Automatic Enforcement of Security Properties

Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes

CAOSD Group, Universidad de Málaga, Andalucía Tech, Málaga, Spain
{horcas,pinto,lff}@lcc.uma.es

Abstract. Ensuring the security requirements of an application is not a straightforward task. Security properties (e.g., confidentiality, anonymity) need to be satisfied in different ways in different parts of the same application. Software architects are usually required to manually define security components and their dependencies with the base application, customize them to the application's requirements, identify the points where security is incorporated, and verify that the selected places are correct. The last two steps are especially complex and error-prone. In our approach, we aim to provide a solution that helps software architects to identify the correct places to incorporate the security functionality and to verify the correctness of the composed application architecture. This is achieved by identifying a set of general structural patterns for incorporating security into the application architecture, and by providing a model-driven SPL solution to customize these patterns to each application's requirements.

Keywords: Encryption · Security pattern · Software architecture · SPL

1 Introduction

It is well known that the development of applications that ensure their security requirements is not a straightforward task [1–3]. This is because security properties (e.g., confidentiality, authentication, etc.) need to be satisfied in different ways in different parts of the same application. Even the same security property usually needs to be satisfied differently in multiple parts of the same application. For instance, in order to preserve confidentiality the application's sensitive data has to be encrypted. But, where and how does the data need to be encrypted?

In our example, confidentiality is guaranteed by adding the encryption behavior in different places of the application. For instance, during the interaction of two components that exchange sensitive data, it could be added at the place where the data is encrypted, and the place where the same data is to be decrypted. However, encryption could be applied differently to different kinds of interactions. For example, only remote interactions, involving components that are deployed in different hosts, may be required to encrypt sensitive informa-tion. Sensitive data could be encrypted for all the interactions managing sensi-tive information, independently from the location of components. Or, encryption is required for all interactions managing sensitive data but providing different

-

security levels — i.e., using different encryption algorithms, depending on the local or remote location of the components. Moreover, guaranteeing the secure storage of the information requires the data to be encrypted before storing it and decrypted after retrieving it. More variability is introduced if we consider that the sensitive data have to circulate securely — i.e., encrypted, through different components of the application. This means that the component where the data is encrypted and the component where the data is decrypted do not directly interact with each other, as there are third components between them. Finally, different kinds of sensitive data may require different levels of security, requiring the use of different encryption algorithms (e.g., RSA, AES,...). Thus, it is not trivial for software architects to correctly answer the where-and-how question.

A first step toward mitigating this problem is applying the separation of concerns principle to model the variability of security from the early stages of the software's development. Thus, security concerns are modeled separately from the base applications and later customized to the application's requirements and composed at particular points of the application model [4,5]. It has been demonstrated that this approach has many advantages [4–6], such as high reusability, low coupled components and high cohesive software architectures. Moreover, security can be more easily customized to the application's requirements. Following this approach, our previous work [5] provided support to: (1) model the variability of security independently from the application, (2) instantiate the security model according to the requirements of a particular application, and (3) compose the customized security model with the application model.

However, in order to compose the customized security model and the application model, first, the join points — i.e., points in the application model where the elements of the security model have to be injected/composed, have to be identified. In our previous work, the join points were identified completely manually. This resulted in a lack of support to guarantee the required security level, since analyzing whether the security components had been introduced in all and the correct places was not possible. Other similar approaches [4,6,7] have the same limitation. Thus, the benefits of reusing the security models are lost. In this paper, we improve upon our previous approach by providing support to ensure that security is correctly incorporated in the applications. This is achieved by: (1) automatically identifying the places in the software architecture where a particular security functionality has to be incorporated, and (2) checking whether the security functionality was incorporated correctly to a software architecture.

In order to automate the identification of the join points we need to understand that security models are not completely oblivious to the application models, and thus some dependencies between them need to be taken into account during both the security modeling and during the incorporation of the security functionality inside the application. Without this, the automatic identification of the join points is impossible. Concretely, as part of the variability of the security properties, we need to model the variability of the different structural and behavioral patterns (e.g., the remote/local direct/indirect interactions, data storage, etc. in the case of encryption) previously discussed. The main reason is that the identification of the join points and the definition of the composition rules largely depend on these patterns. Moreover, formally defining these patterns and the composition rules based on them will provide our approach with the support that is required to verify the correct deployment of security.

In this paper we focus on confidentiality, although our ultimate goal is to identify a set of general structural and behavioral patterns to incorporate many other security properties into an application's architecture, and to customize these patterns to each application's requirements. We use a Software Product Line (SPL) [8] to specify the variability of the composition patterns, and model-to-model (M2M) transformations to identify the join points from the patterns and to guarantee that the final architecture satisfies the security requirements.

