problem frames.

SREF - Security Requirements Engineering Framework by Haley & Laney & Moffett & Nuseibeh (2008) is a framework that defines the notion of security requirements, considers security requirements in an application context, and helps answering the question whether the system can satisfy the security requirements. Their definitions and ideas overlap our approach, but they do not use patterns and they do not give concrete guidance to identify and elicit dependent security requirements.

Moreover, there exist other promising approaches to security requirements engineering, such as the agent-oriented Secure Tropos methodology by Mouratidis & Giorgini (2007) and the goal-driven KAOS - Keep All Objectives Satisfied approach by van Lamsweerde (2004).

A comprehensive comparison of security requirements engineering approaches (including the one presented in this chapter) can be found in (Fabian, B. & Gürses, S. & Heisel, M. & Santen, T. & Schmidt, H., to appear).

Formal Security Specifications

Li & Hall & Rapanotti (2006) use an extended CSP version by Lai & Lai & Sanders (1997) to systematically derive a specification from requirements. Their work does not consider non-functional requirements such as security requirements. Furthermore, biddable domains are not formalized. Since biddable domains are used to model unpredictable parts of the environment (such as honest and malicious users), we believe that this is a key feature to security requirements engineering.

KAOS by van Lamsweerde (2004) addresses security requirements by means of anti-goals. A linear real-time temporal logic is used to formalize these goals. The goals and further ingredients such as domain properties as well as pre- and postconditions form patterns that can be instantiated and negated to describe anti-goals. Furthermore, Mouratidis & Giorgini (2007) added this formal approach to Secure Tropos.

Haley & Laney & Moffett & Nuseibeh (2008, 2004) consider the notion of a *trust assumption*: "… the requirements engineer trusts that some domain will participate 'competently and honestly' in the satisfaction of a security requirement in the context of the problem." (pp. 4) in their SREF approach. To decide whether a system can satisfy the security requirements, Haley & Laney & Moffett & Nuseibeh (2005) make use of structured informal and formal argumentation. A two-part argument structure for security requirement satisfaction arguments consisting of an informal and a formal argument is proposed. In combination with trust assumptions, satisfaction arguments facilitate showing that a system can meet its security requirements.

In a stepwise development process, it is essential that security requirements specified in a certain step are preserved in later steps. This concept corresponds to the *stepwise refinement* concept of formal methods. Moreover, *information flow properties* represent a class of security properties that can be used to formally express informal security requirements. In contrast to safety and liveness properties, information flow properties are generally not preserved under refinement. Mantel (2003) gives an overview of using information flow properties for security requirements specification and preserving information flow properties under refinement.

In contrast to our work, KAOS with anti-goals, Secure Tropos, and SREF with trust assumptions do not allow to express security requirements in terms of information flow properties. Thus, the refinement of security requirements to specifications is only considered based on informal techniques.

Security Architectures

Architectural patterns named architectural styles are introduced by Bass & Clements & Kazman (1998) and Shaw & Garlan (1996). These patterns do not consider security requirements, and they are not integrated in a security engineering process.

Similarly, the AFrames by Rapanotti & Hall & Jackson & Nuseibeh (2004) do not consider security requirements. These patterns correspond to the popular architectural styles Pipe-and-Filter and Model-View-Controller (MVC), which the authors apply to Jackson's problem frames for transformation and control problems.

Hall & Jackson & Laney & Nuseibeh & Rapanotti (2002) extend machine domains of problem diagrams by architectural considerations. They do not deal with security requirements and they do not derive software architectures explicitly. Instead, their extension of the problem frames approach allows one to gather architectural structures and services from the problem environment.

There exist several techniques to evaluate the security properties of architectures, e.g., formal proving of security properties (Moriconi & Qian & Riemenschneider & Gong, 1997), analysis by means of petri nets and temporal logics (Deng & Wang & Tsai & Beznosov, 2003), or evaluation of used security patterns (Halkidis & Tsantalis & Chatzigeorgiou & Stephanides, 2008).

Choppy & Hatebur & Heisel (2005, 2006) present architectural patterns for Jackson's basic problem frames. The patterns constitute layered architectures described by UML (Unified Modeling Language) composite structure diagrams (UML Revision Task Force, Object Management Group (OMG), 2007). The authors also describe how these patterns can be applied in a pattern-based software development process. Hatebur & Heisel (2005) describe similar patterns for security frames. These frames are comparable to security problem frames, which are enhancements of the original security frames presented in (Hatebur & Heisel, 2005). Compared to the architectural patterns presented in this chapter, the mentioned papers do not consider behavioral interface descriptions and operation semantics. Furthermore, only a vague general procedure to derive components for a specific frame diagram is given. Architectural patterns especially for the problem class of confidential data storage using encryption are not described. And as a last difference, a refinement to implementable architectures is not considered. Nevertheless, the papers by Choppy & Hatebur & Heisel (2005, 2006) and Hatebur & Heisel (2005) as well as the idea to systematically preserve quality requirements from early requirements engineering to software design presented by Schmidt & Wentzlaff (2006) constitute the basis for the enhancements presented in this chapter.

Patterns for Security Engineering

Patterns for security engineering are mainly used during the phase that follows the phases presented in this chapter, i.e., they are applied in fine-grained design of secure software. Many authors advanced the field of security design patterns for years, e.g., (Schumacher & Fernandez-Buglioni & Hybertson & Buschmann & Sommerlad, 2005) and (Steel & Nagappan & Lai, 2005). A comprehensive overview and a comparison of the different existing security design patterns is given by Scandariato & Yskout & Heyman & Joosen (2008). Fernandez & Larrondo-Petrie & Sorgente & Vanhilst (2007) propose a methodology to systematically use security design patterns during software development. The authors use UML (UML Revision Task Force, 2007) activity diagrams to identify threats to the system, and they use security design patterns during fine-grained design to treat these threats. Mouratidis & Weiss & Giorgini (2006) present an approach to make use of security design patterns that connects these patterns to the results generated by the Secure Tropos methodology by Mouratidis & Giorgini (2007).

The relation between our concretized security problem frames, which still express problems, and security design patterns is much the same as the relation between problem frames and design patterns: the frames describe problems, whereas the design/security patterns describe solutions on a fairly detailed level of abstraction. Furthermore, since security design patterns are more detailed than our generic security components and architectures, they can be applied after a composed generic security architecture is developed.

Furthermore, the security standard Common Criteria (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2009) and KAOS make use of patterns for security engineering. The Common Criteria introduces security functional requirements, which are textual patterns to express security mechanisms on an abstract level. They are comparable to concretized security problem frames. KAOS provides formal

patterns to describe security goals specified using a linear real-time temporal logic. These patterns can be compared to the effects of security problem frames.

3. OVERVIEW OF A SECURITY ENGINEERING PROCESS USING PATTERNS

We present in this chapter a *security engineering process using patterns* (SEPP). SEPP is an iterative and incremental process that consists of three phases. It follows a top-down and platform-independent approach until a generic security architecture is selected in phase three. Then, it takes a bottom-up and platform-specific approach to search for given security components that realize the generic security architecture.

Phase 1 – Security Requirements Analysis

This phase starts with an initial set of security requirements, which is analyzed in detail by incrementally and iteratively processing six analysis steps. The result of this phase is a consolidated set of security requirements including solution approaches.

Step 1 - Describe Environment The environment in which the software development problem is located is described in detail, developing a context diagram and expressing domain knowledge about the domains that occur in the context diagram. The domain knowledge describes the environment of the machine. This concerns especially potential attackers. The distinction of facts and assumptions is particularly important for security requirements. A machine usually cannot satisfy security requirements unconditionally. It can provide security mechanisms that contribute to system security, but cannot enforce system security on its own.

Step 2 - Select and Instantiate Security Problem Frames The software development problem is decomposed into smaller subproblems. The security-relevant subproblems are analyzed and documented based on security problem frames (SPF).

Step 3 - Select and Instantiate Concretized Security Problem Frames Generic solution approaches are selected for the previously documented security problems. The generic solution mechanisms are documented based on concretized security problem frames (CSPF).

Step 4 - Check for Related SPFs Based on a pattern system of SPFs and CSPFs, SPFs that are commonly used in combination with an already used CSPF can be found.

Step 5 - Analyze Dependencies Based on the pattern system, dependent security problems are identified, which can be either assumed to be already solved or they have to be considered as new security requirements to be solved by generic security mechanisms.

Step 6 - Analyze Possible Conflicts: the pattern system shows possible conflicts between security requirements and generic solution mechanisms. If a conflict is relevant, it must be resolved.

Phase 2 - Security Specifications

A formal CSP model is developed for each instantiated (C)SPF. These models are used to formally express the security requirements and to prove refinements that preserve the specified security requirements. The result is a set of formal behavioral security specifications of the machines and their environment. Moreover, the refinement proofs guarantee that the generic solution approaches selected in phase one are sufficient to realize the security requirements.

Step 1 - Construct Formal CSP Models A formal CSP model is constructed for each (C)SPF instance.

Step 2 - Formally Express Security Requirements The informally described security requirements are expressed formally based on the CSP models.

Step 3 - Show Security-Requirements Preserving Refinements For each SPF instance and the corresponding CSPF instance, the previously constructed CSP models are used to show that the CSPF CSP model refines the SPF CSP model. This refinement proof comprises the functional refinement and the preservation of the formally specified security requirement.

Phase 3 - Security Architectures

Generic security architectures are selected and realized using existing security components from APIs or component frameworks. The result of this phase is a platform-specific and implementable security architecture that realizes the machines of the instantiated CSPFs.

Step 1 - Select Generic Security Architectures A generic security architecture that consists of a set of generic security components is selected for each CSPF instance, based on domain knowledge and constraints of the application domain.

Step 2 – Combine Generic Security Architectures The selected generic security architectures are combined to a single generic security architecture based on relations between the CSPF instances.

Step 3 - Refine Generic Security Architecture The combined generic security architecture is refined to a platform-specific security architecture based on, e.g., existing security components.

Step 4 - Connect Security Components Glue code is written to connect the components according to the chosen generic security architecture.

The process is described as an *agenda* (Heisel, 1998) that summarizes the input development artifacts, the output development artifacts, and validation conditions for each step. Furthermore, each step is complemented by a method describing how to develop the output artifacts from the input artifacts.

4. USING PROBLEM FRAMES FOR SECURITY REQUIREMENTS ENGINEERING

We present security problem frames, concretized security problem frames, the pattern system, and the process steps for security requirements engineering. The described techniques are then applied to the secure text editor case study.

4.1 Security Problem Frames

Jackson (2001) states that his five basic problem frames are "... far from a complete or definitive set" (p. 76). To meet the special demands of software development problems occurring in the area of security engineering, we introduced *security problem frames* (Hatebur & Heisel & Schmidt, 2006). SPFs are a special kind of problem frames, which consider *security requirements*. Similarly to problem frames, SPFs are patterns. The SPFs we have developed strictly refer to the *problems* concerning security. They do not anticipate a solution. For example, we may require the confidential storage of data without being obliged to mention encryption, which is a means to achieve confidentiality. The benefit of considering security requirements without reference to potential solutions is the clear separation of problems from their solutions,

which leads to a better understanding of the problems and enhances the re-usability of the problem descriptions, since they are completely independent of solution technologies.

Each SPF consists of a name, an intent, a frame diagram with a set of predefined interfaces, an informal description, a security requirement template, and an effect. The latter is a formal representation of the security requirement template. Effects are expressed as formulas in Z notation (Spivey, 1992) based on a metamodel for problem frames developed by Hatebur & Heisel & Schmidt (2008). The metamodel formally specifies problem frames and problem frame constituents such as domains and interfaces by means of a UML class diagram (UML Revision Task Force, Object Management Group (OMG), 2007) and OCL (Object Constraint Language) constraints (UML Revision Task Force, Object Management Group (OMG), 2006). We use the instances of the classes of the metamodel as types for the formulas representing effects.

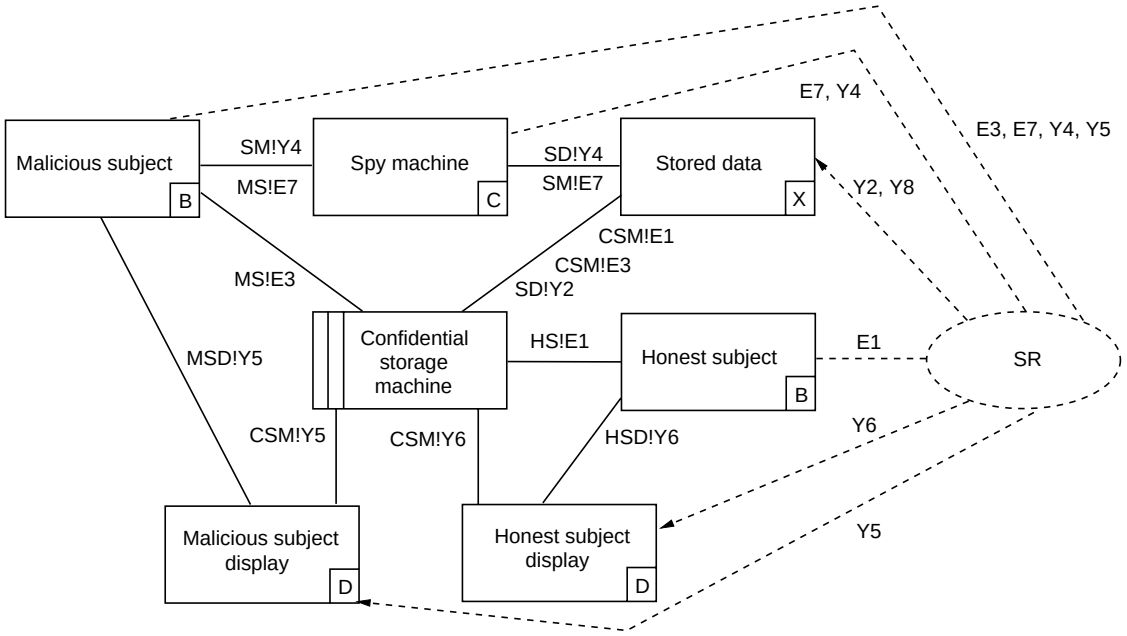

Figure 2: (C)SPF Confidential Data Storage (Using Password-Based Encryption)

As an example, we present in detail the *SPF confidential data storage*, which describes the problem class of confidentially storing data:

Name:

SPF confidential data storage

Intent: Conceal data (e.g., files, folders, metadata, etc.) stored on some storage device (e.g., hard disks, memory cards, smartcard, etc.).

