Model-Based Security Analysis of Feature-Oriented Software Product Lines

Sven Peldszus
University of Koblenz-Landau
Germany
speldszus@uni-koblenz.de

Daniel Strüber
University of Koblenz-Landau
Germany
strueber@uni-koblenz.de

Jan Jürjens
University of Koblenz-Landau
Germany
juerjens@uni-koblenz.de

Abstract

Today's software systems are too complex to ensure security after the fact – security has to be built into systems by design. To this end, model-based techniques such as UMLsec support the design-time specification and analysis of security requirements by providing custom model annotations and checks. Yet, a particularly challenging type of complexity arises from the variability of software product lines. Analyzing the security of all products separately is generally infeasible. In this work, we propose *SecPL*, a methodology for ensuring security in a software product line. SecPL allows developers to annotate the system design model with product-line variability and security requirements. To keep the exponentially large configuration space tractable during security checks, SecPL provides a family-based security analysis. In our experiments, this analysis outperforms the naive strategy of checking all products individually. Finally, we present the results of a user study that indicates the usability of our overall methodology.

CCS Concepts • **Security and privacy → Software security engineering**; • **Software and its engineering → Abstraction, modeling and modularity**; **Unified Modeling Language (UML)**;

Keywords Security, Software Product Lines, OCL, UML

ACM Reference Format:
Sven Peldszus, Daniel Strüber, and Jan Jürjens. 2018. Model-Based Security Analysis of Feature-Oriented Software Product Lines. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3278122.3278126

GPCE '18, November 5–6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6045-6/18/11…$15.00
https://doi.org/10.1145/3278122.3278126

1 Introduction

The omnipresence of software systems has made our everyday lives considerably easier, and yet it gives rise to a rapidly growing multitude of security threats. Security is a business-critical factor in enterprises, since each security issue implies a potential loss of control over internal data and resources, with severe financial and reputation consequences. A recent developer study pinpoints security as the number-one concern to be addressed by future software analysis tools [12].

The paradigm of *security by design* emphasizes that security cannot be addressed merely retroactively, by identifying and fixing security loopholes; security has to be considered as a first-class citizen from the early development stages. A methodology to address security early in the development process is offered by model-based security approaches such as UMLsec [27]. UMLsec can be used to specify security requirements and security-related assumptions in design models, and to analyse the resulting models with regard to security goals based on a predefined threat model. UMLsec has been practically applied in distributed systems [6], mobile communications [30], and protocol engineering [5].

Security becomes yet more challenging during the development of *software product lines* (SPLs, [4]). An SPL is a family of software products sharing a set of core assets and differing in a set of *features*, that is, increments of functionality only present in some of the products. Since SPLs are useful for tailoring products to diverse customer needs, companies such as Bosch, HP, and GM develop business-critical software as SPLs [4]. However, developing an SPL is challenging due to the complexity arising from *variability*: an SPL with n features can include 2^n individual products. In domains like automotive engineering, where SPLs have thousands of features [62], the resulting problems during software engineering tasks, in particular those related to security, are of astronomical scale.

In many cases, practical solutions involve trade-offs between precision and tractability. I.e., during the testing of SPLs, developers rely on *sampling* [51], in which a selection of all products is considered to uncover implementation defects. However, in the case of security, sampling is problematic: a vulnerability affecting any of the products represents a potential leakage of secrets and, therefore, a business risk. Worse, focusing on a selection of all products for security engineering might actually be *harmful*: security measures

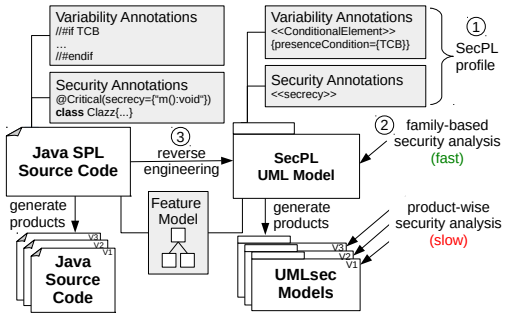

Figure 1. Overview of SecPL.

implemented in a subset of all products can be used by attackers to automatically generate exploits for the remaining products [47]. This calls for an efficient way to specify and analyze the security requirements of *all* products in an SPL.

In this paper, we present SecPL, a methodology for managing security requirements in SPLs systematically. Specifically, as shown in Fig. 1, we make the following contributions:

(1) The *SecPL profile* (Sec. 4), allowing users to specify security requirements and product-line variability in UML models. SecPL uses UMLsec's stereotypes for the specification of security requirements and security-related assumptions. To annotate structural and behavioral elements that only exist in some of the products, model elements can have presence conditions, that is, propositional expressions over a given set of features. The set of features is defined using a feature model, a standard SPL representation.

(2) A *family-based security analysis* (Sec. 5) for efficient security checks. Our analysis assumes an encoding of the security check at hand as an OCL constraint. We provide such encodings for the most prominent UMLsec checks; additional ones may be created by experts. For a given model, our analysis evaluates the constraint using a method called *template interpretation* [15], which results in an propositional formula that describes feature combinations which lead to a security violation. This propositional formula is fed to a SAT solver to obtain either a counterexample, that is, a subset of features giving rise to an insecure product, or a proof that all products are secure. By relying on efficient SAT solvers, we avoid the combinatorial explosion arising if each product is generated and analysed separately.

(3) A *reverse engineering mechanism* (Sec. 6) for extracting SecPL models from variability- and security-annotated code, making our analysis applicable to legacy SPLs.

(4) An *evaluation* (Sec. 7), demonstrating the usability of our methodology based on user feedback, and the performance of our analysis when applied to realistic models.

To our knowledge, our work is the first to support a model-based security analysis of all products in a software product line. While our analysis relies on template interpretation [15], one of our key contributions is provide suitable encodings of security constraints that we feed as input to template interpretation, similar to other analysis techniques that rely on a backend SAT solver. Moreover, to the best of our knowledge, our evaluation is the first to assess the benefit of a template-interpretation-based technique on a set of realistic models. We discuss related work in Sect. 8.

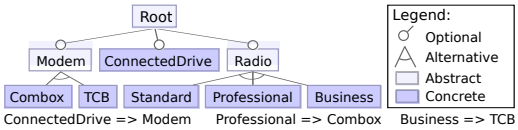

Figure 2. Feature model of an in-car system

Our methodology uses UML-based system models for capturing the system design and annotating it with security requirements. In industry, system models are used for various purposes, including informal communication, documentation, learning, and code generation; UML is the most widely applied modeling language in many software domains [59]. We rely on an existing approach called UMLsec and combine it with feature-based variability engineering. However, our approach is not limited to UML, but can be adapted to modeling languages with similar diagram types as well, for example, SysML [50] for automation systems.

2 Background

In this section, we introduce the background of our approach: model-based security analysis with UMLsec, and variability engineering.

UMLsec provides a UML profile that allows developers to annotate system models with security requirements and security-related assumptions. Based on a variety of provided stereotypes, UMLsec supports various security checks, including the analysis of security policies, secure information flow, and secure communication in protocols. Stereotypes are one of three extension mechanisms of UML and allow to extend it with domain specific language elements and to annotate UML model elements with those [49]. Similar to classifiers, stereotypes can have properties, which are called *tagged values*. UMLsec operates at the level of class diagrams, deployment diagrams, activity diagrams, sequence diagrams, and component diagrams. In the past, UMLsec has been applied for security analyses in diverse contexts such as protocol engineering [5], distributed information systems [6], and mobile communications [30].

Variability engineering is concerned with variability in systems. An important concept are *features*, units of functionality that can be configured, that is, switched on or off [4]. In standard SPL approaches, the set of features and their relations are specified in feature models [31]. Fig. 2 shows a particular feature model that we explain in the upcoming section. To establish traceability, features can be mapped directly to code and to design models: Preprocessor directives can be used to annotate feature-specific code portions, and the entire codebase can be divided into modules [4]. Design

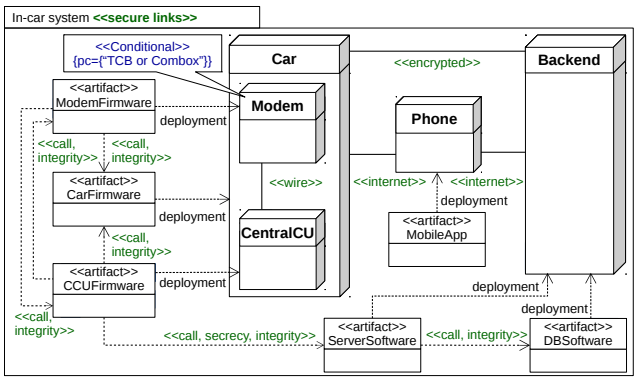

Figure 3. In-car system deployment diagram.