The paper is structured as follows. Section 2 motivates our work with a case study. Section 3 presents our SPL to model and instantiate the variability of the security patterns. Section 4 explains the automatic identification of join points from the patterns. Section 5 qualitatively evaluates our approach. Section 6 discusses related work and Sect. 7 sets out our conclusions and future work.

2 Motivating Case Study

Our case study is an electronic payment (e-payment) application for making payments for different services (taxi, restaurants, donations,. . .) and chasing up receipts. This kind of application requires strong security requirements such as preserving confidentiality of the user's information, integrity of the data, and access control, among others. In this paper, we focus only on confidentiality.

Figure 1 shows a simplified UML software architecture with the basic functionality of the application. The `PaymentApp` component allows users to make payments for a specific service, and request the proof of payments by using the `EPayment` interface. The customer's information (e.g., payment card details) is stored on the user's device (`CustomerProfileManager` component). Additionally, this information can be synchronized between different user devices (`SynchronizationData` component). The server manages the payments through the `EPaymentServer` component that uses the `ServiceDomainResolution` and `BankTransaction` components to identify the service's provider information and to complete the transactions with the banks, respectively. The server also tracks a history of the users' transactions (`TransactionsHistory` component).

Apart from this basic functionality shown in Fig. 1, it is of paramount importance to guarantee the following security requirements, among others:

- *Req. 1: Confidentiality.* Sensitive information (i.e., payment card data) exchanged between the client and the server hosts must be encrypted (e.g., using the RSA algorithm). This means that it is required to encrypt the payment information (information of type `PaymentMethod` or the parameter `payInfo` in Fig. 1) when: (1) a payment is made — i.e., the `PaymentApp` client component calls the `pay` method of the `EPayment` interface; and when (2) the client synchronizes the list of payment methods — i.e., the component `CustomerProfileManager` calls the `synchronize` method of the `SynchInfo` interface. Then, the information has to be decrypted when the

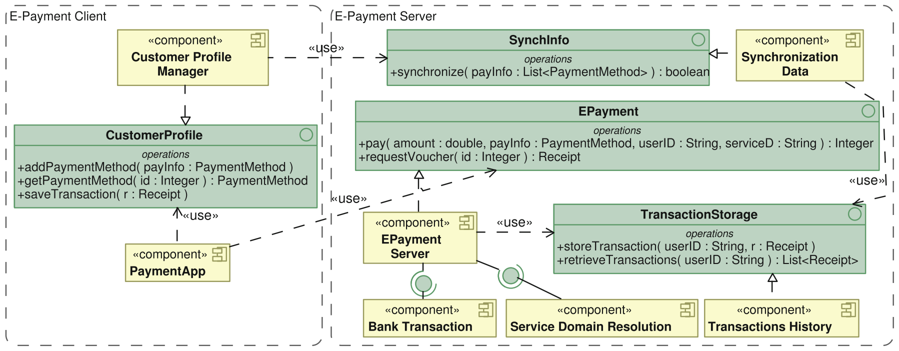

E-Payment Client
E-Payment Server
«component» Customer Profile Manager
«use»
SynchInfo
operations
+synchronize(payInfo : List<PaymentMethod>) : boolean
«component» Synchronization Data
EPayment
operations
+pay(amount : double, payInfo : PaymentMethod, userID : String, serviceD : String) : Integer
+requestVoucher(id : Integer) : Receipt
CustomerProfile
operations
+addPaymentMethod(payInfo : PaymentMethod)
+getPaymentMethod(id : Integer) : PaymentMethod
+saveTransaction(r : Receipt)
«use»
«use»
«use»
«component» PaymentApp
«component» EPayment Server
«use»
TransactionStorage
operations
+storeTransaction(userID : String, r : Receipt)
+retrieveTransactions(userID : String) : List<Receipt>
«component» Bank Transaction
«component» Service Domain Resolution
«component» Transactions History

Fig. 1. Software architecture of the e-payment application.