Frame diagram:

Figure 2 shows the frame diagram of the SPF confidential data storage.

Predefined interfaces:

The interfaces of the SPF confidential data storage are defined as follows:

$E1 = \{OperationsOnStoredData_{HS}\}$

$Y2 = \{ContentOfStoredData\}$

$E3 = \{OperationsOnStoredData_{MS}\}$

$Y4 = \{Obervations_{SM}\}$

Y5 = {Obervations$_{CSM}$}
Y6 = {ContentOfStoredData, Observations}
E7 = {SpyOperations}
Y8 = {OperationsOnStoredData}

Informal Description:

The malicious environment is represented by the domains *Malicious subject, Spy machine, and Malicious subject display.* The domain *Stored data* represents the data to be protected against the malicious environment. The *Malicious subject* domain uses the interface MS!E7 (between *Malicious subject* and *Spy machine*) to spy (*SpyOperations*) on the *Stored data* domain. The interface SM!Y4 (between *Malicious subject* and *Spy machine*) is used by the *Malicious subject* domain to receive some observations (*Obervations$_{SM}$*), e.g., meta-information about *Stored data* such as its length or type, from the *Spy machine* domain. The *Spy machine* domain is connected directly to the *Stored data* domain via interfaces SD!Y4 and SM!E7 to represent that the *Malicious subject* domain is not restricted to only access the *Stored data* domain through the machine domain *Confidential storage machine.* For example, access to the *Stored data* domain can also be possible via the operating system.

The *Malicious subject* domain can execute some operations (*OperationsOnStoredData$_{MS}$*) on the *Stored data* domain using the machine domain via interface MS!E3 (between *Malicious subject* and *Confidential storage machine*). Similarly, the honest environment represented by the domains *Honest subject* and *Honest subject display* can execute some operations (*OperationsOnStoredData$_{HS}$*) on the *Stored data* domain using the machine domain via interface HS!E1 (between *Honest subject* and *Confidential storage machine*).

According to the commands from the (malicious or honest) environment, the machine accesses the domain *Stored data* via interfaces CSM!E1 and CSM!E3. The content of *Stored data* (*ContentOfStoredData*) is received by the machine domain using the interface SD!Y2. Afterwards, the content of *Stored data* and some observations (*Obervations*) is shown to the domain *Honest subject* using the *Honest subject display* domain (via interface CSM!Y6 between *Confidential storage machine* and *Honest subject display* and via interface HSD!Y6 between *Honest subject display* and *Honest subject*).

The domain *Malicious subject* can possibly make some observations (*Obervations$_{CSM}$*), e.g., meta-information about *Stored data* such as its length or type, using the *Malicious subject display* (via interface CSM!Y5 between *Confidential storage machine* and *Malicious subject display* and via interface MSD!Y5 between *Malicious subject display* and *Malicious subject*).

Security requirement template:

The security requirement template (SR) is described as follows: Preserve confidentiality of *Stored data* for honest environment (consisting of *Honest subject* and *Honest subject display*) and prevent disclosure to malicious environment (consisting of *Malicious subject, Spy machine,* and *Malicious subject display*).

Effect:

$HonestEnvironment : \mathbb{P}(HonestSubject \times HonestSubjectDisplay)$
$MaliciousEnvironment : \mathbb{P}(MaliciousSubject \times SpyMachine \times MaliciousSubjectDisplay)$

$$\forall\ cosd: ContentOfStoredData;\ he: HonestEnvironment;\ me: MaliciousEnvironment \bullet conf(cosd, he, me)$$

$\forall$ *sd*: *StoredData*; *he*: *HonestEnvironment*; *me*: *MaliciousEnvironment* •

conf(*sd*, *he*, *me*)

An honest environment consists of an honest subject and an honest subject display, whereas a malicious environment consists of a malicious subject, a spy machine, and a malicious subject display. The set of all honest environments *HonestEnvironment* is a set of pairs consisting of elements of the domains *Honest subject* and *Honest subject display*, as indicated by the powerset operator $\mathbb{P}$. Similarly, the set of all malicious environments *MaliciousEnvironment* is a set of triples consisting of elements of the domains *Malicious subject*, *Spy machine*, and *Malicious subject display*.

Informally speaking, the effect expresses that the confidentiality of the phenomenon *ContentOfStoredData* (see interfaces SD!Y2 between *Confidential storage machine* and *Stored data*, HSD!Y6 between *Honest subject display* and *Honest subject*, and CSM!Y6 between *Confidential storage machine* and *Honest subject display*) and of the domain *StoredData* is preserved for the *HonestEnvironment* and that disclosure by the *MaliciousEnvironment* is prevented.

To formally express this effect, we specify two versions of a relation *conf*. One version of *conf* deals with the confidentiality of a phenomenon, another version deals with the confidentiality of a lexical domain. We define *conf* as a set of triples of a phenomenon (or a lexical domain), an honest environment, and a malicious environment. Each triple describes that the confidentiality of the phenomenon (or of the lexical domain) is preserved for the honest environment and that disclosure by the malicious environment is prevented.

The universally quantified formulas that express the effect make use of the relation *conf*: the first formula expresses that the confidentiality of each possible instance *cosd* of the phenomenon *ContentOfStoredData* is preserved for each possible instance *he* of the *HonestEnvironment* and that disclosure by each possible instance *me* of the *MaliciousEnvironment* is prevented. The second formula expresses a similar condition for each possible instance *sd* of the lexical domain *StoredData*.

Further SPFs exist, e.g., *SPF distributing secrets* that represents the problem to deliver secrets such as a passwords and encryption keys to the correct recipients, *SPF authentication* that represents the problem to authenticate users or systems, *SPF integrity-preserving data transmission* that represents the problem to transmit data over an insecure channel in an integrity-preserving way, and several others. Hatebur & Heisel & Schmidt (2008) present an overview of the available SPFs.

4.2 Concretized Security Problem Frames

Solving a security problem is achieved by choosing generic security mechanisms (e.g., encryption to keep data confidential). The generic security mechanisms are represented by *concretized security problem frames* (CSPF).

Each CSPF consists of a name, an intent, a frame diagram with a set of predefined interfaces, an informal description, a concretized security requirement template, necessary conditions, and a list of related SPFs. The necessary conditions must be met by the environment for the generic security mechanism that the CSPF represents to be applicable. If the necessary conditions do not hold, the effect described in the according SPF cannot be established. The necessary conditions are expressed in Z notation. A *concretized security requirements template* refers to the effect

described in the according SPF and the necessary conditions. More precisely, it is expressed as an implication: if the necessary conditions hold, then the effect is established. The effects of the SPFs and the necessary conditions of the CSPFs serve to represent dependencies between SPFs and CSPFs explicitly. The list of related SPFs serves to exhibit security problems that often occur when the security mechanism represented by the CSPF at hand is applied.

As an example, we present in detail the *CSPF confidential data storage using password-based encryption*. This CSPF represents the generic security mechanism password-based encryption according to the password-based cryptography standard PKCS #5 v2.0 (RSA Laboratories, 1999), which can be used to solve problems that fit to the SPF confidential data storage problem class. Another CSPF that solves such a security problem is the *CSPF confidential data storage using encryption key-based encryption.*

Name:

CSPF confidential data storage using password-based encryption

Intent:

Conceal data (e.g., files, folders, metadata, etc.) stored on some storage device (e.g., hard disks, memory cards, smartcard, etc.) using a password-based encryption mechanism.

Frame diagram:

The frame diagram of the CSPF confidential data storage using password-based encryption is similar to the frame diagram of the SPF confidential data storage shown in Figure 2. For this reason, we do not explicitly show it here. Instead, we briefly describe the differences between the two frame diagrams: The domain *Stored data* is replaced by the domain *Encrypted stored data.* Furthermore, the usage of a password-based encryption mechanism leads to modifications of the interfaces (see informal description for details).

Predefined interfaces:

The interfaces of the CSPF confidential data storage using password-based encryption are defined as follows:

E1 = {$\text{OperationsOnEncryptedStoredData}_{HS}$, Password}

Y2 = {EncryptedContentOfStoredData}

E3 = {$\text{OperationsOnEncryptedStoredData}_{MS}$, WrongPassword}

Y4 = {EncryptedContentOfStoredData, Obervations_{SM}}

Y5 = {Obervations_{CSM}}

Y6 = {ContentOfStoredData, Observations}

E7 = {SpyOperations}

Y8 = {OperationsOnEncryptedStoredData}

Informal Description:

Since the domain *Stored data* is replaced by the domain *Encrypted stored data*, the interface SD!Y4 is replaced by the interface ESD!Y4. Compared to the SPF confidential data storage, the phenomenon *ContentOfStoredData* of the interfaces SD!Y2 (between *Confidential storage machine* and *Encrypted stored data*), ESD!Y4 (between *Encrypted stored data* and *Spy machine*), and SM!Y4 (between *Spy machine* and *Malicious subject*) is replaced by the phenomenon *EncryptedContentOfStoredData.* Accordingly, the phenomenon $\textit{OperationsOnStoredData}_{HS}$ of the interfaces HS!E1 (between *Honest subject* and *Confidential storage machine*) and CSM!E1 (between *Confidential storage machine* and *Honest subject*) is replaced by the phenomenon $\textit{OperationsOnEncryptedStoredData}_{HS}$. These interfaces additionally contain the phenomenon

Password that represents a password used by the honest environment for encryption and decryption. Furthermore, the phenomenon $OperationsOnStoredData_{MS}$ of the interfaces MS!E3 (between *Malicious subject* domain and *Confidential storage machine*) and CSM!E3 (between *Confidential storage machine* and *Malicious subject*) is replaced by the phenomenon $OperationsOnEncryptedStoredData_{MS}$. These interfaces additionally contain the phenomenon *WrongPassword* that represents a password used by the malicious environment for encryption and decryption.

Finally, the phenomenon *OperationsOnStoredData* of the phenomena set Y8 (at the requirement reference connected to *Encrypted stored data*) is replaced by the phenomenon *OperationsOnEncryptedStoredData.*

Concretized security requirement template:

The concretized security requirement template (CSR) is described as follows: If *Password* is unknown to malicious environment, then confidentiality of *Stored data* is preserved for honest environment and disclosure to malicious environment is prevented.

Necessary conditions:

$HonestEnvironment : \mathbb{P}(HonestSubject \times HonestSubjectDisplay)$

$MaliciousEnvironment : \mathbb{P}(MaliciousSubject \times SpyMachine \times MaliciousSubjectDisplay)$

$RightPasswords : \mathbb{P}\ Password$

$WrongPasswords : \mathbb{P}\ Password$

$$RightPasswords \cap WrongPasswords = \varnothing$$

$$\forall\ pwd: Password; he: HonestEnvironment; me: MaliciousEnvironment \bullet conf(pwd, he, me)$$

$$\forall\ pwd: Password; he: HonestEnvironment; me: MaliciousEnvironment \bullet int(pwd, he, me)$$

Passwords used by the honest environment must be different from the ones used by the malicious environment. Otherwise, a password-based encryption mechanism is not applicable. In practice, this necessary condition has to be assumed. That is, we have to assume that the malicious environment does not guess the right password. It cannot be fulfilled by a security mechanism. This necessary condition is formally expressed based on a set *RightPasswords* that represents the valid passwords chosen by the honest environment, and a set *WrongPasswords* that represents the invalid passwords chosen by the malicious environment. Thus, we formally express the necessary conditions by stating that these two sets are disjoint.

Furthermore, passwords used by the honest environment must not be known by the malicious environment. This necessary condition is formally described by the first universally quantified formula which expresses that the confidentiality of each possible instance *pwd* of the phenomenon *Password* is preserved for each possible instance *he* of the *HonestEnvironment* and that disclosure by each possible instance *me* of the *MaliciousEnvironment* is prevented.

Moreover, passwords used by the honest environment must be transmitted to the machine domain in an integrity-preserving way. To formally express this necessary condition, we specify a relation *int* as a set of triples of a phenomenon (or a lexical domain), an honest environment, and a malicious environment. Each triple describes that the integrity of the phenomenon (or of the lexical domain) is preserved for the honest environment and that modification by the malicious environment is prevented. Consequently, the second universally quantified formula expresses the

mentioned necessary condition using the relation *int*: the integrity of each possible instance *pwd* of the phenomenon *Password* is preserved for each possible instance *he* of the *HonestEnvironment* and that modification by each possible instance *me* of the *MaliciousEnvironment* is prevented.

Related SPFs:

- SPF integrity-preserving data storage

Further CSPFs exist, e.g., *CSPF distributing secrets using negotiation* that represents the generic security mechanism to deliver secrets using a negotiation mechanism, *CSPF authentication using passwords* that represents the generic security mechanism to authenticate users by passwords, *CSPF integrity-preserving data transmission using symmetric mechanism* that represents the generic security mechanism to transmit data over an insecure channel in an integrity-preserving way using a symmetric mechanism, and serveral others. Hatebur & Heisel & Schmidt (2008) present an overview of the available CSPFs.

4.3 Pattern System

We developed a catalog of SPFs and CSPFs. Both kinds of frames are arranged in a *pattern system* (Hatebur & Heisel & Schmidt, 2007), which indicates dependent, conflicting, and related frames. The pattern system is represented as a table and is partly shown in Table 1. The complete pattern system can be found in (Hatebur & Heisel & Schmidt, 2008).

