Ontology-Driven Evolution of Software Security

Sven Peldszus[1][0000−0002−2604−0487], Jens Bürger[2][0000−0003−2504−1653], Timo Kehrer[3][0000−0002−2582−5557], and Jan Jürjens[1,4][0000−0002−8938−0470]

[1] Institute for Software Technology, University of Koblenz-Landau, Universitätsstraße 1, 56070 Koblenz, Germany.
{speldszus,juerjens}@uni-koblenz.de
[2] Knipp Medien und Kommunikation GmbH, Martin-Schmeißer-Weg 9, 44227 Dortmund, Germany.
jbuerger@knipp.de
[3] Humboldt-Universität zu Berlin, Unter den Linden 6, 10099 Berlin, Germany.
timo.kehrer@informatik.hu-berlin.de
[4] Fraunhofer-Institute for Software and Systems Engineering ISST, Emil-Figge-Str. 91, 44227 Dortmund, Germany.

Abstract. Ontologies as a means to formally specify the knowledge of a domain of interest have made their way into information and communication technology. Most often, such knowledge is subject to continuous change, which demands for consistent evolution of ontologies and dependent artifacts. In this article, we study ontology evolution in the context of software security, where ontologies may be used to formalize the security context knowledge which is needed to properly implement security requirements. In this application scenario, techniques for detecting ontology changes and determining their semantic impact are required to maintain the security of a software-intensive system in response to changing security context knowledge. Our solution is capable of detecting semantic editing patterns, which may be customly defined using graph transformation rules, but it does not depend on information about editing processes such as persistently managed changelogs. We leverage semantic editing patterns for (i) generating system co-evolution proposals, (ii) adapting the configuration of standard security checks, and (iii) performing incremental security compliance analyses between co-evolved system models and the implementation. We demonstrate the feasibility of the approach using a realistic medical information system known as iTrust.

Keywords: Software engineering · Model-based security · Security context knowledge · Ontology evolution · Semantic editing patterns · Security compliance.

1 Introduction

Many modern software systems are developed for and deployed in security-critical domains such as health care. In such domains, it is essential to keep track of all relevant threats and appropriate measures. Furthermore, whenever security assumptions change, e.g., due to newly discovered attacks or weaknesses, the system has to be co-evolved in order to ensure compliance with the updated security knowledge.

1.1 Context and Problem Motivation

Ontologies as a means for formal and explicit specification of knowledge of a domain of interest have made their way into contemporary information and communication technology, e.g., to foster information semantics and semantic interoperability in various kinds of information systems [21]. Often, the knowledge of a domain is subject to continuous change. This change demands the

Accepted Manuscript of S. Peldszus, J. Bürger, T. Kehrer, and J. Jürjens, "Ontology-Driven Evolution of Software Security", Data & Knowledge Engineering (DKE), 2021, doi: https://doi.org/10.1016/j.datak.2021.101907.

continuous evolution of the respective ontologies and, depending on the application scenario, the consistent co-evolution of dependent artifacts. Managing such ontology evolution is faced with a multitude of technical and organizational challenges [16, 38].

In this article, we focus on the task of detecting ontology changes and determining their semantic impact, which must be supported by a technical solution that accommodates application-specific requirements [64]. As an application scenario, we study ontology evolution in the context of software security. Developing secure software systems requires expert knowledge that is independent of the actual system. Such knowledge, referred to as *Security Context Knowledge (SCK)* in the sequel, comprises, e.g., information on which encryption algorithms can be considered secure and which are known to be compromised. We build upon the tool S^2EC^2O, in which ontologies are used as a means to formally capture and exploit SCK. However, SCK changes over time due to, e.g., newly discovered attacks or regulatory changes. In case the system is insecure regarding the evolved knowledge, it needs to be co-evolved such that the essential security requirements are preserved. In this application scenario, tool support for managing ontological change is faced with a set of requirements which may be summarized as follows (cf. literature pointers for a more detailed discussion of each of the requirements):

(1) Atomic, low-level changes such as adding or deleting single ontology elements are too fine-grained to determine their semantic impact. Instead, changes must be handled on the level of more coarse-grained, *semantic editing patterns* also referred to as composite, complex, or high-level changes [49, 55].
(2) Since changes to the knowledge base may occur ad-hoc and mostly undocumented, occurrences of semantic editing patterns must be *detected after the fact* by comparing the old and new version of an ontology. In particular, they cannot be recorded through a controlled change management process and toolchain [67].
(3) The set of semantic editing patterns supported by the change detection facility must be *customizable and extensible*. This enables the configuration of domain- and system-specific countermeasures addressing different kinds of security threats [11].
(4) To (semi-)automatically execute configured countermeasures, occurrences of detected semantic editing patterns must be *amenable to model-driven engineering tools* [9].

While several approaches to support ontology evolution have been presented that cope with the lifting of atomic ontological changes to semantic editing patterns [15,29,30,49,54,55,63,64] (requirement (1)), none of them meets all of the above-mentioned requirements. Most of them [15, 29, 30, 49,63,64] rely on a well-defined ontology evolution process and assume all ontological changes to be captured through a dedicated tool and/or persistently managed in the form of a changelog. Ad-hoc changes and exchanging ontologies across tool boundaries, which demand a state-based comparison after the fact (requirement (2)), are not supported. Some approaches [54, 55] get rid of the restriction of persistently managed changelogs, however, they only support a fixed set of semantic editing patterns that cannot be adapted (requirement (3)). None of the proposed approaches reports occurrences of semantic editing patterns in a form that is amenable to model-driven engineering tools (requirement (4)).

To overcome these shortcomings, our technical solution to detect ontological changes draws from recent advances in the field of model comparison and versioning. In particular, we adopt a state-based approach to model differencing, which assumes the old and new version of an ontology to be available, but no information about editing processes such as the existence of a changelog [36]. We work on a structural, graph-based representation of ontological models using a fixed meta-model as a type graph. Thereupon, graph transformation rules are used as an intuitive means for specifying semantic editing patterns.

Concerning our application scenario, we leverage semantic editing patterns for maintaining software security in response to changes in the SCK. We extend our conference paper [10], in which we study ontology evolution in the context of a model-based approach to secure software engineering called S^2EC^2O [7]. While the scope of our previous work is on the level of system models, in this article, we extend this scope to the implementation level. For propagating ontology changes over the model level to the implementation level, we combine S^2EC^2O with the GRaViTY approach [57], a graph-based approach for synchronizing system models with the implementation level. Moreover, GRaViTY facilitates various forms of program analyses, which we exploit to adapt the configuration of standard security checks and perform incremental security compliance analyses.

1.2 Research Questions & Methodology

In this work, we aim at studying approaches for detecting changes in security context knowledge and means to leverage these changes to allow a secure co-evolution of software systems. For this purpose, we have to answer four research questions if we want to provide an approach to support system developers in the context of evolving security context knowledge:

RQ1: How can we detect security context knowledge changes and extract their semantic impact?
RQ2: How can we generate model co-evolution proposals to mitigate the changes on the architectural level?
RQ3: How can we co-evolve security check configurations to allow checks against the changed security context knowledge?
RQ4: How can we efficiently verify the compliance of the system's implementation with the co-evolved architecture?

To answer the identified research questions, we followed the design science research methodology [26, 56]. Applying this methodology, we motivate the relevance of the identified problem in detail in Sect. 2. Next, in sections 3 and 4, we introduce our artifact's conceptual design and its concrete realization in Sect. 5. In Sect. 6, we demonstrate the feasibility of the designed artifact to solve the identified problem. We critically discuss our solution and its evaluation in Sect. 7, discuss related works in Sect. 8, and conclude in Sect. 9.

1.3 Contributions & Structure

In summary, the article makes the following contributions:

1. An approach to detect occurrences of semantic editing patterns, specified as graph transformation rules, between two versions of an ontology formulated using the Web Ontology Language (OWL) [53] (Sect. 3).
2. The rule-based leveraging of semantic editing patterns for the secure co-evolution of the system model (Sect. 4.1).
3. The co-evolution of security check configurations to allow security compliance checks after the evolution of security context knowledge (Sect. 4.2).
4. Incremental static compliance checks, also specified as graph transformation rules, between the security context knowledge captured in the ontologies and the implementation of the system (Sect. 4.3).
5. A prototypical implementation of the approach based on the Eclipse Modeling Framework (EMF) technology stack [78], along with an open catalog of semantic editing patterns on OWL ontologies implemented in the Henshin transformation language [79] (Sect. 5).

6. An application to the medical information system iTrust [24], showing the feasibility of leveraging semantic editing patterns for the sake of system model and implementation co-evolution in model-based secure software engineering (Sect. 6).

We present contributions 1 and 2 without conceptual extensions as we introduced them in our conference paper [10]. Contributions 3 and 4 are entirely new contributions of this article. Contributions 5 and 6 were originally introduced in the conference paper [10], which we significantly extend in this article: First, our tool prototype has been extended to cover also the source code level besides security ontologies and design-time models. Second, our feasibility study has been extended to cover the co-evolution of security check configurations and security compliance checks between design-time models and their implementation.

2 Problem Analysis and Overview of the Approach

In this section, we introduce our application scenario of secure software engineering and how this relates to security context knowledge, which is captured in an ontological model. In particular, we motivate the main problems which need to be solved for specifying and detecting changes in the security context knowledge to be exploited for the sake of maintaining the security of a long-living software-intensive system. Thereby, we also provide background information on the approaches we are building on.

2.1 Context Knowledge in Secure Software Engineering

Secure software engineering is a software development paradigm that incorporates security as a primary quality attribute into software development. One of the key aspects that distinguishes security from many other quality attributes (e.g., performance, fault-tolerance, robustness, etc.) is that the security of a software-intensive system may be affected by external security knowledge [7]. Consider an information system that processes private data and which shall meet the Essential Security Requirement (ESR) that all database communications must be encrypted using a secure encryption algorithm. Obviously, what can be considered a secure encryption algorithm is not system-specific but part of a more general body of knowledge to which we refer as Security Context Knowledge (SCK). ESRs are called *essential* since they cover abstract, technology-independent security needs, while the SCK provides the knowledge required to properly implement ESRs for a specific system. In our example, the SCK provides information on which encryption algorithms can be considered secure.

In this article, we follow a model-based approach to secure software engineering called S^2EC^2O (Secure Software in Evolving Contexts via CO-evolution), in which ESRs are refined to concrete security requirements using the SCK. These security requirements are linked to appropriate security mechanisms and to model-based system implementations through a series of design decisions and using well-established approaches for secure, model-based software engineering, including the selection of concrete implementation technologies. We refer to the design artifacts describing a software system in terms of architecture and/or behavior as a *system model*, which we assume to be specified using UMLsec [31], a security-oriented extension of the Unified Modeling Language (UML).

On the implementation level, we assume the executable software artifacts to be written in Java. For the sake of security-related analysis and maintenance, we abstract from the actual source code by working with so-called program models provided by the GRaViTY framework [57]. A *program model* provides a structural representation of an object-oriented program, basically, a high-level abstraction of the program's Abstract Syntax Tree (AST) equipped with additional information (e.g.,

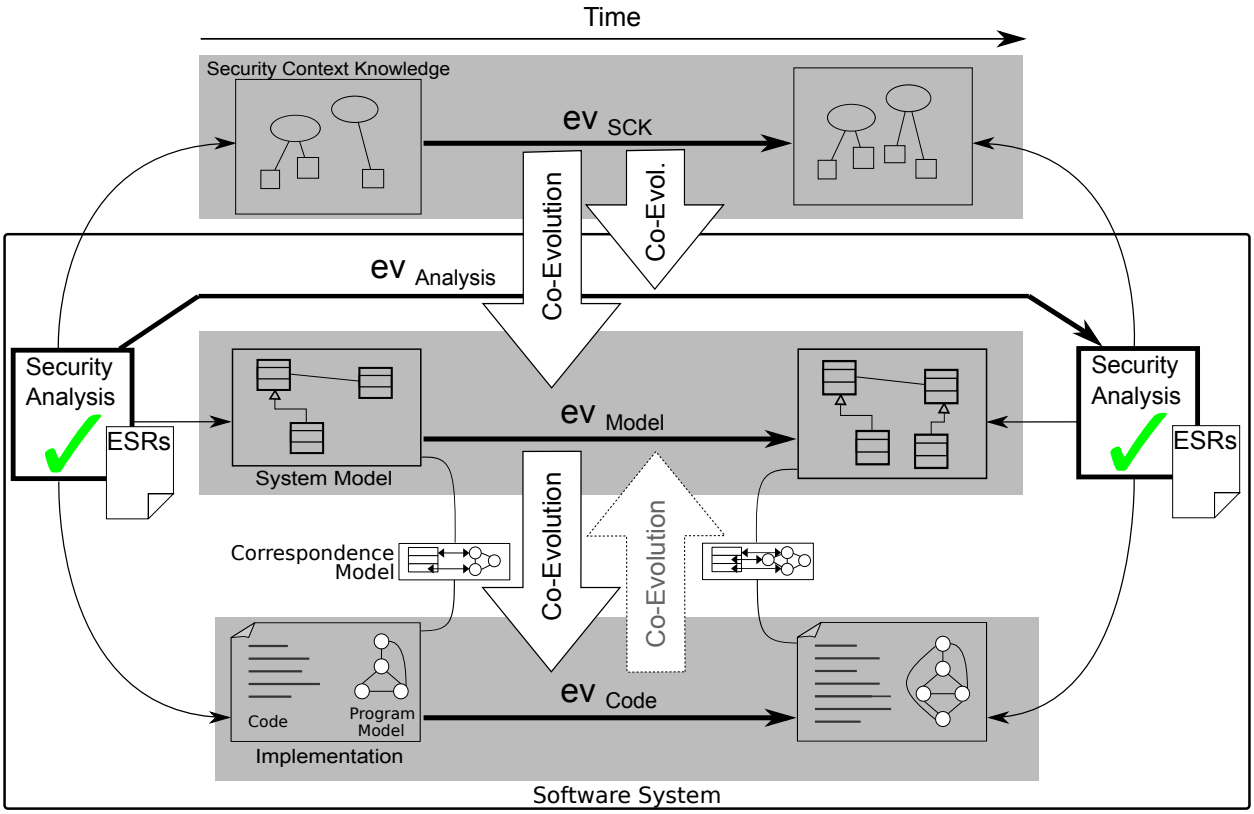

Fig. 1. Co-evolution in secure software engineering with evolving context knowledge.

cross-tree references representing call and access edges) facilitating program analysis and transformation (e.g., refactorings [59, 60], anti-pattern detection [61] and elimination [68], or compliance checks with models [62]). Moreover, GRaViTY provides the functionality to keep the system model, source code, and program model synchronized in case of changes. For this purpose, a correspondence model between these artifacts is maintained, expressing which elements correspond to each other. Fig. 1 illustrates the relation between the SCK (top), the system model (center), and the implementation (bottom). Thereby, the system model and implementation but also security checks and their configuration comprise the development artifacts of the software system under development.