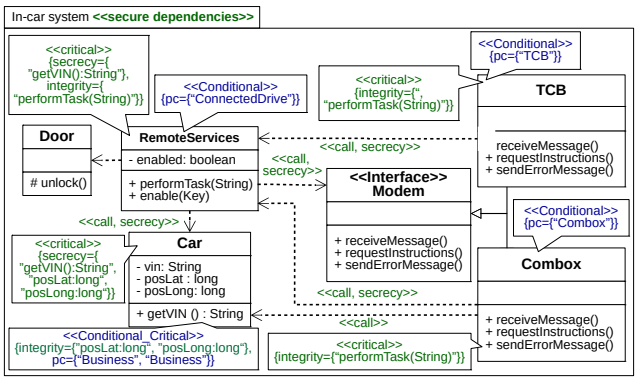

Figure 4. In-car system class diagram.

models can be annotated with presence conditions over a set of features [14].

3 Running Example

Modern cars are highly configurable software-intensive systems, made up of a magnitude of reusable hardware and software components. For example, consider an SPL of in-car systems for modern cars. An in-car system comprises traditional components such as the radio, but it may also furnish the driver with services such as traffic information, news playback, and a remote-control app, e.g., to unlock the driver's door using a mobile phone. The functionality available in a specific car can be expressed using features.

Fig. 2 shows an excerpt of a feature model for an in-car system, based on typical features of BMW cars. Customers can select between three kinds of radios; the radio selection determines which modem, if any, is built into the car, and if BMW's ConnectedDrive services can be used in this car.

Consider the following twist. In May 2015, an IT magazine demonstrated that ConnectedDrive was vulnerable to a security threat allowing the attacker to unlock the car doors [58]. The following root causes of the threat were revealed: (i) In its authentication protocol, the access protocol component relied on the secrecy of the vehicle identification number (VIN), while another component revealed the VIN with one of its error messages. (ii) Some versions of the software used DES, an encryption algorithm that was declared insecure in 2005. ConnectedDrive was launched in 2011. (iii) The component was prone to replay attacks, where the attacker could capture and replay a legitimate unlock message.

Each of these issues could have been avoided with an appropriate security mechanism in place. To capture this situation, in the following, we show an excerpt of a design model for an in-car system, focusing on the threats (i) and (iii). The used encryption algorithms can be specified in system models as well, using deployment diagrams in a similar way as presented here. The design model is annotated with SecPL's security-specific and variability stereotypes and tagged values. However, to introduce the example in a general way, let us temporarily ignore these annotations.

According to the deployment diagram shown in Fig. 3, the in-car system incorporates three types of devices: the car, a mobile phone, and back-end servers. These devices and their sub-devices are connected via physical communication paths based on Internet connections. In addition, the car's subsystems are related via physical communication path as well, based on wire. Moreover, the deployment diagram shows implementation artifacts with their logical dependencies and their deployment onto the included devices. For example, the *CCUFirmware* deployed onto the *Central Computing Unit (CCU)* can call the *CarFirmware* to unlock the driver's door.

In our example, we focus on the subsystem deployed onto the car. Specifically, Fig. 4 shows a class diagram containing the relevant parts for the example. The class *Car* stores the car's VIN and position, a class *Door* provides the functionality to unlock the car doors, a *Modem* such as *Combox* or *TCB* enables the communication with the mobile app and the backend, and a class *Remote Services* represents the following functionality: (1) enabling the remote services provided by ConnectedDrive, and (2) processing the commands received from the mobile app and the backend. The information if remote services are enabled is stored in a property *enabled* that can be set using the operation *enable(Key)*. Modems use the short-message service [57] for receiving text messages, to obtain instructions from the back-end over an Internet connection, and to respond error text-messages. The instructions obtained from the back-end such as unlocking the door are processed by the operation *performTask(String)* of the *RemoteServices* class.

Opening the driver's door via the mobile app involves the following actions in the car subsystem:

1. The modem receives a *requestInstructions* text message, establishes a connection to the Internet and requests detailed instructions from the back-end over a http-get request.
2. If the back-end provided a non-empty set of instructions, the modem calls *RemoteServices* with the instructions. Otherwise, it sends an *error* text-message to the app.
3. If there have been valid instructions the RemoteServices processes those and performs the assigned tasks like opening the door or enabling the RemoteServices otherwise the Modem is called to send an error message.

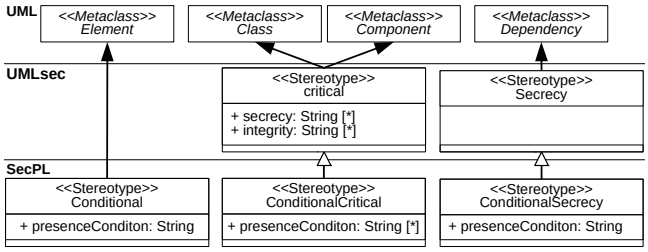

Figure 5. Excerpt from the SecPL UML Profile.

At the first glance, this procedure seems safe and secure. However, in the details, the aforementioned issues manifest themselves.

First, step 1 lacks an authorization of the *get instructions* request. Anyone capable of faking a GSM basestation—to this end, *femtocells*, a type of fully-featured short-range basestation, can be used—can send the required text message to the car and provide any instructions. This problem can be handled by enforcing encryption on the communication path between the car and the phone. To this end, we will use an extended form of UMLsec's *secure links* check.

Second, one may at least assume that disabling the remote services is an effective way of shielding the system against unauthorized access. Unfortunately, in the original setting [47], this is not the case for all products of the SPL: An issue in the implementation of the Combox firmware allows any third person to enable the remote services in products containing this modem: The key required for enabling the remote services is the VIN of the car. However, for identification reasons, error reports sent by the Combox always include the VIN! Attackers can exploit this behavior by opening a femtocell and sending text messages to perform invalid instructions. As a result, they receive an error report containing the VIN.

A possible explanation for this glaring, but nonetheless real security issue is the lack of a shared mental model of the system and its security requirements: The developer responsible for programming the Combox appears to have been unaware of the use of the VIN as a key for RemoveServices by another developer, leading to a data breach. In order to detect such breaches automatically, we present a methodology that supports the explicit specification of a system's security requirements as part of the system model. In the concrete example, to verify whether data declared as sensitive are treated accordingly, we use an extended form of UMLsec's *secure dependencies* check, as explained in the next section.

4 Security and Variability Profiles

We provide a UML extension to support the specification of security requirements and product-line variability. The *SecPL profile* extends UML with 17 security-specific and variability stereotypes and tagged values. The security-specific concepts of SecPL are built atop of those of UMLsec [27]; annotating elements with variability-specific presence conditions is inspired by solutions such as *model templates* [14].

4.1 UMLsec Security Checks

UMLsec provides an UML profile for annotating UML models with security requirements and various checks for checking those security requirements. In what follows, we introduce two of those checks that are particularly interesting for the design of a secure system, as they cover the security requirements of data on both the logical and physical levels of the system: *secure dependencies* and *secure links*.

Secure dependencies is a check concerning the static structure of the system. It ensures that call and send dependencies between objects respect the security requirements on the data that may be communicated along them. *Secure dependencies* can be thought as a contract between calling and called objects. The following definition adapted from [27] addresses *secrecy*; the *integrity* case is entirely analogous. We assume that objects have a set of members, that is, operations and properties, and a list of *secrecy*-stereotyped members, as can be specified using tagged values of the *«critical»* stereotype as specified in the center of Fig. 5. To be more precise every *Class* or *Component* in an UML diagram can be stereotyped with *«critical»* and the set of *secrecy*-stereotyped members is given as a List of signatures in the tagged value *secrecy*.