The pattern system is constructed by analyzing the necessary conditions of the different CSPFs and the effects of the different SPFs. We check the necessary conditions of a CSPF and syntactically match them with the effects of all SPFs. For example, the necessary conditions of the CSPF confidential data storage using password-based encryption and the CSPF confidential data storage using encryption key-based encryption require integrity-preserving and confidential paths for the passwords and the encryption keys, respectively. The effect of the SPF integrity-preserving data transmission provides an integrity-preserving path and the effect of the SPF confidential data transmission provides a confidential path. For this reason, the mentioned CSPFs depend on the SPF integrity-preserving data transmission and the SPF confidential data transmission. Consequently, the rows that belong to these CSPFs in Table 1 are marked at the positions of the columns that belong to the SPF integrity-preserving data transmission and the SPF confidential data transmission with the letter “D”. Furthermore, the necessary conditions of the CSPF confidential data storage using encryption key-based encryption require that encryption keys must be distributed and that these encryption keys are confidentially stored. Thus, this CSPF depends on the SPF confidential data storage and on the SPF distributing secrets. The row that belongs to this CSPF in Table 1 is marked at the positions of the columns that belong to the SPF confidential data storage and SPF distributing secrets with the letter “D”.

The fact that a CSPF concretizes an SPF is represented in Table 1 by the letter “C”. For example, in the row of the CSPF confidential data storage using password-based encryption is a letter “C” at the position of the column of the SPF confidential data storage.

	…	SPF confidential data storage	SPF confidential data transmission	SPF integrity-preserving data storage	SPF integrity-preserving data transmission	SPF Distributing Secrets	…
⋮							
CSPF confidential data storage using password-based encryption		C	D	R	D		
CSPF confidential data storage using encryption key-based encryption		C, D	D	R	D	D	
⋮							

Table 1: (C)SPF Pattern System

Furthermore, the "Related" sections of the CSPFs are represented in Table 1 by the letter "R". The rows of the CSPFs confidential data storage using password-based encryption and using encryption-key based encryption are marked with the letter "R" at the positions of the column of the SPF integrity-preserving data storage. The "Related" sections are helpful, since they indicate at an early stage of software development possible security problems that commonly occur in combination with the generic solution mechanism at hand and the security problem it solves.
Additionally, possible interactions between generic security mechanisms represented as CSPFs and security requirements represented as SPFs are indicated by the pattern system. We do not discuss this part of the pattern system here. Hatebur & Heisel & Schmidt (2008) describe this part of the pattern system in detail based on a case study of a software to handle legal cases.

The (C)SPFs we developed form a self-contained pattern system: for any necessary condition of a CSPF covered by the pattern system, there exists at least one SPF contained in the pattern system that provides a matching effect. Therefore, the (C)SPFs contained in the pattern system can be used to *completely* analyze a given security problem, whose initial security requirement is covered by one of the frames.

The explicit knowledge of the dependencies between the security SPFs and their concretized counterparts increases the value of our approach. The guidance provided by the dependency relations of the pattern system helps to structure the security requirements engineering process, to avoid confusion, and to analyze security problems and their solution approaches in depth. Hence, the security requirements engineering process will result in a consolidated set of security requirements and solution approaches, which is complete with respect to the initial set of security requirements. Compared to the initial set of security requirements, the final set of security requirements additionally contains dependent and related security requirements that may not have been known initially.

4.4 Security Requirements Analysis Method

Figure 3 shows an overview of SEPP's security requirements analysis lifecycle. The arrows are annotated with inputs or with conditions (in square brackets). The latter must be true to proceed with the step the arrow under consideration is pointing at. The arrow pointing at the "End" state is annotated with SEPP's overall output of the first phase. Each of the steps is described according to the following template:

- Input: artifacts necessary to accomplish the step
- Output: artifacts that are created or modified during the execution of the step
- Validation conditions: necessary semantic conditions that an output artifact must fulfill in order to serve its purpose properly (Heisel, 1998)

Before we explain SEPP step-by-step, we note that some activities to be executed for a comprehensive security requirements analysis are not explicitly mentioned in the descriptions of SEPP's security requirements analysis steps. These activities concern the maintenance of the following development artifacts:

- **attacker model** describes assumptions about potential attackers. For example, the Common Evaluation Methodology (CEM) (International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2006) defines the attack potential of a potential attacker as a function of time, expertise, knowledge, and equipment. It also identifies two numeric values for each of these factors. The first value is for identifying and the second one is for exploiting a vulnerability. By assuming values for the input variables of the function, we can calculate the attack potential.
- **results from threat analysis** represent potential threats. For example, attack trees by Schneier (1999) can be applied.
- **results from risk analysis** represent the risk of an attack and the resulting loss. For example, a risk analysis method such as CORAS by Braber & Hogganvik & Stølen & Vraalsen (2007) can be applied.
- **glossary** contains all used names, type information (if applicable), and references to artifacts that contain the names.

These artifacts are initially constructed in SEPP's first step. After every refinement step, i.e., the steps from requirements analysis and specification to architectural and fine-grained design and finally implementation, new threats can arise. For example, the decision for a particular security mechanism, the definition of a certain length for cryptographic keys, and the usage of a specific security component of a component framework enlarge the attack surface of the system to be developed. Hence, the attacker model as well as the results from threat and risk analysis must be kept up-to-date in all phases of the software development life-cycle. Note that SEPP covers the early phases of software development, i.e., requirements analysis, specification, and design. Consequently, it does not cover the analysis of security problems arisen during fine-grained design and implementation. Nevertheless, the threats identified in the late phases of software development can result in new security requirements to which SEPP can be applied.

Moreover, in each step results from functional requirements analysis may be used to support the construction of the security-relevant artifacts. For example, a context diagram that emerged from functional requirements analysis can be used as a starting point for the construction of a context diagram that contains security-relevant entities and relations.

SEPP starts given a textually described software development problem with an initial set of security requirements *SR*. For example, the initial security requirements can be obtained by using the methods proposed by Gürses & Jahnke & Obry & Onabajo & Santen & Price (2005) and by Fernandez & la Red M. & Forneron & Uribe & Rodriguez G. (2007).

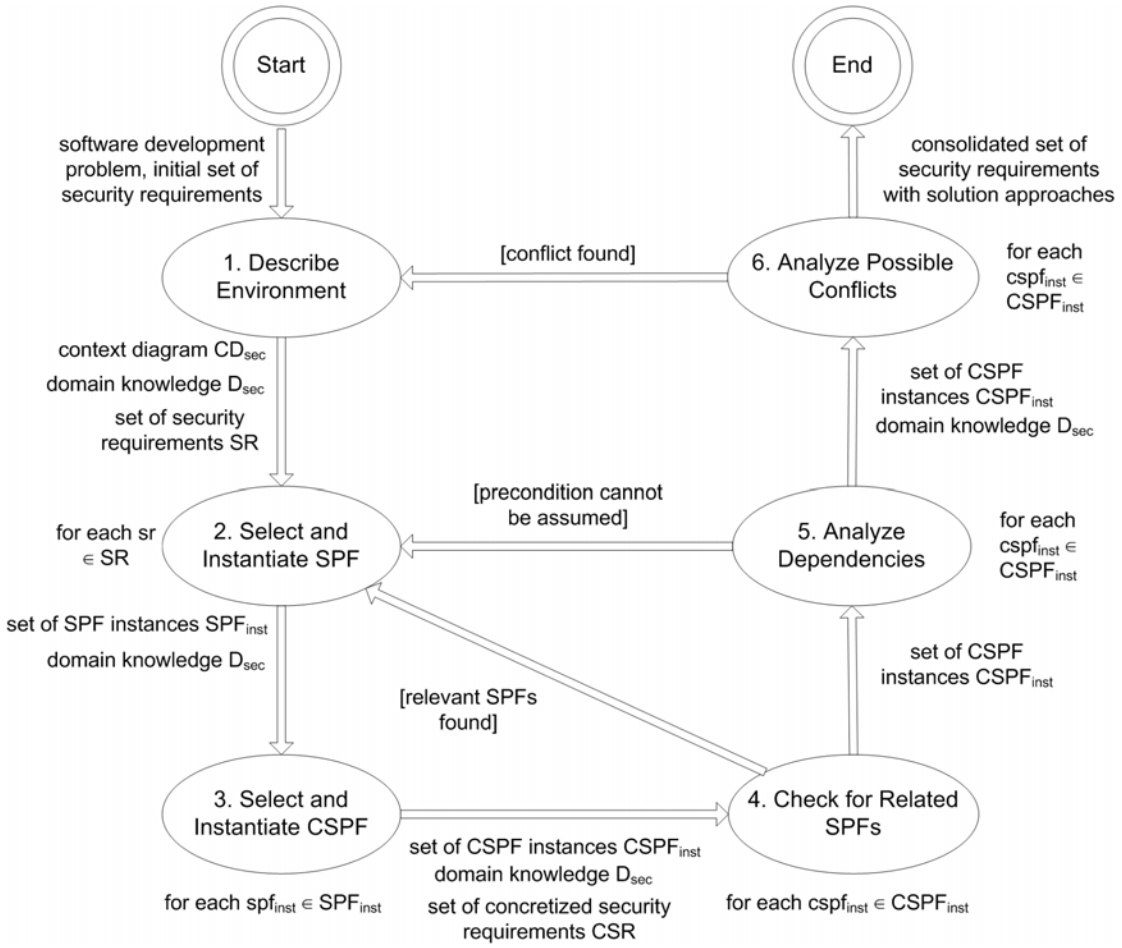

Figure 3: SEPP's Security Requirements Analysis Lifecycle

Step 1 - Describe Environment

All security-relevant entities contained in the environment and relations between them are modeled. Given a context diagram that emerged from functional requirements analysis, it is extended by security-relevant entities and relations. The result is a context diagram CD_{sec}. An example is shown in Figure 4. The hatched area named “Malicious environment” represents the extension by security-relevant entities and relations.

Domain knowledge, i.e., facts and assumptions, about the environment in which the software development problem is located is collected and documented. Especially, domain knowledge about the malicious environment is considered. An example for domain knowledge about a malicious environment is the assumed strength of a potential attacker. The result is a set of security-relevant domain knowledge D_{sec} that consists of a set of security-relevant facts F_{sec} and a set of security-relevant assumptions A_{sec}. Collecting the domain knowledge D_{sec} involves the construction of an attacker model, a threat model, and the application of a risk analysis method.

Input:

- textual description of software development problem
- set of initial security requirements *SR*

Output:

- context diagram CD_{sec} including security-relevant domains and phenomena
- security-relevant domain knowledge $D_{sec} \equiv F_{sec} \wedge A_{sec}$
- attacker model
- results from threat analysis
- results from risk analysis

Validation Conditions:

- domains and phenomena of context diagram CD_{sec} must be consistent with *SR* and D_{sec}
- context diagram CD_{sec} must contain malicious environment

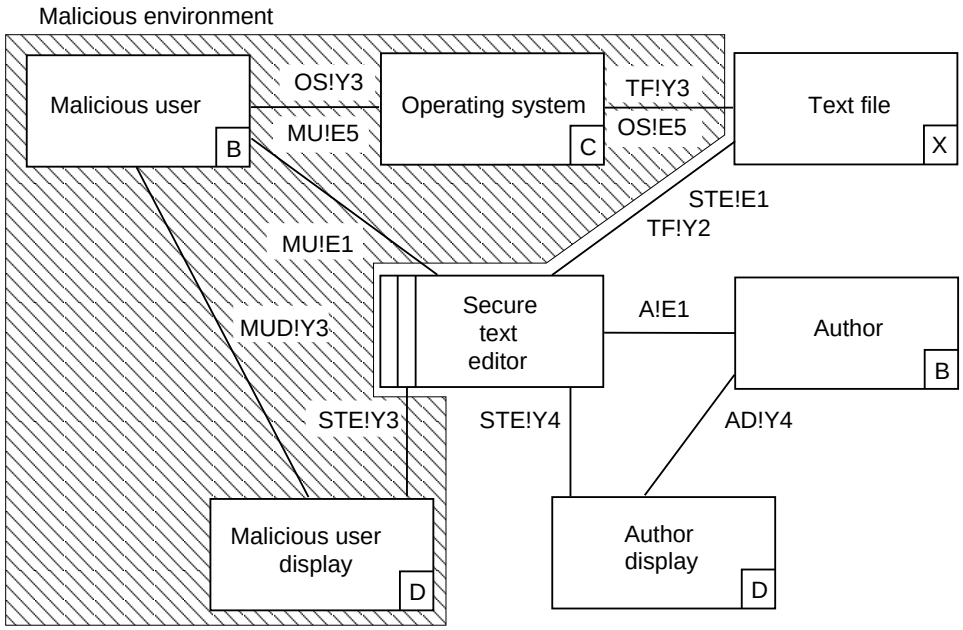

Figure 4: Context Diagram "Secure Text Editor"

Step 2 - Select and Instantiate SPF

This step must be executed for each security requirement $sr \in SR$. To determine an SPF that is appropriate for the given environment and the security requirement $sr \in SR$, the latter is compared with the informal descriptions of the security requirement templates of the SPFs contained in our pattern system. The result is a set of SPFs candidates from which the SPF to be instantiated is selected by considering the security-relevant environment represented by CD_{sec} and the security-relevant domain knowledge D_{sec}.

More precisely, the context diagram CD_{sec} represents the environment of a complex problem, which is decomposed into subproblems that fit to SPFs using decomposition operators such as “leave out domain” or “combine several domains into one domain”. Thus, an SPF candidate that fits to the decomposed environment of the corresponding subproblem is selected.

Afterwards, the SPF is instantiated by assigning concrete values to the domains, phenomena, interfaces, effect, and the security requirement template. The instantiation of an SPF may result in additional security-relevant domain knowledge, which is added to the set of security-relevant domain knowledge D_{sec}. For example, if in the course of the problem decomposition a domain is split into several domains, domain knowledge about these new domains is collected and documented.

After this step is executed for each security requirement $sr \in SR$, the result of this step is a set of security problems SPF_{inst} represented as instantiated SPFs. Furthermore, the set of security-relevant domain knowledge D_{sec} may be updated.