When a system is initially developed, it is ideally compliant with all of its ESRs regarding the SCK, i.e., it passes a security analysis (left) [31]. However, SCK changes over time due to, e.g., newly discovered attacks or regulatory changes. When the SCK evolves (ev_{SCK}), the system model and its implementation need to be co-evolved (ev_{Model} and ev_{Code}) such that they pass the security analysis against the evolved security knowledge again. In some cases, it may be necessary to co-evolve the configuration of the security analysis itself ($ev_{Analysis}$), e.g., by updating the list of known weak encryption algorithms. Nevertheless, while preserving the possibility of executing an entire compliance check again, due to high computational effort we have to avoid entire compliance checks. A possible solution is to check only changed parts of the system again.

As the SCK is globally accepted knowledge, changes within the software system cannot immediately trigger the SCK's co-evolution. While low-level code changes can require a co-evolution of the system models to reflect these changes, in this article, we do not focus on this kind of changes but rely on the change propagation of GRaViTY to keep the system model and implementation consistent. After every manual change and every automated propagation of changes, the security compliance checks have to be re-executed to ensure compliance with the SCK. If the software sys-

tem is not compliant, additional iterations of manual changes and change propagation are required until a compliant software system is reached.

While there are approaches that support the specification and verification of security properties on models or source code, the main challenge lies in the integration of these approaches. Thereby, we have to find ways for bridging the conceptual differences between the SCK, design-time system models, and concrete implementations.

Our overall goal is to detect security-relevant changes to the SCK, determine the impact on the system's security, and to facilitate appropriate co-evolutions to (i) recover the system's compliance with the ESRs or (ii) to perform an updated analysis accounting for changes in the SCK. Moreover, security-relevant changes shall be used to trigger incremental security compliance checks, thereby avoiding a re-execution of the entire security analysis. To that end, we build upon concepts and techniques drawn from our previous work [7,57], notably S^2EC^2O's idea of an ontological formalization of SCK, its abstract concept of so-called security maintenance rules, and GRaViTY's program model and its synchronization facility. The main challenge addressed in this article is to handle the evolution of the SCK in terms of semantic editing patterns which can be later exploited for the sake of maintaining security.

2.2 Ontological Modeling of Security Context Knowledge

A basic prerequisite is the formal and explicit representation of SCK. To that end, we adopt an ontological approach leveraging the Web Ontology Language (OWL) as standardized by the W3C [53].

While its semantics is based on description logics (DLs) [5], the OWL notation is largely motivated by terminology known from object-oriented programming: Objects (or entities) are called *individuals*, and each object is an *instance of* a dedicated *class*. On the type level, OWL supports the definition of inheritance hierarchies through *subclass* relationships. On the instance level, so-called *object properties* may be attached to an individual by relating it to other individuals. Type and instance level correspond to what is usually referred to as TBox ("terminological part") and ABox ("assertional part") in DLs, respectively.

An ontological model of SCK is structured into three layers, each of them having a specific security focus. Lower layers may import and use the concepts defined by upper layers. In DL terminology, elements of the upper ontology are part of the TBox, domain and system ontologies can contain elements of both TBox and ABox.

Upper The upper ontology is independent of a particular software domain or application. It represents the most general software security concepts, such as *encryption algorithm* or *attack*. An upper ontology of software security concepts may be found in [11, 20].

Domain Domain ontologies capture domain knowledge as well as concrete security issues and measures. They have to be created for each domain anew and can be shared by different systems in the same domain.

System System ontologies express the security-relevant knowledge about a concrete system. They can be produced or enriched from existing artifacts, such as a UML-based system model or the implementation of the system.

Fig. 2 shows an excerpt of an ontological model of SCK which is relevant for our example. In the initial version 1 on the left, it includes elements from different layers, namely a security property (`Secure Encryption`), system components and concepts (`Database Connection`, `Communication Path`), and an encryption algorithm (`RC4`) which is considered to provide secure encryption.

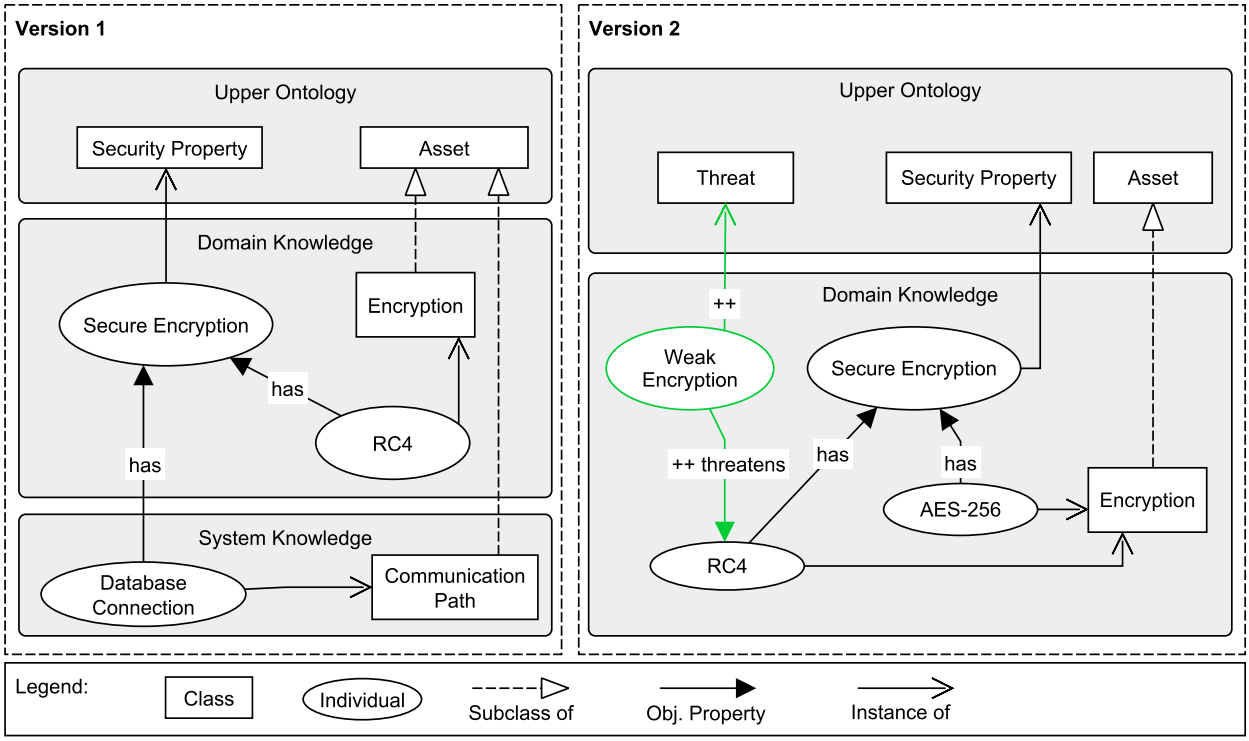

Fig. 2. Example of an ontological model of security context knowledge subject to evolution: An encryption algorithm is discovered to be vulnerable.

SCK is usually gathered from natural language documents of various kinds, e.g., the IT baseline protection guidelines proposed by the German Federal Office for Information Security, or attack and vulnerability reports as provided in the Common Vulnerabilities and Exposures (CVE) database. Moreover, community knowledge regarding known vulnerabilities and mitigations is available, e.g., in terms of the Common Weakness Enumeration (CWE) database. Finally, individual persons such as white hats or developers can contribute to the SCK.

2.3 Security Context Knowledge Evolution

As already mentioned, SCK may change over time. This holds in particular for domain and system knowledge, while we assume the upper ontology to be stable. An example of such SCK evolution this is illustrated in Fig. 2. The initial version of the ontological model evolves into version 2 shown on the right. Essentially, `Weak Encryption` of `RC4` is added as a new `Threat`.

In general, two main approaches to obtain a difference between two versions of a structural model, such as our ontological representation of SCK, have been proposed in the literature; *logging* (also known as *recording* or *operation-based differencing*) and *state-based comparison* [32]. The idea of logging dates back to Lippe et al. [48], who assumed an object-oriented database management system to be used for storing documents, and each edit command to be executed as a transaction on the object base. The idea was to exploit the logging facilities of the database management system to generate changelogs. The same basic idea was later adapted to changelogs maintained in syntax-based editors (e.g, [6, 71]). Logging-based approaches have the advantage that changes

https://www.bsi.bund.de/EN/Topics/ITGrundschutz/itgrundschutz_node.html
https://cve.mitre.org
https://cwe.mitre.org

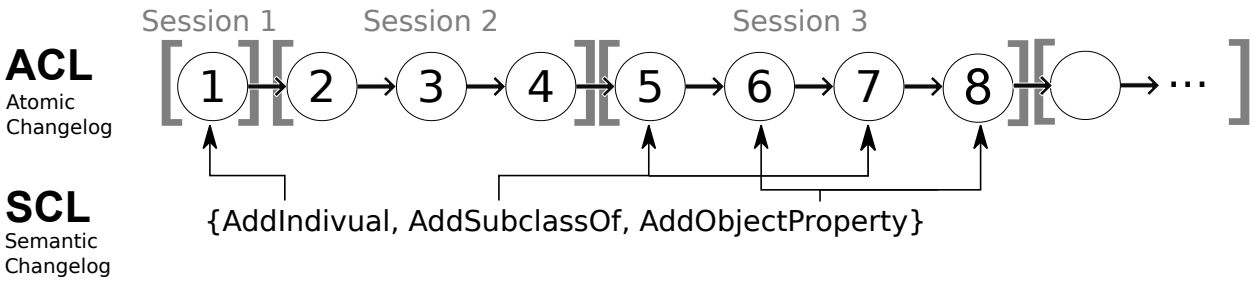

Fig. 3. Low-level changelog (top) vs. semantic editing patterns (bottom) comprising a set of intermixed low-level changes.

can be obtained directly by logging editing commands. In principle, editing commands could be designed to work on the level of semantic editing patterns, such as adding Weak Encryption of RC4 as a new Threat. In practice, however, change recording is faced with the following organizational and technical challenges.

From an organizational point of view, ontologies representing the SCK may be updated by a variety of different tools and processes in a largely uncontrolled and ad-hoc manner. In some cases, parts of the ontological model of SCK are automatically extracted from external knowledge sources [25], which means that there is no dedicated editing process at all but parts of the SCK are simply replaced by a new version. As a consequence, occurrences of semantic editing patterns as the one in our example must be detected after the fact, i.e., by comparing the old and new version of an ontology [67]. They cannot be recorded through a dedicated tool or software library such as the OWL API of Protégé. In general, the absence of a changelog is a well-known problem when models are exchanged across tool boundaries [45].

Technically, even if all changes would be applied through a common tool or API, the resulting changelogs are bound to the fixed set of edit operations being offered by the editing environment. In the case of Protégé, the generated changelogs capture ontological changes on a level of granularity which is way too fine-grained to determine the semantic impact of the recorded changes [54, 55]. Moreover, different semantic editing patterns can be performed in an interleaved manner [67]. As an example, consider the changes between the two ontology versions 1 and 2 in Fig. 2. Assume that, according to the illustration in Fig. 3, a security expert performs this task in two editing sessions (Session 1 and Session 3) which are interrupted by an externally triggered change (Session 2). In Session 1, (s)he adds the individual `WeakEncryption`, and (s)he completes the editing task in Session 3 by declaring `WeakEncryption` as an instance of class `Threat` and by adding an object property targeting `RC4`. Logging through the OWL API would result in a stream of low-level changes as illustrated in the upper part of Fig. 3. Note that in OWL, relationships are represented as first class citizens connected to their source and target elements. Thus, the addition of each of the relationships introduced in Session 3 boils down to first adding the element representing a relationship which is then connected to its source and target. Consequently, all low-level changes need to be grouped to form occurrences of pre-defined semantic editing patterns such as "addEncryptionThreat", as the lower half of Fig. 3 exemplifies.

While the technical problem of recording low-level changes could be solved by lifting change APIs of tools like Protégé to the level of semantic editing patterns (see also the discussion on related work in Sect. 8), the organizational problem remains: Not all the changes to the SCK can be recorded through a controlled change management process and system, but must be detected after the fact by

`https://protege.stanford.edu`

The effect of Session 2 is not illustrated in Fig. 2.

comparing the old and the new version of the ontological representation of SCK. To that end, state-based differencing techniques do not assume any logging information, but are only based the original and the revised version of the models which are to be compared. Virtually all of them (see, e.g., [41] for a survey), have a similar processing pipeline like the basic differencing pipeline (steps 1 and 2) shown in Fig. 6: A matching procedure first searches for pairs of corresponding model elements which are considered the same in both models, all elements not involved in a correspondence are then considered to be deleted or created. However, this kind of difference derivation basically yields the same low-level changes as reported by the changelog of Protégé. They need to be lifted to the level of semantic editing patterns which can be leveraged for further analysis of the evolution of SCK. This challenge which will be addressed in the next section.

3 Detection of Security Context Knowledge Changes

Before we can mitigate security context knowledge changes using a combined approach of S^2EC^2O and GRaViTY, we have to detect such changes and have to extract the semantic information of what has been changed. In this section, we answer RQ1 regarding the detection of security context knowledge changes and extraction of semantic changes. For this purpose, we introduce how we represent the security context knowledge and how to detect and extract their semantic impact in the security context knowledge.

3.1 Structural Representation of Ontologies

As illustrated in the previous section, semantic editing patterns rely on a structural, graph-based representation of ontologies. To that end, we adopt the concept of meta-modeling that has been established in model-based software engineering. Following this approach, an ontology is considered as a typed attributed graph, the meta-model defines the allowed types of the nodes and edges. Conceptually irrelevant details, notably the layout of an external diagram notation such as the one used in Fig. 2, are ignored. In our prototypical implementation, we work with a MOF-based meta-model for OWL. MOF-based meta-models are based on basic principles of object-oriented modeling. In particular, node types are specified by classes which can be related by generalization relationships. Edge types are specified by associations equipped with multiplicity constraints.

Fig. 4 shows an excerpt of our MOF-based OWL meta-model. Entities, i.e., `NamedIndividual`s, `Class`es and `ObjectProperties`, do not have attributes that make them distinguishable directly, but their "identity" is given by relating them to an `URI` node via an edge of type `entityURI`. Instance-of relationships are represented by `ClassAssertion`s relating an individual to its class via edges of type `individual` and `classExpression`. `ObjectProperties` relate two individuals via edges of type `sourceIndividual` and `targetIndividual`.

