SoSPa: A System of Security Design Patterns for Systematically Engineering Secure Systems

Phu H. Nguyen§*, Koen Yskout†, Thomas Heyman‡, Jacques Klein*, Riccardo Scandariato†‡, and Yves Le Traon∗

§ Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway
* SnT-University of Luxembourg, 4 rue Alphonse Weicker, L-2721 Luxembourg
†‡iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
* Chalmers & University of Gothenburg, Sweden

Abstract—Model-Driven Security (MDS) for secure systems development still has limitations to be more applicable in practice. A recent systematic review of MDS shows that current MDS approaches have not dealt with multiple security concerns systematically. Besides, catalogs of security patterns which can address multiple security concerns have not been applied efficiently. This paper presents an MDS approach based on a unified System of Security design Patterns (SoSPa). In SoSPa, security design patterns are collected, specified as reusable aspect models to form a coherent system of them that guides developers in systematically addressing multiple security concerns. SoSPa consists of not only interrelated security design patterns but also a refinement process towards their application. We applied SoSPa to design the security of crisis management systems. The result shows that multiple security concerns in the case study have been addressed by systematically integrating different security solutions.

I. INTRODUCTION

Model-Driven Security (MDS) specialises Model-Driven Engineering approach for secure systems development, but still has limitations to be more applicable. Our recent systematic review of MDS [11] shows that multiple security concerns have not been addressed systematically by existing MDS studies. Indeed, the interrelations or dependencies among security solutions have not been considered systematically by current MDS approaches. Developing modern secure systems must always address multiple security concerns to minimise different security leaks and to make these systems resilient to different security attacks. A solution to address a specific security concern often depends on other solutions addressing other security concerns. For instance, most authorisation mechanisms depend on authentication mechanisms because before an authorisation decision, the authorisation mechanism should have known the identity of the requester. Authentication mechanisms often rely on encryption mechanisms, especially for distributed systems. Furthermore, there could be a lot of different variations of security solutions to address the same security concern. All urge for an MDS approach that can systematically address multiple security concerns, considering interrelations among security solutions and their variants.

Besides, from the security engineering’s point of view, one of the best practices is the use of security patterns to guide security at each stage of the development process [12]. Patterns are applied in the different architectural levels of the system to realise security mechanisms. So far, catalogs of security patterns are the most accessible, well organised, documented resources of different security solutions for different security concerns, e.g. [2, 12, 14]. But the results of a recent empirical study [15] show that using existing catalogs of security patterns improves neither the productivity of the software designer, nor the security of the design. Indeed, security patterns could be applied at different levels of abstraction, e.g. architectural design rather than detailed design. Moreover, the levels of quality found in security patterns are varied, not equally well-defined like software design patterns [4]. Particularly, many security patterns are too abstract or general, without a well-defined, applicable description. There is also a lack of some coherent specification of the interrelations among security patterns, and with other quality properties like performance, usability. In some catalogs, each security pattern is described having its related patterns, and possible impacts on other quality properties mentioned, but without any more practical or implementation details. To the best of our knowledge, none of existing MDS approaches has proposed a System of Security design Patterns which provides not only well-defined security design patterns but also the interrelations among security patterns that can guide developers in systematically dealing with multiple security concerns.

In this paper, we propose an MDS framework based on a System of Security design Patterns (SoSPa) that allows practitioners to systematically address multiple security concerns in secure systems development. Our security patterns in SoSPa are theoretically based on well-known security design patterns (e.g. in [2, 12, 14]). They are collected, specified as reusable aspect models (RAM) [6] to form a coherent system of them. While not only specifying security patterns at the abstract level like in security patterns catalogs, SoSPa also provides a refinement process supported by RAM to derive the detailed security design patterns closer to implementation. In other words, a software designer can reuse our security design patterns that are specified at different abstraction levels as RAM models. By using SoSPa, an integrated security solution dealing with multiple security concerns can be systematically engineered into a system. Not only the security design patterns but also their interrelations are specified in SoSPa. Based on SoSPa, the conflicts and inconsistencies among the applied
security solutions in a system design can be detected, resolved, or eliminated systematically. This may help to improve the security in a system design against different security threats. Because we propose an MDS development framework, SoSPa is built on a meta-model, which is extended from RAM meta-model. Our MDS framework allows selecting, refining, composing security design patterns to systematically build security solution models, and then automatically integrating them into a target system design. The contribution of this paper is three fold: 1) hierarchical RAM models with a refinement process for specifying security design patterns from abstract level till detailed design level; 2) the explicitly specified interrelations among security design patterns for systematically dealing with multiple security concerns; 3) an MDS framework supporting secure systems development based on SoSPa.