Definition 1 (Secure dependencies). *A subsystem fulfills secure dependencies iff for all «call» or «send» dependencies d from an object C to an object D the following conditions hold:*
(i) for all $s \in D.members$*:* $s \in C.secrecy \Leftrightarrow s \in D.secrecy$*,*
(ii) for all $s \in D.members$*:* $s \in C.secrecy \Rightarrow d$ *is stereotyped «secrecy», where s refers to the signature of a member.*

For instance, for the class diagram in Fig. 6, *secure dependencies* is not fulfilled: *Car* specifies *secrecy* for the signature `getVIN()`. However, since *Combox* does not specify *secrecy* for `getVIN()` as well, and the *«call»* dependency relating both classes does not contain the *«secrecy»* stereotype, properties (i) and (ii) are violated.

Secure links is a check concerning the physical deployment of a system. It analyses whether the network of nodes with their communication paths respects the user-specified security requirements wrt. a given attacker model.
Attacker Model. The check is formulated relative to a given attacker type, such as *default* or *insider* attackers, with distinct capabilities of compromising the system [28, p. 59]. For each pair of an attacker and communication path type, a set of threats is known. In this section, we focus on the threats posed by the default attacker, which represents an outsider adversary with modest capability. This kind of attacker can *read*, *modify*, and *delete* messages sent over a plain Internet connection, whereas in the case of an encrypted connection, they can only *delete* messages (e.g., using a fake GSM basestation). However, a default attacker would not be able to read

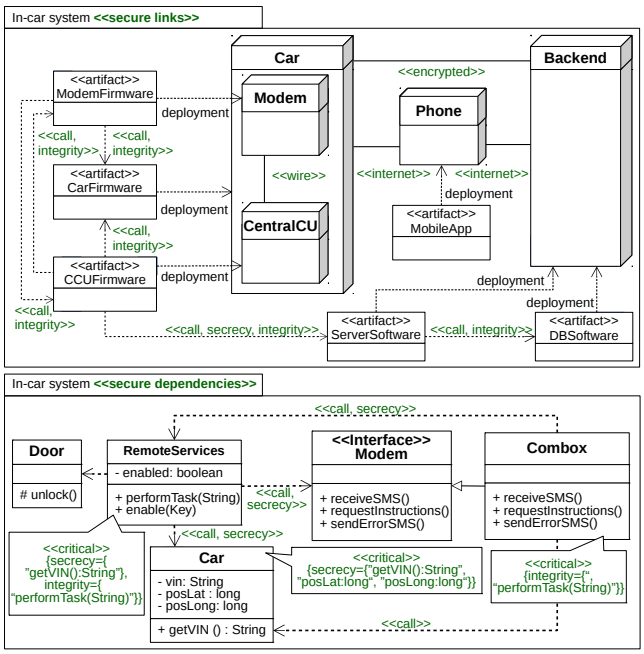

Figure 6. Deployment and class diagram of the product obtained for the configuration {ConnectedDrive, Professional, Combox}.

the plaintext messages or insert messages encrypted with the right key. Of course, this assumes that the encryption is set up in a way such that the adversary does not get hold of the secret key. The default attacker is assumed not to have direct access to the local area network (LAN) and therefore not to be able to eavesdrop on those connections, nor on wires connecting security-critical devices (e.g. a smart-card reader).

We recall a definition [27] for the security requirement *«integrity»*. A corresponding definition for *«secrecy»* is obtained by replacing the considered threat with *read*.

Definition 2 (Secure links)**.** *A subsystem fulfills secure links iff for all «integrity» dependencies d between objects on different nodes n, m, $\exists$ communication path p between n and m with a stereotype s s.t. $write \notin Threats(s)$, where $Threats(s)$ is a set of threats posed by an outside attacker to s-stereotyped communication paths.*

For example, in the deployment diagram in Fig. 3, *secure links* holds under the condition that the communication path between Car and Backend is stereotyped with *«encrypted»*. Due to the *«integrity»*-stereotype dependency between CCUFirmware and ServerSoftware, *secure links* does not hold when only a *«Internet»* communication path is available, because outsider attackers can perform a man-in-the-middle-attack to compromise integrity.

4.2 SecPL Profile

To support variability we extended the stereotypes of UMLsec with presence conditions. Fig. 5 shows an excerpt with three of SecPL's stereotypes and their relationship to UML and UMLsec. The stereotype *«Conditional»* extends the UML meta-class *Element*, whereas the stereotypes *«ConditionalCritical»* and *«ConditionalSecrecy»* generalize their non-conditional counterparts from UMLsec. All in all the SecPL profile consists out of 17 stereotypes similar to the presented ones and includes validation rules for the well-formedness of the presence conditions.

«Conditional» can be added to all UML elements to specify variability on the model level. The tagged value *presenceCondtion* specifies the condition under which the annotated element is present in products of the SPL. We support propositional formulas with negations, conjunctions, and disjunctions over the set of features. For instance, the *Modem* node in Fig. 3 is present if either feature *TCB* or *Combox* is selected.

«ConditionalCritical» generalizes UMLsec's *«critical»* in order to specify security requirements. In this work we mainly focus on the security requirements of protection from unauthorized view access (*secrecy*) and unauthorized modification *(integrity)*. However, we also cover the other security requirements provided by the UMLsec profile. To this end, *«ConditionalCritical»* inherits *«critical»*'s tagged values for these requirement kinds. Each of these tagged values stores a list of operation signatures and property signatures. Variability of security requirements is specified using a list of presence conditions, which are mapped to the corresponding signatures based on their position in the list. For example, in Fig. 4, the position of the car requires additional to treatment with secrecy treatment with integrity when the business radio has been selected by the customer. Multiple requirements on the same element are supported by leaving certain positions in the lists of the tagged values empty, so that each presence condition is mapped to precisely one entry. While the process of managing presence conditions can be complicated, adequate tool support is a promising strategy to support users during such tasks [32].

«ConditionalSecrecy» controls the existence of a *«secrecy»* stereotype in products of the SPL.

In the following sections we are focusing on the *«Conditional»* stereotype. The introduced methodology can be applied to the other stereotypes in the same way.

Deriving Products. Products of the SPL are derived by configuring the features, that is, selecting a specific subset of features. As a result, model elements and security requirements whose presence conditions evaluate to *false* are removed from the model, yielding a regular UMLsec model. Fig. 6 shows the result of deriving a product of our example SPL. Inactive elements such as the class *TCB* have been removed.

Adaptation to Domain-Specific Languages. Our profile can be applied in combination with domain-specific languages that are based on UML profiles.

For example, a central diagram type in SysML models are block diagrams. The blocks in block diagrams are elements

of the UML type Class with the stereotype *«Block»*. Accordingly SysML blocks can own properties, just like like classes in class diagrams can do. The properties in SysML are more fine-grained, reflected in additional SysML-specific stereotypes such as *«AdjunctProperty»* or *«DistributedProperty»*. Since the categorization of properties in these stereotypes is orthogonal to the included security requirements, the Secure Dependency check can be applied to block diagrams straightforwardly, by applying both the SysML and the SecPL stereotypes to the underlying UML model.

5 Family-based Security Analysis

A prime benefit of model-based security approaches is the possibility to perform a security analysis on design models. For example, using the *secure dependencies* check, we can determine if objects in the system respect the security requirements of the data they send and receive. In the product line setting addressed by SecPL, performing such analysis on each product of a SPL separately is infeasible since the number of products can grow exponentially with the number of features. Therefore, we propose a family-based security analysis, which lifts checks such as *secure dependencies* from the level of individual products to the entire SPL.

Our analysis assumes an encoding of the to-be-performed check as an OCL constraint. We provide such encodings for the most prominent and important UMLsec checks; additional ones may be provided by an expert user. To evaluate this constraint against the design model at hand, we use a method called *template interpretation* [15]. Template interpretation was originally designed for checking well-formedness properties, such as "*each association has at least two member ends*", in unstereotyped UML models with variability. To address our security setting, our OCL constraints also take stereotypes into account. Template interpretation generates a certain propositional formula which can be evaluated using a SAT solver. In the formula, features are represented as variables. If the formula is satisfiable, the SAT solver returns a satisfying example, that is, a subset of features giving rise to an insecure product. Else, we have proof that the security property is fulfilled in each product.