3.2 Rule-based Specification of Semantic Editing Patterns

Using a graph-based representation of ontologies, we consider the effect of applying a semantic editing pattern as an in-place graph transformation and use the transformation language Henshin [79] for specifying editing patterns of interest. Henshin is based on graph transformation concepts, which enables us to specify editing patterns as declarative graph transformation rules. A Henshin rule $r : L \to R$ consists of two graphs L and R referred to as left-hand side and right-hand side, respectively. The notation $L \to R$ symbolizes a partial mapping which, by adopting notations from

https://www.omg.org/mof
https://www.w3.org/2007/OWL/wiki/MOF-Based_Metamodel

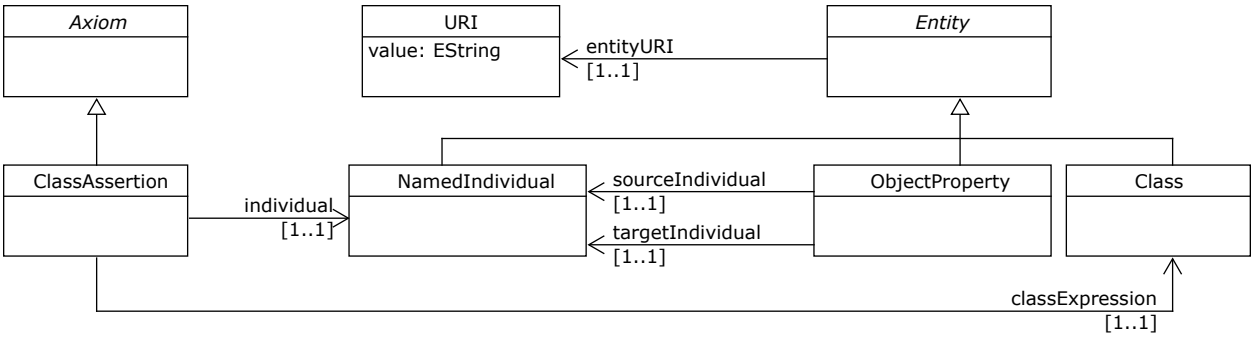

Fig. 4. MOF-based meta-model serving as a type graph for the structural, graph-based representation of OWL ontologies.

set theory loosely, induces the graph patterns to be found and preserved ($L \cap R$), to be deleted ($L \setminus R$), and to be created ($R \setminus L$) by a rule. In the visual Henshin transformation language, the left- and right-hand side of a rule are integrated in a "unified graph", the graph patterns $L \cap R$, $L \setminus R$ and $R \setminus L$ are marked by stereotypes « preserve », « delete » and « create », respectively.

To give an example, Fig. 5 shows the Henshin rule *addEncryptionThreat*. This rule formally specifies the semantic editing pattern turning version 1 of the ontology shown in Fig. 2 into its revised version 2. The parts which specify the changes illustrated in Fig. 3 are indicated by dashed ellipses. Change actions indicated by ① specify the creation of an individual named *WeakEncryption* (represented by nodes of type `Class` and `URI` with value `cwe326#Weak_Encryption`). Change actions indicated by ⑤ and ⑦ specify the creation of an instance-of relationship (node of type `ClassAssertion`) which declares the new individual to be an instance of *Threat* (represented by nodes of type `Class` and `URI` with value `upper#Threat`). The class *Threat* itself already exists, it is imported from the upper ontology. Change actions indicated by ⑥ and ⑧ specify the attachment of the object property *threatens* (nodes of type `ObjectProperty` and `URI` with value `upper#threatens`) to the new individual. The object property targets an individual (node of type `NamedIndividual`) which is an instance of *Encryption* (represented by nodes of type `ClassAssertion`, `Class` and `URI` with value `cwe326#Encryption`. Analogously to the class *Threat*, the object property *threatens* as well as the individual which is an instance of *Encryption* already exist (imported from the upper ontology).

Note that the manual specification of semantic editing patterns may be supported by semi-automated techniques, e.g., by learning from examples [33] or past evolutions [39]. Their exploration is out of the scope of this paper.

3.3 Recognition of Pattern Occurrences

Our approach to recognizing occurrences of semantic editing patterns between two versions of the structural representation of an OWL ontology adopts the model differencing approach and tool known as SiLift [34]. The differencing pipeline implemented in SiLift basically consists of the three successive steps depicted in Fig. 6. While a detailed description of the generic aspects of each of these steps is out of the scope of this paper and may be found in [36], we give a brief illustration for the sake of self-containedness, concentrating on how we adapt the pipeline to OWL ontologies.

In the first step, a model matcher identifies the corresponding elements which are considered "the same" in both versions v_1 and v_2. The matcher is an exchangeable component within the SiLift framework. Currently, we use a rather simple one which matches entities based on their URIs, relying on the assumption that equally named entities refer to the same conceptual elements. Relationships are matched if they connect the same source and target elements. In principle, this

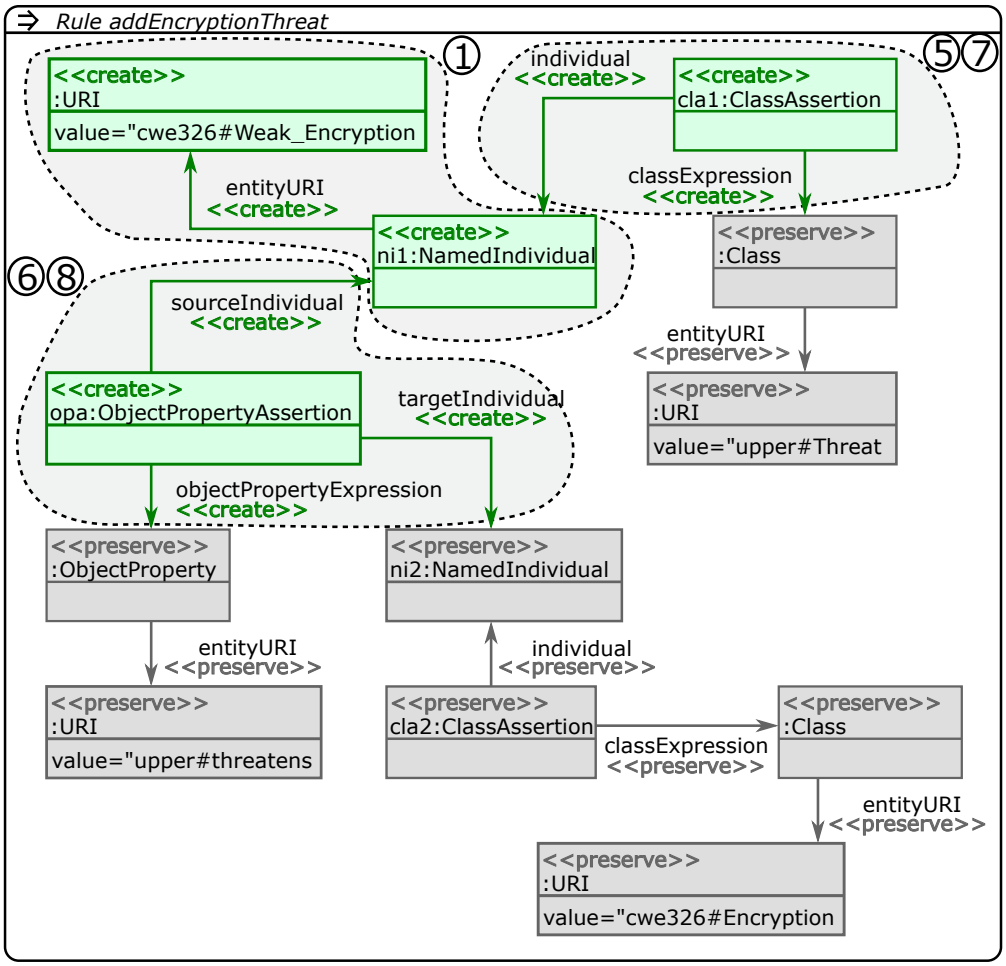

Fig. 5. Rule-based specification of a semantic editing pattern updating the SCK: Addition of a *Threat* to an existing *Encryption* of the Security Context Knowledge.

rather simple matching strategy implemented in our research prototype can be replaced by more sophisticated matchers, such as an adaptable model matcher [35] or more semantically oriented ontology matcher [52].

Subsequently, a low-level difference is derived from the matching. Elements of v_1 which do not have a corresponding element in v_2 are considered to be deleted. Analogously, elements exclusively comprised by v_2 are considered to be created. The low-level difference derivator is a generic component which can be adopted from the SiLift framework without the need for any adaptation.

Finally, a pattern detection algorithm recognizes occurrences of semantic editing patterns in a given low-level difference. The pattern detection engine is configured by formal specifications of semantic editing patterns using the Henshin transformation language, as presented in the previous section. Essentially, SiLift exploits the fact that the application of a semantic editing pattern leads to well-defined change pattern in the given low-level difference. All low-level changes involved in such a change pattern are grouped to a so-called semantic change set, a set of low-level changes as indicated by the semantic editing log in the lower part of Fig. 3.

Pattern occurrences may be accessed through the SiLift API. Basically, all rule graph elements comprised by the respective Henshin rule can be traced to their occurrence in the low-level difference between the two versions v_1 and v_2. Hence, semantic editing pattern occurrences being observable

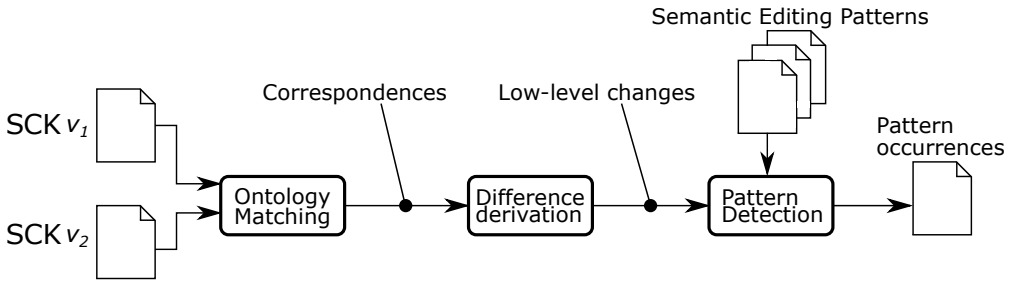

Fig. 6. Processing pipeline of the model differencing tool SiLift [34] adapted to detect occurrences of semantic editing patterns in evolving OWL ontologies.

in the evolution of the SCK can be automatically processed in subsequent steps, as we will illustrate in the following sections.

To answer RQ1 on how to detect security context knowledge changes and extract their semantic impact, we propose to use rule-based semantic editing patterns for detecting semantic changes in SCK maintained in ontologies. We have shown how semantic editing patterns can be specified and instanced be detected on ontologies.

4 Maintaining Security by Leveraging Semantic Editing Patterns

To keep the system in a secure state in case of changes in the security context knowledge, we have to adapt the system to be compliant with the SCK again. Fig. 7 shows our approach for adapting and verifying the security compliance of a system with the SCK. The adaptation of the system starts with the detection of changes in the SCK using the semantic editing patterns introduced in the previous section. To react to such changes and answer RQ2, we introduce a rule-based approach for adapting the system model and security check configurations (RQ3) to meet the changed security requirements and execute additional compliance checks. The adaptation is propagated over the correspondence model into the implementation. To verify the compliance of the implementation with the system model and the SCK, as considered in RQ4, we introduce security violation patterns that leverage the information stored in the correspondence model for security compliance checks.

In the remainder of this section, we first introduce how security can be maintained by leveraging semantic editing patterns. Next, we show how configurations of state of the art security checks can be co-evolved to allow checks against changed security knowledge. Afterwards, we introduce how implementation-level security compliance can be enforced using security violation patterns.

4.1 Generation of Model Co-Evolution Proposals

To mitigate security vulnerabilities as introduced in the previous section and to generate model co-evolution proposals as required for answering RQ2, we propose to trigger the maintenance mechanism of S^2EC^2O based on the detected changes. S^2EC^2O follows a rule-based approach to security maintenance using so-called Security Maintenance Rules (SMRs) [8]. The goal of an SMR is to recover the security of a system when a threat is discovered w.r.t. an ESR. Therefore, an SMR proposes possible adaptations of the system model mitigating the threat. In general, SMRs follow the *Event-Condition-Action* principle: An external event triggers an SMR which, if its condition evaluates to true, causes a set of adaptations to be carried out as the action part of the SMR. In particular, SMRs may be triggered by changes in the SCK, and they may leverage occurrences of semantic editing patterns to determine possible adaptations.

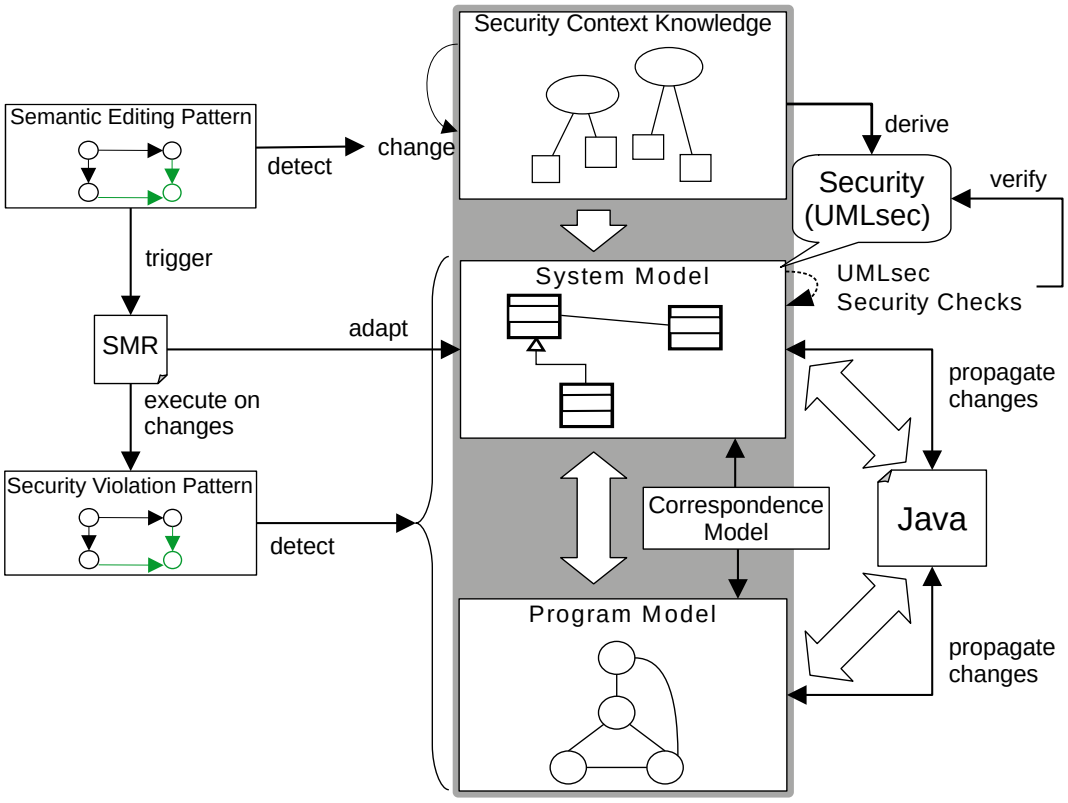

Fig. 7. Approach for the detection of SCK changes, semi-automated co-evolution of the system, and incremental security compliance checks.

The execution of an SMR follows the *Event-Condition-Action* principle. As an event, we consider a security context knowledge change that has been detected in terms of an occurrence of a semantic editing pattern. Each pattern occurrence can be associated with SMRs that are executed on the detected change. SMRs are specified by implementing two hook methods, dedicated to the Condition and Action parts of the principle. In what follows, we concentrate on the condition and action part which need to be tailored for a particular threat:

- `checkConditions(deltaList DeltaList)` is used to realize the *Condition* part. By using the parameter `deltaList`, the SMR may access occurrences of semantic editing patterns on the evolving SCK and, if a threat is discovered, generate proposals on how to mitigate the treat.
- As soon as the user or an algorithm has selected a proposal, the method `apply(proposal Proposal)` is called, realizing the *Action* part. The proposal which is to be used by the SMR to co-evolve the system model is supplied as a parameter.