In the remainder of this paper, Section [I] provides some fundamental concepts and motivational examples for our MDS approach. Then, we present our MDS approach based on SoSPa in Section [III] and some key security design patterns of SoSPa in Section [IV]. Section [V] shows how our approach has been evaluated and discussed. Related work is given in Section [VI]. Finally, Section [VII] presents our conclusions and future work.

II. BACKGROUND AND MOTIVATIONAL EXAMPLES

A. System of Security Patterns

According to Schumacher et al. [12], a security pattern describes a particular recurring problem that arises in specific contexts and presents a well-proven generic scheme for its solution. Security patterns typically do not exist in isolation because applying one solely can not make a system secure to different threats. Related patterns should be integrated for working together to address more complex security problems in the real world. A system of security design patterns is an integrated collection of patterns for designing secure systems, together with guidelines for their implementation, combination, and practical use in secure systems development. Some catalogs of security patterns exist but might not be considered as systems of security design patterns in which patterns are well, concretely interconnected.

B. Reusable Aspect Models

RAM [6] is an aspect-oriented multi-view modelling approach with tool support for aspect-oriented design of complex systems. In RAM, any concern or functionality that is reusable can be modelled using class, sequence, and state diagrams in an aspect (RAM) model. A RAM model can be (re)used within other models via its clearly defined usage and customisation interfaces [1]. The usage interface of a RAM model consists of all the public attributes and methods of the class diagrams in the model. The customisation interface of a RAM model consists of all the parameterised model elements (marked with a vertical bar |) of the partially defined classes and methods in the model. A RAM model can be (re)used by composing the parameterised model elements with the model elements of other models. A RAM model can also reuse other RAM models in a hierarchical way. RAM weaver is used to flatten aspect hierarchies to create the composed design model.

We find that security patterns can be well specified by using RAM approach. RAM’s multi-view modelling ability makes it possible to capture even complex semantics of security patterns. The hierarchical modelling support of RAM enables a refinement process in which security patterns can be refined from abstract level till detailed design level. Fig. [1] shows a RAM model of a Session pattern which reuses the generic RAM model of Map (or ZeroToManyAssociation alternatively) [6]. The RAM model of Map [6] is composed of a generic data container |Data using a “Map” structure to store data in pairs of |Key and |Value. Session reuses Map by composing parameterised elements of Map with model elements of Session as can be seen in the “Map instantiation” box. For example, the mapping |Value to |Attribute means that attributes in a session are stored as |Value objects managed by the Map structure. The |Attribute element itself is a parameter. Any object can be stored in a session by mapping it to the |Attribute parameter. The RAM model of Session itself has the customisation interface comprises the parameterised classes |Session, |SessionID, |Attribute, and |AttributeType. For example, |SessionID is a parameterised class which will be instantiated as a unique identity associated with a session.

C. Motivational Examples

This section first recalls the case study of designing and developing Crisis Management Systems (CMS) described in [7]. Next, CMS’s potential misuse cases related to multiple security concerns are described. Then, we show why dealing with multiple security concerns systematically is very hard, even by leveraging security patterns in existing catalogs.

Briefly mentioning, in CMS a crisis can be created, processed by executing the rescue missions defined by a super observer, and then assigning internal and/or external resources. Fig. [2] shows a partial design of CMS, for creating a rescue mission. CMS are also security-critical systems whose different users must be authenticated, authorised to execute certain operations, and their communication be secured.
networks must be protected, and responsibility of users must be clearly traced. In [7], only a simple use case for CMS user authentication is provided. We show different security threats/misuse cases of CMS (informally, not exhaustively due to space restrictions) as follows.

**Misuse cases related to user accounts, access control:** [MUC-A1] An attacker impersonates a CMSEmployee after obtaining the user password by guess and try; [MUC-A2] A colleague of an authenticated user misuses the system on the authenticated user’s working device while it is being left unattended and accessible; [MUC-A3] A CMSEmployee gains disallowed access to the protected resources. CMSEmployees must only have access rights according to their assigned roles; [MUC-A4] A CMSEmployee has access rights to the system as a FirstAidWorker but also a SuperObserver. Conflict-of-interest roles, e.g. FirstAidWorker and SuperObserver, cannot be assigned to the same user.