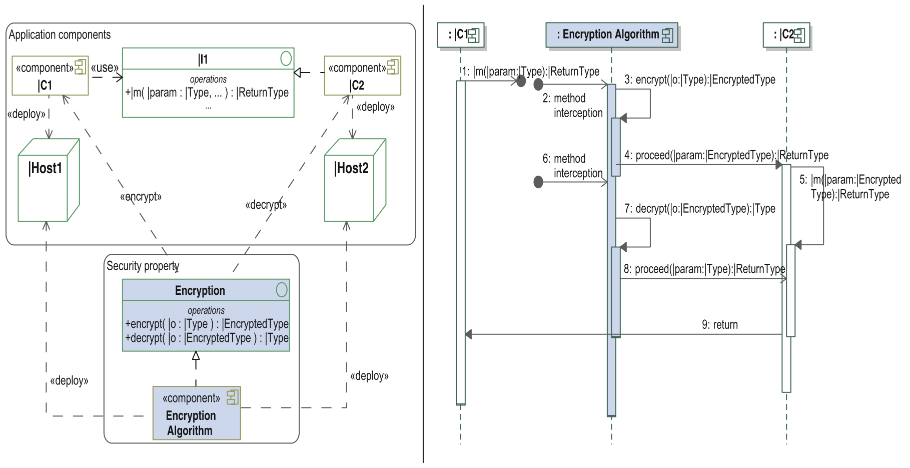
server receives it, in both `EPaymentServer` and `SynchronizationData`. Following our approach the software architect only needs to indicate that the payment card data is the sensitive data and then our automatic join point identification approach indicates those join points where encryption and decryption need to be incorporated in order to ensure the confidentiality of the sensitive data.

- *Req. 2: Confidentiality.* All information exchanged between the `EPaymentServer` and `BankTransaction` components inside the server must also be encrypted, regardless of the type of information or the interface used. In this case, the software architect indicates that interactions between two specific components need to be encrypted and all the affected join points are automatically identified by our approach.
- *Req. 3: Confidentiality.* The payment card details are stored in the user's device using a different encryption algorithm from the one for communications (e.g., AES). Thus, another encryption algorithm is required to encrypt the payment methods information when they are stored/retrieved in the user's device. In this case, both encrypting and decrypting functionalities are required by the same component (`CustomerProfileManager`). Here, our approach inspect the application looking for a structural pattern that represents a data storage and the join points would be automatically detected.

3 Capturing the Security Variability

To accomplish the automatic identification of the joint points, we identify a set of structural patterns that specifies the relationships with the application. The variability of the patters is modelled in an SPL, together with the variability of the security functionality. Then, the software architect instantiates the patterns and the security functionality according to the application's requirements.

A security pattern describes a particular, recurring security problem (e.g., applying encryption) that arises in specific contexts, and presents a well-proven generic solution for it [9]. In the case of the confidentiality property, we need


Application components
«component»
|C1
«use»
|I1
operations
+|m(|param : |Type, ...) : |ReturnType
...
«component»
|C2
«deploy»
«deploy»
|Host1
|Host2
«encrypt»
«decrypt»
Security property
Encryption
operations
+encrypt(|o : |Type) : |EncryptedType
+decrypt(|o : |EncryptedType) : |Type
«deploy»
«deploy»
«component»
Encryption Algorithm
: |C1
: Encryption Algorithm
: |C2
1: |m(|param:|Type):|ReturnType
2: method interception
3: encrypt(|o:|Type):|EncryptedType
4: proceed(|param:|EncryptedType):|ReturnType
5: |m(|param:|EncryptedType):|ReturnType
6: method interception
7: decrypt(|o:|EncryptedType):|Type
8: proceed(|param:|Type):|ReturnType
9: return

(a) Structural view (b) Behavioural view

Fig. 2. Encryption pattern for secure communications.

a set of patterns that allows to apply the encryption functionality to different parts of the application such as remote/local direct/indirect interactions, and data storage. Encryption is usually defined as a component that provides two main functionalities: `encrypt` and `decrypt` (see Fig. 2), which normally intercept (*crosscut*) the application functionality at specific points. For example, the pattern to apply encryption in a secure communication is shown in Fig. 2 and states that the `encrypt` method intercepts a "sender component" (`|C1`) in order to encrypt the message information (`|param`) before sending it (i.e., calling the method `|m`), while the `decrypt` method crosscuts a "receiving component" (`|C2`) to decrypt the message information after receiving it. Figure 2 represents a parameterizable structural (a) and behavioral (b) view of this pattern for encrypted communications. The top of Fig. 2(a) shows the dependencies of the pattern with the architectural elements of the application architecture (`Application components`). The bottom of Fig. 2(a) shows the encryption component (`EncryptionAlgorithm`) with the provided functionality (`Encryption` interface). The pattern captures the information that is required to incorporate encryption into the interaction between two components. Throughout this paper, we only use the structural view for the sake of simplicity, but patterns can be complemented with additional views as shown in Fig. 2(b).

The partial or total instantiation of the parameters of this pattern, with information obtained from the application's requirements, allows correctly applying encryption for straightforward communications — i.e., applying encryption to two communicating components that use a common interface. However, this pattern does not capture all the situations in which encryption may have to be incorporated. For instance, it does not allow applying encryption in situations that do not involve a communication, such as storing encrypted data in a device, or applying encryption to interactions between two non-adjacent components.