Input:

- all results of step 1

Output:

- set of SPF instance SPF_{inst}
- security-relevant domain knowledge added to D_{sec}

Validation Conditions:

- each security requirement $sr \in SR$ is covered by some SPF instance $spf_{inst} \in SPF_{inst}$
- each SPF instance $spf_{inst} \in SPF_{inst}$ can be derived from the context diagram CD_{sec} by means of certain decomposition operators

Step 3 - Select and Instantiate CSPF

This step must be executed for each SPF instance $spf_{inst} \in SPF_{inst}$. To solve a security problem characterized by an instance of an SPF, a generic security mechanism based on the CSPFs linked to the applied SPF is chosen. The pattern system indicates the CSPFs linked to an SPF by positions marked with "C" in the SPF's column. From the different generic security mechanisms that are represented by CSPFs, an appropriate CSPF is selected. To decide if a CSPF is appropriate, the security-relevant environment represented by CD_{sec} and the security-relevant domain knowledge D_{sec} is considered. For example, if users should select secrets for an encryption mechanism, a password-based encryption mechanism should take precedence over an encryption-key based mechanism. Furthermore, the selection can be accomplished according to other quality requirements such as usability or performance requirements or according to the presumed development costs of the realizations of the generic security mechanisms represented by the different CSPFs.

After a CSPF is selected, it is instantiated by assigning concrete values to the domains, phenomena, interfaces, necessary conditions, and the concretized security requirement template. Normally, domains and phenomena contained in the SPF instance are re-used for the instantiation of the corresponding CSPF. The instantiation of a CSPF may result in additional security-relevant domain knowledge, which must be added to the set of security-relevant domain knowledge D_{sec}, e.g., domain knowledge about passwords or encryption keys.

After this step is executed for each SPF instance $spf_{inst} \in SPF_{inst}$, the result is a set of CSPF instances $CSPF_{inst}$ and a corresponding set of concretized security requirements CSR. Furthermore, the security-relevant domain knowledge D_{sec} may be updated.

Input:

- all results of step 2

Output:

- set of CSPF instances $CSPF_{inst}$
- set of concretized security requirements *CSR*
- security-relevant domain knowledge added to D_{sec}

Validation Conditions:

- for each security requirement $sr \in SR$ there exist a concretized security requirement $csr \in CSR$, and vice versa
- each concretized security requirement $csr \in CSR$ is covered by some CSPF instance $cspf_{inst} \in CSPF_{inst}$
- domains and phenomena of each CSPF instance $cspf_{inst} \in CSPF_{inst}$ are re-used from the corresponding SPF instance

Step 4 - Check for Related SPFs

This step must be executed for each CSPF instance $cspf_{inst} \in CSPF_{inst}$. SPFs that are commonly used in combination with the described CSPF are indicated in the pattern system by positions marked with "R" in the CSPF's row. This information helps to find missing security requirements right at the beginning of the security requirements engineering process.

After this step is executed for each CSPF instance $cspf_{inst} \in CSPF_{inst}$, the result of this step is a set of related security requirements and a corresponding set of SPFs. The related security requirements are added to the set of security requirements *SR* and the SPFs are instantiated by returning to step 2.

Input:

- all results of step 3

Output:

- related security requirements added to the set of security requirements *SR*
- set of SPFs that correspond to the related security requirements

Validation Conditions:

- the new security requirements are relevant for the given software development problem

Step 5 - Analyze Dependencies

This step must be executed for each CSPF instance $cspf_{inst} \in CSPF_{inst}$. The necessary conditions of a CSPF instance are inspected to discover dependent security problems. Two alternatives are possible to guarantee that these necessary conditions hold: either, they can be *assumed* to hold, or they have to be established by instantiating a further SPF, whose effect matches the necessary conditions to be established. Such an SPF can easily be determined using the pattern system: the corresponding positions in the row of the instantiated CSPF are marked with "D".

Only in the case that the necessary conditions *cannot* be assumed to hold, one must instantiate further appropriate SPFs. Then, steps 2 - 4 must be applied to the dependent SPFs.

The security-relevant domain knowledge D_{sec} helps to decide whether the necessary conditions can be assumed to hold or not. For example, assumptions on the strength of passwords chosen by honest users lead to the assumption that malicious users cannot guess the honest user passwords. In contrast, if encryption keys must be delivered to the correct recipients over an insecure

network, additional security mechanisms to authenticate the recipients and to transmit the encryption keys in a confidential and integrity-preserving way must be taken into consideration.

This step is executed for each CSPF instance $cspf_{inst} \in CSPF_{inst}$ until all necessary conditions of all CSPF instances can be proved or assumed to hold. The result of this step is a set of dependent security requirements and a corresponding set of SPFs.

Input:

- all results of step 4

Output:

- dependent security requirements added to the set of security requirements *SR*
- set of SPFs that correspond to the dependent security requirements

Validation Conditions:

- each necessary condition of each $cspf_{inst} \in CSPF_{inst}$ is either assumed to hold or treated by some SPF
- if a necessary condition is assumed, a justification is stated

Step 6 - Analyze Possible Conflicts

This step must be executed for each CSPF instance $cspf_{inst} \in CSPF_{inst}$. The pattern system indicates possible conflicts between the SPF instances and the CSPF instances by rows marked with "I" (for "interaction"). If a possible conflict is discovered, it must be decided if the conflict is relevant for the application domain using D_{sec}. In the case that it is relevant, the conflict must be resolved by modifying or prioritizing the requirements. An example of relaxing security requirements for the benefit of usability requirements can be found in Schmidt & Wentzlaff (2006). This step can result in modified sets of security requirements and concretized security requirements. In such a case, all previous steps must be re-applied to the modified security requirements.

This step is executed for each CSPF instance $cspf_{inst} \in CSPF_{inst}$ until all possible conflicts are analyzed and all relevant conflicts are resolved. Finally, we obtain a set of CSPF instances $CSPF_{inst}$ that can solve the security problems represented by the SPF instances as well as modified sets of security requirements *SR* and concretized security requirements *CSR* with all conflicts resolved.

Input:

- all results of step 5

Output:

- consolidated set of security requirements *SR*
- consolidated set of concretized security requirements *CSR*
- security-relevant domain knowledge added to D_{sec}

Validation Conditions:

- the set of security requirements *SR* contains no more conflicts and is complete with respect to the initial set of security requirements
- the set of concretized security requirements *CSR* contains no more conflicts and is complete with respect to the initial set of security requirements

All in all, the security requirements analysis method results in a consolidated set of security problems and solution approaches that additionally cover all dependent and related security problems and corresponding solution approaches, some of which may not have been known initially.

4.5 Case Study: Secure Text Editor

We now apply the techniques introduced in the previous sections to the following software development problem:

> A graphical secure text editor should be developed. The text editor should enable a user to create, edit, open, and save text files. The text files should be stored confidentially.

The informal security requirement (*SR*) can be described as follows:

> Preserve confidentiality of *Text file* except for its file length for honest environment and prevent disclosure to malicious environment.

Note: We decide to focus on confidentially storing text files. The given software development problem can also be interpreted such that the security requirement also covers confidential editing operations, e.g., confidential clipboard copies. For reasons of simplification, this is not covered in the security requirements analysis. For the same reason, the create and edit functionality of the secure text editor is not covered in our case study. Practically, it is very difficult to develop 100% confidential systems. Hence, as an example, we discuss a SR that allows the secure text editor to leak the file length.

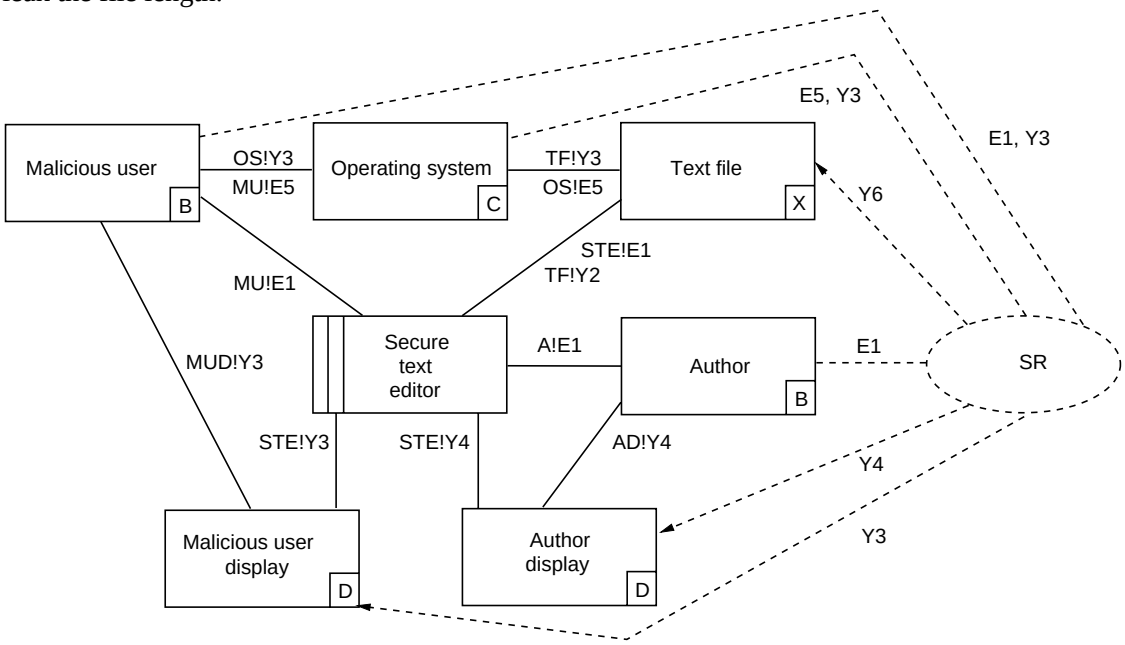

Figure 5: Instantiated SPF Confidential Data Storage "Secure Text Editor"

According to the first step of our security requirements analysis method, the context diagram of this software development problem shown in Figure 4 is developed. Note: we do not show the interfaces of the context diagram explicitly, since they are similar to the interfaces of the

instantiated SPF confidential data storage shown in Figure 5. Furthermore, the attacker model and the results of the threat and risk analysis are not shown explicitly. For example, the attacker model comprises assumptions on the strength and the abilities of potential attackers and an analyzed threat covers the usage of the domain *Operating system* to access the domain *Text file*.

In the second step, we instantiate the SPF confidential data storage as shown in Figure 5 to capture the *SR*. The interfaces of the SPF confidential data storage are instantiated as follows:

E1 = {Save, Open}
Y2 = {TextFile}
Y3 = {LengthOfTextFile}
Y4 = {TextFile, LengthOfTextFile}
E5 = {Spy}
Y6 = {Saving, Opening}

According to the commands from the (malicious and honest) environment, the machine accesses the domain *Text file*, and opens a file or saves one. An opened file is shown to the domain *Author* using the *Author display* domain. We assume that the domain *Malicious user* can at most observe the length of the opened file via the domains *Malicious user display* and *Operating system*.

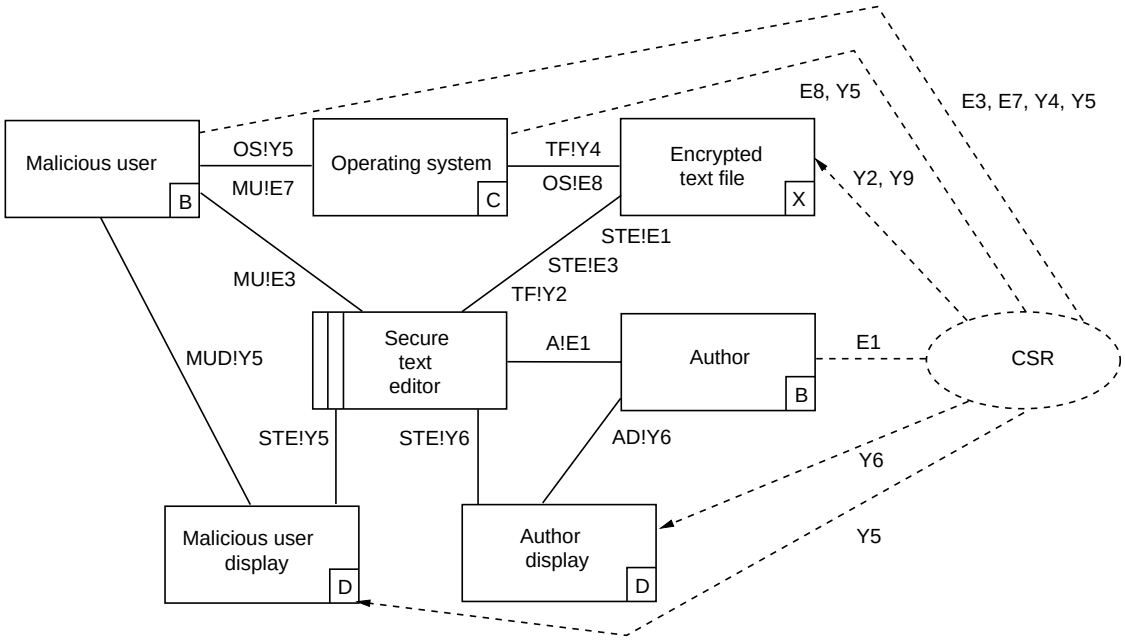

Figure 6: Instantiated CSPF Confidential Data Storage Using Password-Based Encryption "Secure Text Editor"

In the third step, we decide to use a password-based encryption mechanism to conceal text files. For such an encryption mechanism, passwords are necessary. The passwords should be generated and memorized by the users. Since the users must memorize the passwords, a symmetric encryption mechanism is to be preferred over an asymmetric one. This is a trade-off between the usability and the security of the password-based encryption mechanism: asymmetric encryption keys must be much larger compared to symmetric keys to achieve a similar level of

encryption strength. Because of the encryption key lengths, it is more difficult for users to memorize asymmetric keys than symmetric ones.

Using passwords for encryption leads to the assumptions that the users do not reveal their passwords and that they choose passwords that guarantee a certain level of security.

Note that the latter assumption can be transformed into a requirement: the users should not be able choose trivial passwords. Such a requirement can be realized by password checking mechanisms to prevent users from choosing trivial passwords, e.g., words from dictionaries, proper names, and so on.