**Misuse cases related to accountability data:** [MUC-B1] A CMSEmployee has received mission information concerning a car crash but ignores or overlooks some crucial information, and does not accept the mission. The rescue mission fails. When confronted, the CMSEmployee denies having received the mission information. [MUC-B2] A SuperObserver wrongly created a rescue mission which led to its failure. The SuperObserver manages to delete the corresponding log entry of his wrong action, and denies it.

**Misuse cases related to transmitted data:** [MUC-C1] An attacker intercepts usernames and passwords barely transmitted from client to server to impersonate a valid CMSEmployee; [MUC-C2] An attacker intercepts the mission information about a crisis barely transmitted from the system to CMSEmployee. Similarly, an attacker may intercept victim’s identity and/or medical history information transmitted from the HospitalResourceSystem to the CMS and/or from the CMS to a FirstAidWorker. Advanced attacker even could modify the transmitted victim’s medical history information.

For most security experts, not saying security novices, finding the best possible solutions addressing multiple security concerns, e.g. described in the misuse cases, and integrating them properly into the CMS would be a very big challenge. Developing and integrating different security solutions addressing multiple security concerns into a system is hard, making them work together consistently is harder. Leveraging security patterns seems to be a good approach for practitioners because security patterns are fairly well documented to address different security concerns. Moreover, some security patterns also contain some informal inter-pattern relations, and constraints regarding other quality properties such as performance to guide the patterns selection process. For tackling [MUC-A1], one may decide to use the patterns in [12], e.g. to ensure the complexity of user passwords, make password reset frequently, combine user passwords with one-time-password (OTP). [MUC-A2] and [MUC-A3] can be mitigated by using the patterns of access control in [2][12][14]. For tackling [MUC-B1] and [MUC-B2], the Audit Interceptor and/or Secure Logger patterns [14], or the Security Logger and Auditor pattern [2] can be used. For tackling [MUC-C1] and [MUC-C2], one may decide to use the Secure Channel pattern or Secure Pipe pattern [14], or the TLS pattern in [3].

But there is still a big gap between the intention and practical application of security patterns. Security patterns are often too abstract with good intention but no clear semantics that make them difficult to be implemented and applied, especially together. One can see that in the existing catalogs of security patterns, e.g. [2,12,14], the interrelations among patterns and other constraints are only briefly mentioned but not concretely specified to be applicable. All of these could lead to inappropriate implementation and application of security patterns. For example, a not well-thought design decision could lead to a weak user passwords authentication solution that allows a FirstAidWorker to guess and successfully impersonate a SuperObserver. Improperly integrating authentication, user session, and authorisation solutions could lead to access rights misused, and sensitive data leaked. Even worse, wrongly implementing an encryption channel for data transmission and also an auditing mechanism that intercepts and records the transmitted data may result in encrypted log entries that are useless for auditing purposes. Similarly, constructing a logging solution for accountability must be aware of an existing authorisation solution in the same system to produce the logs correctly. Depending on how these two work together, the logs might contain nothing, or meaningless info, or different types of info about successful executions of method calls, or failed authentication/authorisation checks for the method calls, or sometimes also successful authentication/authorisation checks. A sound approach for systematically addressing multiple security concerns in secure systems development is needed, but has not existed yet at least in the MDS research area.

### III. Our Model-Driven Security Approach Based on SoSPA

#### A. Overview of Our Approach

Our MDS approach is based on a System of Security design Patterns (SoSPA). Fig. 3 displays our meta-model of SoSPA (SoSPa-MM) that is an extension of RAM meta-model [6]. The core elements of SoSPa-MM are depicted in white. The rest are core elements of RAM meta-model. SoSPA aims at systematically addressing the globally accepted security concerns such as confidentiality, integrity, availability, accountability. Thus, SoSPA is composed of an extensible set of security solution blocks, e.g. authentication, authorisation, cryptography. Each security solution block consists of interrelated security design patterns. To support the selection of security design patterns, we use feature modelling as in software product line engineering to capture the variability and interrelations of security patterns. Specified by a feature model (FM), each security solution block can be used to form a specific, customised security solution. Each security design pattern in SoSPA contains all well-structured elements such as context, problem, consequences as can be seen in well-documented security patterns of existing catalogs. More than that, inter-pattern relations are captured and explicitly specified at the conceptual level as well as model level by
Fig. 3. The Meta-model of the System of Security design Patterns (SoSPa-MM)