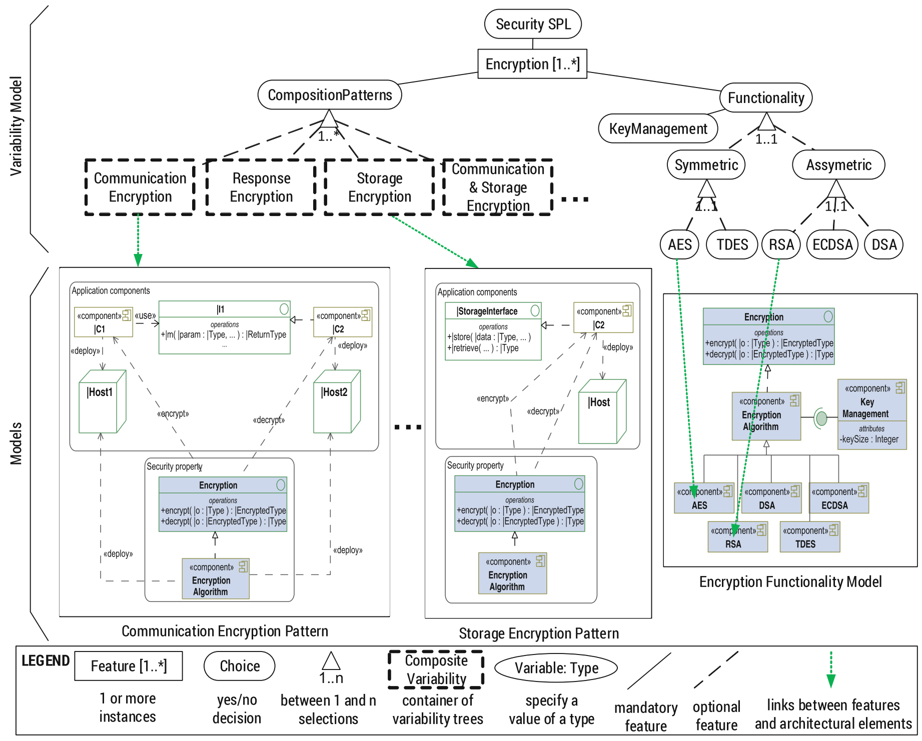


Fig. 3. Variability model for the encryption patterns and encryption functionality.

Thus, as it is impossible for only one pattern to cover all kinds of interactions with the application, we propose modeling the variability of the security patterns following a Software Product Line (SPL) [8]. Concretely, we extend the variability model modeling the security functionality in our previous work to enhance it with the variability of the security patterns. An SPL[1] allows us to specify the commonalities and variabilities of a product and then generate specific configurations of the product according to different requirements.

Figure 3 shows a variability model that specifies, using *features* in an abstract level (top of Fig. 3), the variability of the encryption patterns (left of the figure) and the variability of the encryption functionality (right of the figure). Then, specific models (e.g., the software architecture of encryption, the structural/behavioral patterns for encryption) are linked to the features of the abstract tree. Note that using existing tools for SPL (e.g., CVL [10], SPLOT [11]), the architectural models in the bottom of Fig. 3 will be automatically instantiated according to the features selected from the abstract tree. Concretely, the bottom of Fig. 3 shows two parameterizable encryption patterns and the model of the encryption functionality (`Encryption Functionality Model`), including all the

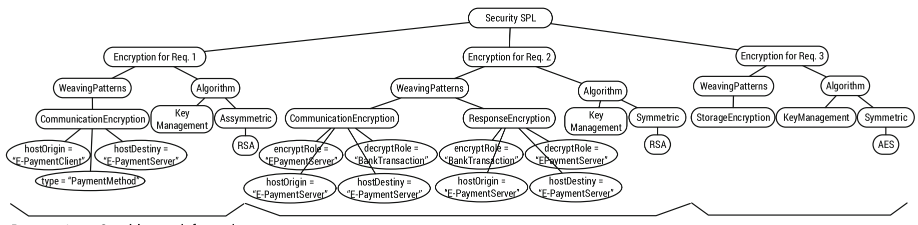
[1] http://www.sei.cmu.edu/productlines/.

variable architectural elements. Notice that security properties are usually modeled by much more complex architectures, though in this example we only represent the encryption algorithms for the sake of simplicity. For the `Encryption Functionality Model`, a selection of a particular feature in the tree selects the encryption algorithm that will be used. The multiplicity feature (`Encryption [1..*]` in Fig. 3) indicates that both the algorithms and the patterns can be instantiated multiple times in order to use different algorithms and patterns.

3.1 Resolving the Variability of the Application

Once all the variability of the security functionality and the patterns has been defined in the SPL (only once) by the domain experts, the software architect can use our approach to generate different configurations of the patterns and the security functionality according to each application's requirements.