To give an example, Listing 1.1 shows the SMR which is dedicated to the knowledge evolution declaring RC4 as a vulnerable encryption algorithm. Its condition and action part are illustrated using pseudo-code.

In the condition part, first, all currently used encryption algorithms are queried from the system model (line 2). After that, all alternative algorithms are gathered (line 3). Therefore, the SCK is queried for encryption algorithms that are currently not threatened. Next, we walk through the changes in the SCK (for-each loop at line 4), checking for the occurrence of the semantic editing pattern "addEncryptionThreat" (line 5). The threatened encryption algorithms are obtained from the occurrence of the semantic editing pattern through the SiLift API (line 6) and added to a globally accessible collection (line 7). Next, the SMR checks whether the threatened algorithm is currently also used in the system model (line 8). If so, for every possible combination of exchanging this specific algorithm with a non-threatened one, a proposal mitigating the threat is generated and added to the collection of proposals (line 10).

```
1  checkConditions(DeltaList deltaList) {
2   currentAlgorithms = queryModelForUsedAlgorithms();
3   alternativeAlgorithms = querySckForAlternativeAlgorithms();
4   for (pattern : d.getSemanticEditingPatterns()){
5    if (pattern.getName().equals("addEncryptionThreat")){
6     threatened = pattern.getParameter("alg");
7     threatenedAlgorithms.add(threatened);
8     if (currentAlgorithms.contains(threatened)){
9      for (alternative : alternativeAlgorithms){
10      Proposal.addAlternative("Replace" + threatened + " by " + alternative);
11     }
12    }
13   }
14  }
15 }
16
17 apply(Proposal p) {
18  for (cp : queryModelForCommunicationPaths) {
19   if (threatenedAlgorithms.contains(cp.getAlgorithm)){
20    alterModel(p.getChoice());
21   }
22  }
23 }
```

Listing 1.1. Condition and action part of the SMR mitigating the detection of a vulnerable encryption algorithm in the evolving SCK.

After a developer has made a choice on which proposal shall be selected to adapt the system, the action part of the SMR is being executed (lines 17-21). It iterates over all communication

paths of the system model, and every threatened encryption algorithm is replaced according to the chosen proposal.

Next, the implementation has to be adapted to reflect the changes performed on the system model. While structural changes on the system model, such as the extraction of information into a new class, can be propagated into the implementation by GRaViTY, manual actions of developers are required in this case. To change an encryption algorithm on the system model it is sufficient to change the value of the property specifying the used algorithm. However, this change might require more changes in the implementation. For example, using an entirely different API or even switching to a different cryptographic library could be necessary.

To answer RQ2 on how to generate model co-evolution proposals to mitigate the changes on the architectural level, we propose to specify SMRs corresponding with the recognition rules for semantic editing patterns. Every time an occurrence of a semantic editing pattern is detected, the corresponding SMRs are automatically triggered to provide model co-evolution proposals for mitigating the change. On the concrete example of the addition of a new encryption threat, we have shown how such an SMR has to be specified.

4.2 Adaptation of Standard Security Checks

As a reaction to an evolution of the SCK, it is not sufficient to adapt only the system model, but the option of conducting an entire security compliance analysis should be preserved. Following RQ3, we have to co-evolve security check configurations to allow checks against the changed security context knowledge. For example, between the SCK and the model-level UMLsec security checks, such as shown in the upper right part of Fig. 7. One of these UMLsec checks is the *secure links* check that verifies whether sensitive information is only communicated over communication paths with sufficient protection, such as strong encryption. Using the SMR in Listing 1.1 we have adapted the system model to not contain a weakness due to new vulnerable encryption. However, the UMLsec security check initially used to ensure the security compliance of the system model with the SCK is not capable of detecting uses of new vulnerable encryption algorithms, e.g., RC4 as introduced in the previous subsection. Accordingly, we cannot any longer use this security check to ensure compliance after changes or additions to the system model. The same holds for security checks on the implementation level. Here, developers can be supported in implementing secure code by static analysis tools such as SonarQube [12,76]. Comparable to the UMLsec secure link check, SonarQube provides a check to detect uses of vulnerable encryption algorithms (S5547) [74].

In what follows, we consider the integration of SonarQube into the Eclipse IDE, called SonarLint [75]. SonarLint runs a local SonarQube Server for performing analyses and is integrated into the Eclipse IDE. By default, the SonarQube check S5547 is not adaptable. However, SonarQube offers parameters for adding configuration information to custom rules. Using a parameter, we changed this check to not consider a hardcoded list of vulnerable encryption algorithms but to check against a configurable list of vulnerable encryption algorithms. In Listing 1.2, we show our modification to the implementation of the check. The field `DEFAULT_VULNERABLE_ALGORITHMS` contains the static initialization of the known vulnerable algorithms. We use this as the default value for the field `VULNERABLE_ALGORITHMS` which is accessed when the check is executed. We marked this field as `@RuleProperty` which contains a key with the value `vulnerable`. Using this key, custom values for this field can be provided. If no value is provided, the default value is used. As the default implementation of the security check in method `checkIssue` (lines 17-22) needs a collection of vulnerable algorithms, we added a helper method `getVulnerableAlgorithms` that splits the comma-separated list of algorithms. Immediately modifying the field `VULNERABLE_ALGORITHMS` is not supported by the parameters used in SonarQube.

```
1  @Rule(key = "S5547")
2  class StrongCipherAlgorithmCheck extends AbstractMethodDetection {
3
4   private static final String DEFAULT_VULNERABLE_ALGORITHMS = Stream.of("DES", "RC2", "
       Blowfish").collect(Collectors.joining(",");
5
6   @RuleProperty(key = "vulnerable",
7    description = "Comma separated list of vulnerable encryption algorithms",
8    defaultValue = "" + DEFAULT_VULNERABLE_ALGORITHMS)
9   public String VULNERABLE_ALGORITHMS = DEFAULT_VULNERABLE_ALGORITHMS;
10
11  Collection<String> getVulnerableAlgorithms() {
12   return Arrays.asList(VULNERABLE_ALGORITHMS.toUpperCase().split(","));
13  }
14
15  ...
16
17  void checkIssue(ExpressionTree argumentForReport, String algo) {
18   String[] transformationElements = algo.split("/");
19   if (transformationElements.length > 0 && getVulnerableAlgorithms().contains(
        transformationElements[0].toUpperCase())) {
20    reportIssue(argumentForReport, MESSAGE);
21   }
22  }
23 }
```

Listing 1.2. Adaptable implementation of SonarQube rule S5547 [74].

```
1  checkConditions(DeltaList deltaList) {
2   for (pattern : deltaList.getSemanticEditingPatterns()){
3    if (pattern.getName().equals("addEncryptionThreat")){
4     Proposal.addAlternative("Update checks to consider as vulnerable algorithm: " +
         pattern.getParameter("alg"));
5    }
6   }
7  }
8
9  apply(Proposal p) {
10  newVulnerable = p.getChoice().getData();
11  sl = UMLsec.getConfig(project).getCheck("Secure Links");
12  if(sl != null) {
13   sl.getThreatendedAlgorithms().add(newVulnerable);
14  }
15
16  rules = SonarLintGlobalConfiguration.readRulesConfig();
17  rule = rules.getAny(r->"S5547".equals(r.getKey()));
18  rule.getParams().merge("vulnerable", newVulnerable, (o,n) -> o+","+n);
19  SonarLintGlobalConfiguration.saveRulesConfig(rules);
20  if(rule.isActive()) {
21   AbstractAnalyzeProjectJob.create().schedule();
22  }
23 }
```

Listing 1.3. SMR for the co-evolution of security-check configurations.

Similar to the generation of co-evolution proposals, we can specify SMRs for the adaptation of the configurations of security checks. In Listing 1.3 we show an SMR that adapts the configuration of the UMLsec secure links check as well as the SonarQube rule S5547 for the detection of uses of vulnerable cipher algorithms. In Listing 1.3, we show the condition and action part of this SMR as pseudocode. Similarly to the SMR in Listing 1.1, we check the applicability and generate proposals in the condition part of this SMR. Again, we iterate over all detected occurrences of semantic editing patterns and collect all that added a new encryption threat (lines 2 and 3). For every new encryption threat, we propose to update the configurations of the security checks to consider the new vulnerable algorithm. For the application of the SMR, at first, the SMR reads the new vulnerable encryption algorithm from the selected proposal (line 10). Afterwards, it adapts the

UMLsec secure links check (lines 11 - 14) and the SonarQube rule S5547 (lines 16-22). For the adaptation of the UMLsec secure links check, we read the UMLsec configuration of the project containing the system model and access the configuration part specific to secure links (line 11). If secure links is present in this configuration, we add the new vulnerable encryption algorithm to the list of vulnerable encryption algorithms (lines 12-13). Regarding SonarQube, we also get all rule configurations from the central SonarLint configuration and search for the rule S5547 that we want to adapt (lines 16 - 17). For rule S5547 we access the parameter for vulnerable encryption algorithms with the key `vulnerable` and merge the new vulnerable encryption algorithm into the already configured vulnerable algorithms (line 19). Afterwards, we save the adapted configuration.

As opposed to the system model, where it has been ensured by the SMR in Listing 1.1 that the system is in a secure state, no automated adaptation is applied to the implementation. To inform developers about locations at which an adaptation is required, we execute SonarQube to detect all locations where a vulnerable encryption algorithm is used if the rule S5547 is active in the current SonarLint configuration (lines 20 - 21).

To answer RQ3 on how to co-evolve security check configurations to allow checks against the changed security context knowledge, we propose to utilize SMRs for this purpose. In this subsection, we have shown how this utilization of SMRs can be realized for adapting the security check configuration of the state of the art static analysis tool SonarLint. Similar SMRs should be defined for all security checks relevant for the software systems security.

4.3 Incremental Security Compliance Analyses

While GRaViTY allows the propagation of structural changes made by SMRs into the implementation [58, 60], we need a verification of the adapted design-time security properties on the implementation level. As a first idea, we could re-execute all implementation-level security checks. However, this comes with two drawbacks. First, even when these have been adapted to reflect the current SCK, these are usually disconnected from the security requirements specified on architectural models of the system. Second, a full compliance check is usually very time-consuming. For this reason, following RQ4, we need an efficient verification of the compliance of the system's implementation with the co-evolved architecture.

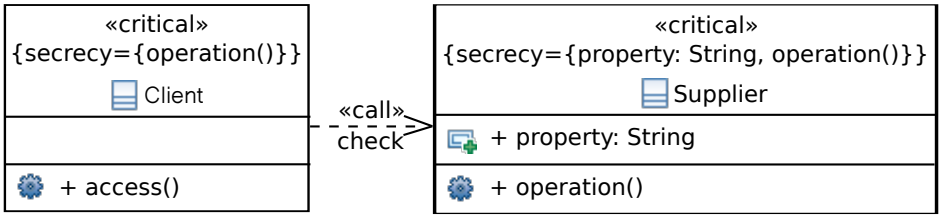

Fig. 8. Example for UMLsec secure dependency.

As a connection between GRaViTY and S^2EC^2O, we can use the UMLsec security specifications edited and checked by SMRs on the system model-level. One example of such a UMLsec security specification is the *secure dependency* check [31]. Secure dependency aims to structure the application into different security levels for critical class members. Accesses to members on such a security level are only allowed by entities that obey these security levels according to their security specifications. Thereby, the information of which class members are critical, e.g., attributes holding sensitive data, might be subject to security context knowledge changes.

Fig. 8 shows a UML class diagram with UMLsec stereotypes following UMLsec secure dependency. A supplier specifies a property and a method that can be accessed by a client, providing information to the client. The security level of information transmitted along a «call»-dependency can be specified by the «critical» stereotype of UMLsec. In the example, both the method and the property are put to the security level of secrecy by a security expert. For every incoming dependency to this class, all accessing classes have to specify their compliance by adding a corresponding critical stereotype containing the signatures of the critical members on the required security levels. In the example, the client is on the security level of secrecy for the signature `operation()`. However, there is a violation of the secure dependency property for `property:String` as the «critical» stereotype on `Client` does not contain the signature `property:String` on the level of secrecy.

Using the tool support of UMLsec, security experts can find such violations against the structuring of the application into security levels. The concrete violation of the example can be mitigated by removing the violating dependencies or by adding the `Client` to the security level of secrecy for the violated security level of a member.

Which kind of information should be put to which security level is captured in the SCK. To show that an implementation that is compliant with the security levels derived from the SCK, we have to prove that the implementation does not contain violating accesses to elements representing critical members in the system model. If we execute such a compliance check as a reaction to model-level changes made by an SMR, an entire compliance check is not necessary but it is sufficient to only check all changed parts.

For compliance checks, we leverage the information stored in the correspondence model between the UML system model and the program model. For every class in the system model, we have one or more corresponding classes in the implementation. The same holds for the operations and attributes of classes in the system model. These are corresponding to one or more methods or fields in the implementation. For example, in Java, it is a common practice to encapsulate a property from the system model by the use of getter and setter methods at the implementation level.

Comparable to the semantic editing patterns on the ontology, we can specify security violation patterns using the Henshin transformation language. In Fig. 9, we show a rule for detecting a violation of the system model-level secure dependency specification on the implementation level for the security level of secrecy. The rule shows on the left the elements from the UML models, in the center the correspondences from the correspondence model, and on the right the elements from the program model.

On the UML model, the rule matches in part ① of the rule every feature (operations or properties) contained in a class `supplier` with the stereotype critical, that contains the signature of the feature in the list of signatures on the level of secrecy. Thereby, the containment is expressed in the condition on top of the rule. Also on the UML model, in part ②, the rule matches a `client` class which does not specify the signature of the feature in a critical stereotype and which is connected with the other class over the implementation. This connection to the implementation is expressed in part ③ showing the correspondences between the matches in the system model and the match in the program model that have to be present. In part ④ of the rule, an access from a member of a type in the implementation corresponding with the class `client` to a member corresponding with the feature of the class `supplier` is matched. If this rule matches, we found a security violation in the implementation regarding the model-level security specification.