In the remainder of this section, we present our security checks with their OCL encodings, we illustrate the generation of a certain formula via template interpretation, and we wrap up.

5.1 UMLsec Checks as OCL Constraints

We focus on UMLsec's *secure links* and *secure dependencies* checks [27]. In combination, these checks support an analysis of security requirements on the physical and logical system levels. We consider the security requirements *secrecy* and *integrity* from Sect. 4.

We specified an OCL version of the secure dependencies check. For brevity, the illustration in Fig. 7 focuses on an

```
1 context Model inv:
2 let callSendRelations = self.allOwnedElements()→
3  select(e|e→oclIsKindOf(Package) and
       (e.has(securedependencies) or
       self.has(securedependencies)))→
4   select(p|p.allOwnedElements()→
5   select(d|d→oclIsKindOf(Dependency)) and (d.has(call)
        or d.has(send))) in
6 callSendRelations→forAll(cs|
7  ((cs.target→forAll(trg | trg→oclIsKindOf(Interface)
8  or (trg→oclIsKindOf(Class) and trg.has(critical))))
9 and cs.source→forAll(src |
       src.getStereotypeApplications(critical)→
10  forAll(srcCritical | srcCritical.getSecrecy()→
11   forAll(srcSecrecy | cs.has(secrecy) and
12   cs.target→select(trg|trg→oclIsKindOf(Class))→
13    forAll(trg | trg.getMembers()→
14      forAll(mem | mem.getName() <> srcSecrecy) or
15      trg.getStereotypeApplications(critical)→
16       exists(trgCritical | trgCritical.getSecrecy()→
17        exists(trgSecrecy | trgSecrecy = srcSecrecy)))
18 )))))
```

Figure 7. Secure dependencies (*«secrecy»* case, excerpt)

excerpt, capturing the "⇒" direction of property (i) and the full property (ii) of the secure dependency check. In the full constraint, the opposite direction and the *integrity* case are considered analogously. Dependencies representing a potential *d* are aggregated on lines 1–5. On lines 1–3, we consider both *models* and *packages*, since both concepts may represent subsystems. On lines 7–8, we check whether the dependency's target class has a *«critical»* stereotype, so that the set of *secrecy* members exists. Note that we use the function *has* as a shortcut to check if an element has a particular stereotype. Interfaces do not need to have this stereotype, since their implementing classes do. On lines 9–17, we iterate over the *secrecy*-stereotyped members of the source class to check if the dependency has the required *«secrecy»* stereotype (line 11), and if the operation in question is tagged with *secrecy* in the *«critical»* stereotype of the target class, in case it exists. As a simplification of the shown OCL constraint we show instead of iterating over both `getOperations()` and `getProperties()` a method `getMembers()`.

Similar to the secure dependencies check, we encoded the secure links check as an OCL constraint.

5.2 Template Interpretation

Template interpretation [15] supports the evaluation of OCL constraints on models in which model elements are annotated with presence conditions (such models are called *model templates* in [15]). The key idea is to replace the standard OCL semantics with a variability-aware one: The result of evaluating a constraint is not a plain value, but a set of value-formula pairs, where the formulas specify the condition under which each of the values occurs. This condition, in turn, depends on the presence condition of model elements. Based on this set, to find out if a particular value can actually occur, we combine its formula with the constraints specified in the feature model. We feed the result to a SAT solver to efficiently

check whether the formula can be satisfied. Since our aim is to establish if a particular constraint—representing a security check—holds in all configuration, we feed the negation of the condition under which it evaluates to *true* to the SAT solver. Note that we do not translate OCL constraints into SAT problems, but calculate all possible outcomes of the OCL constraint execution and the conditions under which they can occur in respect to the feature model.

For instance, to check if a particular class is stereotyped with the stereotype *«critical»*, we can evaluate the constraint `class.has(critical)` on the class. Assuming standard OCL semantics, the result of this check is `true` or `false`. But with template interpretation, we take presence conditions into account. For the class *TCB* in Fig. 4, the following result is obtained: `{(true, TCB), (false, ¬TCB)}`. The paper [15] explains how to generate such sets of value-formula pairs for arbitrary OCL constraints, including those with complex operators such as `forAll()` and `size()`.

We need to answer the question if a OCL constraint c representing a security check `sec`, such as *secure links*, on an element e holds in all products of a considered SPL. This question can be represented as the following SAT problem:

$$s = f \wedge (p^*(e) \Rightarrow \neg c_{true}) \quad (1)$$

Here, f is the conjunction of the feature constraints in the feature model, $p^*(e)$ is e's extended presence condition, and c_{true} is the condition under which c evaluates to *true*. The feature constraints are taken into account because they determine the allowed set of configurations. E.g., considering the constraint `self.getImplementations()→size()>0` for the interface *Modem* in Fig. 4, we would obtain that $c_{true} = TCB \vee Combox$. However, in a version of the feature model where *Modem* is a mandatory feature, either *TCB* or *Combox* is always selected as well, and consequently, s should yield *true*. The implication allows to neglect irrelevant configurations in which e is absent and thus, cannot violate the constraint. The extended presence condition $p^*(e)$ accounts for the containment hierarchy: the presence of an element depends on the presence of its container objects. Therefore, $p^*(e)$ is obtained via the conjunction of $e's$ presence condition with the presence conditions of its container elements.

The output of evaluating s with a SAT solver is either the result that c is *true* in all configurations, that is, `sec` holds in e for all products, or a *witness*, that is, a configuration leading to a product in which `sec` is not fulfilled in e.

5.3 Discussion

Performance. The performance of the overall security analysis depends on the generation of the formula as well as the SAT check. As argued in [15], the generation procedure has polynomial complexity in relation to the size of the input model. For most of OCL's operators, the generation is linear; however, in the case of `size`, it requires quadratic time, since it considers the cross product of model elements. SAT solving is NP complete in general, but state-of-the-art SAT solvers can handle a million of variables and several millions of constraints efficiently [20], which is more than sufficient for typical product line scenarios.

Tool Support. The analysis is implemented as a prototypical plugin for the Eclipse IDE. Since our OCL constraints are formulated in a rather coarse-grained fashion, based on the model- and package-level, determining the root cause of a failed check can be a non-trivial task. However, for debugging purposes, users can use the produced witnesses to inspect a single product where the issue occurs, rather than the full SPL representation. During this task, he can use full-fledged tool support as e.g. provided by CARiSMA [11] for the analysis of the detected insecure product.

Correctness. The correctness of template interpretation relies on the argumentation in [15]. The correctness of our implementation, including the OCL constraints, was studied by systematic testing. Specifically, we systematically extended the test cases of the existing implementation with variability: We considered all possible combinations of annotating the involved elements with variability. The resulting test suite comprises 54 test cases. As test oracle, we used the existing Java-based implementation of UMLsec's checks in CARiSMA, the standard implementation of UMLsec. For a given SecPL-based test model, we enumerated all products, producing a set of UMLsec models on which we performed the CARiSMA check. The results of the variability-aware security check and the single CARiSMA checks were equivalent in all cases, yielding confidence in the correctness of our analysis.

Extensibility. We provide OCL encodings for the most important UMLsec checks: *secure links* and *secure dependencies*. As illustrated in the example, in combination, these checks aim to protect secrecy and integrity on the physical and the logical level. Our solution is extensible in the sense that expert users can define additional checks by providing additional stereotypes with a corresponding OCL encoding. These checks can be used by end-users for annotating and checking UML models transparently, without using or understanding OCL.