A complete configuration of the variability model from requirements `Req. 1`, `2`, and `3` of our case study is shown in Fig. 4. There are three instances of the encryption feature: one for each requirement. The first instance (`Encryption for Req. 1`) is configured with the `RSA` algorithm, and uses the `Communication-Encryption` pattern instantiated as shown in Fig. 5(a), with the goal of encrypting the sensitive information exchanged between the client and the server host. The second instance (`Encryption for Req. 2`) is also configured with the `RSA` algorithm, but uses two patterns (`CommunicationEncryption` and `Response-Encryption`) to apply encryption in both directions of the communications between the components `EPaymentServer` and `BankTransaction` of the same host (`E-PaymentServer`). Since all information exchanged between these two components is required to be encrypted, no information regarding the type of the data, nor the interface, methods, etc. is provided by the software architect. Finally, the third instance (`Encryption for Req. 3`) is configured with the `AES` algorithm, and uses the `StorageEncryption` pattern in order to storage the payment information in a secure way. In this case, the software architect has not instantiated any parameter of the pattern, as shown in Fig. 5(b). This could



Req. 1. Sensitive information (`PaymentMethod`) exchanged between the client (`E-PaymentClient`) and the server (`E-PaymentServer`) hosts must be encrypted using the `RSA` algorithm.

Req. 2. All information exchanged between the `EPaymentServer` and `BankTransaction` components inside the server (`E-PaymentServer`) host must be encrypted using the `RSA` algorithm.

Req. 3. The payment card details are stored in the user's device using the `AES` algorithm.

Fig. 4. Instance of the variability model for the encryption functionality and patterns.

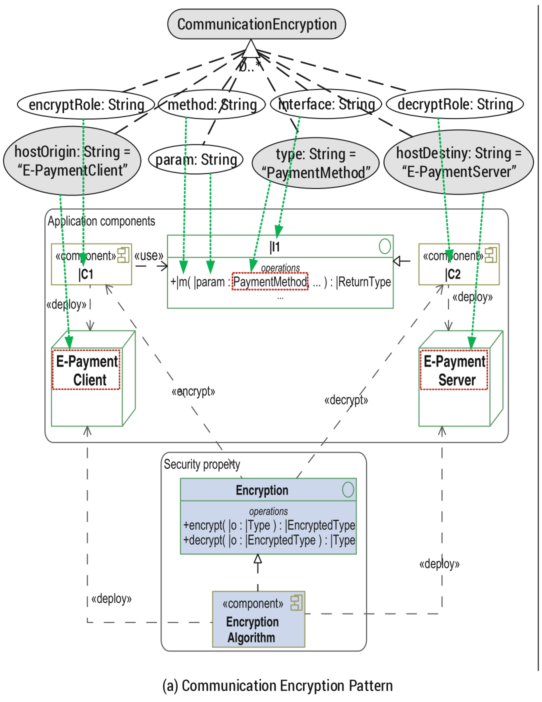
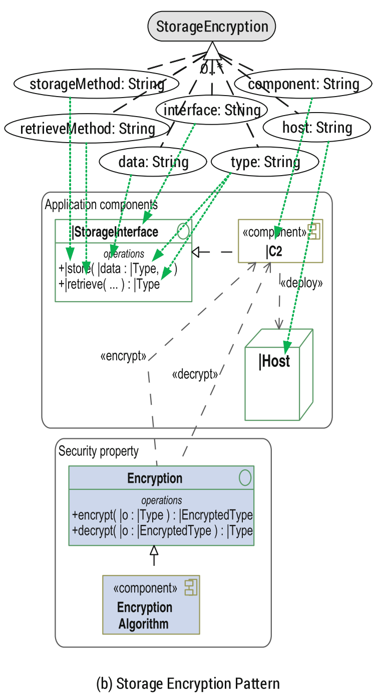

CommunicationEncryption
0..*
encryptRole: String
method: String
interface: String
decryptRole: String
hostOrigin: String = "E-PaymentClient"
param: String
type: String = "PaymentMethod"
hostDestiny: String = "E-PaymentServer"
Application components
«component» |C1
«use»
|I1
operations
+|m(|param : |PaymentMethod, ...) : |ReturnType
«component» |C2
«deploy»
E-Payment Client
E-Payment Server
«encrypt»
«decrypt»
Security property
Encryption
operations
+encrypt(|o : |Type) : |EncryptedType
+decrypt(|o : |EncryptedType) : |Type
«component» Encryption Algorithm
«deploy»
(a) Communication Encryption Pattern


Fig. 5. Instances of the encryption patterns for (a) *Req 1.* and (b) *Req. 3.*

occur when the requirements do not provide enough information, the software architect does not know how to interpret the requirements, or does not have the required knowledge to instantiate the pattern. In this case our approach identifies a larger set of join points and the software architect has to manually select the correct ones. At least it knows all the matching points that need to be considered.