RAM models. Fig. 3 shows that each SecurityDesignPattern is associated with ReusableAspect(s) that realised the pattern. In other words, security design patterns are specified as reusable aspect models (RAM) to form a coherent system of them, i.e. SoSPa. The interrelations among security patterns are categorised into five types as specified in RelationshipType. We capture the core relation types among security patterns, i.e. depends on, benefits from, alternative to, impairs, conflicts with. These relations can be transitive and/or symmetrical.

Five main security solution blocks of SoSPa are Authentication, Authorisation, Cryptography, Auditing, and Monitoring as can be seen in Fig. 4. Derived from security requirements, a customised security solution can be built up from a combination (OR relation) of these security solution blocks. Each feature (node) of the FM can be associated directly with a RAM model. For example, Authentication feature is directly specified by a RAM model named Authentication. The features with underlined names are security patterns which can be refined by composing the hierarchical RAM models realising them. For example, Security Session pattern is realised by the RAM model SecuritySession, and also the other relevant RAM models like SessionManager, Session. Some low-level features are not really security patterns but generic RAM models which help building security patterns, e.g. Map, ZeroToManyAssociation. Because in SoSPa, security patterns are built on hierarchical RAM models (see Section IV), the interrelations among security patterns are actually specified at the model/design level. In other words, the interrelations are specified based on the relations of RAM models that the security patterns are built on. We elaborate more on this in Section III-C

B. Pattern-Driven Secure Systems Development Process

This section presents the development process in three main stages, especially emphasising the selection and composition of security design patterns into a target system design.

[Security threats identification & analysis]: This is not the focus of this paper. We assume that misuse cases are created in this stage. Attack models might be created from risk analyses, e.g. using the CORAS framework as discussed in [3].

[Security design patterns selection and application]

Step 1 - Constructing security solutions from the security patterns in SoSPa: For each security concern, the interrelations specified in the feature model (Fig. 4) are used to select the most appropriate security design patterns, i.e., the pattern that best matches with the context and the security problem, most satisfies the interrelations with the other already selected security design patterns, and maximises the positive impact on relevant quality properties like usability, performance. All the RAM models of the selected security design patterns and other required RAM models are woven into the RAM model of the top most feature in the hierarchy corresponding to the security concern of the FM. This step derives a detailed RAM design of a customised security solution for the concern, including its customisation interface and usage interface. For example, to construct a customised authentication solution, all the selected features under Authentication feature are woven into it. The output of this step is a complete RAM model, i.e. the woven RAM model of the authentication solution. This woven RAM model of the authentication solution later can be integrated into a base system model via its customisation interface. More details can be find in the case study described in Section V.

Step 2 - Defining mappings to integrate the newly built security solutions to a base system model: For each selected security pattern, use the customisation interface of the generated design to map the generic design elements to the application-specific context. This step generates the mappings of the parameterised elements in the security design pattern with the target elements in the target system design. Any constraints/conflicts between mappings of all the selected security design patterns need to be resolved. Most constraints are predefined by SoSPa for the obvious interrelations (e.g. L1...
C. The Interrelations of Security Patterns in SoSPa

This section shows how five interrelations among security patterns at conceptual level can be realised at detailed design (model) level in SoSPa.

1) Depend-on Relation: Security pattern X depends on security pattern Y means that X will not function correctly without Y. This relation is not symmetrical, but transitive. In SoSPa, this relation is specified as the mandatory “required” relation among RAM models that realise the security patterns. Let security pattern X be realised by RAM model A; let security pattern Y be realised by RAM model B. X depends on Y means that model A (directly or indirectly) depends on (requires) model B. Thus, model A realising security pattern X can only be applicable to any base system if all the RAM models that A depends on, such as model B, have been woven into A. In Fig. 4 the patterns of Authorisation and Auditing depends on the patterns of Authentication. This means security solution of Authorisation must be completed by a security solution of Authentication.