6 Reverse Engineering of SecPL Models

Despite our primary intention to support *security by design*, in practice, security concerns often need to be addressed in codebases long after they were initially deployed. Apart from poor planning, a root cause are migration scenarios where the original application was developed for an offline context [17]. In this section, we study the application of our methodology to situations where the goal is to *harden* an existing system. To this end, we provide a mechanism for the reverse engineering of SecPL models from existing codebases. Our mechanism extends the state-of-the-art methodology for *model-based reverse engineering*, which is concerned with

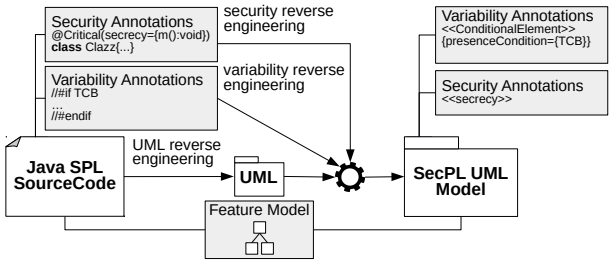

Figure 8. Reverse engineering mechanism.

the process of obtaining useful higher-level representations of legacy systems [7].

The key idea is to let the developers annotate security-critical parts of the source code of the input SPL. We can then generate a SecPL class model which is amenable to the analysis capabilities introduced in Sect. 5. We assume that the input SPL was originally developed using *Antenna*, a widely-spread preprocessor mechanism for annotating Java code with variability [53]. However, other preprocessor mechanisms with similar annotations could be potentially supported without much additional effort. On top of that, the developer uses custom annotations such as `@Secrecy` to specify security requirements and security-related assumptions on fields and methods that will be extracted and added to the output model.

Fig. 8 gives an overview of our mechanism's internal workings. We use the reverse engineering tool MoDisco [9] to extract a UML model from the existing Java codebase. In addition, we parse the Antenna preprocessor annotations from the code to add corresponding *«Conditional»* stereotypes in the UML model. Finally, we parse the UMLsec Java annotations and combine them with the variability information. Annotations being nested into an Antenna condition become *conditional* stereotypes from our SecPL profile. Other annotations become regular UMLsec stereotypes.

In the following, we show some of the regular expressions used during our parsing process, which simultaneously act as a light-weight specification of possible annotations. *Ifdef directives* respecting the following expression are represented as presence conditions of elements in the class diagram.

$$//\backslash s * \#if(def)?. * (n|r) \quad (2)$$

This expression matches every line comment starting from the beginning specified by the // to the end of the line which has the keyword `if` or `ifdef` directly after the start characters, ignoring white space. The rest of the line contains the presence condition.

The following kind of annotations are required to produce SecPL and UMLsec annotations:

$$@((Integrity)|(Secrecy)|(High)|(Critical(\backslash(. * \backslash))?)) \quad (3)$$

We match the combinations of the @-symbol with which every Java-annotation starts with the names of our annotations. To obtain the tagged values of the *«critical»* we match everything with the parentheses of the `@Critical` and analyze this group of the regular expression in a subsequent step.

Listing 1 shows a code excerpt illustrating the use of our annotations. The first regular expression matches lines 1 and 5. Based on the position of those matches in the source code and the positions of the *endif directives* we can calculate which Java elements are covered by such an annotation. The second regular expression matches line 2 of the example. Again we can assign the annotated Java elements based on the position of the match.

```
1 //#ifdef ConnectedDrive
2 @Critical(secrecy={"getVIN():String"},
      integrity={"performTask(String)"})
3 public class RemoteServices {
4    ...
5    //#if Business
6    ... // #endif
7 } //#endif
```

Listing 1. Source code with Antenna and SecPL annotations

In our evaluation, we applied our reverse engineering mechanism to multiple large open-source projects.

7 Evaluation

We designed a methodology for specifying and analyzing security requirements in software product lines. In this section, we evaluate the following aspects of our methodology:

- **RQ1: Efficiency** To what extent does our family-based analysis improve the efficiency of the security analysis?
- **RQ2: Scalability** How does our analysis scale to product lines with large feature models and domain models?
- **RQ3: Usefulness** Is our methodology easily understandable, usable and applicable to realistic software engineering projects?

For this evaluation we implemented the SecPL analysis in a prototypical version as plugin for the Eclipse IDE using the Papyrus UML editor for creating and annotating UML models. During the task of annotating UML models the user is supported with well-formedness checks of presence conditions, an overview of feature usages in the UML model as well as the option to execute our check on all products. If a product with security violations is detected, the standard UMLsec check is executed on this product to generate detailed error messages, using the standard implementation of UMLsec by the CARiSMA tool.

We performed all experiments on a Windows 10 PC with an Intel i5-3570K, 8 GB of RAM and Oracle JDK 8 inside of an Eclipse Neon.3 instance which was allowed to allocate up to 6 GB memory.

7.1 RQ1: Efficiency

To evaluate our methodology based on realistic subjects, we collected a suite of models suitable for our security- and variability-oriented setting. The collection was performed

Table 1. Subjects

Project name	Input artifacts	#Elements	#*Call*	#Features	#Products
BMW	Magazine article	116	13	16	54
E2E	UMLsec models	130	14	7	94
BCMS	UML models	3,034	4	8	254
JSSE	Java	24,077	28	6	64
Notepad	Java + Antenna	252	4	13	512
MobilePhoto	Java + Antenna	4,069	35	13	3,072
Lampiro	Java + Antenna	29,045	24	20	5,892

based on convenience sampling, in most cases by reusing evaluation samples from the existing literature on software product lines and model-based security. We give an overview of our subjects in Table 1 with relevant information, including the number of dependencies with *«call»* and *«send»* stereotypes, since they are a key part in both considered checks. The models stem from a variety of sources that can be divided into two groups.

The first group represents original modeling examples. First, we created a model based on the *BMW* running example introduced in Sect. 3. Second, we used an UMLsec scenario obtained from the CARiSMA developers from their prior collaboration with an industry partner and extended it with variability: *EndToEndEncryption* (E2E) is based on a set of system models specifying different versions of *Munich Re*'s IT infrastructure [54]. For our evaluation, we refactored those models into a product line. Third, the *Barbados Car Crash Management System* (bCMS) [10] is based on a requirements specification of car crash management system SPL. For our evaluation, we used an available UML implementation in the form of enumerated products [60] and manually refactored it into a SecPL model. While the bCMS model is relatively large, only a small part of the model required security annotations, resulting in four relevant calls.

The second group is made up of projects from the open-source Java context. We produced SecPL models by applying our reverse engineering mechanism to the available codebases. While most projects featured Antenna annotations specifying variability on the source code level, this was not the case for OpenJDK's [3] implementation of the *Java Secure Socket Extension* (*JSSE*), a particularly interesting security-critical scenario. We extended JSSE's codebase with variability, by assigning features to the different supported protocols, and security annotations based on security critical keywords like "keystore". *Notepad* [16] is a text editor in which the opening and writing of files are security-critical. For example many iOS apps have been infected by a corrupted editor [23]. *MobilePhoto* [44] is a mobile multi-media platform supporting for sharing media over an Internet connection. *Lampiro* [38] is an instant messaging client which has been naively developed as software product line. In these cases, we added security annotations to the codebasis. Our Lampiro model is the largest one considered, comprising 29K elements, including classes, dependencies, and operations. MobilePhoto and Lampiro have already been subject to earlier SPL research [34].

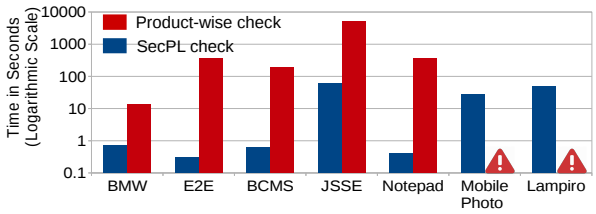

Figure 9. Execution Times for RQ1: Efficiency.

Set-up. We experimentally evaluated the efficiency of our analysis using the models described above, using the state-of-the-art tool, CARiSMA, as baseline. For each model, we compared the execution time for checking the SecPL model using our analysis (SecPL check) to the sum of the execution times for checking all products using CARiSMA, which supports regular UMLsec checks on single products (product-wise check). In both cases we measure the timespan from loading a UML model with SecPL stereotypes to the delivery of the analysis results for all products. Our analysis is more efficient if the execution times of the SecPL implementation are significantly lower than those of CARiSMA.