The encryption patterns show the parameters that can be customized according the application's requirements. The patterns for applying encryption to the communications (`Communication Encryption Pattern`) and to the storage of information (`Storage Encryption Pattern`) are described in more details in Fig. 5(a) and (b), respectively. Providing a value in the feature tree means that this element in the pattern will be instantiated with the provided value. For instance, to satisfy *Req. 1*, the software architect has instantiated the `Communication Encryption Pattern` of Fig. 5 (a) with the following parameters: the data type of the information to be encrypted (i.e., the `PaymentMethod` type), and the identifiers of the client and the server hosts (`E-PaymentClient` and `E-PaymentServer`). The following section explains how we correctly identify the join points from the previously customized patterns in our approach.

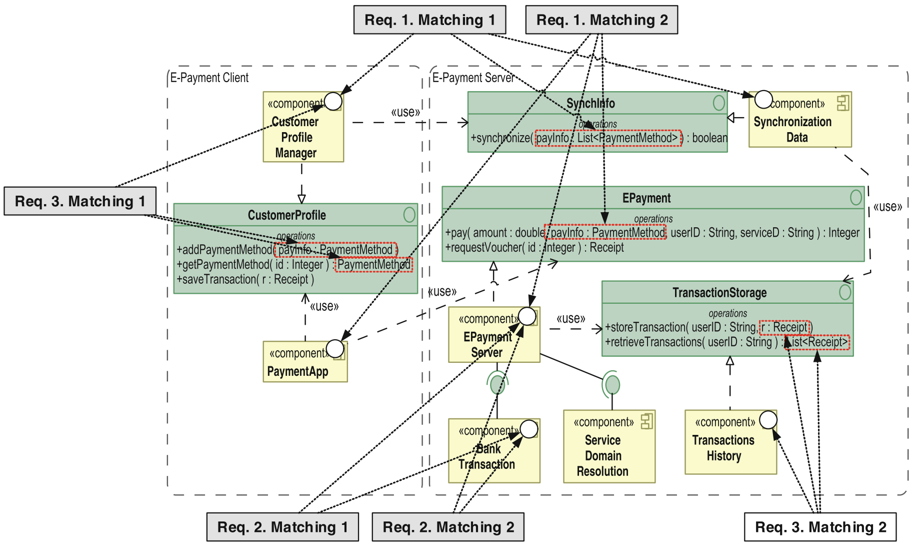
4 Supporting the Composition Process

Once the variability model has been instantiated, now the security and the application models are composed. To achieve this, the composition patterns customized in the previous step are mapped on structures in the application architecture by using M2M transformations. The mapping process can be used in two complementary ways: (1) guiding the software architects in selecting the correct join points, and (2) supporting them in verifying their choices of join points.

4.1 Automatically Identifying the Join Points

In order to identify the join points where the customized patterns have to be applied and guaranteeing that the final architecture satisfies the security requirements, the model transformations can be treated as a separate previous step to the composition process. Before the composition process, checking each instantiated pattern with our e-payment application architecture (Fig. 1) finds all possible matchings where the pattern can be applied (Fig. 6). The number of identified matchings directly depends on the number of parameters for which a specific value was provided during the instantiation of the variability model.

For instance, the first instantiated pattern (`Encryption for Req. 1`) matches the application model in the join points `Req. 1. Matching 1` and `Req. 1. Matching 2` in Fig. 6. The software architect provided just the data type to be encrypted (`PaymentMethod`) and the hosts' information, while the concrete components where the encrypt and decrypt methods will be composed has



Req. 1. Matching 1
Req. 1. Matching 2
E-Payment Client
E-Payment Server
«component» Customer Profile Manager
«use»
SynchInfo
operations
+synchronize(payInfo : List<PaymentMethod>) : boolean
«component» Synchronization Data
Req. 3. Matching 1
CustomerProfile
operations
+addPaymentMethod(payInfo : PaymentMethod)
+getPaymentMethod(id : Integer) : PaymentMethod
+saveTransaction(r : Receipt)
EPayment
operations
+pay(amount : double, payInfo : PaymentMethod, userID : String, serviceD : String) : Integer
+requestVoucher(id : Integer) : Receipt
«use»
«use»
«use»
«use»
TransactionStorage
operations
+storeTransaction(userID : String, r : Receipt)
+retrieveTransactions(userID : String) : List<Receipt>
«component» PaymentApp
«component» EPayment Server
«component» Bank Transaction
«component» Service Domain Resolution
«component» Transactions History
Req. 2. Matching 1
Req. 2. Matching 2
Req. 3. Matching 2

Fig. 6. Matchings for the encryption patterns in the e-payment architecture.