According to the pattern system, we decide to select the CSPF confidential data storage using password-based encryption. The structure of the CSPF instance shown in Figure 6 is similar to the instantiated SPF confidential data storage shown in Figure 5 with the difference that the domain *Text file* is replaced by the domain *Encrypted text file*. The differences between the SPF instance and the CSPF instance are located in the interfaces. The interfaces of the CSPF instance are instantiated as follows:

E1 = {Save, Open, Password}
Y2 = {EncryptedContentOfTextFile}
E3 = {Save, Open, WrongPassword}
Y4 = {EncryptedContentOfTextFile, LengthOfTextFile}
Y5 = {WrongContentOfTextFile, LengthOfTextFile}
Y6 = {TextFile, LengthOfTextFile}
E7 = {Spy, WrongPassword}
E8 = {Spy}
Y9 = {Saving, Opening}

Since the text files are encrypted by the encryption mechanism, the interfaces TF!Y2 (between *Confidential storage machine* and *Text file*) and TF!Y4 (between *Text file* and *Operating system*) contain the phenomenon *EncryptedContentOfTextFile*. The authors can enter passwords for encrypting and decrypting text files. Therefore, the interfaces A!E1 (between *Author* and *Confidential storage machine*) and STE!E1 (between *Confidential storage machine* and *Text file*) contain the phenomenon *Password*. The malicious users can also enter passwords under the assumption that they cannot guess passwords of authors. Therefore, the interfaces MU!E3 (between *Malicious user* and *Confidential storage machine*), STE!E3 (between *Confidential storage machine* and the *Text file*), and MU!E7 (between *Malicious user* and *Operating system*) contain the phenomenon *WrongPassword*. The concretized security requirement (*CSR*) derived from the *SR* is phrased as follows:

> If *Password* is unknown to malicious environment, then confidentiality of *Text file* except for its file length for honest environment is preserved and disclosure to malicious environment is prevented.

In step four, the pattern system is inspected to check for related security problems. According to the pattern system, the SPF integrity-preserving data storage is related to the CSPF confidential data storage using password-based encryption. This SPF is indeed relevant in the given application context: the integrity of text files stored by authors should be preserved and modification by malicious environment should be prevented. However, for reasons of simplification, this related SPF is not considered here.

In step five, the instantiated necessary conditions (not shown, since they are similar to the necessary conditions of the CSPF except for the types) of the instantiated CSPF are inspected. The first necessary condition is assumed to hold, because we assume that a malicious user cannot guess passwords of an author. The second necessary condition is assumed to hold, because we assume that an author does not reveal passwords to a malicious user. The third necessary condition is assumed to hold, because we assume that there is no malicious user able to intercept and modify passwords of an author.

In step six, our security requirements engineering method proceeds with an analysis of potential conflicts between security requirements. Since only one SPF (and one CSPF) is instantiated, it is not necessary to analyze any potential conflicts between security requirements.

No further SPFs must be instantiated, because each necessary condition is covered by an assumption.

5. DEVELOPMENT OF A SECURITY SPECIFICATION

According to Jackson (2001), a specification is “an optative description: it describes the machine's behaviour at its interfaces with the problem domains” (p. 55). In contrast to the requirements, a machine specification gives an answer to the question: “How should the machine act, so that the system, i.e. the machine together with the environment, fulfills the requirements?” Specifications are descriptions that are sufficient for building the machine. They are implementable requirements.

SEPP supports two methods of constructing a security specification:

- Construction of a semi-formal security specification based on *generic security protocols* (Hatebur & Heisel & Schmidt, 2006), which are expressed using UML sequence diagrams (UML Revision Task Force, Object Management Group (OMG), 2007).
- Construction of a formal security specification according to Schmidt (2009) expressed in CSP (Communicating Sequential Processes) (Hoare, 1986).

In the following, we discuss the second method of constructing a security specification in detail. The software development principle of *stepwise refinement* is popular in software engineering, and is also well supported by *formal methods*. When performing stepwise refinement, software is developed by creating intermediate levels of abstraction. Starting with the requirements, an abstract *specification* is constructed, which is refined by a more concrete *implementation*. Then, the implementation must be verified against the specification, and further refinement steps are accomplished until the desired level of abstraction is reached.

We consider the step from the instantiation of an SPF to the instantiation of a corresponding CSPF a refinement step that not only preserves functional correctness but also the security requirement. To prove this refinement, we apply formal techniques to the (C)SPF approach. Refinement is traditionally either data-refinement or behavior-refinement. Since the (C)SPFs deal with interfaces and communicating domains rather than with states, we decided to describe them using CSP.

CSP is a process algebra that can be used to describe parallel processes that communicate synchronously via message passing. Furthermore, with the model-checker FDR2 (Failure-Divergence Refinement) from Formal Systems (Europe) Limited, sophisticated tool support is available for CSP.

Applying CSP and stepwise refinement to the (C)SPF approach has several benefits:

- CSP models of the (C)SPF instances enable a developer to formally express security requirements captured by (C)SPF instances.
- The CSP models provide a point of contact to the formal probabilistic (and possibilistic) security requirement descriptions by Santen (2008).
- Since problem frames and (C)SPFs as such only provide a static view of a system, we obtain an understanding of the dynamic aspects of (C)SPF instances.
- The CSP models allow a developer to verify that the functional and the security requirements of an SPF are *correctly* implemented by an associated CSPF, i.e., that the functionality and the security requirement are preserved.
- Verification is tool-supported by the model-checker FDR2.

5.1 Introduction to CSP_M

We make use of the CSP ASCII notation named CSP_M since this is a prerequisite for formal verification using the model-checker FDR2. Using CSP_M notation, we define *processes* that interact only by communicating. Communication takes the form of visible *events* or *actions*. A sequence of events produced by a process is called a *trace*. The set of all traces that can be produced by a process *P* are denoted *traces*(*P*). Let *a* be an action and *P* be a process; then *a* -> *P* is the process that performs *a* and behaves like *P* afterwards. This is called *prefixing*. A process can have a name, e.g., *Q* = *a* -> *P*. *Recursion* makes it possible to repeat processes and to construct processes that go on indefinitely, e.g., *Q* = *a* -> *Q*.

We can make use of input and output data: the expression *in*?*x* binds the identifier *x* to whatever value is chosen by the environment, where *x* ranges over the type of the *channel in*. The expression *out*!*y* binds an output value to the identifier *y*, where *y* ranges over the type of channel *out*. The variables *x* and *y* can then be used in the process following the prefix. By convention, ? denotes input data and ! denotes output data.

A process acts in a *nondeterministic* way when its behavior is unpredictable because it is allowed to make internal decisions that affect its behavior as observed from outside. The *replicated internal choice operator* |~| models these internal decisions: let *P* be a process and *X* a finite and non-empty data type, then |~| *a* : *X* • *P*(*a*) behaves according to the selected *a*. This operator gives the environment no control over which data item is chosen. In contrast, the *replicated external choice operator* [] models external decisions: [] *a* : *X* • *P*(*a*) behaves according to the *a* selected by the environment.

5.2 Security Specification Method

The security specification method consists of three steps, which are discussed in detail in the following.

Step 1 - Construct Formal CSP Models

This step must be executed for each SPF and CSPF instance. To formalize a given (C)SPF instance, we describe each of its domains as a recursive CSP process. The interfaces and the control direction of the shared phenomena (control flow) of a domain are translated into CSP channels as well as input and output events. For lexical shared phenomena, we define data types and declare the corresponding channels to be of one of these data types.

Note that when using a model-checker such as FDR2 to analyze real-world problems, we have to address the state explosion problem. A common approach to keep the model-checking effort manageable is to simplify the system to be analyzed. For that reason, we usually must define simplified data types.

We describe a (C)SPF instance as a CSP process consisting of the CSP processes of all of its domains. The processes are combined using *synchronized parallel communication* denoted by [| |]. The synchronization is accomplished over the channels modeling the interfaces that connect the domains.

Finally, the CSP models are checked using the model-checker FDR2 to prove that they are deadlock-free and livelock-free. The model-checker can also be used to debug CSP models.

After this step executed for each SPF and CSPF instance, the result is a set of CSP models that formally specify the (C)SPF instances.

Input:

- all results of phase one

Output:

- set of CSP models of SPF instances CSP_{spf}
- set of CSP models of CSPF instances CSP_{cspf}

Validation Conditions:

- each domain is described by at least one CSP process
- each interface is described by exactly one channel
- each phenomenon is described by an event or an element of a data type
- for each (C)SPF instance, one CSP process expresses a (C)SPF instance by combining all corresponding domains using synchronized parallel communication over those channels that represent the interfaces of the (C)SPF
- all CSP processes are deadlock-free and livelock-free (to be checked using FDR2)

Step 2 - Formally Express Security Requirements

SEPP's security requirements analysis phase currently covers integrity and confidentiality requirements. Integrity requirements can be formally expressed as correctness properties, since they require to preserve the correctness of data.

Confidentiality requirements can be expressed as *information flow properties* of two flavors:

- *possibilistic:* based on the fact that an ICT system has a system behavior, which produces observations visible to the environment, there must exist at least one alternative possible system behavior that produces the same observation.
- *probabilistic:* stochastic system behavior is taken into account.

This step must be executed for each SPF instance CSP model $csp_{spf} \in CSP_{spf}$. In the following, we focus on confidentiality requirements in terms of *possibilistic* information flow properties. In general, we call the formal description of a confidentiality requirement a *confidentiality property*. To formally specify a confidentiality requirement, we apply the framework for the specification of confidentiality requirements by Santen (2008). There does not exist *the* confidentiality property that allows us to express every (informal) confidentiality requirement. Instead, an adequate confidentiality property depends on the confidentiality requirement that it formalizes. Mantel (2003) gives a comprehensive overview of possibilistic information flow properties.

Furthermore, the framework by Santen (2008) discusses different confidentiality properties that formalize different confidentiality requirements.

We apply the techniques presented by Santen (2008) to a CSP model that formally specifies an SPF instance. First, a confidentiality property that fits the informal security requirements description is chosen. Second, the confidentiality property is expressed based on the CSP model of the SPF instance, and it is proven that the CSP models fulfills the confidentiality property.

Note: since confidentiality properties are predicates on *sets* of traces of a CSP model, they cannot be modeled directly in CSP, and thus cannot be verified using FDR2. Nevertheless, we can mathematically specify a confidentiality property and prove that a given machine and environment (i.e., an SPF instance) satisfy the property.

After this step executed for each informal security requirement description, the result is a set of formal security requirement descriptions.

Input:

- informal security requirement description *sr*
- CSP models of SPF instances $csp_{spf} \in CSP_{spf}$

Output:

- formal descriptions of security requirements *SR*

Validation Conditions:

- each formal description of a security requirement $sr \in SR$ refers to traces that are produced by the CSP model of the corresponding SPF instance

Step 3 – Show Security-Requirements Preserving Refinements

Refinement is the transformation of an abstract specification into a concrete specification (implementation). CSP supports three types of process refinements:

- **Trace refinement** A process *Q* trace-refines a process *P*, if all the possible sequences of communications, which *Q* can perform, are also possible in *P*: $traces(Q) \subseteq traces(P)$
- **Failure refinement** Trace refinement extended by consideration of deadlocks.
- **Failure-divergence refinement** Failure refinement extended by consideration of livelocks.

This step must be executed for each formal security requirement description. First, it is proven on a functional level that the CSPF instance failure-divergence refines the SPF instance. Since all structural elements of the SPF are preserved in an associated CSPF, we can show a failure-divergence refinement after we reduce the structural additions of the CSPF instance to the SPF instance structure:

- We hide events that can only be communicated in the CSPF instance model using the *hiding* operator \ of CSP
- We map those events that have a more concrete structure in the CSPF instance model to events that are compatible with events of the SPF instance model. This mapping constitutes a *data refinement*. The mapping is accomplished using the *relational renaming* operator [[<-]] of CSP.

This refinement proof is tool-supported by the model-checker FDR2.

Second, it is proven that the confidentiality requirement is preserved in the CSPF instance. This proof depends not only on the CSP models of (C)SPF instances but also on the confidentiality property. For this proof, tool support is not available. For example, to prove that the confidentiality property *concealed behavior* (Santen, 2008) is satisfied by a given CSPF instance, a set inclusion of sets of traces must be proven.

After this step is executed for each formal security requirement description, the result is a set of functional and confidentiality-preserving refinement proofs.

Input:

- all results of steps 1 and 2

Output:

- refinement proofs of functional requirements (e.g., using FDR2)
- refinement proofs of confidentiality-preserving refinement

Validation Conditions:

- for each pair consisting of the CSP model of an SPF instance $csp_{spf} \in CSP_{spf}$ and the CSP model of the corresponding CSPF instance $csp_{cspf} \in CSP_{cspf}$, $csp_{cspf} \sqsubseteq_{FD} csp_{spf}$ must hold (the concrete CSPF instance model failure-divergence refines the abstract SPF instance model)

5.3 Case Study: Secure Text Editor

According to step one of the previously described method to construct security specifications, we develop CSP models for the instantiated SPF confidential data storage and the instantiated CSPF confidential data storage using password-based encryption of the secure text editor case study. In the following, the suffix *_S* (for specification) of a process or channel name denotes that it is part of the CSP model of the abstract SPF instance, whereas the suffix *_I* (for implementation) denotes that it is part of the CSP model of the concrete CSPF instance.

Construction of the CSP Model of the Instantiated SPF Confidential Data Storage

The basic ingredients of a CSP model are type and function definitions as well as channel declarations. We define a simple data type named *Plaintext* with four values *p1, p2, p3, p4*:

datatype Plaintext = p1 | p2 | p3 | p4

Then, we declare the channel *TextFile_Y2_S* that corresponds to the interface TF!Y2 in Figure 5 to be of this data type, i.e., all events communicated over this channel are *p1, p2, p3,* or *p4*. Furthermore, we model a *spy* command by a data type *SpyCommand* with only one action *spy* that represents an operating system command to open a text file:

datatype SpyCommand = spy

We declare the channels *MaliciousUser_E5_S* and *OperatingSystem_E5_S* to be of this type. We model the user command *open* to securely open text files:

datatype UserCommand = open

We declare the channels *Author_E1_S*, *MaliciousUser_E1_S*, and *SecureTextEditor_E1_S* to be of this type.