Results. The product-wise check produced a result for five out of seven subjects, *BMW*, *E2E*, *BCMS*, *JSSE*, and *Notepad*. In these cases, the SecPL checks were between one and three orders of magnitude faster. For the subjects *MobilePhoto* and *Lampiro*, the product-wise check terminated with a GC overhead exception after 700 to 1,000 checked products and 3 to 5 hours of runtime, whereas the SecPL checks took below 100 seconds. On average, the product-wise check spent 91.6% of the time generating the products, and the remaining 8.4% performing the checks. We observed that the runtime of the product-wise check mainly depends on the number of products, whereas the SecPL check is mainly influenced by the model size and the number of relevant calls. In sum, SecPL outperformed the product-wise check constantly.

7.2 RQ2: Scalability

For our scalability evaluation, we needed to freely control the size of our test models. To this end, we generated synthetic models. Our rationale was to create models being representative of the realistic examples, which we address as follows.

To study the effect of the model size, we generated large class models, being amenable to the *secure dependencies* check. Based on typical cases in the security-critical portions of the realistic examples, we incrementally added classes with on average four operations and three dependencies. Our initial model contained two classes with one call dependency between them and one operation each. In each iteration, we added a class with a *«critical»* stereotype and a normally distributed number of operations, on average four, and a normally distributed number of dependencies, on average three. We added all member signatures of classes reachable over a dependency to the *secrecy* tag the class's *«critical»* stereotype. The resulting model is potentially expensive to

check: (i) it comprises many involved dependencies and operations, and (ii) since it fulfills secure dependencies–every class treats all relevant signatures with *secrecy*—, the check does not terminate early with a counterexample.

To study the effect of the feature model size, we took a randomly generated UML model from the model-size experiment with 4K classes, incrementally added independent features to the feature model successively, and assigned each feature to one class in the model via a suitable *«conditional»* stereotype. We checked models annotated with between zero and 4K features, adding 50 features in each iteration.

Set-up. To experimentally evaluate scalability, we measure the execution times of our SecPL implementation on different synthetic models with a growing number domain model elements and features as described above. Our analysis is scalable if the execution time avoids exponential growth for increasingly larger domain models and feature models.

Results. In our scalability experiment regarding model size, the largest generated model we checked had 524K UML elements, including 66K classes with an average number of four operations and three call dependencies to other classes. As shown in Fig. 10 the execution of this test case took 97.3 minutes. The regression function we calculated from this measurements is second order polynomial and fits the measured data with a coefficient of determination (R^2) of nearly one (0.999985). This observation is in line with the performance considerations for template interpretation. For our scalability experiment regarding the number of features, we used a randomly generated model with 4K UML classes (32K UML elements) and successively added up to 4K independent features ($1.04 \cdot 10^{1233}$ products). The measured data points as illustrated in Fig. 10 show a higher variance compared to those from the previous experiment. The analysis took between 57 and 58 second up to a number of 1.7K features, and started oscillating between 58 and 60 seconds until around 4K features. In sum, our analysis showed scalable behavior up to thousands of features, the magnitude of large product lines in automotive engineering [62].

7.3 RQ3: Usefulness

To evaluate the usefulness of our methodology, we conducted a user experiment with participants from academia and industry.

Set-up. We recruited nine participants from academia, two of them with a significant background in industry, and one representative of an industry partner. The participants from academia came from three universities and one private research institute and had their focus in the security, SPL, and modeling domains. The industry-experienced academics had long-running backgrounds as IT freelancers. Moreover, one of them was employed at a large steel-based technology group at the time of the experiment. The industry partner,

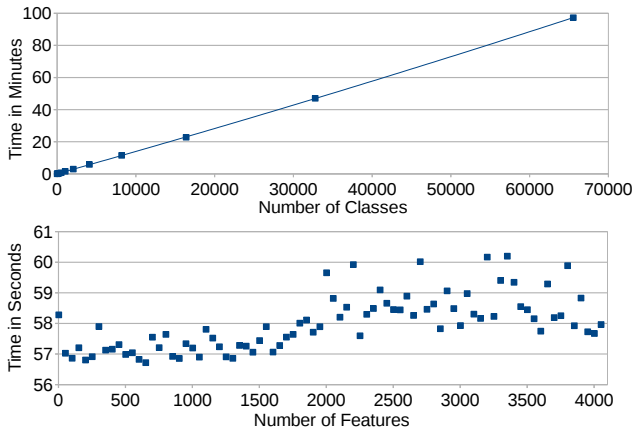

Figure 10. RQ2: scalability regarding *number of classes* and *number of features*.

SinnerSchrader, is Germany's fourth-largest digital marketing company and cooperates with many major international companies.

After a short introduction to SecPL, we asked the participants to perform a development task based on the example in Section 3. The task was to extend the included model with a new modem type by using our tool prototype, while addressing the included security requirements. Afterwards, the participants filled in a questionnaire in which they rated their subjective experience in eight questions based on a five-point Likert scale. Five questions addressed usability concerns, such as the difficulty of specifying a new security requirement; three questions were concerned with understandability, such as the certainty that the participant's understanding of the used stereotypes was correct. We provide a replication package including the task, questions, and results together with the submission.

After the experiment, we conducted informal interviews with all participants, in which we asked for feedback concerning usability and understandability. In the interviews with the industry-based and -experienced participants, we additionally asked them to comment on the applicability of our methodology to their business segments and those of their customers.

Results: Usability and Understandability. The answers to our questions indicate that our methodology is easily usable and understandable. According to Fig. 11 in both categories more than 70 percent of the answers suggest a high or very high usability and understandability, an impression confirmed by the feedback in the interviews. On the downside, some participants perceived the editing of annotations through Papyrus's user interface as cumbersome, as reflected by some of the negative scores for usability. Moreover, some participants were worried that a larger model "cluttered" with annotations may become hard to read. A promising strategy to deal with these issues is by providing improved tool support, for example, to support the editing

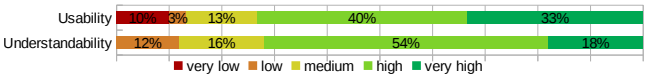

Figure 11. Aggregated answers from our user study.

of large models based on custom-tailored views [8], including views on individual products of the product line [33]. A further question raised by participants was where to start when annotating the model with security requirements. To this end, approaches analyzing higher-level security specifications and suggesting SecPL security annotations can be helpful [1]. Moreover, a textual UML notation may further help to improve the usability. Despite the mostly positive understandability ratings, one participant reported considerable problems while understanding the stereotypes. An interactive help system may help to further improve understandability.

Results: Practical Applicability. According to our industry partner's representative, our notation for specification and analysis of security requirements on product lines is an accurate fit for their business needs. As an example for a possible application, they mentioned a current collaboration in the automotive domain on real-time car software upgrades based on changing customer needs. They want to dynamically advertise and sell upgrades according to customers needs by dynamically reconfiguring the car – e.g. to sell the usage of the trailer hitch for some days when the customer is relocating. The specification and analysis of security requirements on software product lines is essential for this concept. The participant deemed our graphical notation on UML models as a possibility to realize the specification in a user-friendly way.

One of the industry-experienced participants conjectured that our approach might be very helpful for developers familiar with modeling, but felt that he was not proficient enough in this topic to really judge applicability.

The other industry-experienced participant, who is also employed for a steel-based technology group, stated that our methodology could be used for coordinating the development of security-critical software in multiple distributed teams. If the project has been planned using UML, special trained team-members can easily annotate the models with required and provided security properties. However, for a direct use in industry, the tool support has to be improved; distributed and parallel editing of UML models has to be supported. Nevertheless, he pointed out that these are general issues with model-based development, and that they are by no means necessarily aggravated by incorporating SecPL.

To conclude, these first impressions give a promising outlook on the applicability of our methodology in industry. Since we do not require any artifacts beyond those involved in typical software development processes, our participants found that an alignment of our methodology with these processes seems generally possible.

7.4 Threats to Validity

External validity is threatened by our limited selection of models that may not be representative of all realistic models. While our suite of test subjects selected for RQ1 represents a broad variety of use cases, we cannot generalize our findings to arbitrary models. The models generated for our scalability measurements in RQ2 were inspired by the realistic ones for RQ1; their purpose was to illustrate the effect of an increased model size and feature number. The model in RQ3 is by no means representative for all possible usage contexts; however, it was chosen as a critical example inspired by a real case.