2) Benefit-from Relation: Security pattern X benefits from security pattern Y means that implementing Y will add to the value already provided by implementing X. At design level, X benefits from Y if RAM model A realising X optionally depends on RAM model B realising Y. For example, Authentication patterns can benefit from SecuritySession pattern. But it is up to designers to decide if the chosen Authentication pattern needs to use SecuritySession.

3) Alternative-to Relation: Security pattern X is alternative to security pattern Y means that X provides a similar security solution like Y’s. The designer can choose either X or Y for addressing the same security problem. For instance, DirectAuthentication pattern is alternative to ThirdPartyAuthentication.

4) Impair-with Relation: Security pattern X impairs with security pattern Y means that X and Y are not recommended together. In case X and Y are both selected for working together, they may result in inconsistencies. A conflict resolution could be provided to make them working consistently together. For example, Load Balancer pattern impairs with SecuritySession pattern. If no conflict resolution can be found, we say that X conflicts with Y.

5) Conflict-with Relation: Security pattern X conflicts with security pattern Y means that implementing Y in a system that contains X will result in inconsistencies. An example is that the AuditInterceptor pattern could conflict with the SecureChannel pattern.

IV. SECURITY DESIGN PATTERNS IN SoSPA

Due to space restrictions, this section only presents some key security design patterns of SoSPA. We show how hierarchical RAM models are used to specify security patterns from abstract level till detailed design level. Besides, each security pattern is presented with its interrelations to the others.

A. Authentication Patterns

The Authentication feature is specified by a RAM model with the most basic authentication logic (Fig. 5). For every call to any protected method |m| of any protected class |ProtectedClass|, the caller must be already authenticated before method |m| is executed. The underlying authentication mechanisms are abstract and to be refined by composing the model elements |Authentication, authenticate, and check with the parameterised elements of the selected RAM models, e.g. SecuritySession, DirectAuthentication that Authentication depends on. The FM shows that designer could choose different alternatives below the Authentication feature.
for designing a customised authentication solution, e.g. DirectAuthentication, ThirdPartyAuthentication\(^1\) or JAASAuthentication\(^2\).

Fig. 6 shows the RAM model of DirectAuthentication pattern which is based on the AuthenticationEnforcer pattern in \([14]\) and the Authenticator pattern in \([2]\). RequestContext contains the user’s principal and credential extracted from the protocol-specific request mechanism. An instance of RequestContext is often provided by a specific implementation framework. We do not discuss this RequestContext class in details. By using the input Principal, the DirectAuthentication retrieves the corresponding Credential from an identity store that it manages using a Map aspect. The input Credential is checked against the retrieved Credential. Using DirectAuthentication requires designer to decide on what kinds of shared secrets to be used for authentication. The abstract aspect AuthenticationMeans shows that share secrets could be user password, biometics, or one time password (OTP). These features could be used together, e.g. user password with OTP. If using user password for authentication, the Password\(^*\) pattern will be woven into the DirectAuthentication aspect.

Third-party authentication provider (ThirdPartyAuthentication\(^*\)) can also be used to validate client’s credentials. The main idea of this pattern is to map the authentication process to a proxy to call the authentication method provided by a third-party authentication provider. Shared secrets among the third-party provider and their clients are invisible to the authentication solution being constructed.

On the other hand, session can bring more benefits to an authentication solution, e.g. for maintaining an authenticated status. We describe security session patterns in the next section. The SecuritySession pattern is optional to Authentication.

Note that the order of dependencies specified on the top of each RAM model is important to make the patterns work consistently together. For example, the order of weaving dependencies into Authentication must be from left-to-right, and then top-down, i.e. LimitedAttempts, SecuritySession, DirectAuthentication. The orders of weaving are also part of SoSPa to provide for designers. The RAM weaver can execute the orders of weaving dependencies automatically. We elaborate more on this in the case study given in Section\(\bigstar\).