been automatically identified. To satisfy *Req. 2*, two patterns were instantiated: `CommunicationEncryption` and `ResponseEncryption`. Each of them matches the application architecture in `Req. 2. Matching 1` and `Req. 2. Matching 2`, respectively. In this case, the identifiers of the two communicating components were directly provided: the `PaymentServer` and the `BankTransaction` components. All information exchanged between these two components will be encrypted before sending and decrypted after being received. Finally, for *Req. 3*, the `StorageEncryption` pattern was instantiated without specifying any parameters. This implies that there will be multiple matchings for this pattern, as `Req. 3. Matching 1` and `Req. 3. Matching 2` in Fig. 6. Both matchings are correct for this pattern. However, *Req. 3* only specifies that the payment card information that is stored in the user's device need be encrypted (`Req. 3. Matching 1`) and, thus, the information about the receipts of the transactions (`Req. 3. Matching 2`) does not need to be encrypted. In such a case, we give the software architect the opportunity to make an explicit choice of the matching, or to instantiate the pattern again by providing more specific information such as the type of data to be encrypted (e.g., the `PaymentMethod` in this case).

4.2 Verifying the Security Requirements

To demonstrate that the security functionality has been applied in the correct way and places, the M2M transformations can be applied to a model where the security model has already been composed with the application model. For instance, when the join points were manually identified. The same instance of the variability model shown in Figs. 4 and 5 can now be used to verify that the security property was correctly added to all the matchings of the final application model (the base application model composed with the security model). In our case study, a software architect could verify whether the final application model includes encryption in all the matchings shown in Fig. 6. Comparing the final application model and the matchings, the software architect may realise that encryption was not added to some of the identified matchings. This supposes a hole in the security of the application, but it can be easily resolved according to the information provided automatically by our approach. Thus, the tool supporting the instantiation of the patterns can also be used to check that the software architect has applied them in all the correct places.

5 Evaluation Results and Discussion

We have tested the validity of our approach by implementing the patterns and M2M transformations using the Henshin transformation language [12].[2] We have used two case studies: the e-payment application used throughout this paper, and an electronic voting application.[3] In this section we qualitatively argue the correctness, extendibility, and reusability of our approach.

[2] They are available at http://150.214.108.91/code/interfacesfqa/tree/master.

[3] http://inter-trust.eu/.

Correctness. The correctness of our approach depends on the correctness of the specification of the patterns and on the implementation of the M2M transformations. The patterns and M2M transformations are formally modeled conforming to a specific metamodel, so if the domain experts do their job correctly, the identification and checking of the join points will also be correct. Moreover, separately modeling the security functionality and the base application considerably facilitates the verification of the security properties of an application since a security expert can rely on the automatic output provided by applying the patterns, instead of manually checking all the modules in the base application to ensure that all security requirements have been correctly enforced. Finally, our approach is able to ensure the level of security required by an application even when the software architect is not completely aware of the elements in the application architecture that are affected by the security requirements, but is able to indicate at least, the structural patterns that are affected by security (e.g., encrypt the communications between components, or encrypt the data store in a data storage). In this case, our approach identifies a larger set of join points because most of the pattern's parameters are not specified. We can ensure that all of them are correct. The software architect then has two options: (1) add encryption to all the identified join points. This will guarantee that the security of the application is ensured, or (2) manually select a subset of them. In this case, our approach is not responsible for the security gaps that may be introduced.

Extendibility. In this paper we have focused on the confidentiality property and have shown in the SPL only the variability of the encryption algorithms. However, the SPL can be easily extended to cover more security properties such as authentication, integrity, anonymity, etc. Moreover, the variability model can consider any variable security functionality such as the management of the keys for encryption or the passwords for the authentication concern, not just the variation between algorithms. Note that although the intricacy of this approach may seem inadequate for only adding encryption to an application, our final goal is much more ambitious as this work is part of an approach to separately modeling the variability of quality attributes [5]. Concretely, we have an SPL modeling the variability of several quality attributes, not only security (e.g., contextual help, persistence) and the approach presented in this paper is applicable to all of them.

Reusability. Our approach improves the reusability of the security concerns by modeling the security functionalities separately from the core functionality of the application, from early stages. This reduces the coupling and increases the cohesion of software architectures. Also, thanks to the combined use of the separation of concerns and the SPLs, we can reuse the same security functionality and patterns with different applications.

6 Related Work

Security is usually achieved in several ways, but most of the approaches present security as a set of non-functional properties [6,9,13,14], instead of focusing on the functional part of the security concerns as we have done for confidentiality in this paper. For instance, in [6], the authors present a systematic approach for weaving non-functional requirements into software architecture using architectural tactics similar to our composing patterns. However, we consider security as extra-functionality that needs to be present as functional components inside the application architecture to satisfy the requirements. So, we focus on the identification of the correct places where security must be incorporated, instead of providing the systematic steps to perform the composition of the patterns, as we also did in previous work on composing security functionalities [5,15].