Regarding internal validity, a potential threat concerns the correctness of our implementation. In Sec. 5, we argued for the correctness of our OCL implementations of the considered UMLsec checks by using CARiSMA as a test oracle. Since both implementations were developed independently from another, the identical results from the test suite give us a high level of confidence in the correctness of our implementation. For additional user-specified constraints, correctness has to be ensured as well, for example, by providing a similar test suite. Moreover, while we aimed to systematically specify all security requirements in the considered examples, we cannot guarantee the completeness of our security annotations.

With regard to conclusion validity, a more definitive verdict on the practical applicability of our methodology requires the involvement of a larger sample of practitioners. To this end, we plan to conduct a broader developer survey in the future. In particular, we did not evaluate if users can work with our reverse-engineered models effectively, which depends on the employed model editor's usability during the editing of larger models.

Regarding construct validity, our methodology is based on existing technology, such as template interpretation and the Papyrus UML editor, that also impact its applicability. Our evaluation assesses the applicability of these techniques *in the domain of software security*, which has not been done in previous work. Moreover, to the best of our knowledge, we also provide the first evaluation of a template-interpretation-based technique on a set of realistic models.

8 Related Work

Model-Based Security Analysis. An overview of model-based security analysis can be found in [39], which reviews existing approaches for security analysis of model-based object-oriented software designs, and identifies ways in which these approaches can be improved and made more rigorous. Some research addresses linking the model to the code level within model-based security through model-driven reverse engineering [41, 52]; similar to our work, Martínez et al. [41] use OCL to specify security policies. Other work addresses

the model-based use of security patterns [36, 48, 63]. Further research makes use of aspect-oriented modeling for model-based security [19]. [22] proposes an approach for model-based security verification.

UMLsec [29] provides a model-based approach to develop and analyze security critical software systems, in which security requirements such as secrecy, integrity, and availability are expressed in UML diagrams [49]. UMLsec is provided as an UML profile, containing different stereotypes and tagged values to annotate UML diagrams with security properties. The CARiSMA tool performs the corresponding security analysis [25]; it has been applied to various industrial applications (e.g. [26]).

While these works are relevant as foundations for our technique, none of them addresses software product lines.

Security of Software Product Lines. Sion et al. [56] present a research agenda towards systematically addressing security concerns in software product lines in a way which considers security separate from other variability dimensions by allowing to express security and its variability, select the right solution, properly instantiate a solution, and verify and validate it. This research agenda seems certainly relevant and worthwhile, but there do not seem to be results published to date. Myllärniemi et al. [45] propose a kind of modeling language for specifying security and functional variability at the architectural level of a system. Their solution allows the user to select among multiple countermeasures; however, a security analysis in the style of our work is not possible in this solution, since security requirements on the level of threats and assets are deliberately left outside the scope of this work. Nadi and Krüger [46] use the modeling language Clafer, which combines feature modeling and meta-modeling, for modeling cryptographic components. In comparison, their work could be considered a specific product line of security-relevant software products, whereas our goal is to apply security concepts to harden arbitrary software product lines. Mellado et al. [42, 43] present approaches which deal with security requirements from the early stages of the product line lifecycle in a systematic and intuitive way especially adapted for product line based development. These works do not address the system design, as we do here. Fægri and Hallsteinsen [18] presents a software product line reference architecture for security. This work does not use a model-based design approach, as we do.

Analysis of Software Product Lines. Scalability issues arising due to variability have motivated a variety of software analyses for SPLs; for an overview, see the comprehensive survey by Thüm et al. [61]. A key distinction is that between *product-based* approaches that operate on a selection of all products, and *family-based* ones that lift the analysis to a representation of the overall SPL. Product-based approaches are useful in scenarios where the result does not need to be complete, a prime example being testing. *Model-based testing* of SPLs [2, 13, 24, 40] focuses on the use of dedicated test models for this purpose. To improve test coverage, Cichos et al. [13] derive test cases from a "150%" test model for the SPL, and Johansen et al. [24] use a certain notion of covering arrays that can be derived from the feature model. Ali et al. [2] propose a methodology for reducing the specification effort during model-based testing of SPLs. Lachmann et al. prioritize products by their risk for failures for integration tests of SPLs [37]. These approaches do not aim to ensure a complete analysis of all products of the product line.

Our security analysis falls into the category of family-based analysis. Most works in this category focus on program analyses, such as syntax and type checking [35], static program analysis [21], or model checking [21]. A seminal model-level work is the well-formedness analysis for model templates by Czarnecki and Pietroszek [15] that we used as a foundation for our analysis (see Sect.5). While this work operates on vanilla UML models to validate well-formedness constraints, our analysis works on stereotyped UML models for checking security properties. Salay et al's [55] work on the lifting of transformation rules to model-based SPLs includes a matching step that can be considered a family-based analysis. However, none of these works addresses security.

9 Conclusion and Future Work

Security is one of the hardest properties of software to accomplish in practice. With this work, we provide a comprehensive methodology for the model-based security analysis of software product lines. Using the SecPL profile, users specify security requirements as well as variability information as part of the system model. Our analysis addresses the scalability issues encountered in this setting by lifting the analysis to the level of the entire product line rather than individual products. In our evaluation, this solution enabled the analysis of realistic product lines in realistic cases where the naive approach terminated without a result; a user study indicates the usefulness of our methodology.

In the future, we aim to extend our work in three directions. First, we intend to broaden the scope of our reverse engineering approach. When models and code may be subject to evolution, keeping security properties synchronized on both levels is challenging. Second, we aim to apply our methodology to a broader selection of use-cases. Since UMLsec has been used in protocol engineering [5], a promising application involves protocol families. Finally, an extended form of our analysis could inform the automated configuration of a product line, e.g., by considering the established security degree and the cost for security measures to assess solutions.

Acknowledgements

This work was supported by the Deutsche Forschungsgemeinschaft (DFG), project *SecVolution@Run-time*, no. 221328183.

References

[1] Amir Shayan Ahmadian, Sven Peldszus, Qusai Ramadan, and Jan Jürjens. 2017. Model-based Privacy and Security Analysis with CARiSMA. In *FSE*. 989–993. https://doi.org/10.1145/3106237.3122823

[2] Shaukat Ali, Tao Yue, Lionel C. Briand, and Suneth Walawege. 2012. A Product Line Modeling and Configuration Methodology to Support Model-Based Testing: An Industrial Case Study. In *MoDELS*. 726–742.

[3] Oracle Corporation and/or its affiliates. 2018. OpenJDK. (2018). http://openjdk.java.net/

[4] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer.

[5] A. Bauer and J. Jürjens. 2010. Runtime Verification of Cryptographic Protocols. *Computers and Security* 29, 3 (2010), 315–330.

[6] B. Best, J. Jürjens, and B. Nuseibeh. 2007. Model-based Security Engineering of Distributed Information Systems using UMLsec. In *ICSE*. ACM, 581–590.

[7] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2012. Model-driven Software Engineering in Practice. *Synthesis Lectures on Software Engineering* 1, 1 (2012), 1–182.

[8] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. 2017. A feature-based survey of model view approaches. *Software & Systems Modeling* (2017), 1–22.

[9] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. 2010. MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In *ASE*. ACM, 173–174.

[10] Afredo Capozucca, Betty Cheng, Geri Georg, Nicolas Guelfi, Paul Istoan, Gunter Mussbacher, Adam Jensen, Jean-Marc Jézéquel, Jörg Kienzle, Jacques Klein, et al. 2011. Requirements Definition Document for a Software Product Line of Car Crash Management Systems. *ReMoDD* (2011).

[11] CARiSMA 2018. CARiSMA Tool. (2018). http://carisma.umlsec.de

[12] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *ASE*. 332–343.

[13] Harald Cichos, Sebastian Oster, Malte Lochau, and Andy Schürr. 2011. *Model-Based Coverage-Driven Test Suite Generation for Software Product Lines*. Springer, 425–439.

[14] Krzysztof Czarnecki and Michal Antkiewicz. 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *GPCE*. 422–437.