B. Security Session Patterns

The FM in Fig. 4 also shows how the aspects related to security session are organised. The patterns for security session are based on the SecuritySession pattern in \([12]\), the ControlledAccessSession pattern in \([2]\), and the AuthenticationEnforcer, SecureSessionObject patterns in \([14]\). In SoSPa, the SecuritySession pattern depends on the SessionManager\(^*\) aspect for managing sessions. The designer can choose between a generic Session pattern showed in Fig. 1 or the SecureSessionObject\(^*\) pattern \([14]\). SecuritySession presented in Fig. 7 specifies how a SecuritySession object is created and used for maintaining the authenticated status of a subject. Logically, if the validation process in authentication returned by the \(\text{check}\) method is successful, a new session object is created. Then, any security-related information can be stored in this object, e.g. the validated Principal. The authenticated status is associated with the session as long as the session is active. The SessionTimeout\(^*\) pattern provides a timing mechanism that requests authenticating again if the corresponding session is expired.

C. Authorisation Patterns

Fig. 8 shows the corresponding Authorisation RAM model. Authorisation can be employed together with Authentication if the optional AuthenticationDependency aspect is selected (see Fig. 9). AuthenticationDependency presented in Fig. 9 shows that authentication must be done before the method \(\text{in}\) called. The Authorisation RAM model itself contains the Authorisation class with evaluateReq function to evaluate any request AccessRequest to method \(\text{in}\) of a protected resource \(\text{ProtectedClass}\). The AccessRequest is created with all necessary elements of a method call such

\(^{1}\)Due to space restrictions, patterns marked with a star * are not presented.
The AuthorisationEnforcer pattern in Fig. 10 contains the AuthorizationEnforcer class that processes any request access based on other RAM models such as PolicyBasedAC*, ControlledAccessSession*.

D. Auditing (Accountability)

In critical systems, the ability to keeping tracks of who did what and when is very important. Security patterns for auditing can solve this accountability concern. The RAM models for auditing and their interrelations specified in Fig. 4 are based on the Secure Logger, Audit Interceptor patterns [14], and the Security Logger and Auditor pattern [2]. Two common patterns for auditing are SecureLogger and AuditInterceptor* in which the latter depends on the former. Fig. 11 shows the structural view and message view of the SecureLogger pattern. The classes and methods being traced are mapped to the parameterised Traced class and method |m| once the Trace object and the identity of caller are created, they are sent to the SecureLogger for being logged. How a Trace object can be created is specified by the generic Traceable* aspect mentioned before. The LogManager is responsible for the actual serialisation of the log (using LogFactory* aspect) to a secure storage (either using SecureLogStore* pattern or SecureDataLogger* pattern [14]).

SecureLogger in Fig. 11 is a generic logger that just simply logs a trace whenever the method |m| is called. There is no specification on whether |m| has been successfully authenticated and/or authorised, or actually executed. Different variations of when and how a trace is logged are provided in different logging strategy aspects. For example, aspect AuthenticationDependence in Fig. 12 specifies that a trace is logged only if the caller to |m| has been authorised successfully and |m| has been executed.

To summarise, we have presented some key security patterns of SoSPa and their dependencies to each others, and to other RAM models. Note that the order of dependencies on top of each RAM model matters to the order of weaving. How dependencies are woven into a RAM model are specified by instantiation mappings.

V. EVALUATION AND DISCUSSION

A. Case Study and Results

By using SoSpa with the patterns selection and application process described in Section III-B multiple security concern/misuse cases of CMS can be addressed/mitigated properly. We demonstrate the three main steps of the patterns selection and application process as follows.
about the features below SecuritySession. This hierarchical aspects composition is applied to every feature in the feature model. Thus, all the selected features below Authentication are automatically woven into it, according to the order of dependencies specified on top of Authentication, and the model elements mappings defined in the corresponding RAM models. For example, once LimitedAttempts is selected, it is woven into Authentication first. In this way, a concrete authentication solution, namely wovenAuthentication RAM model has been built and ready to be integrated into CMS base design to fulfil its user authentication requirements and mitigate its potential misuse cases.

Similarly, a concrete authorisation solution can be built to mitigate the misuse cases [MUC-A3] and [MUC-A4]. Assuming AuthorisationEnforcer with PolicyBasedAccessControl' and Role-Based Access Control (RBAC)' have been selected. All the selected RAM models for the authorisation solution such as AuthorisationEnforcer, PolicyBasedAccessControl' are woven into the Authorisation RAM model to create a wovenAuthorisation RAM model.