If we have a closer look at the name of the rule, we can see that this is followed by parameters of the format `kind name:type`. These parameters make the rule elements with the same name accessible to the conditions of the rule and the caller of the rule. The parameter kinds in the rule are `var` and `in`. While `var` parameters only serve as internal variables, parameters of kind `in` can be bound to an element when calling the rule. For the shown rule, the parameter `supplierCritical`

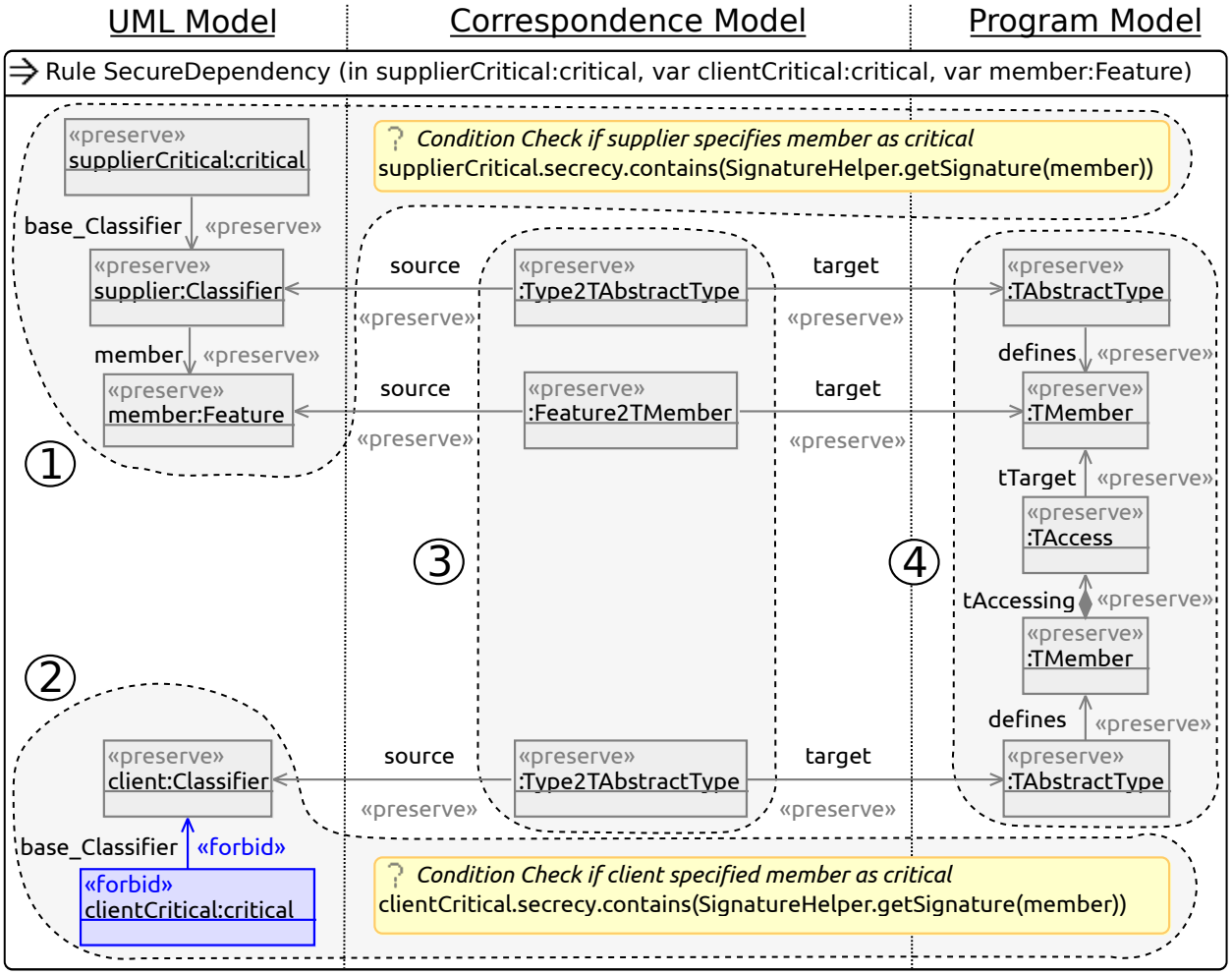

Fig. 9. Rule-based specification of a security violation pattern for the detection of a violation in the implementation according to the design-time security specification.

may be bound by the rule's caller. This allows us to bind this parameter to the « critical » stereotypes that have been modified by an SMR and to restrict the compliance check only to the changed security properties.

To answer RQ4 on how to efficiently verify the compliance of the system's implementation with the co-evolved architecture, we introduced security violation patterns. As shown in this section, using security violation patterns, we can specify security violations on the implementation level with respect to security properties specified in architectural models. In case of changes, a matching of these patterns can be initialized with the changed parts to restrict the search-space and only execute compliance checks on the changed parts.

5 Prototypical Tool Support

The S^2EC^2O tool as well as GRaViTY and the extensions presented in this work are realized as an extension of the Eclipse platform. Fig. 10 gives an overview of the tool's architecture.

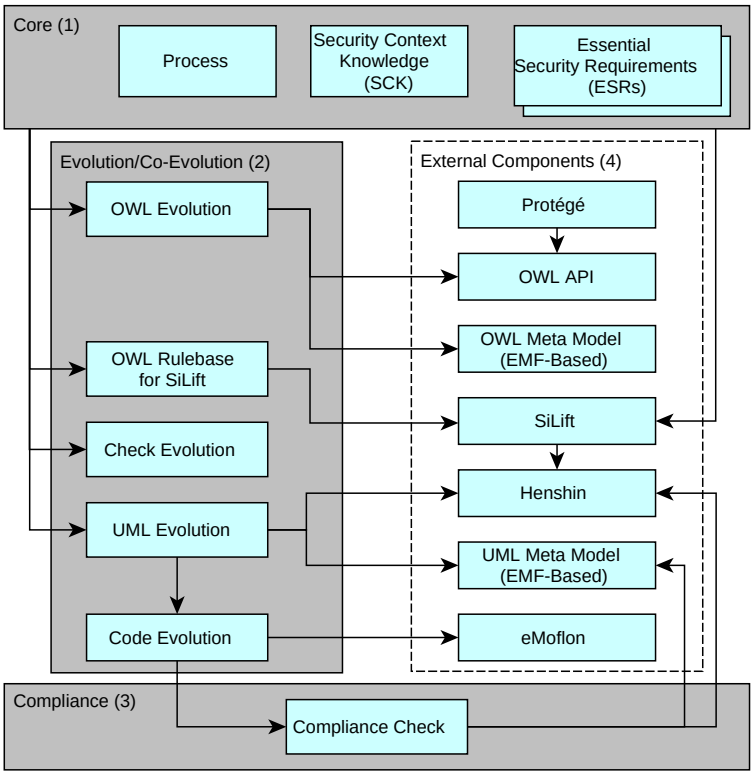

Fig. 10. Overview of the used prototype components.

The tool architecture consists of three sub-systems and uses a set of external components (arrows between components represent dependencies):

Core. `Process` is the main component of the *core* sub-system. It realizes an engine for the S^2EC^2O process, supporting both automated and interactive parts. Moreover, the core sub-system provides access to the SCK and ESRs.

Evolution/Co-Evolution. We implemented four components to support the *evolution and co-evolution* of OWL ontologies (representing the SCK), system models (realized as UML(sec) models), the implementation, and security check configurations.

`OWL Evolution` is responsible for the transformation from API-based representations of OWL ontologies to instantiations of the EMF-based OWL meta-model, i.e., the structural representation we work with (s. Sect. 3). Furthermore, reasoners as provided by the OWL API can be used. Thus, the plug-in uses the OWL API as well as the OWL EMF meta-model.

`OWL Rulebase` adapts the SiLift model differencing framework in order to recognize occurrences of semantic editing patterns in evolving OWL ontologies. This is done by using SiLift and leveraging the SCK ontologies transformed into the EMF format.

`Check Evolution` implements SMRs for the co-evolution of the configuration state of the art security checks using the provided Java interface. Currently, we support UMLsec checks and SonarLint with a refined version of SonarQube.

`UML Evolution` realizes model co-evolution for UMLsec models by interpreting SMRs, as illustrated in the previous sub-section. To realize analyses of UMLsec models, it makes use of an EMF-based implementation of the UML meta-model. Co-Evolutions are then realized and applied by leveraging the Henshin graph transformation framework. Additionally, supplementary Java code can be used to support queries as well as co-evolutions.

`Code Evolution` provides the synchronization between the implementation level and the system model. To keep the artifacts consistent, we employ triple graph grammars (TGGs) [72]. In contrast to the Henshin transformation language which is well-suited for detecting patterns inside a single model (as used in this work for detecting security knowledge changes and specifying compliance checks), TGGs provide a bi-directional transformation between multiple models that allows us to synchronize the used artifacts. A TGG consists of transformation rules that express which elements should be translated to each other, i.e., in the UML model and program model, given a context of already translated model elements. Thereby, a correspondence model between the different artifacts is maintained for capturing which elements have been translated to each other. After this initial translation, synchronization of changes can be performed by undoing rule applications for deleted elements and translating new elements with additional rule applications.

Compliance. The component `Compliance Check` implements security compliance checks in terms of security violation patterns and their execution. As input, the component uses the correspondence model and program model maintained by the `Code Evolution` as well as a system model according to the UML meta-model. The security violation patterns are specified and executed using the Henshin graph transformation framework.

External Components. As for *external* tools, we chose `Protégé` to manage OWL knowledge bases, and we use EMF-based implementations of the OWL and UML meta-model, respectively. For the realization of the synchronization between the system model, implementation, and program model, GRaViTY uses the eMoflon transformation tool [14, 47]. For this purpose, eMoflon implements triple graph grammars [72]. Finally, the model transformation framework Henshin is not only used by SiLift but also used for a declarative specification of SMRs and Security Violation Patterns.

6 Feasibility Study

In the previous sections , we introduced the design of a technical approach that was guided by four research questions, and a prototypical realization of the proposed solutions. While we have shown

the concepts on concrete examples, we have not shown that these solutions can be applied to a real-world system. For this reason, in this section, we study if our solution as implemented in our prototype is feasible to solve the identified problems on a real-world system. First, we are interested in whether the developed toolchain works also in practice and not only in theory as shown in the previous sections. Second, we want to show that real-world SCK changes can be mitigated using our approach. Finally, it has to be possible to execute the toolchain with a reasonable time after a change of the SCK has occurred. In summary, we consider feasibility regarding the following two objectives:

O1 The feasibility study should show that the approach can be applied to an evolution scenario on a real-world system.
O2 The feasibility study should show that the execution of the steps which are automated by our research prototype is possible within a reasonable time.

In what follows, we first focus on investigating O1 by applying our tool prototype to a real-world system. Afterward, we investigate the execution times of the approach as considered in O2.

6.1 O1: Application to a Real-World Example

To apply our approach to a larger subject system, we chose the medical information system iTrust [24], which supports the electronic communication of all participants (patients, doctors, lab technicians, etc.) of the medical processing in a smart hospital. To date, iTrust comprises 39 use cases being implemented in about 120k lines of code distributed over more than 900 Java classes. Thus, the system is of a size that bares the risk of overlooking details when managing security without proper tool support.

In the remainder of this section, we investigate the feasibility of our approach regarding O1 on two concrete security evolution scenarios and show how semantic editing patterns detect the addition of threats and how they can be leveraged to co-evolve dependent artifacts of our subject system. In sum, we analyzed three kinds of context changes that have the potential to put security at stake: changes to privacy laws, trust in external libraries, and encryption algorithm exploits. For full details of the study, we refer to [7]. We investigated concrete contextual changes that lead to the following security vulnerabilities, all of them can be categorized in CWE catalog entries (the vulnerability of RC4 is categorized by CWE-327):

- CWE-284: Improper Access Control,
- CWE-311*: Missing Encryption of Sensitive Data,
- CWE-732*: Incorrect Permission Assignment for Critical Resource,
- CWE-327*: Use of a Broken or Risky Cryptographic Algorithm,
- CWE-20: Improper Input Validation, and
- CWE-502: Deserialization of Untrusted Data.

The entries with an asterisk are among the 2011 CWE/SANS 25 most dangerous software errors. In every case, we were able to show that changes leading to these vulnerabilities can be detected by analyzing evolution in ontologies, and our approach guided the user to manage the risk by recommending alternatives.

We selected the two application scenarios to cover different kinds of SCK evolution followed by different measures and effected parts of the system. First, we selected an evolution due to changes in technology. The reaction to this change causes a co-evolution of model-level security requirements

http://cwe.mitre.org/top25/

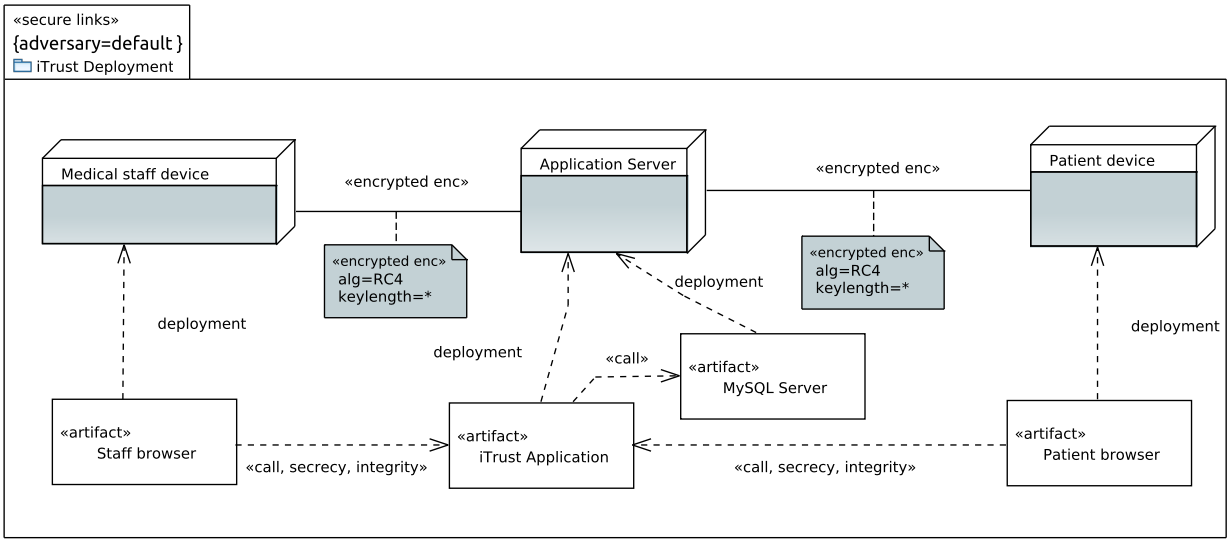

Fig. 11. Deployment diagram for iTrust with security annotation «secure links».

and check configurations that have to be followed by manual adaptions. Second, we selected an SCK evolution triggered by a legal change. Again, this change´ leads to a model co-evolution but no co-evolution of security check configurations. Instead, the co-evolution leads to existing security compliance checks in terms of security violation patterns to fail.

Security Vulnerability by Context Evolution due to Changes in Technology Knowledge

At first, we focus on leveraging the detected SCK changes for the adaptation of the system model using an SMR. As an example, we use a change in the knowledge about technology which causes security violations that have to be mitigated. These changes can be advances in technology but also the discovery of technological weaknesses such as a new attack on an encryption algorithm.

The evolution of security-relevant context knowledge indicated by the example shown in Fig. 2 represents a security vulnerability that has actually taken place, broke assumptions, and led to the necessity that software systems needed to be adapted. The cipher suite RC4 has been popular over a long period of time and was used in Transport Layer Security (TLS) to provide security for HTTP sessions. However, after the publication of an attack that could be carried out in merely 75 hours [81], the use of RC4 has been prohibited in a Request for Comments (RFC) by the Internet Engineering Task Force [65]. At that time, the estimation of TLS traffic relying on RC4 was approximately 30%. Moreover, numerous business applications communicating through HTTP-based REST-APIs were affected by this vulnerability. All of the affected web servers and distributed systems needed to be adapted.