[15] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying Feature-based Model Templates Against Well-formedness OCL Constraints. In *GPCE*. 211–220.

[16] Hugo Sica de Andrade, Eduardo Santana de Almeida, and Ivica Crnkovic. 2014. Architectural bad smells in software product lines: an exploratory study. In *WICSA*. 12:1–12:6.

[17] Premkumar T Devanbu and Stuart Stubblebine. 2000. Software Engineering for Security: A Roadmap. In *ICSE*. ACM, 227–239.

[18] Tor Erlend Fægri and Svein O. Hallsteinsen. 2006. A Software Product Line Reference Architecture for Security. In *Software Product Lines - Research Issues in Engineering and Management*. 275–326.

[19] Geri Georg, Indrakshi Ray, Kyriakos Anastasakis, Behzad Bordbar, Manachai Toahchoodee, and Siv Hilde Houmb. 2009. An Aspect-oriented Methodology for Designing Secure Applications. *INFSOF* 51, 5 (2009), 846–864.

[20] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. 2008. Satisfiability Solvers. *Foundations of Artificial Intelligence* 3 (2008), 89–134.

[21] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. 2008. Modeling and Model Checking Software Product Lines. In *FMOODS*. Springer, 113–131.

[22] Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean. 2008. Applying Formal Methods to a Certifiably Secure Software System. *IEEE Trans. Software Eng.* 34, 1 (2008), 82–98.

[23] Alex Hern and agencies. 2015. Apple removes malicious programs after first major attack on app store. The Guardian online. (2015). https://goo.gl/phxmRR

[24] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. 2012. Generating Better Partial Covering Arrays by Modeling Weights on Sub-product Lines. In *MoDELS*. 269–284.

[25] J. Jürjens. 2000. Secure Information Flow for Concurrent Processes. In *CONCUR*, C. Palamidessi (Ed.), Vol. 1877. 395–409.

[26] Jan Jürjens. 2001. Modelling Audit Security for Smart-Card Payment Schemes with UML-Sec. In *Trusted Information: The New Decade Challenge*, Michel Dupuy and Pierre Paradinas (Eds.). Springer, 93–107.

[27] Jan Jürjens. 2002. UMLsec: Extending UML for Secure Systems Development. In *UML*. 412–425.

[28] Jan Jürjens. 2005. Model-based Security Engineering with UML. In *FOSAD*. Springer, 42–77.

[29] Jan Jürjens. 2005. *Secure Systems Development with UML*. Springer.

[30] J. Jürjens, J. Schreck, and P. Bartmann. 2008. Model-based Security Analysis for Mobile Communications. In *ICSE*. ACM, 683–692.

[31] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. DTIC Document.

[32] Christian Kästner and Sven Apel. 2009. Virtual Separation of Concerns-a Second Chance for Preprocessors. *Journal of Object Technology* 8, 6 (2009), 59–78.

[33] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *ICSE*. ACM, 311–320.

[34] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. 2012. Type Checking Annotation-based Product Lines. *TOSEM* 21, 3, Article 14 (July 2012), 14:1–14:39 pages.

[35] Christian Kästner, Paolo G Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation. *ACM SIGPLAN Notices* 46, 10 (2011), 805–824.

[36] Basel Katt, Matthias Gander, Ruth Breu, and Michael Felderer. 2011. Enhancing Model Driven Security through Pattern Refinement Techniques. In *FMCO*. 169–183.

[37] Remo Lachmann, Simon Beddig, Sascha Lity, Sandro Schulze, and Ina Schaefer. 2017. Risk-based integration testing of software product lines. In *VaMoS*. ACM, 52–59.

[38] Lampiro 2011. Lampiro. (2011). https://github.com/pinturic/lampiro/tree/master/lampiro

[39] Kevin Lano, David Clark, and Kelly Androutsopoulos. 2002. Safety and Security Analysis of Object-Oriented Models. In *SAFECOMP*. Springer, 82–93.

[40] Malte Lochau, Sven Peldszus, Matthias Kowal, and Ina Schaefer. 2014. Model-based Testing. In *Formal Methods for Executable Software Models*. Springer, 310–342.

[41] Salvador Martínez, Valerio Cosentino, and Jordi Cabot. 2016. Model-based Analysis of Java EE Web Security Configurations. In *MiSE*. ACM, 55–61.

[42] Daniel Mellado, Eduardo Fernández-Medina, and Mario Piattini. 2008. Towards Security Requirements Management for Software Product Lines: A Security Domain Requirements Engineering Process. *Computer Standards & Interfaces* 30, 6 (2008), 361–371.

[43] Daniel Mellado, Haralambos Mouratidis, and Eduardo Fernández-Medina. 2014. Secure Tropos Framework for Software Product Lines Requirements Engineering. *Computer Standards & Interfaces* 36, 4 (2014), 711–722.

[44] MobilePhoto 2008. MobilePhoto. (2008). http://homepages.dcc.ufmg.br/~figueiredo/spl/icse08/

[45] Varvana Myllärniemi, Mikko Raatikainen, and Tomi Männistö. 2015. Representing and Configuring Security Variability in Software Product Lines. In *QoSA*. 1–10.

[46] Sarah Nadi and Stefan Krüger. 2016. Variability Modeling of Cryptographic Components: Clafer Experience Report. In *VaMoS*. 105–112.
[47] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. 2015. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching. In *SP*. IEEE, 692–708.
[48] Phu Hong Nguyen, Koen Yskout, Thomas Heyman, Jacques Klein, Riccardo Scandariato, and Yves Le Traon. 2015. SoSPa: A System of Security Design Patterns for Systematically Engineering Secure Systems. In *MoDELS*. 246–255.
[49] Object Management Group (OMG). 2011. UML 2.5 Superstructure Specification. (2011).
[50] OMG. 2017. *OMG System Modeling Language*. Technical Report. Object Management Group.
[51] Sebastian Oster, Florian Markert, and Philipp Ritter. 2010. Automated Incremental Pairwise Testing of Software Product Lines. In *SPL*. Springer, 196–210.
[52] Salvador Martínez Perez, Joaquín García-Alfaro, Frédéric Cuppens, Nora Cuppens-Boulahia, and Jordi Cabot. 2013. Model-Driven Extraction and Analysis of Network Security Policies. In *MoDELS*. Springer, 52–68.
[53] Jörg Pleumann, Omry Yadan, and Erik Wetterberg. 2010. Antenna Preprocessor. (2010). http://antenna.sourceforge.net/
[54] Max Reininger. 2006. *End-to-End Security in a Reinsurance Company, Remote Access to the Company Network*. Master's thesis. TU Munich.
[55] Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, and Marsha Chechik. 2014. Lifting Model Transformations to Product Lines. In *ICSE*. ACM, 117–128.
[56] Laurens Sion, Dimitri Van Landuyt, Koen Yskout, and Wouter Joosen. 2016. Towards Systematically Addressing Security Variability in Software Product Lines. In *SPLC*. 342–343.
[57] SMS 2002. *ISO/IEC 21989:2002: Information technology – Telecommunications and information exchange between systems – Private Integrated Services Network – Specification, functional model and information flows – Short message service*. Technical Report. International Organization for Standardization, https://www.iso.org/standard/36050.html.
[58] Dieter Spaar and Fabian A. Scherschel. 2015. Beemer, Open Thyself! – Security vulnerabilities in BMW's ConnectedDrive. (2015).
[59] Harald Störrle. 2017. How are Conceptual Models used in Industrial Software Development?: A Descriptive Survey. In *EASE*. ACM, 160–169.
[60] Daniel Strüber, Timo Kehrer, Thorsten Arendt, Christopher Pietsch, and Dennis Reuling. 2016. Scalability of Model Transformations: Position Paper and Benchmark Set. In *Workshop on Scalable Model Driven Engineering*. 21–30.
[61] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 6:1–6:45.
[62] Len Wozniak and Paul Clements. 2015. How Automotive Engineering is Taking Product Line Engineering to the Extreme. In *SPLC*. 327–336.
[63] Koen Yskout, Riccardo Scandariato, and Wouter Joosen. 2015. Do Security Patterns Really Help Designers?. In *ICSE*. 292–302.