And so on, misuse cases [MUC-B1] can be mitigated by constructing a suitable Auditing solution, e.g. using SecureLogger. [MUC-B2] is also mitigated by using SecureLogger because either SecureLogStore* or SecureDataLogger* is employed to protect the logged data from being tampered. All the selected RAM models for SecureLogger are woven into the SecureLogger RAM model, resulting in a wovenAuditing RAM model.

To mitigate misuse cases [MUC-C1] and [MUC-C2], SecureChannel pattern [14], or TLS pattern as presented by [3], can be employed to secure transmitted data. We do not discuss more details about this due to space restrictions.

Step 2 - Mapping the security solutions to the CMS base design: For each security solution built in the previous step, its parameterised model elements can be mapped to the target elements in the CMS design to integrate the security solution into CMS. Note that the customisation interface of the top-most RAM model (e.g. Authentication) in the hierarchy of a security solution is also the customisation interface of that security solution (wovenAuthentication). Let us make the createMission function of CMS secure and its execution logged. We would come up with the following mappings: wovenAuthentication.|ProtectedClass→CrisisManager wovenAuthentication.|m→createMission wovenAuthorisation.|ProtectedClass→CrisisManager wovenAuthorisation.|m→createMission wovenAuditing.|Traced→CrisisManager wovenAuditing.|m→createMission

As we see, the constraints among the mappings have to be resolved. From the security requirements of CMS, the authorisation solution has to work with the existing (already built) authentication solution. Thus, the AuthenticationDependence feature (or L1 in Fig. 4) must also be selected for the authorisation solution wovenAuthorisation. That means wovenAuthentication is woven into AuthenticationDependence, and their woven
RAM model is then woven into `wovenAuthorisation` to create a `wovenAuthenticationAuthorisation`. By doing so, the constraints among `wovenAuthentication` and `wovenAuthorisation` have been resolved. Similarly, to log a wrongly created rescue mission action of a `SuperObserver` in CMS as described in [MUC-B2], the `AuthorisationDepend` feature (or L2 in Fig. 4) is needed for `wovenAuditing`. By weaving `wovenAuthenticationAuthorisation` with `AuthorisationDependence` and then into `wovenAuditing` to create a `wovenAllSolutions` model, all the constraints have been resolved. After that, the instantiation directives for integrating all the security solutions into the CMS base design are straightforward:

**wovenAllSolutions.**

**Step 3 - Weaving the security solutions into the CMS base design:** This step can be automatically done by the RAM weaver once all the mappings and constraints have been specified in the previous step. Fig. [3] shows final woven model. For space reasons, we only display the key parts of the customised authentication, authorisation, and auditing solutions woven into the base model. The woven model shows that the `createMission` action now can only be executed by an authenticated user that is authorised to execute this task, and a trace of this action is securely logged. Of course, a formal analysis of the woven model against attack models could show a formal proof that the woven model is resilient to different attacks. Constructing attack models in a similar way as security solution models and then employing formal analysis techniques could be a good direction for future work.

**B. Discussion**

Our work raises an important question related to the abstraction level proposed by SoSPa. More specifically, the question is how can we guarantee that the level of details is sufficient? Or how to guarantee that there is no need to develop new RAM models or security patterns? Our answer is that the required level of details strongly depends on the expected use of SoSPa. Obviously, if SoSPa was used to generate code that can run, the low-level details of a design would have to be provided. If the goal was to generate the code skeleton of a secure system and test cases, the low-level details could be omitted. The level of details of RAM models in SoSPa is not enough for full code generation, but enough for specifying all the important semantics of security patterns and their interrelations. That means the code skeleton of a secure system can be generated that preserves the important semantics of the employed security patterns and their interrelations.

Each security design pattern can also be associated with the side effects of its adoption on other quality properties, e.g. performance, usability. An impact model of the security design patterns for each quality property can be built as discussed in [1]. Impact models are useful for analysis of the trade-off among alternatives which leads to a thoughtful decision on systematically selecting the right security design patterns for the job. On the other hand, attack models can also be specified using SoSPa-MM. These attack models are associated to the security concerns, and can be woven into the system model to generate misuse models for formal analysis of security, or generate test cases for testing. In this paper, we only focus on the interrelations among security patterns, not yet considering the relations to other quality properties and attack models. The types of interrelations in this paper are aligned with the five types of inter-pattern relations presented in [16].