We represent the interfaces MUD!Y3, OS!Y3, STE!Y3, and TF!Y3 (see Figure 5) by channels of the datatype *Length:*

datatype Length = short | long

The data items leaked over this channel are defined by a *leakage function f*:

f(p) = if p==p1 or p==p2
then short
else long

As an example, the leaked data items are *short* and *long*, and they correspond to the lengths of the plaintexts sent over the channel *TextFile_Y2_S.*

The interfaces AD!Y4 and STE!Y4 are represented by channels of the datatype *Plaintext.Length,* i.e. the plaintext and its length concatenated to it.

We describe each domain as a recursive CSP process, e.g., the process *HonestUser_S* that represents the *Author* domain:

Author_S = (|~| *ucmd* : *UserCommand* @ *Author_E1_S*!*ucmd* -> *Author_S*)
[] (*AuthorDisplay_Y4_S*?*pt* -> *Author_S*)

This process arbitrarily chooses a user command *ucmd* and sends this command over the channel *Author_E1_S* or it receives a plaintext *pt* over the channel *AuthorDisplay_Y4_S*. Afterwards, the recursive call of *Author_S* ensures that the process is repeated.

We specify the instantiated SPF in Figure 5 as a process *SecureTextEditor_ SPF_CONFIDENTIAL_DATA_STORAGE*(*pt*) that combines all formalized domains of the instantiated SPF confidential data storage. It has a parameter *pt* that is initialized by an arbitrary chosen element of *Plaintext*. For example, the process

(*Author_S* [| {|*AuthorDisplay_Y4_S*|} |] *AuthorDisplay_S*)

combines the processes *Author_S* and *AuthorDisplay_S.* They synchronize over the channel *AuthorDisplay_Y4_S*, or, informally speaking, the domain *Author* reads data from the domain *Author display*.

Construction of the CSP Model of the Instantiated CSPF Confidential Data Storage Using Password-Based Encryption

Instead of describing the instantiated CSP model of the CSPF confidential data storage using password-based encryption completely, we present those parts that are responsible for the usage of the password-based encryption mechanism.

We introduce data types *Password* and *Ciphertext*, and the functions *encr* and *decr*:

datatype Ciphertext = c1 | c2 | c3 | c4

datatype Password = pwd1 | pwd2 | pwd3 | pwd4

encr(p1,pwd1) = c1	…	*decr(c1,pwd1) = p1*	…
encr(p1,pwd2) = c2	…	*decr(c1,pwd2) = p2*	…
encr(p1,pwd3) = c1	…	*decr(c1,pwd3) = p1*	…
encr(p1,pwd4) = c2	…	*decr(c1,pwd4) = p2*	…

Note: the definitions of the *encr* and *decr* functions are not complete. The functions *encr* and *decr* model a length-preserving cryptographic mechanism.

The implementation of the previously presented process *Author_S* makes use of the declared passwords:

Author_I(*password*) = (|~| *ucmd : UserCommand @*
Author_E1_I!*ucmd.password -> Author_I*(*password*))
[] *(AuthorDisplay_Y6_I?pt -> Author_I*(*password*))

The process *Author_I*(*password*) is parameterized by the *password* selected by the author. This password is passed together with the user command *ucmd* over the channel *Author_E1_I* to the *SecureTextEditor_I* process. The type of the channel *Author_E1_I* is *UserCommand.Password.*

The process *OperatingSystem_I* makes use of the declared ciphertexts and decryption function:

OperatingSystem_I = (MaliciousUser_E7_I?scmd.pwd -> OperatingSystem_E8_S!scmd
-> OperatingSystem_I_mempwd(*pwd*))

OperatingSystem_I_mempwd(password) = OperatingSystem_I
[] (*TextFile_Y4_I?ct*
-> OperatingSystem_Y5_I!*decr*(*ct,password*).
f(*decr*(*ct,password*))
-> OperatingSystem_I)

The process *OperatingSystem_I* receives a spy command *scmd* and a password *pwd* over the channel *MaliciousUser_E7_I.* The spy command *scmd* is passed over to the process *TextFile_I* representing the *Text file* domain via the channel *OperatingSystem_E8_S*. Then, the process *OperatingSystem_I* behaves as defined by the process *OperatingSystem_I_mempwd*(*password*).

This process either behaves as defined by the process *OperatingSystem_I* or it behaves as follows: it receives a ciphertext *ct* over the channel *TextFile_Y4_I*. The operating system applies the decryption function to the ciphertext using *decr*(*ct,password*) to obtain the (wrong) plaintext. Furthermore, it calculates the length of this plaintext using *f*(*decr*(*ct,password*)). The plaintext and the result of the length calculation are concatenated and sent over the channel *OperatingSystem_Y5_I* to the *MaliciousSubject_I* process, which represents the *Malicious subject* domain.

The role of the channel between the malicious user and the operating system has changed: the channel *OperatingSystem_Y5_I* not only leaks the lengths of the transferred data items to the environment, but also the (wrong) plaintext obtained by applying the *decr* function to the ciphertext *ct* using the password *pwd*. Under the assumption that the password *pwd* selected by the malicious user is unequal to the password selected previously by the author to encrypt the plaintext yielding the ciphertext *ct*, the malicious user will only be able to decrypt the ciphertext *ct* to a wrong plaintext.

In summary, we constructed CSP models of the instances of the SPF confidential data storage and the corresponding CSPF confidential data storage using password-based encryption. Using FDR2, we successfully verified that the presented CSP models are deadlock-free and livelock-free.

Formally Expressing the Confidentiality Requirement

Following step two of phase two, we formally express the confidentiality requirement according to Schmidt (2009). The concept of indistinguishable traces presented by Santen (2008) is the foundation for defining confidentiality properties. Given a set of channels *W*, two traces *s*, *t* $\in$ *traces(P)* of a process *P* are *indistinguishable* by *W* (denoted $s \equiv_W t$) if their projections to *W* are equal: $s \equiv_W t \Leftrightarrow s \upharpoonright W = t \upharpoonright W$, where $s \upharpoonright W$ is the projection of the trace *s* to the sequence of events on *W*. The *indistinguishability class* $J_W^{P,k}(o)$ contains the traces of *P* with a length of at most *k* that produce the observation *o* on *W*.

Applied to the previously presented CSP models, this means that any distinction (e.g., data item length is *short* or *long*) the malicious subject can make about the internal communication of the system (e.g., appearance of different plaintexts and ciphertexts) based on the observations on, e.g., *OperatingSystem_Y3_S* and *OperatingSystem _Y4_I* is information revealed by the system. Conversely, any communication that cannot be distinguished by observing, e.g. *OperatingSystem _Y3_S* and *OperatingSystem _Y4_I* is concealed by the system. We can determine two indistinguishability classes: one that contains those traces that produce the observation *short* on the monitoring channel, and another one that contains those traces that produce the observation *long* on the monitoring channel.

An *adversary model* according to Santen (2008) is a system model that consists of the machine to be developed, the honest environment, the malicious environment, and their interfaces. The previously presented CSP models constitute valid adversary models.

A *mask* $\mathcal{M}$ for an adversary model is a set of subsets of the traces over the alphabets (i.e., the events supported by a process) of the processes modeling the machine to be developed, the honest user environment, and the malicious environment such that the members of each set are indistinguishable by observing the monitoring channels (i.e., the channels that leak the wrong plaintexts and the lengths) of the adversary environment *W*: $\forall M : \mathcal{M} \bullet \forall t_1, t_2 : M \bullet t_1 \equiv_W t_2$

All traces of the form

$t_0(pt) =$
$\quad$ < *TextFile_Y3_S.f*(*pt*), *OperatingSystem_Y3_S.short*> if $pt \in \{p1, p2\}$,
$\quad$ < *TextFile_Y3_S.f*(*pt*), *OperatingSystem_Y3_S.long*> else,

where $pt \in$ *Plaintext*, produce the observation *short* or *long* on *OperatingSystem _Y3_S* for *Malicious user*. According to the informal confidentiality requirement as it has been stated in

SEPP's phase one, this observation should not allow *Malicious user* to infer the transferred plaintext.

A mask $\mathcal{M}_0$ supporting the confidentiality requirement needs to require that for a given length *l* all variations of plaintexts *pt* in the parameter list of the trace t_0 are possible causes of the observation *OperatingSystem _Y3_S.l*.

Therefore, the sets $M_0 = \{t_0(p1), t_0(p2)\}$ and $M_1 = \{t_0(p3), t_0(p4)\}$ should be members of $\mathcal{M}_0$. If the traces in a set $M \in \mathcal{M}$ are indistinguishable by observing the monitoring channels, then the differences between these traces are kept confidential. This confidentiality property is named *concealed behavior* (Santen, 2008). It is formalized based on a set inclusion $M \subseteq J_W^{QE,k}(o)$, where the process *QE* is a *variant*, i.e., a purely deterministic process, of the adversary model. It is required that members of $\mathcal{M}$ are either completely contained in an indistinguishability class, or not at all. One says that the set of indistinguishability classes $\mathcal{I}$ *covers* $\mathcal{M}$.

In general, a given adversary model satisfies a confidentiality property, which is defined based on a *basic confidentiality property* (Santen, 2008), if there exists a probabilistic deterministic realization of a machine that satisfies the basic confidentiality property in all admissible environments. In the case of concealed behavior, the question is if there is an adversary model that covers a given mask.

To show that the adversary model represented by the CSP model of the SPF instance conceals the mask $\mathcal{M}_0$, a deterministic machine realization must be found such that its composition with all realizations of the environment covers $\mathcal{M}_0$.

We choose the implementation of the CSP model of the SPF instance that resolves all nondeterministic choices by probabilistic choices with equal probabilities for all alternatives.

The admissible environments consist of realizations that deterministically produce traces according to the pattern $t_0(pt)$, where $pt \in Plaintext$. The members M_0 and M_1 of $\mathcal{M}_0$ are covered by the indistinguishability classes of all resulting variants of the CSP model of the SPF instance, because the chosen machine realization does not exclude any of the traces $t_0(pt)$, where $pt \in Plaintext$.

In summary, we presented a formal description of the informal confidentiality requirement description of the instance of the SPF confidential data storage.

Proving Confidentiality-Preserving Refinement

The next and last step of phase two comprises the verification of a confidentiality-preserving refinement based on the CSP model of the (C)SPF instances and the formally specified confidentiality requirement.

To prove the functional refinement, we prove that the CSP model of the CSPF instance failure-divergence refines the CSP model of the SPF instance using the model-checker FDR2.

Based on the CSP model of the CSPF instance, we create a re-abstracted CSP model by hiding events that can only be communicated in the CSPF model, e.g., *Password* and *WrongPassword*, and we map those events that have a more concrete structure in the CSPF model to events that are compatible with the events of the SPF model, e.g., *Ciphertext* events are substituted by *Plaintext* events. The data refinement is characterized by the fact that a plaintext is refined by a pair consisting of a ciphertext and a password. The resulting CSP process failure-divergence refines the CSP process that models the SPF instance, which can be verified using FDR2.

To prove the confidentiality-preserving refinement, we calculate the indistinguishability classes of the CSP model of the CSPF instance. As an example, we present the indistinguishability class $J^{QE,2}_{OperatingSystem\_Y4\_I}(OperatingSystem\_Y5\_I.p1.l)$:

$$J^{QE,2}_{OperatingSystem\_Y4\_I}(OperatingSystem\_Y5\_I.p1.l) =$$
$$\{<TextFile\_Y4\_I.c1, OperatingSystem\_Y5\_I.p1.short>,$$
$$<TextFile\_Y4\_I.c2, OperatingSystem\_Y5\_I.p2.short>,$$
$$<TextFile\_Y4\_I.c3, OperatingSystem\_Y5\_I.p1.long>,$$
$$<TextFile\_Y4\_I.c4, OperatingSystem\_Y5\_I.p2.long>\}$$

Since the confidentiality property concealed behavior refers to both, the monitoring channel *OperatingSystem_Y5_I* and the data, we must relate the concrete monitoring channel and the data back to the abstract ones originally referred to by the confidentiality property. Applied to concealed behavior, this general concept provides a basis for defining *refined concealed behavior*. After the re-abstraction, we must check if the re-abstracted traces are members of M_0 and M_1, respectively.

We find out that the re-abstracted traces are the same traces as the abstract ones. For this reason, they are contained in the sets M_0 and M_1. Hence, the CSP model of the CSPF instance conceals the mask M_0, and the confidentiality property concealed behavior is preserved in the CSP model of the CSPF instance. Therefore, the CSP model of the CSPF instance comprises a valid specification for the machine to be developed.

6. DEVELOPMENT OF A SECURITY ARCHITECTURE

In the following, we move on to the design phase of software development, i.e., the construction of a software architecture using:

- *architectural patterns* to construct a *platform-independent* secure software architecture that realizes the specified security requirements and
- a method to construct a *platform-specific* secure software architecture based on a previously developed platform-independent secure software architecture and a component framework or an application programming interface (API).

6.1 Generic Security Components and Architectures

Software components are reusable software parts. We represent software components by means of UML composite structure diagrams (UML Revision Task Force, Object Management Group (OMG), 2007) and *interface specifications*. The latter consist of several parts: structural and syntactic descriptions are expressed as UML class diagrams (interface classes). Semantic descriptions of the operations provided and used by the components' interfaces are expressed as OCL pre- and postconditions. Behavioral descriptions are expressed as UML sequence diagrams.

The generic security components discussed in this section constitute special software components that realize concretized security requirement templates. We call them “generic”, because they are a kind of conceptual pattern for software components. They are platform-independent. An example for a generic security component is an encryption component defined neither referring to a specific encryption mechanism such as AES or DES nor specific encryption keys, such as encryption keys with a certain length.

We use generic security components to structure the machine domain of a CSPF. They describe the machine's interfaces to its environment and the machine-internal interfaces of its components. Each CSPF is linked to a set of generic security components.