Fig. 11 shows a deployment diagram of iTrust. Its concrete security requirements are specified using the UMLsec approach and its «secure links» annotation. Concrete encryption algorithms are annotated. The deployment reflects the typical setting of a distributed information system: There is an application server, executing the iTrust application as well as the database. Apart from that, there are two kinds of devices (medical staff and patients) to act as clients, both running a browser. The database runs on the same node as the iTrust application, and it thus does not require communication path encryption. On the contrary, the communication between the server and the clients shall ensure the integrity and the secrecy of the data transmitted over the communication paths.

At a point in time before the vulnerability of RC4 is discovered, according to the SCK, RC4 is selected as the encryption algorithm to secure the communication paths between the application server and the client devices' browsers. As illustrated in the previous sections, the vulnerability of RC4 is identified by detecting an occurrence of the semantic editing pattern "addEncryptionThreat" in the evolving SCK, being a trigger for the SMR shown in Listing 1.1. In subsequent steps, our approach first checks if a risky encryption algorithm is in use and, if so, guides the user to select appropriate, state-of-the-art alternatives (again, by using the SCK). This way, the iTrust system model could be successfully adapted in a (semi-)automated fashion.

After the adaptation of the system model, the configuration of the UMLsec secure links check and SonarQube rule S5547 have been co-evolved as shown in Listing 1.3 by adding RC4 to the list of vulnerable encryption algorithms in both checks. As expected, manual execution of the UMLsec secure link check finds no violation in the system model, which confirms that the adaptation performed in the SMR from Listing 1.1 was successful. SonarLint has been executed automatically. As in the original iTrust example, the transport encryption depends on the configuration of a Tomcat server iTrust is executed on, the original implementation contains no explicit use of any encryption. Also, we are not aware of any security checks for Tomcat configuration files. For this reason, we added the initialization of a cipher using RC4 which is detected by SonarLint as expected after the adaptation.

Security Vulnerability by Context Evolution due to new Laws

In this scenario of our feasibility study, we focus on the verification of the compliance of the implementation with the SCK after the SCK changed. As an example, we use a legal change, namely the release of the EU General Data Protection Regulation (GDPR) in which the European Parliament has adopted stricter regulations for the use of personal data [17].

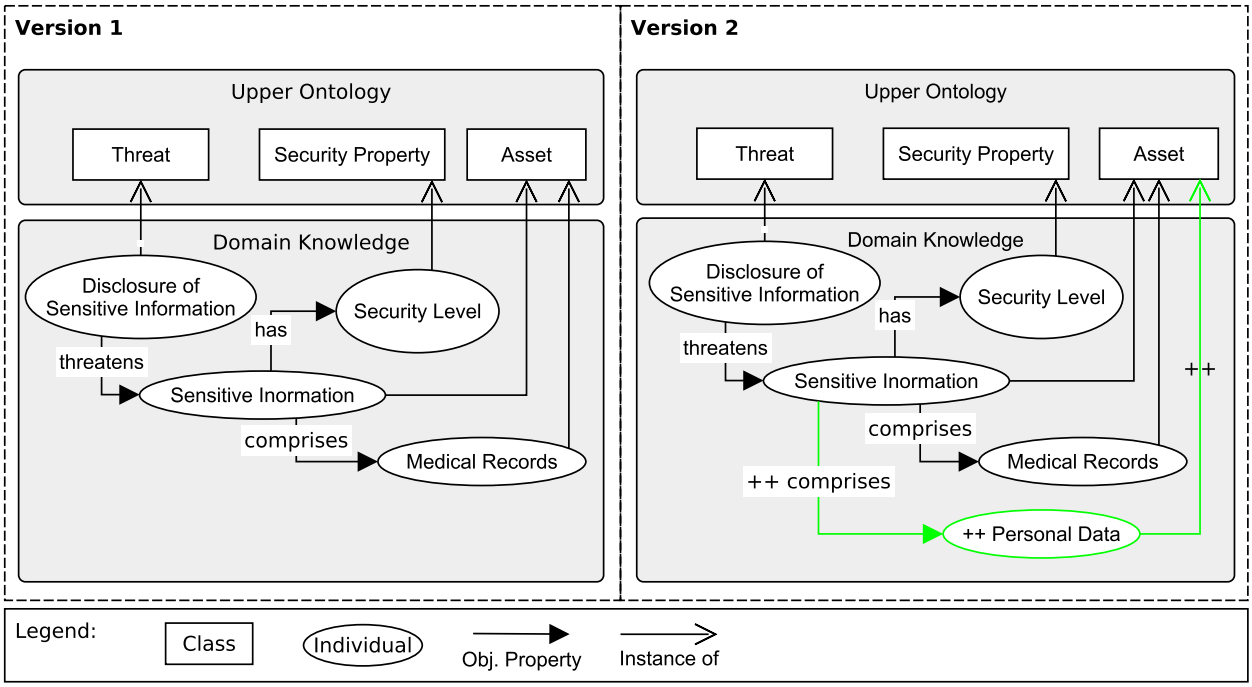

Fig. 12. SCK evolution due to the GDPR

For simplicity, in the considered scenario, we assume that the protection of personal data has not strictly been regulated by now. According to the ontology version 1 in Fig. 12, it is only regulated that medical records have to be treated as sensitive information and require explicit protection against their disclosure. This protection should be realized by assigning a security level to sensitive information and restricting access to this security level. Technically, this can be done by applying UMLsec secure dependency.

Fig. 13 shows an excerpt from the system model of the iTrust medical application [50] with applied UMLsec secure dependency stereotypes. On the right, the users of the system are shown. These can be doctors or patients. For both, a hashed version of the password and personal information like their home address is stored. On the left, we see different actions that can be performed in the system. These actions are realized as controls. One of these controls is the `DiagnosisControl` that allows users, depending on their rights, to read or edit medical diagnoses. To access this control, a user has to log in using the `LoginControl`. To check if a user can log in and determine her rights, the `LoginControl` accesses the `User` object captured as «call»-dependency (shown on the bottom of the diagram). Thereby, the `LoginControl` potentially has access to all information captured by the `User` class.

From a security perspective before the change of the regulation, only the password stored in the `User` class is sensitive and access has to be limited, e.g., by restricting access to this information to entities that are on a required security level. Accordingly, in Fig. 13, we put the information stored in the property `password` on the level of secrecy by adding the signature of this property to the `secrecy` tag of the «critical» stereotype on the class.

After a release of the GDPR adopting stricter regulations for the use of personal data, the SCK changes according to version 2 in Fig. 12. Now, sensitive information also comprises every kind of personal data.

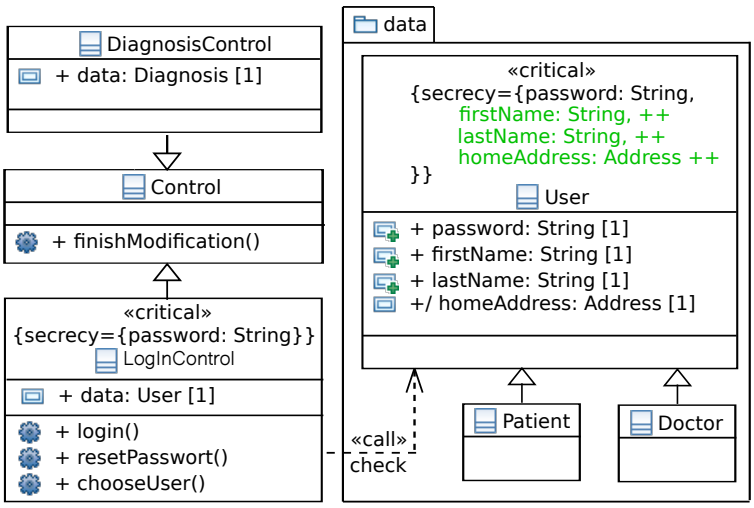

Fig. 13. Excerpt from the system model of iTrust after adaptation by an SMR.

As in the first scenario of technological changes, the detection of the SCK change triggers the execution of an SMR. This SMR is performing the following actions taking the new kind of sensitive information as input:

1. Detection of every instance of the new kind of sensitive information in the system model. In the example shown in Fig. 13, these are the properties `firstName`, `lastName`, and `homeAddress`.

2. Adding the detected instances to the security level of secrecy as shown in Fig. 13. The changes are highlighted in green and indicated by a ++.
3. Searching all incoming dependencies of the changed classes and creating proposals for the mitigation of the introduced violation of secure dependency. For the shown example the proposals are:
   (a) Deletion of the dependency called `check`.
   (b) Extension of the security level to the class `LoginControl`, the source of the dependency.
   (c) Extraction of sensitive information into a new class.
4. After mitigation has been performed by a developer, the security violation pattern shown in Fig. 9 is executed for detecting violations of the new security level on the implementation.

As the class `LoginControl` has to access the class `User` to verify the password of the user, the deletion of the dependency in step 3 (a) is not possible. Also, for the implementation of this class, the developers have already to consider the security level of the class `User`. For this reason, a security expert decides to extend the security level as proposed in option (b). For all other dependencies, she decides similarly.

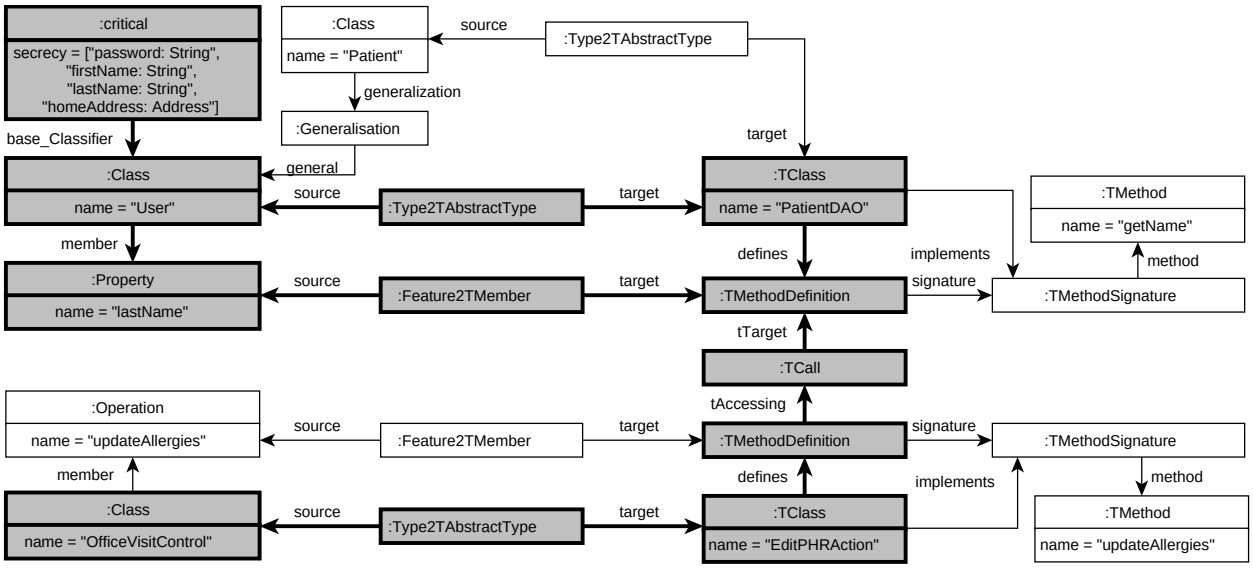

Fig. 14. Security violating match of a security violation pattern

Afterwards, matching the security violation pattern against the program model of the iTrust implementation detects the occurrence illustrated in Fig. 14, meaning that a concrete security violation has been detected on the implementation. The elements in Fig. 14 are arranged as in the security violation pattern shown in Fig. 9: On the left, we see the elements from the system model, the center shows the elements from the correspondence model, and right-hand part comprises the elements from the program model. The concrete violation is the access to a getter method of the property `lastName` by the method `updateAllergies` of the class `OfficeVisitControl`.

The corresponding source fragment of the violating access is shown in Listing 1.4. The detected security violation takes place in the implementation that allows doctors to edit health records as part of an office visit. To be more precise, in a method implementing the update of a patient's allergies. The concrete violation is the call to the method `getName` (line 6). This method is part of a `PatientDAO` that is a data access object for patient data. As no access to personal information has been planned in the system model, the whole editing of health records should be done over a

patient ID which is resolved at line 6 and violating the defined security level. Even more dangerous is that the only use of the personal information is as part of a status message (line 10) if an allergy has already been recorded which might even be written to log files. As mitigation of the security violation, the personal information has to be removed from this status message which makes access to personal information obsolete.

```
1  public class EditPHRAction extends PatientBaseAction {
2   private PatientDAO patientDAO;
3   ...
4   public String updateAllergies(long pid, String description){
5    ...
6    String patientName = patientDAO.getName(pid);
7    List<AllergyBean> allergies = allergyDAO.getAllergies(pid);
8    for (AllergyBean current : allergies){
9     if (current.getDescription().equals(bean.getDescription())) {
10     return "Allergy " + bean.getNDCode() + " - " + bean.getDescription() + " has
           already been added for " + patientName + ".";
11    }
12   }
13   ...
14  }
15  ...
16 }
```

Listing 1.4. Security violating code fragment of the iTrust implementation.

6.2 O2: Feasibility of the Execution Times

As there might be manual effort required in between the single automated steps, we investigated our tool prototype's feasibility regarding execution time separately for every step. Following objective O2, we have to investigate if our approach's automated steps can be executed within a reasonable time after an SCK change took place. Here, the essential executions are the detection of occurrences of semantic editing patterns, the reaction to the change, and the verification of consistency after the mitigation.

We measured all three executions on an Intel Core i5-6200U mobile CPU running at 2.30GHz with 8GB of memory. As the execution environment, we used Ubuntu 20.04LTS and OpenJDK 14.

We used Protégé to analyze the complexity of the ontologies representing the SCK in our study. The highest complexity measure was identified as $\mathcal{ALU}$ (attributive language with concept union). However, the complexity regarding the description logic is not relevant for SiLift, which works on a syntactical level rather than inferring logical relations. Regarding the runtime performance of our tooling, detecting semantic editing patterns took at most 17 seconds for the considered evolution scenarios. The time for a complete run of S^2EC^2O, including the generation of co-evolution proposals, took no longer than 30 seconds. Given that we are experimenting with a research prototype that is not optimized for performance, we believe that the obtained results demonstrate our approach's applicability in a real-world setting.

Regarding the propagation of the changes from the model-level to the implementation, usually, manual effort is included for resolving conflicts. However, this manual effort can be reduced by using our approach for propagating changes. For this reason, we measured the time needed for an initial propagation of changes into the source code but not the time needed for manual changes afterward. Thereby, we considered the following changes in the UML models:

- Deletion of a security-violating dependency.
- Separating sensitive information from public information by adding a new property in a class.
- Extraction of security-critical operations into a new class.

– Moving an operation to a different class to group security-critical functionality.

The propagation of a change from the UML models into the implementation took 50 seconds on average and 51.7 seconds in the worst case. Currently, we use a non-incremental code-generator that takes most of the time (98.5%). For the pure propagation of the changes from the UML models into an implementation model, from which code is generated, only 0.75 seconds are needed. Thereby, we did not notice any significant difference between the changes.