A second question is related to the completeness and extendibility of SoSPa. As previously discussed, the set of security patterns mentioned in this paper is not complete, mainly due to space constraints. We have nonetheless considered the most illustrative for the purpose of our work. However, an interesting feature of SoSPa is that it is fully extensible. As a result, if a user realises that some details or some security patterns are missing, he can extend it. This is eased by the use of RAM and the explicitness of the relationships among the security patterns.

In a final note, we recall that to the best of our knowledge, SoSPa is the first attempt to provide an extensive set of concrete design models of security patterns that can be integrated systematically. We provide RAM models that users can explore according to several dimensions: not only from high to low level of details, but also in the variability perspective traversing a large number of possible security solutions.

**VI. RELATED WORK**

Different MDS approaches can be found in [11]. This section briefly presents the closest related work only. Aspect-oriented modelling; Georg et al. [3] propose a methodology that allows not only security mechanisms but also attacks to be modelled as aspect models. The attacks models can be composed with the primary model of the application to obtain the misuse model. The authors then use the Alloy Analyser
to reason about the misuse model and the security-treated model. Mouheb et al. [9] develop a UML profile that allows specifying security mechanisms as aspect models. The aspect models often go together with their integration specification to be woven automatically into UML design models. Recently, Horcas et al. [5] propose a hybrid AOSD and MDE approach for automatically weaving a customised security model into the base application model. By using the Common Variability Language (CVL) and xπ, different security concerns can be woven into the base application in an aspect-oriented way, according to weaving patterns. However, dependencies between the security aspects and their application orders have not been taken into account. In general, all these approaches have not considered security aspects as a coherent system capturing their interrelations and constraints. Patterns-based: Three main catalogs of security patterns are presented in [2] [12] [14]. These catalogs only contain abstract security patterns without any refinement process towards their application and any MDS framework. Shiroma et al. [13] propose an approach to leverage model transformations for specifying and implementing application procedures of security patterns, including inter-pattern dependencies. The inter-pattern dependencies here mean the order of consecutive applications of security patterns by performing consecutive transformations. In fact, the approach only presents the dependencies in terms of the order of application of security patterns. There are many other important interrelations that one must consider such as conflicts, benefits, and alternatives among security patterns. Their approach is only able to deal with 8 per 27 security patterns in [12] because the other 19 patterns do not have structures described. With our extensive, extensible SoSPa, all key interrelations among security patterns are considered, not only the order of patterns application.

Our MDS approach based on SoSPa initially explored in the position paper [10] is inspired by [1]. Alam et al. [1] propose an approach based on RAM for designing software with concern as the main unit of reuse. They show how their ideas can be realised by using an example of low-level design concern, i.e. the Association concern. The adoption of their approach to high-level concerns like security has not been dealt with yet. In this paper, we realised the ideas in [10] by developing a system of RAM models specifying multiple security patterns and their interrelations to form SoSPa. We extend the concept of using variation interface in [1] for specifying the interrelations among security patterns. With SoSPa, abstract security patterns can be specified, refined as detailed designs with concrete semantics. Additionally, the interrelations among patterns can be concretely specified in the variation interfaces and in the detailed designs, thanks to RAM. We also plan to extend the idea of using impact model in [1] for specifying the constraints of security patterns with other quality properties like performance, usability.

VII. Conclusions and Future Work

This paper has presented an MDS approach based on a System of Security design Patterns (SoSPa) to systematically guide and automate the application of multiple security patterns in secure systems development. Our SoSPa is specified by using an extended meta-model of RAM, namely SoSPa-MM. Based on SoSPa-MM, security patterns are collected, specified as reusable aspect models to form a coherent system of them. Our MDS framework allows systematically selecting security design patterns, constructing security solutions, and automatically composing them with a target system design. We evaluated our approach by leveraging the System of Security design Patterns to design the security of crisis management systems. The result shows that multiple security concerns of the case study have been addressed. More importantly, the different security solutions are thoughtfully selected and systematically integrated. Our work on SoSPa opens three points for future work: 1) extending the impact models in [1] for specifying the side-effects of security patterns regarding other quality properties like performance, usability; 2) incorporating the attack models to generate misuse models for formal analysis of security [3]; 3) using test templates embedded into the security patterns for testing their application [8].

Acknowledgments

Supported by the Fonds National de la Recherche (FNR), Luxembourg, under the MITER project C10/IS/783852.

References