A machine domain of a CSPF can be structured by means of generic security components according to the following principles:

- Each interface of the machine with the environment must coincide with an interface of some component.
- Components of the same purpose can be combined, e.g., several storage management components can be combined to one such component.
- For each interface between the machine and a biddable or display domain a user interface component must be introduced. Interfaces to another display or machine domain can result in an additional user interface component (especially if such an interface is security-critical, e.g., an interface to enter a password).
- For each interface from the machine to a lexical domain, a storage management component must be introduced. Symbolic phenomena correspond to return values of operations or to getter/setter operations.
- For each interface of the machine domain with a causal domain, a driver component must be introduced. Causal phenomena correspond to operations provided by driver components.
- For password or encryption key handling, key management components or key negotiation components must be introduced.
- For encryption key generation, random number generator and encryption key generator components must be introduced.
- For symmetric / asymmetric encryption / decryption, corresponding encryptor / decryptor components must be introduced.
- For integrity mechanisms, hash and MAC calculation components must be introduced.

Following the described principles, we developed a catalog of generic security components for each available CSPF. These components can be combined to obtain a set of generic security architectures that realize the concretized security requirement template of a CSPF.

The generic security components constructed for a CSPF can be combined to obtain generic security architectures according to the following principles:

- An adequate basic software architecture to connect the generic security components has to be selected, e.g., a layered architecture.
- If components can be connected directly, one connects these components.
- If components cannot be connected directly (e.g., because a component provides incompatible input for another component), additional components to interconnect them must be introduced.
- Interfaces between the machine and its environment must be introduced in the generic security architecture according to the generic security components that provide or use these interfaces.

As examples, we present generic security components and architectures for the CSPF confidential data storage using password-based encryption.

6.2 Generic Security Components for CSPF Confidential Data Storage Using Password-Based Encryption

Figure 7 shows on the left-hand side a generic security component for handling passwords expressed as a UML composite structure diagram and a class diagram.

The component *PasswordReader* provides an interface *PwdRIf*. It consists of the operations *readPassword()* and *destroyPassword()*.

The behavior of the *PasswordReader* component is described using the UML sequence diagram in Figure 8. After the operation *readPassword()* is called via the interface *PwdRIf*, the component calls the operation *showPasswordDialog()* via the used interface *EnvIf* that shows a dialog to the user and requests a password from her/him. Then, the user can submit a password *pwd* to the component, which returns this password to the caller of the operation *readPassword()*. Afterwards, this password must be wiped out from memory using the operation *destroyPassword()*.

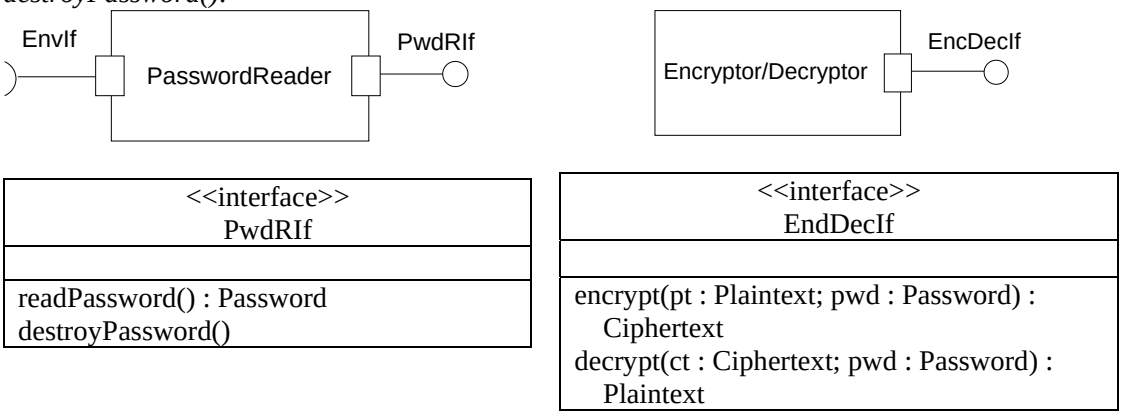

Figure 7: Generic Security Components "PasswordReader" and "SecretManagement" with Interfaces Classes

OCL pre- and postconditions can be used to enrich the generic security component *PasswordReader* with security-relevant operation semantics. For example, constraints on the quality of the password captured by the operation *readPassword()*: e.g., a minimal password length, occurrence of special characters, etc. can be expressed.

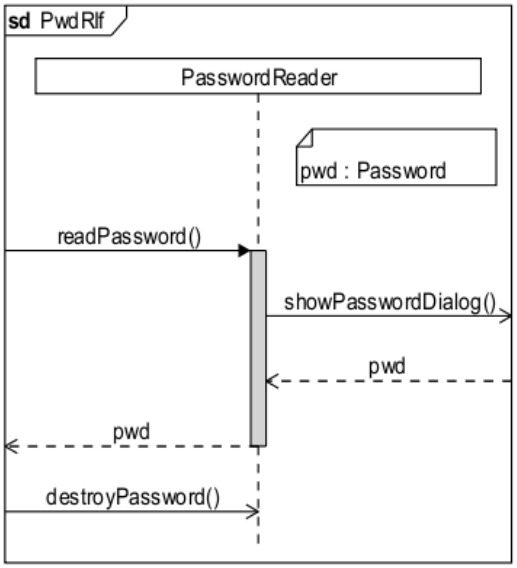

Figure 8: Behavior of the Generic Security Component "PasswordReader"

The generic security component Encryptor/Decryptor described on the right-hand side of Figure 7 provides an operation *encrypt(pt : Plaintext; pwd : Password)* that encrypts a plaintext *pt* using a password *pwd* to a ciphertext *ct*. Additionally, it provides an inverse operation *decrypt(ct : Ciphertext; pwd : Password)* that calculates the plaintext *pt* given the ciphertext *ct* and the password *pwd*.

The generic security components described previously must be combined to obtain generic security architectures. Since generic security components are platform-independent, so are generic security architectures. Each CSPF is linked to a set of generic security architectures that realize the concretized security requirement template.

As an example, we present a generic security architecture for the CSPF confidential data storage using password-based encryption in Figure 9.

According to the previously presented general procedure to set up a generic security architecture of a CSPF's machine domain, we introduce *UserInterface* and *StorageManager* components to let a user interact with the software and to access a storage device, respectively. To realize the concretized security requirement template of the CSPF, we introduce the generic security components *PasswordReader* and *Decryptor/Encryptor*. These components are arranged in a *layered architecture* using an *Application* layer component.

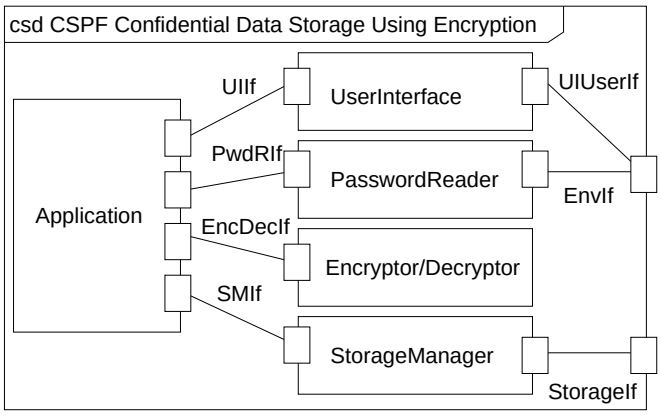

Figure 9: Generic Security Architecture "CSPF Confidential Data Storage Using Password-Based Encryption"

6.3 Security Architecture Method

The security architecture method consists of four steps, which are discussed in detail in the following.

Step 1 - Select Generic Security Architectures

A generic security architecture that consists of a set of generic security components is selected for each CSPF instance, based on domain knowledge and constraints of the application domain.

Input:

- all results of phase one

Output:

- set of generic security architectures including sets of generic security components

Validation Conditions:

- the set of generic security architectures is suitable to realize the concretized security requirements *CSR*

Step 2 - Compose Generic Security Architectures

Generic security architectures can also be applied to software development problems of higher complexity, i.e., problems that are divided into several subproblems, by composing the according sub-architectures. Note that we refer to generic security architectures as generic security sub-architectures if a software development problem described by more than one CSPF is considered. Choppy & Hatebur & Heisel (2006) developed for a set of subproblems a corresponding set of sub-architectures that solve these subproblems. Moreover, the sub-architectures are composed based on dependencies between the subproblems. The authors identified three different kinds of dependencies:

- parallel subproblems
- sequential subproblems
- alternative subproblems

For subproblems that are instances of CSPFs, sequential dependencies can be identified based on the effects and necessary conditions of the subproblems. If a necessary condition of an instantiated $CSPF_A$ is considered as an additional security requirement, then a $CSPF_B$ that solves

this security requirement depends on $CSPF_A$. More precisely, the effect of $CSPF_B$ that solves the new security requirement must be established before the mechanism represented by $CSPF_A$ can work correctly.
According to Choppy & Hatebur & Heisel (2006), we must "decide if two components contained in different subproblem architectures should occur only once in the global architecture, i.e., they should be merged". We adopt the principles to merge components presented by Choppy et al. to our approach. We consider the dependencies of functional subproblems (e.g., problem frame instances) and security subproblems (CSPF instances). The principles to merge components contained in generic security sub-architectures are as follows (Choppy & Hatebur & Heisel, 2006):

- Two components that belong to sequential or alternative subproblems should be merged into one component.
- Two components that belong to parallel subproblems and that share some output phenomena should be merged into one component, because the output must be generated in a way satisfying both subproblems.
- If two components that belong to parallel subproblems and that share some input phenomena do not share any output phenomena, one can merge the components or keep them separate. In the latter case the common input must be duplicated.
- Two components that belong to parallel subproblems and that do not share any phenomena should be kept separate.

The result of the composition procedure is a platform-independent global generic secure software architecture. To show that the components in a global generic security architecture work together in such a way that they fulfill the security requirements corresponding to the different subproblems, one can model the global generic security architecture with *UMLsec* by Jürjens (2003). It can then be analyzed using the tool suite provided for UMLsec (see (Jürjens, 2003, pp. 133) for details) to check if the security requirements are fulfilled in the global generic security architecture.

Especially, confidentiality requirements must be treated carefully, since composition of incompatible components can lead to non-fulfillment of confidentiality requirements (see (Santen & Heisel & Pfitzmann, 2002) for details).

Input:

- all results of step one

Output:

- a generic security architecture that combines all generic security architectures selected in step one

Validation Conditions:

- the combined generic security architecture is suitable to realize the concretized security requirements *CSR*

Step 3 - Refine Generic Security Architecture

In the following, we consider the refinement of the platform-independent generic security architecture to a platform-specific and implementable security architecture. It consists of several substeps:

1. Select an adequate component framework or API, e.g., the security APIs BouncyCastle or Sun's javax.crypto.*.

2. Select given components from the chosen component framework or API:
    a. Compare the interfaces, the operation semantics, and the behavioral description of the generic security components with the documentation of the component framework or API to find adequate existing components.
    b. Normally, several existing components must be used to realize one generic security component; in this case glue code must be written to connect the existing components in such a way that the specification of the generic security component is fulfilled.
    c. In the rare case that the specification of an existing component matches the specification of a generic security component, the existing component can be used without customization.
    d. Those generic security components that cannot be treated by the selected component framework or API must be implemented from scratch, based on the specification of the generic security component.

Input:

- combined generic security architecture from step two

Output:

- a refined security architecture including a set of refined security components

Validation Conditions:

- each generic security component is either realized using an existing security component or it is developed from scratch

Step 4 - Connect Security Components

Glue code is written to connect the components according to the refined generic security architecture.

Input:

- all results of step three

Output:

- a security architecture that realizes the concretized security requirements *CSR*

Validation Conditions:

- the refined security component are connected according to the refined security architecture

6.5 Case Study: Secure Text Editor

In step one of phase three, we select a generic security architecture for each applied CSPF. For the secure text editor case study, the generic security architecture shown in Figure 9 is adequate, since it comprises a component for password-based encryption/decryption and a component to obtain a password from a user.

It is not necessary to apply step two, because we selected only one generic security architecture.

We refine this generic architecture in step three using the Java Standard Edition 6 API provided by Sun. As examples, we present the refined *PasswordReader* component in Figure 10

and the refined *Encryptor/Decryptor* component in Figure 11. The refined *PasswordReader* component consists of a Wrapper component that represents the glue code necessary to combine the components pwdField, *pbeKeySpec*, *secretKeyFactory*, and *secretKey* provided by Sun's Java Standard Edition 6 API. The *pwdField* component provides a graphical text field to retrieve a password from a user. It makes the user input unreadable, while the password is entered into text field.

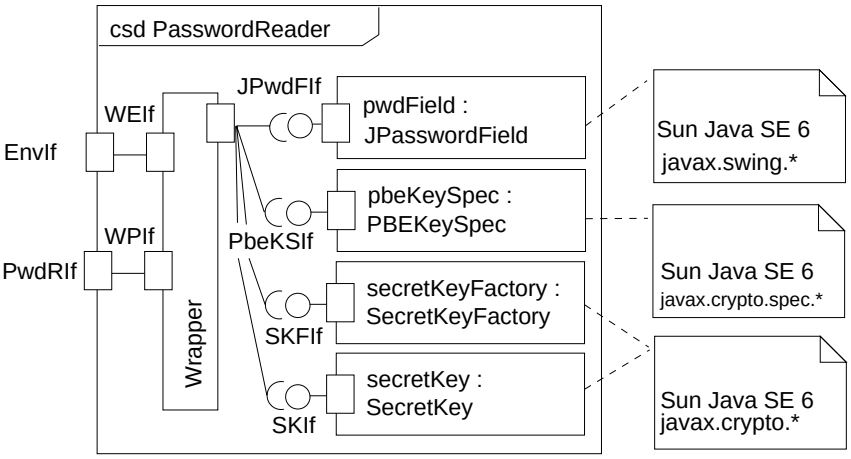

Figure 10: Refined "PasswordReader" Component

The other components are used to construct a symmetric encryption key compliant to a specific password-based cryptography specification (e.g., PKCS #5) from the user password.