Regarding the application of security violation patterns, for the verification of UMLsec secure dependency on the implementation level, two violation patterns are required for each security level. First, for the client not being annotated with the required security property, as shown in Fig. 9. Second, for the opposite direction, the supplier not being annotated with the required security property. We applied these two patterns after two kinds of changes. First, changes that resulted in a security violation and, second, changes that did not affect the security compliance. Here, we did not change the structure of the implementation but edited the security annotations to introduce a security violation. To quantify the benefit of incremental security violation patterns, we executed the security violation patterns incrementally and in terms of a complete security compliance check. For a security-compliant implementation, the incremental security violation patterns' execution took on average 235 seconds, while the complete security compliance check did not terminate within 60 minutes. When investigating a change that led to a security violation, the execution time of the incremental security violation patterns increased to 440 seconds on average. The full security compliance check did not terminate within a reasonable time.

Reconsidering objective O2, our tool prototype shows a runtime sufficient for automatic execution, e.g., as part of a continuous integration pipeline. While there is still potential for optimizing the prototype's implementation, e.g., an incremental code-generation, we already achieved feasible execution times on a consumer computer. Furthermore, we assume a continuous integration pipeline to be executed on a server with relatively high computing power.

In this feasibility study, we investigated the feasibility of our proposed solution regarding two objectives. First, we showed that, by using semantic editing patterns, the detection of SCK changes can be leveraged using SMRs to bring the design-time models into a security compliant state. Also, we showed how these SMRs can be used for adapting the configurations of security checks and the application of security violation patterns to verify the compliance of the implementation with the adapted design-time models. Thereby, we demonstrated the realizability of our proposed toolchain. Furthermore, we showcased the applicability to practical problems by using the iTrust electronics health records application as the subject system (O1). Finally, we showed that all steps of our approach can be executed within minutes. Here, the security compliance check between the implementation and the design-time models had the longest execution time but is still within a time suitable for a continuous integration pipeline. To sum up, this also results in feasibility from the execution-time perspective (O2).

7 Discussion

In this section, we discuss implications for successfully applying our approach to the maintenance of a software system in the context of security knowledge changes, its current limitations with respect to providing formal guarantees, as well as possible threats to the validity of our demonstration of feasibility.

7.1 Assumptions and Implications

Our article's main assumption is that our approach will be applied to systems developed using a model-driven development approach. The presence of systems models as a prerequisite for using our approach might lead to limitations of its applicability. In this regard, we consider two factors that might limit the practical applicability of our approach.

First, the required model-driven development approach might not be applicable in the context of agile project development. However, considering the legal requirements in many security-critical domains, standards such as the ISO/EC 62304 for the development of medical device software [27], the development and maintenance of the artifacts required by our approach is a prerequisite in most cases. Such standards do not require specific development processes as long as the required artifacts are provided. The same applies to our approach. For example, Rumpe demonstrates how systems can be developed agile using model-based development [69].

Second, our approach may not be able to reimburse the costs incurred to create the required models. However, as our approach's main scope is large software systems developed for strongly regulated areas, these artifacts are likely required by standards the software systems have to comply with. In this case, there are no additional costs for using our approach. For all other systems, the application of our approach might require additional effort to create those artifacts. From this side, the only reason standing against our approach might be that, up to some size, manually keeping the security context knowledge up to date and manually selecting measures in case of changes might be more cost-efficient than using our approach for (semi-)automating these activities. Here, the automated reverse-engineering of UML models can be a cost-efficient solution to apply our approach. Either way, if developers want to apply our approach, they have to implement model-driven development practices. While there are approaches to reverse engineer system models from the implementation of a software system [40] or to restore traces between design models and the implementation [62], in this article, we assume that the software system has been developed from the very beginning using UML system models, S^2EC^2O, and GRaViTY.

Furthermore, we assume a security expert to use the upper ontology of Gärtner et al. for the specification of the SCK. However, other upper ontologies could also be used for specifying the SCK, e.g., the Unified Foundational Ontology (UFO) of Guizzardi et al. [22, 23] or the Basic Formal Ontology (BFO) of Smith et al. [2]. As the intention of the SCK is to only capture security knowledge, we decided against such a general-purpose upper ontology [77]. Besides our security ontology, there are also more ontologies for capturing security knowledge. For example, Souag et al. [77] present a core and generic security ontology we could use. Again, this is possible in principle, but the structuring according to the changeability of the security concepts fits our approach better than the structure of Souang et al. into the dimensions of organization, risk, and treatments.

Besides the fact that our assumptions might limit the applicability of our approach, when applying our approach, developers implicitly apply other best practices for secure systems development.

First, our approach requires developers to explicitly specify knowledge about the security context using a formal approach. In case of changes in the formalized security context knowledge, these changes are automatically detected. However, making security context knowledge explicit, might also contribute to a deeper security-related understanding of the software system.

Second, by using the proposed approach, developers would implicitly follow the principle of security by design. Security is considered from the very beginning, and our approach enforces the reuse and compliance with security properties.

Furthermore, our approach is compliant with multiple popular concepts of secure software development. The discussed automated security compliance checks in case of changes are also one main characteristic of SecDevOps. If our approach is deployed within a continuous integration framework,

it can be integrated into SecDevOps, complementary to other vulnerability detection techniques such as penetration testing or static code analysis. In this case, our approach adds a new level of automatization beyond local security checks on single artifacts. Our approach will be executed together with other automated security tests that focus on fine-grained local security properties while our approach contributes the tracing and compliance checks between security properties. As shown in our feasibility study, considering such an integration, our approach is even capable to update the configurations of the other security checks.

7.2 Formal Guarantees

The framework presented in this article can be considered as a recommender system which, if configured properly, generates various kinds of proposals of how to adapt a system in response to changing SCK. Configuration takes place in a rule-based manner, notably rule-based specifications of semantic editing patterns, SMRs, and security violation patterns. While this provides a dedicated method of how to deal with evolution of SCK in a systematic manner, the specification of these rules is left to the discretion of a security expert. To date, there is no automated way of assessing the quality of these rules, in particular, for checking their soundness and completeness: Given a configuration of our framework, will a generated co-evolution proposal indeed be a valid solution for the kind of SCK evolution it attempts to mitigate (soundness), and can we guarantee to generate valid co-evolution proposals for all possible kinds of changes to the SCK (completeness)?

While we aim at an extension of our framework by such formal proofs and assurances in the future, this is out of the scope of this article which focuses on demonstrating the feasibility of our approach. As for completeness, a first property which needs to be fulfilled is that, given a set of rule-based specifications of semantic editing patterns, all possible changes to the SCK can be expressed using editing patterns available in this set. A starting point for this could be the approach and tool suite presented in [37, 66], which guarantees to generate a complete set of basic editing patterns for a given type of models, such as our ontological representation of SCK.

7.3 Threats to Validity

The validity of our demonstration of feasibility might be subject to some threats discussed in what follows. Thereby, we differentiate between internal and external threats.

An internal threat to validity is that all experiments have been performed by ourselves, precisely knowing how our tool prototype works. Nevertheless, this still shows that our prototypical tool is suitable to solve the problem. However, this might not be the way external users want to use the tool.

Also, the runtime-measurements are subject to an internal threat to validity. The runtime performance of the automated tasks supported by our tooling and carried out in terms of our feasibility study has been evaluated in a non-closed system. Thus, we cannot rule out other computational tasks or processes we were unaware of to impact our measurements negatively. Moreover, performance measurements could be biased by just-in-time compilation overheads of the Java runtime. However, we did not aim for high-precision micro benchmarking in terms of our feasibility study but to report about the maximal runtimes that we could observe in terms of our study to showcase the applicability of our tooling in a real-world setting.

The selection of iTrust as a subject system to demonstrate our approach's feasibility gives rise to an external threat. We cannot guarantee that iTrust is representative for all other software systems our approach could be applied to.

Also, there is the threat that the used ontology for capturing the SCK does not represent security ontologies as security experts from industry would have defined them. To reduce this risk, we based our security ontology on an upper ontology from existing works [11, 20]. Furthermore, these works are based on best practices identified in systematic literature reviews.

There is a threat that the considered changes in our feasibility study do not represent all possible kinds of real-world changes. To lower the risk for our observations being not generalizable, we cover notable changes with different feasibility study effects. However, our feasibility study still shows the practical applicability of our approach, even if the outcomes are not generalizable. In future work, we plan to mitigate this threat by studying the adaptation of further systems based on real-world security context knowledge changes.

8 Related Work

In this section, we discuss related works. The section is structured according to the main areas discussed in this article. At first, we discuss related works for capturing the evolution of knowledge stored in ontologies. Afterward, we discuss approaches for tracing system properties between different artifacts. Last, we discuss related approaches for the specification and verification of software security.

8.1 Ontology Evolution

Ontology evolution as the process to adapt and change ontologies and dependent artifacts in a consistent manner has been studied in the literature for many years, surveys can be found in [16, 38, 83, 85]. Various techniques have been proposed to support dedicated tasks of this process. The most closely related approaches to ours deal with the management of changes. The first approach which distinguishes changes on an atomic, composite and complex level has been presented by Maedche et al. [49], where ontological changes are recorded through a dedicated tool known as OntoLogging. A configurable approach to detect ontological changes has been presented by Plessers et al. [63, 64]. They introduce a change definition language which allows tool users to define sets of change patterns. The detection is realized by interpreting change definitions as temporal queries on a persitently managed ontology version log. Thus, in contrast to ours, their approach is still bound to the restriction of logging ontology change information through a dedicated tool environment. The same limitation applies to approaches which have been later proposed by Djedidi et al. [15] and Javed et al. [29,30]. The approach proposed by Papavassiliou et al. [54,55] gets rid of this limitation, however, only supports a fixed set of semantic editing patterns which cannot be extended by customly defined change patterns. In sum, none of the proposed approaches fully meets our specific requirements described in the introduction.

8.2 Traceability between System Specification Artifacts

Winkler and Pilgrim performed a survey on traceability in requirements engineering and model-driven development [84]. While traceability in both domains has much in common, they are still separated. In requirements engineering traceability mainly means to follow requirements throughout the development process and in model-driven engineering traceability mainly means to explicitly create trace links between corresponding artifacts. While we consider security requirements and follow them in case of security context knowledge changes, in our work tracing mainly takes place in the area of model-driven development.

For the tracing of the system specification, Atkinson et al. propose the use of a single underlying model (SUM) [4]. Suitable views according to the current task are extracted from this model. The system model considered in our approach can be specified using standard UML. In contrast to this, SUM is based on a modified UML meta-model. While in the SUM approach the tracing down to the implementation is not supported, VITRUVIUS provides a realization of the SUM approach that also supports Java source code [43]. The Codeling tool of Konersmann [42] provides integration of architecture model information with the program code. While we maintain a correspondence model between the implementation and the system model, in Codeling these correspondences are written to the code and the system model is extracted from the code if required. Both SUM and Codeling enrich the implementation with design-time information about the system model with might decrease the readability of the implementation. In contrast to this, we leverage a correspondence model that is invisible to the developers of the system for synchronization and security compliance checks. Also, none of the discussed approaches has been applied to the tracing of software security specifications and their enforcement.

Explicitly designed for the tracing of the security structure and properties is the SecSTAR approach of Fang et al. [18]. SecSTAR monitors the system execution and traces security-related information throughout the system. Based on the recorded diagrams for security analysis are generated. These diagrams can be seen as part of the system model considered by us. Nhlabatsi et al. present an approach to monitor assumptions about security requirements, such as the location of devices, at run time [51]. Thereby, security requirements are specified upfront and might be subject to knowledge changes as considered by us.

8.3 Software Security

While many works are focusing on specific security properties, only a few of them take the whole system into account. Besides the UMLsec approach used by us, there are various approaches for considering security at the model-level. An overview of model-based security analysis can be found in [46]. Newer model-based approaches comprise SecDFDs [80] and SecBPMN [70]. On the implementation level, static code analysis is usually used to detect security issues during software implementation [13]. Also, many approaches locally analyze calls to critical APIs [19, 44] and whether the chosen parameters have been selected securely, e.g., for a crypto API, or to detect leaks of secret data using secure data flow analysis [3, 73, 82]. While these approaches are important for the development of secure systems, one of their main problems is the need for precise information where encryption is required or where secret data is stored. Here, we showed how existing checks can be integrated to verify changed security properties on different levels of a system.

Certifications of software, e.g., according to Common Criteria (CC) [28], play an important role in ensuring the security of systems. Usually, design specifications and test results have to be provided for the certification. Which artifacts have to be provided depends on the assurance level of CC or other domain-specific standards, e.g., ISO/IEC 62304 for medical device software [27]. The certification is usually performed manually and incremental re-certification or revocation in case of changed SCK is currently not supported. Anisetti et al. present a security certification scheme for evolving services [1]. While we focus on security compliance in case of changes in the SCK, they focus on changes in the implementation of the service. Here, our incremental compliance checks could also be applied.

9 Conclusion

We presented an approach and prototypical implementation to detect occurrences of semantic editing patterns in evolving OWL ontologies and to trigger actions for ensuring the security compliance

of the system with the evolved security knowledge. Compared to existing approaches that aim at the semantic lifting of low-level ontological changes, the most distinguishing features of our approach are that it does not rely on tool-specific information such as persistently managed edit logs and that the set of supported semantic editing patterns can be easily customized and extended.

We developed our approach driven by four research questions. First, we propose rule-based semantic editing patterns for detecting changes in SCK maintained in ontologies. Second, we propose the specification of SMRs corresponding with the semantic editing patterns for the generation of model co-evolution proposals. Also, SMRs should be used to co-evolve security check configurations as required in our third research question. Finally, for security compliance checks between the (adapted) models and the implementation, we propose incremental rule-based security violation patterns.

We motivated the need for these features in the context of a model-based security engineering approach which incorporates so-called security context knowledge into system design and evolution. However, application scenarios leading to similar requirements can be found in other areas as well. In software engineering, for example, knowledge management is increasingly considered as a key aspect having a strong impact on the maintainability of various quality attributes of software systems. Besides traditional development artifacts, manifold information such as rationale for development decisions may need to be gathered. Whenever such knowledge is documented and exploited through the use of ontologies that may undergo uncontrolled and ad-hoc change over time, we believe that our approach provides a suitable basis for successfully managing the respective ontology evolution.

We demonstrated the applicability and usefulness of the developed techniques in a feasibility study on a medical information system. Thereby, we focused on two aspects of feasibility. First, we considered the application to real-world problems and, second, whether execution times are acceptable. As part of this study, we showcased how occurrences of detected semantic editing patterns can be directly exploited by model-driven engineering tools for to enforce the security compliance of a system. This study includes the system model's co-evolution with the changing security knowledge and co-evolution propagation into the implementation. To verify the security of the system, we introduced the co-evolution of the configurations of state of the art security checks to consider the evolved security context knowledge. We introduced how security compliance checks leveraging security specifications on the system model can be specified using security violation patterns. For these security violation patterns, we demonstrated how to apply these incrementally to detect security violations on the implementation level in case of changes in the system model.

To sum up, semantic editing patterns for detecting changes in security context knowledge leveraged by SMRs for the adaption of design-time models and security check configurations combined with incremental security violation patterns are an effective technique for bringing a software system into a security-compliant state after SCK evolution.