Cuevas et al. [7] also describe a generic solution for non-security experts using security patterns. The solution captures a security pattern that provides access control to sensor data based on light-weight encryption and grant provision. No means of how and where applying encryption functionality to the software architecture is described. Only the properties and functionalities that are common to all implementations of the encryption-based access control are captured using the security patterns, and thus, the customization of the patterns is too limited because of the lack of variability. QADA [16] is a specific method for designing SPL architectures by transforming systematic functionality into software architectures, but this proposal does not explicitly take into account the security requirements, so the semantic correctness of the final architecture cannot be checked, in order to assure the quality of the system.

Another approach that separately models the security functionality from the base application is CORE (Concern-Oriented REuse) [4]. Nevertheless, as the other existing work [5–7,15], they do not provide mechanisms to guarantee that security is deployed in all and correct places of the application architecture.

7 Conclusions and Future Work

In this paper we have presented an approach towards the automation of the composition process between application and security models. Specifically, the approach consists in modeling the variability of a set of patterns to incorporate the security functionality in the correct places of the application architecture, according to its requirements. This means that we provide the software architect with support for automatically identifying the join points where the security functionality has to be incorporated. So, instead of manually identifying the join points, as existing approaches propose, the system offers the software architect a set of join points. It also provides support to verify the correctness of a composed application architecture, so that the requirements of the system can be assured. As future work we plan to improve our approach by defining the patterns in terms of a security conceptual model that will allow the join points to be selected based on semantic instead of just syntactic information [17].

Acknowledgment. This work is supported by the project Magic P12-TIC1814 and by the project HADAS TIN2015-64841-R (co-financed by FEDER funds).

References

1. Preda, S., Cuppens-Boulahia, N., Cuppens, F., Garcia-Alfaro, J., Toutain, L.: Model-driven security policy deployment: property oriented approach. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 123–139. Springer, Heidelberg (2010)
2. Ayed, S., Idrees, M.S., Cuppens-Boulahia, N., Cuppens, F., Pinto, M., Fuentes, L.: Security aspects: a framework for enforcement of security policies using AOP. In: SITIS, pp. 301–308 (2013)
3. Mouelhi, T., Fleurey, F., Baudry, B., Le Traon, Y.: A model-based framework for security policy specification, deployment and testing. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 537–552. Springer, Heidelberg (2008)
4. Alam, O., Kienzle, J., Mussbacher, G.: Concern-oriented software design. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 604–621. Springer, Heidelberg (2013)
5. Horcas, J.M., Pinto, M., Fuentes, L.: An automatic process for weaving functional quality attributes using a software product line approach. J. Syst. Softw. **112**, 78–95 (2016)
6. Kim, S., Kim, D.K., Lu, L., Park, S.: Quality-driven architecture development using architectural tactics. J. Syst. Softw. **82**(8), 1211–1231 (2009)
7. Cuevas, A., Khoury, P.E., Gomez, L., Laube, A.: Security patterns for capturing encryption-based access control to sensor data. In: SECURWARE, pp. 62–67 (2008)
8. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, New York (2005)
9. Schumacher, M., Fernandez, E., Hybertson, D., Buschmann, F.: Security Patterns: Integrating Security and Systems Engineering. Wiley, Chichester (2005)
10. Haugen, Ø., Wasowski, A., Czarnecki, K.: CVL: common variability language. In: Software Product Line Conference, SPLC, vol. 2, pp. 266–267 (2012)
11. Mendonca, M., Branco, M., Cowan, D.: S.P.L.O.T.: software product lines online tools. In: Object Oriented Programming Systems Languages and Applications, OOPSLA, pp. 761–762. ACM (2009)
12. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010)
13. Yu, H., Liu, D., He, X., Yang, L., Gao, S.: Secure software architectures design by aspect orientation. In: ICECCS, pp. 47–55 (2005)
14. Hafiz, M., Adamczyk, P., Johnson, R.E.: Organizing security patterns. IEEE Softw. **24**(4), 52–60 (2007)
15. Horcas, J.M., Pinto, M., Fuentes, L.: An aspect-oriented model transformation to weave security using CVL. In: MODELSWARD, pp. 138–147 (2014)
16. Matinlassi, M., Niemelä, E., Dobrica, L.: Quality-driven Architecture Design and Quality Analysis Method: A Revolutionary Initiation Approach to a Product Line Architecture (2002)
17. Pires, P.F., Delicato, F.C., Pinto, M., Fuentes, L., Marinho, É.: Software evolution in AOSD: a MDA-based approach. In: CBSE, pp. 193–198 (2011)