The refined *Encryptor/Decryptor* component is constructed similarly. It also consists of a *Wrapper* component that connects the components *cipher* and *pbeParamSpec* from Sun's Java Standard Edition 6 API. The *cipher c*omponent provides the functionality of a cryptographic cipher for encryption and decryption. The *pbeParamSpec* component is necessary to construct a parameter set for password-based encryption as defined in the PKCS #5 standard.

Due to space limitations, we do not describe the component-internal interfaces in detail and we do not show the other refined security components. The result of SEPP's last phase applied to the secure text editor case study is a security architecture that makes use of given components provided by Sun's Java Standard Edition 6 API. The next last step of phase three is programming the *Wrapper* components and the glue code to connect the refined generic security components. Finally, testing and deployment have to be performed.

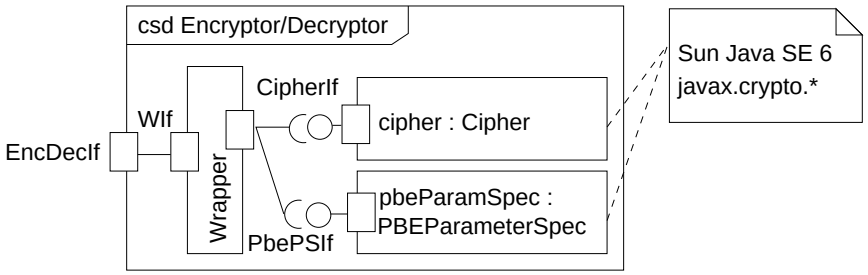

Figure 11: Refined Encryptor/Decryptor Component

We implemented two versions of the case study, one based on the BouncyCastle API and another one based on Sun's javax.crypto.* API. Both versions re-use existing modules of the APIs according to the refined security components shown in Figure 10 and Figure 11.

The amount of glue code to implement the wrapper components ranges between 20 to 50 lines of code per component.

7. FUTURE RESEARCH DIRECTIONS

In the future, we intend to find new patterns to extend the catalogue of SPFs and CSPFs.

We would like to consider probabilistic confidentiality properties and the compositionality of confidentiality-preserving refinement. Moreover, we plan to elaborate more on composition principles to combine generic security architectures to a combined generic security architecture that preserves the security requirements of the different generic security subarchitectures.

Furthermore, we intend to describe the generic security components and the generic security architectures using CSP models. We would like to formally show refinements between the CSPFs and the generic security components/architectures.

8. CONCLUSION

We presented SEPP, a security engineering process that makes use of different kinds of patterns. It covers security requirements engineering, formal security specifications, and the construction of security architectures.

SEPP starts with an extensive security requirements engineering phase, which is based on SPFs and CSPFs. These special kinds of problem frames are arranged in a pattern system. They serve to structure, characterize, analyze, and finally solve software development problems in the area of software and system security. SEPP supports to obtain a complete set of security requirements by analyzing the necessary conditions of the used CSPFs and deciding if they can be assumed or must be established by applying more frames.

Afterwards, formal security specifications in CSP are developed based on instantiated (C)SPFs. These models can be used to formally express security requirements. Given an SPF CSP model and a corresponding CSPF CSP model, we can formally prove a refinement that preserves the security requirement.

SEPP's final phase covers the development of security architectures, which are constructed based on generic security architectures and generic security components. The generic security architectures are refined using existing or tailor-made security components. The results are platform-specific and implementable software architectures that realize the specified security requirements.

9. REFERENCES

Bass, L. & Clements, P. & Kazman, R. (1998). Software Architecture in Practice. Addison-Wesley, 1st edition.

BouncyCastle API. Retrieved June 24th, 2009, from http://www.bouncycastle.org/.

Braber, I. F. & Hogganvik, M. S. L. & Stølen, K. & Vraalsen, F. (2007). Model-based security analysis in seven steps - a guided tour to the CORAS method, (pp.101-117). *In BT Technology Journal,* 25(1).

Choppy, C. & Hatebur, D. & Heisel, M. (2005). Architectural patterns for problem frames. *IEEE Proceedings - Software, Special Issue on Relating Software Requirements and Architecture*, (pp. 198-208). IEEE Computer Society.

Choppy, C. & Hatebur, D. & Heisel, M. (2006). Component composition through architectural patterns for problem frames. *In Proceedings of the Asia Pacific Software Engineering Conference (APSEC)* (pp. 27-34). IEEE Computer Society.

Coplien, J. O. (1992). Advanced C++ programming styles and idioms. Addison-Wesley.

Côté, I. & Hatebur, D. & Heisel, M. & Schmidt, H. & Wentzlaff, I. (2008). A systematic account of problem frames. *In Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP)* (pp. 749-767). Universitätsverlag Konstanz.

Deng, Y. & Wang, J. & Tsai, J. J. P. & Beznosov, K. (2003). An approach for modeling and analysis of security system architectures, (pp. 1099 - 1119). IEEE Transactions on Knowledge and Data Engineering, 15(5). IEEE Computer Society.

Fabian, B. & Gürses, S. & Heisel, M. & Santen, T. & Schmidt, H. (to appear). A comparison of security requirements engineering methods. Requirements Engineering.

Failure-Divergence Refinement (FDR) 2 by Formal Systems (Europe) Limited. Retrieved June 24th, 2009, from http://www.fsel.com/index.html.

Fernandez, E. B. & Larrondo-Petrie, M. M. & Sorgente, T. & Vanhilst, M. (2007). Integrating security and software engineering: Advances and future visions. In H. Mouratidis & P. Giorgini (Eds.), (pp. 107-126).

Fernandez, E. B. & la Red M., D. L. & Forneron, J. & Uribe, V. E. & Rodriguez G., G. (2007). A secure analysis pattern for handling legal cases. *In Latin America Conference on Pattern Languages of Programming (SugarLoafPLoP)* (2007). Retrieved June 24th, 2009, from http://sugarloafplop.dsc.upe.br/wwD.zip.

Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (1995). Design patterns - elements of reusable object-oriented software. Addison Wesley.

Gürses, S. & Jahnke, J. H. & Obry, C. & Onabajo, A. & Santen, T. & Price, M. (2005). Eliciting confidentiality requirements in practice. *In Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, (pp. 101-116). IBM Press.

Haley, C. B. & Laney, R. & Moffett, J. & Nuseibeh, B. (2008). Security requirements engineering: A framework for representation and analysis, (pp. 133-153). *IEEE Transactions on Software Engineering*, 34(1). IEEE Computer Society.

Haley, C. B. & Laney, R. & Moffett, J. & Nuseibeh, B. (2005). Arguing security: Validating security requirements using structured argumentation. *In Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS).*

Haley, C. B. & Laney, R. & Moffett, J. & Nuseibeh, B. (2004). Picking battles: The impact of trust assumptions on the elaboration of security requirements. In C. D. Jensen & S. Poslad & T.

Dimitrakos (Ed.), Proceedings of the International Conference on Trust Management (iTrust), (pp. 347-354).LNCS 2995. Springer Berlin / Heidelberg / New York.

Halkidis, S. T. & Tsantalis, N. & Chatzigeorgiou, A. & Stephanides, G. (2008). Architectural risk analysis of software systems based on security patterns, (pp. 129 - 142). IEEE Transactions on Dependable and Secure Computing, 5(3). IEEE Computer Society.

Hall, J. G. & Jackson, M. & Laney, R. C. & Nuseibeh, B. & Rapanotti, L. (2002). Relating Software Requirements and Architectures using Problem Frames. *In Proceedings of IEEE International Requirements Engineering Conference (RE),* (pp. 137-144). IEEE Computer Society.

Hatebur, D. & Heisel, M. & Schmidt, H. (2008). A formal metamodel for problem frames. *In Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 68–82). LNCS 5301. Springer Berlin / Heidelberg / New York.

Hatebur, D. & Heisel, M. & Schmidt, H. (2007). A pattern system for security requirements engineering. *In Proceedings of the International Conference on Availability, Reliability and Security (AReS)* (pp. 356-365). IEEE Computer Society.

Hatebur, D. & Heisel, M. & Schmidt, H. (2006). Security engineering using problem frames. In G. Müller (Ed.), *Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS)* (pp. 238-253). LNCS 3995. Springer Berlin / Heidelberg / New York.

Hatebur, D. & Heisel, M. (2005). Problem frames and architectures for security problems. In B. A. Gran & R. Winter & G. Dahll (Ed.), *Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP)* (pp. 390-404). LNCS 3688. Springer Berlin / Heidelberg / New York.

Heisel, M. (1998). Agendas - a concept to guide software development activities. In Proceedings of the IFIP TC2 WG2.4 working Conference on Systems Implementation: Languages, Methods and Tools (pp. 19-32). Chapman & Hall London.

Hoare, C. A. R. (1986). Communicating Sequential Processes. Prentice Hall. Retrieved June 24th, 2009, from http://www.usingcsp.com.

International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) (2006). Common evaluation methodology 3.1, ISO/IEC 18405. Retrieved June 24th, 2009, from http://www.commoncriteriaportal.org.

Jackson, M. (2001). Problem Frames. Analyzing and structuring software development problems. Addison-Wesley.

Jackson, M. & Zave, P. (1995). Deriving Specifications from Requirements: an Example. In Proceedings of the Internation Conference on Software Engineering (SE) (pp. 15-24). ACM Press New York / USA.

Jürjens, J. (2003). Secure systems development with UML. Springer Berlin / Heidelberg / New York.

Lai, L. & Lai, L. & Sanders, J. W. (1997). A refinement calculus for communicating processes with state. *In Proceedings of the Irish Workshop on Formal Methods: Electronic Workshops in Computing.* Springer Berlin / Heidelberg / New York.

van Lamsweerde, A. (2004). Elaborating security requirements by construction of intentional anti-models. *In Proceedings of the International Conference on Software Engineering (ICSE),* (pp. 148-157). IEEE Computer Society.

Li, Z. & Hall, J. G. & Rapanotti, L. (2008). From requirements to specifications: a formal approach. *In Proceedings of the International Workshop on Advances and Applications of Problem Frames (IWAAPF)* (pp. 65-70). ACM Press.

Lin, L. & Nuseibeh, B. & Ince, D. & Jackson, M. (2004). Using abuse frames to bound the scope of security problems. *In Proceedings of IEEE International Requirements Engineering Conference (RE)* (pp. 354-355). IEEE Computer Society.

Mantel, H. (2003). *A Uniform Framework for the Formal Specification and Verification of Information Flow Security.* Unpublished doctoral dissertation, Universität des Saarlandes, Saarbrücken, Germany.

Moriconi, M. & Qian, X. & Riemenschneider, R. A. & Gong, L. (1997). Secure software architectures. *In Proceedings of the IEEE Symposium on Security and Privacy* (pp. 84 – 93). IEEE Computer Society.

Mouratidis, H. & Giorgini, P. (2007). Secure Tropos: A security-oriented extension of the Tropos methodology, (285-309). *International Journal of Software Engineering and Knowledge Engineering*, 17(2).

Mouratidis, H. & Weiss, M. & Giorgini, P. (2006). Modelling secure systems using an agent oriented approach and security patterns. International Journal of Software Engineering and Knowledge Engineering (IJSEKE), (471-498). 16 (3).

Rapanotti, L. & Hall, J. G. & Jackson, M. & Nuseibeh, B. (2004). Architecture Driven Problem Decomposition. *In Proceedings of IEEE International Requirements Engineering Conference (RE),* (73-82). IEEE Computer Society.

RSA Laboratories. (1999). *Password-Based Cryptography Standard PKCS #5 v2.0.* Retrieved June 24th, 2009, from ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf

Santen, T. (2008). Preservation of probabilistic information flow under refinement, (pp. 213-249). *Information and Computation*, 206(2-4).

Santen, T. & Heisel, M. & Pfitzmann, A. (2002). Confidentiality-preserving refinement is compositional - sometimes. In Proceedings of the European Symposium on Research in Computer Security (ESORICS), (pp. 194-211). LNCS 2502. Springer Berlin / Heidelberg / New York.

Scandariato, R. & Yskout, K. & Heyman, T. & Joosen, W. (2008). Architecting software with security patterns (Report No. CW515). Katholieke Universiteit Leuven - Department of Computer Science.

Schmidt, H. (2009). Pattern-based confidentiality-preserving refinement. *In Engineering Secure Software and Systems - First International Symposium (ESSoS)*, (pp. 43-59). LNCS 5429. Springer Berlin / Heidelberg / New York.

Schmidt, H. & Wentzlaff, I. (2006). Preserving software quality characteristics from requirements analysis to architectural design. *In Proceedings of the European Workshop on Software Architectures (EWSA)*, (pp. 189-203). LNCS 4344/2006. Springer Berlin / Heidelberg / New York.

Schneier, B. (1999). Attack trees. Dr. Dobb's Journal. Retrieved June 24th, 2009, from http://www.schneier.com/paper-attacktrees-ddj-ft.html.

Schumacher, M. & Fernandez-Buglioni, E. & Hybertson, D. & Buschmann, F. & Sommerlad, P. (2005). *Security Patterns: Integrating Security and Systems Engineering.* Wiley & Sons.

Shaw, M. & Garlan, D. (1996). Software Architecture - Perspectives on an Emerging Discipline. Prentice-Hall.

Spivey, M. (1992). The Z Notation - A Reference Manual. Prentice Hall. Retrieved June 24th, 2009, from http://spivey.oriel.ox.ac.uk/mike/zrm.

Steel, C. & Nagappan, R., & Lai, R. (2005). Core security patterns: Best practices and strategies for J2EE, web services, and identity management.

SUN Java Standard Edition 6 API. Retrieved June 24th, 2009, from http://java.sun.com/javase/6/docs/api/overview-summary.html.

SUN javax.crypto.* API. Retrieved June 24th, 2009, from http://java.sun.com/javase/6/docs/api/javax/crypto/package-summary.html.

UML Revision Task Force, Object Management Group (OMG) (2007). OMG Unified Modeling Language: Superstructure. Retrieved June 24th, 2009, from http://www.omg.org/spec/UML/2.1.2/.

UML Revision Task Force, Object Management Group (OMG) (2006). Object Constraint Language Specification. Retrieved June 24th, 2009, from http://www.omg.org/docs/formal/06-05-01.pdf.