‘

Acknowledgments

This research is partly founded by the European Commission as part of the Horizon 2020 research and innovation programme under grant agreement No 871493 (DataPorts).

References

1. Anisetti, M., Ardagna, C.A., Damiani, E.: A Low-Cost Security Certification Scheme for Evolving Services. In: ICWS. pp. 122–129 (2012)
2. Arp, R., Smith, B., Spear, A.: Building Ontologies With Basic Formal Ontology. MIT Press (2015)

3. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In: PLDI (2014)
4. Atkinson, C., Stoll, D., Bostan, P.: Orthographic Software Modeling: A Practical Approach to View-based Development. In: ENASE, pp. 206–219. Springer (2009)
5. Baader, F., Horrocks, I., Sattler, U.: Description Logics. In: Handbook on Ontologies, pp. 3–28. Springer (2004)
6. Bruegge, B., Dutoit, A.H., Wolf, T.: Sysiphus: Enabling informal collaboration in global software development. In: 2006 IEEE International Conference on Global Software Engineering (ICGSE'06). pp. 139–148. IEEE (2006)
7. Bürger, J.: Recovering Security in Model-based Software Engineering by Context-driven Co-evolution. Ph.D. thesis, University of Koblenz-Landau (2019)
8. Bürger, J., Jürjens, J., Ruhroth, T., Gärtner, S., Schneider, K.: Model-based Security Engineering: Managed Co-Evolution of Security Knowledge and Software Models. In: Foundations of Security Analysis and Design VII, pp. 34–53. Springer (2013)
9. Bürger, J., Jürjens, J., Wenzel, S.: Restoring Security of Evolving Software Models Using Graph Transformation. STTT **17**(3) (2015)
10. Bürger, J., Kehrer, T., Jürjens, J.: Ontology Evolution in the Context of Model-Based Secure Software Engineering. In: RCIS. pp. 437–454. Springer (2020)
11. Bürger, J., Strüber, D., Gärtner, S., Ruhroth, T., Jürjens, J., Schneider, K.: A Framework for Semi-automated Co-evolution of Security Knowledge and System Models. JSS **139**, 142–160 (2018)
12. Campbell, G.A., Papapetrou, P.P.: SonarQube in Action. Manning Publications Co., USA (2013)
13. Damiani, E., Ardagna, C.A., El Ioini, N.: Open Source Systems Security, chap. Formal Methods for Software Verification, pp. 1–26. Springer (2009)
14. eMoflon Developer Team: eMoflon – A tool for building tools, `http://www.emoflon.org/`
15. Djedidi, R., Aufaure, M.A.: Ontology change management. In: I-SEMANTICS. pp. 611–621 (2009)
16. Djedidi, R., Aufaure, M.A.: Ontology evolution: State of the art and future directions. In: Ontology theory, management and design: advanced tools and models. IGI Global (2010)
17. EU Parliament: Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). Official Journal of the European Union **L119**, 1–88 (2016), `http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC`
18. Fang, W., Miller, B.P., Kupsch, J.A.: Automated tracing and visualization of software security structure and properties. In: VizSec. pp. 9–16 (2012)
19. Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., Tao, L.: A Static Analysis Framework For Detecting SQL Injection Vulnerabilities. In: COMPSAC (2007)
20. Gärtner, S., Ruhroth, T., Bürger, J., Schneider, K., Jürjens, J.: Maintaining requirements for long-living software systems by incorporating security knowledge. In: RE. IEEE (2014)
21. Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge acquisition **5**(2), 199–220 (1993)
22. Guizzardi, G.: Ontological Foundations for Structural Conceptual Models. Ph.D. thesis, University of Twente (2005)
23. Guizzardi, G., Wagner, G.: A Unified Foundational Ontology and some Applications of it in Business Modeling. In: EMOI-INTEROP. pp. 129–143 (2004). https://doi.org/10.4018/978-1-59140-339-5.ch013
24. Heckman, S., Stolee, K., Parnin, C.: 10+ Years of Teaching Software Engineering with iTrust: The Good, the Bad, and the Ugly. In: Software Engineering Education and Training. pp. 1–4. IEEE (2018)
25. Hesse, T.M., Gärtner, S., Roehm, T., Paech, B., Schneider, K., Bruegge, B.: Semiautomatic Security Requirements Engineering and Evolution using Decision Documentation, Heuristics, and User Monitoring. In: ESPRE. pp. 1–6. IEEE (2014)
26. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS Q. **28**(1), 75–105 (2004), `http://dl.acm.org/citation.cfm?id=2017212.2017217`
27. IInternational Organization for Standardization (ISO): Medical Device Software — Software Life Cycle Processes. International Standard IEC 62304:2006 (2007)
28. Intl. Standardization Organization (ISO): ISO 15408:2007 Common Criteria for information technology security evaluation, version 3.1, revision 2, CCMB-2007-09-001, CCMB-2007-09-002 and CCMB-2007-09-003 (2007)
29. Javed, M., Abgaz, Y.M., Pahl, C.: A pattern-based framework of change operators for ontology evolution. In: OTM Confederated Intl. Conferences" On the Move to Meaningful Internet Systems". pp. 544–553. Springer (2009)
30. Javed, M., Abgaz, Y.M., Pahl, C.: Ontology change management and identification of change patterns. Journal on Data Semantics **2**(2-3), 119–143 (2013)
31. Jan Jürjens: Secure Systems Development with UML. Springer (2005)

32. Kehrer, T.: Calculation and propagation of model changes based on user-level edit operations: a foundation for version and variant management in model-driven engineering. Ph.D. thesis, University of Siegen (2015), http://dokumentix.ub.uni-siegen.de/opus/volltexte/2015/963/
33. Kehrer, T., Alshanqiti, A., Heckel, R.: Automatic inference of rule-based specifications of complex in-place model transformations. In: Intl. Conf. on Theory and Practice of Model Transformations. pp. 92–107. Springer (2017)
34. Kehrer, T., Kelter, U., Ohrndorf, M., Sollbach, T.: Understanding model evolution through semantically lifting model differences with SiLift. In: Software Maintenance (2012)
35. Kehrer, T., Kelter, U., Pietsch, P., Schmidt, M.: Adaptability of model comparison tools. In: ASE. pp. 306–309. IEEE (2012)
36. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: ASE (2011)
37. Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically deriving the specification of model editing operations from meta-models. In: International Conference on Theory and Practice of Model Transformations. pp. 173–188. Springer (2016)
38. Khattak, A.M., Batool, R., Pervez, Z., Khan, A.M., Lee, S.: Ontology evolution and challenges. Journal of Information Science and Engineering **29**(5) (2013)
39. Kögel, S., Tichy, M., Chakraborty, A., Fay, A., Vogel-Heuser, B., Haubeck, C., Taentzer, G., Kehrer, T., Ladiges, J., Grunske, L., Ulbrich, M., Bougouffa, S., Getir, S., Cha, S., Kelter, U., Lamersdorf, W., Busch, K., Heinrich, R., Koch, S.: Learning from evolution for evolution. In: Managed Software Evolution, pp. 255–308. Springer (2019)
40. Kollmann, R., Selonen, P., Stroulia, E., Systa, T., Zundorf, A.: A Study on the Current State of the Art in Tool-supported UML-based Static Reverse Engineering. In: WCRE. pp. 22–32 (2002). https://doi.org/10.1109/WCRE.2002.1173061
41. Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different models for model matching: An analysis of approaches to support model differencing. In: 2009 ICSE Workshop on Comparison and Versioning of Software Models. pp. 1–6. IEEE (2009)
42. Konersmann, M.: Explicitly Integrated Architecture-an Approach for Integrating Software Architecture Model Information with Program Code. Ph.D. thesis, University of Duisburg-Essen (2018)
43. Kramer, M.E., Burger, E., Langhammer, M.: View-centric Engineering with Synchronized Heterogeneous Models. In: VAO (2013)
44. Krüger, S., Nadi, S., Reif, M., Ali, K., Mezini, M., Bodden, E., Göpfert, F., Günther, F., Weinert, C., Demmler, D., Kamath, R.: CogniCrypt: Supporting Developers in using Cryptography. In: ASE (2017)
45. Küster, J.M., Gerth, C., Förster, A., Engels, G.: Detecting and resolving process model differences in the absence of a change log. In: International Conference on Business Process Management. pp. 244–260. Springer (2008)
46. Lano, K., Clark, D., Androutsopoulos, K.: Safety and Security Analysis of Object-Oriented Models. In: SAFECOMP (2002)
47. Leblebici, E., Anjorin, A., Schürr, A.: Developing eMoflon with eMoflon. In: ICMT. pp. 138–145 (2014)
48. Lippe, E., Van Oosterom, N.: Operation-based merging. In: Proceedings of the fifth ACM SIGSOFT Symposium on Software Development Environments. pp. 78–87 (1992)
49. Maedche, A., Motik, B., Stojanovic, L., Studer, R., Volz, R.: Managing multiple ontologies and ontology evolution in ontologging. In: Intl. Conf. on Intelligent Information Processing. Springer (2002)
50. Meneely, A., Smith, B., Williams, L.: iTrust electronic health care system case study. In: Software and Systems Traceability, pp. 425–438. Springer (2012), https://github.com/ncsu-csc326/iTrust, (accessed Nov 15th, 2020)
51. Nhlabatsi, A., Yu, Y., Zisman, A., Tun, T., Khan, N., Bandara, A., Nuseibeh, B.: Managing Security Control Assumptions using Causal Traceability. In: SST (2015)
52. Otero-Cerdeira, L., Rodríguez-Martínez, F.J., Gómez-Rodríguez, A.: Ontology matching: A literature review. Expert Systems with Applications **42**(2), 949–971 (2015)
53. OWL Working Group, W.: OWL 2 Web Ontology Language: Document Overview. W3C Recommendation (2009)
54. Papavassiliou, V., Flouris, G., Fundulaki, I., Kotzinos, D., Christophides, V.: On detecting high-level changes in rdf/s kbs. In: Intl. Semantic Web Conference. pp. 473–488. Springer (2009)
55. Papavassiliou, V., Flouris, G., Fundulaki, I., Kotzinos, D., Christophides, V.: High-level change detection in rdf (s) kbs. TODS **38**(1), 1 (2013)
56. Peffers, K., Tuunanen, T., Gengler, C.E., Rossi, M., Hui, W., Virtanen, V., Bragge, J.: The Design Science Research Process: A Model for Producing and Presenting Information Systems Research. In: Design Science Research in Information Systems and Technology. pp. 83–106 (2006)
57. Peldszus, et al.: GRaViTY Website, http://gravity-tool.org, (accessed: Nov 15th, 2020)
58. Peldszus, S.: Model-driven Development of Evolving Secure Software Systems. In: EMLS (2020)
59. Peldszus, S., Kulcsár, G., Lochau, M.: A Solution to the Java Refactoring Case Study using eMoflon. In: TTC (2015)
60. Peldszus, S., Kulcsár, G., Lochau, M., Schulze, S.: Incremental co-evolution of Java programs based on bidirectional graph transformation. In: PPPJ. pp. 138–151 (2015)

61. Peldszus, S., Kulcsár, G., Lochau, M., Schulze, S.: Continuous Detection of Design Flaws in Evolving Object-Oriented Programs using Incremental Multi-pattern Matching. In: ASE (2016)
62. Peldszus, S., Tuma, K., Strüber, D., Jürjens, J., Scandariato, R.: Secure Data-Flow Compliance Checks between Models and Code based on Automated Mappings. In: MODELS (2019)
63. Plessers, P., De Troyer, O.: Ontology change detection using a version log. In: Intl. Semantic Web Conference. pp. 578–592. Springer (2005)
64. Plessers, P., De Troyer, O., Casteleyn, S.: Understanding ontology evolution: A change detection approach. Web Semantics: Science, Services and Agents on the World Wide Web **5**(1), 39–49 (2007)
65. Popov, A.: RFC 7465: Prohibiting RC4 cipher suite (2015), `https://tools.ietf.org/html/rfc7465`, (accessed: Nov 15th, 2020)
66. Rindt, M., Kehrer, T., Kelter, U.: Automatic generation of consistency-preserving edit operations for mde tools. Demos@ MoDELS **14** (2014)
67. Ruhroth, T., Gärtner, S., Bürger, J., Jürjens, J., Schneider, K.: Versioning and evolution requirements for model-based system development. In: Intl. Workshop on Comparison and Versioning of Software Models (2014)
68. Ruland, S., Kulcsár, G., Leblebici, E., Peldszus, S., Lochau, M.: Controlling the Attack Surface of Object-Oriented Refactorings. In: FASE (2018)
69. Rumpe, B.: Agile Modeling with UML: Code Generation, Testing, Refactoring. Springer (2017)
70. Salnitri, M., Dalpiaz, F., Giorgini, P.: Modeling and Verifying Security Policies in Business Processes. In: BPMDS (2014)
71. Schneider, C., Zündorf, A., Niere, J.: Coobra-a small step for development tools to collaborative environments. In: Workshop on Directions in Software Engineering Environments (2004)
72. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: WG. pp. 151–163 (1994)
73. Schwartz, E.J., Avgerinos, T., Brumley, D.: All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In: SP (2010)
74. Sonar Source SA: Java Implementation of the SonarQube Strong Cipher Algorithm Check (S5547) (2020), `https://github.com/SonarSource/sonar-java/blob/edbb43cfab0b54cd512102b112e21e87c1d25858/java-checks/src/main/java/org/sonar/java/checks/StrongCipherAlgorithmCheck.java`, (accessed: Nov 15th, 2020)
75. SonarSource S.A: SonarLint Website, (accessed: Nov 15th, 2020)
76. SonarSource SA: SonarQube Website, `https://www.sonarqube.org/`, (accessed: Nov 15th, 2020)
77. Souag, A., Salinesi, C., Mazo, R., Comyn-Wattiau, I.: A Security Ontology for Security Requirements Elicitation. In: ESSoS. pp. 157–177 (2005). https://doi.org/10.1007/978-3-319-15618-7_13
78. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education (2008)
79. Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M.: Henshin: A Usability-focused Framework for Emf Model Transformation Development. In: ICGT. pp. 196–208. Springer (2017)
80. Tuma, K., Scandariato, R., Balliu, M.: Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis. In: ICSA (2019)
81. Vanhoef, M., Piessens, F.: All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In: USENIX Security Symposium. pp. 97–112 (2015)
82. Wang, X., Qin, X., Hosseini, M.B., Slavin, R., Breaux, T.D., Niu, J.: GUILeak: Tracing Privacy Policy Claims on User Input Data for Android Applications. In: ICSE (2018)
83. Wardhana, H., Ashari, A., Sari, A.K.: Review of ontology evolution process. Journal of Computer Applications **45** (2018)
84. Winkler, S., von Pilgrim, J.: A Survey of Traceability in Requirements Engineering and Model-driven Development. SoSyM **9**, 529–565 (2010)
85. Zablith, F., Antoniou, G., d'Aquin, M., Flouris, G., Kondylakis, H., Motta, E., Plexousakis, D., Sabou, M.: Ontology evolution: A process-centric survey. The knowledge engineering review **30**(1), 45–75 (2015)