Security Compliance in Model-Driven Software Development

Sven Peldszus

Abstract To ensure the security of a software system, it is vital to keep up with changing security precautions, attacks, and mitigations. Although model-based development enables addressing security already at design-time, design models are often inconsistent with the implementation or among themselves. Such inconsistencies hinder the effective realization and verification of secure software systems. In addition, variants of software systems are another burden to developing secure systems. Vulnerabilities must be identified and fixed on all variants or else attackers could be well-guided in attacking unfixed variants. To ensure security in this context, in the thesis (Peldszus, Security Compliance in Model-driven Development of Software Systems in Presence of Long-Term Evolution and Variants. Springer, Berlin; 2022), we present GRaViTY, an approach that allows security experts to specify security requirements on the most suitable system representation. To preserve security, based on continuous automated change propagation, GRaViTY automatically checks all system representations against these security requirements. To systematically improve the object-oriented design of a software-intensive system, GRaViTY provides security-preserving refactorings. For both continuous security compliance checks and refactorings, we show the application to variant-rich software systems. To support legacy systems, GRaViTY allows to automatically reverse-engineer variability-aware UML models and semi-automatically map existing design models to the implementation. Besides evaluations of the individual contributions, we demonstrate applicability of the approach in two real-world case studies, the iTrust electronics health records system and the Eclipse Secure Storage. This book chapter provides a summary of the thesis, focusing on the addressed problems, identified and answered research questions, the general solution, and its application of it to two case studies. For details on the

The author obtained his Doctorate from the University of Koblenz-Landau.

S. Peldszus (✉)
Ruhr University Bochum, Bochum, Germany
e-mail: sven.peldszus@rub.de

E. Bodden et al. (eds.), *Ernst Denert Award for Software Engineering 2022*,
https://doi.org/10.1007/978-3-031-44412-8_4

individual solutions, please refer to the thesis and the corresponding publications referenced in this book chapter.

1 Introduction

Software has become a considerable part of today's life, and we rely on it to be safe and secure and respect our privacy. Even in critical domains like healthcare, modern medical imaging devices are exposed to the Internet. Furthermore, software systems tend to be used on a long-term basis in environments prone to changes, and at the same time successors of a software system are developed rapidly. A successor is often a variant of the previous system as significant parts are reused. Besides, multiple variants of a software system can exist at the same time. In all cases, to ensure the security of a software-intensive system, all changes, e.g., due to maintenance or extension, have to be continuously reflected in the whole software system, including all variants. These circumstances result in significant challenges regarding the security of evolving software systems and their variants.

Traditionally, manufacturers ensure security by implementing security standards such as the *Common Criteria*. Currently, such security standards focus more on the processes of how the software is developed than the concrete artifacts. Concerning today's short product cycles and the vast amount of product versions, certifying each product manually is impossible. One missing key to improve security is integrated tool support covering all software development phases. Furthermore, it can already support avoiding security violations during implementation.

A widely accepted development approach is *Model-Driven Development* (MDD) [3, 10] that allows planning well-structured software systems. To support evolution, MDD can include systematic variation points for future extensions or variants. Furthermore, it enables us to address security in the early phases of the software design using approaches such as UMLsec [14] or SecDFD [43]. Design models are annotated with security requirements, and the approaches provide reasoning about their consistency. In many domains, establishing appropriately documented design-time artifacts is mandatory due to legal requirements, e.g., according to the ISO/IEC 62304 for medical device software. Unfortunately, these artifacts are often inconsistent with the implementation [11], eventually causing security issues and a significant effort for harmonizing all artifacts before a certification.

One reason for this inconsistency is the way software is developed. Programming practices involve successive steps of edits, updates, and refinements to improve the implementation and incrementally meet ever-changing requirements [36]. Unfortunately, these changes are often not reflected in the design-time models. In addition, this continuous evolution causes internal decay that can lead software systems to end up in incomprehensible or even inconsistent states [23]. This continuous evolution increases the effort required to extend and maintain a software system and paves the way for security problems. Ultimately, this leads to certification issues as the implementation does not comply with the security design.

In practice, software systems need frequent restructuring to keep them maintainable [9]. To support the efficient restructuring of a software system, refactorings have been proposed and documented in a human-readable form. Despite intense studies and widespread application, a verifiable specification of refactoring operations and the execution of this specification is still an open problem. The same applies to the interaction of refactorings with nonfunctional properties of the software system, such as security.

In summary, the increasing amount of security-critical data and the faster-changing environments are a burden to develop secure software systems. However, there are already some approaches to address the individual sub-problems. However, there is a lack of holistic security engineering support throughout the development life cycle, especially with respect to tracing security requirements and verifying the compliance of all artifacts produced with them.

2 Background and Problem Identification

Software security has been addressed in various ways, but the systematic development of secure software-intensive systems is still not fully addressed when it comes to supporting the entire software development life cycle, as evidenced by frequent news of security incidents. Considering existing solutions, we identify four main reasons that hinder the effective development of secure software systems.

2.1 Non-integrated Solutions

Several approaches have been developed to support the development of secure software systems. MDD-based approaches allow planning of the software system and allow developers to incorporate security considerations from the beginning, but only abstractly [14, 43]. Similarly, common threat modeling approaches, such as STRIDE [40], abstractly model the system to identify security threats. In contrast, implementation-level approaches support the verification of concrete aspects, such as the correct use of cryptographic APIs [15], but not whether these are used where needed. In the best case, design-time security considerations should be reused until the final product is certified, but in practice, there are many non-integrated solutions. Security-related information collected in design or threat models must be manually transferred to the implementation in order to use appropriate security tools or perform manual code reviews.

2.2 Inconsistency and Missing Traceability

Often, the initial security requirements of a software-intensive system and the documentation of the system are inconsistent with the implementation, making it difficult to reason about security at the system level. Checking whether an object in a medical management system contains personal or medical information, and the resulting security requirements, can become a nontrivial task. To enable traceability, the continuous changes in security assumptions and design must be reflected in both the design-time models and the implementation. Currently, developers must manually trace between the various artifacts to identify and apply the necessary changes in the right places. In practice, this often leads to models not being used at all, despite their obvious benefits, such as systematic threat modeling planning for secure system designs. Therefore, we need to maintain correspondences between artifacts used in all development phases and automate the underlying mapping process.

2.3 Security-Aware Restructuring

As software systems are continuously subject to changes, we have to continuously check their security compliance, e.g., with design-time security requirements. In the best case, we can evaluate the desired change before applying it. Current refactoring approaches do not consider nonfunctional properties such as security. We can only evaluate the impact of a refactoring on security aspects after executing the refactoring, e.g., to notice that medical information has been moved to an object that is sent over a non-encrypted connection. This entails the risk of not being able to undo the change entirely. In summary, security-preserving restructurings are required to support the restructuring of security-critical systems without requiring a complete re-certification. If changes cannot be checked upfront, we need means to efficiently check only security properties that might be affected.

2.4 Variant-Rich Software Systems

While existing security approaches can be applied to each product or variant of a software product line, due to the vast amount of possible product configurations, this is not feasible within a reasonable time. We need means for applying security compliance checks and security-preserving refactorings to software product lines without enumerating every single variant. Consequently, the intended measures discussed above must also support software systems with many variants.

3 Research Questions

Based on the problems identified, we formulate five research questions that are answered in the thesis [26]. Figure 1 maps the research questions to the development artifacts considered by GRaViTY. We introduce the research questions discussed in the thesis in detail in what follows.

3.1 RQ1: How Can Security Requirements Be Traced Among System Representations Throughout the Development Process?

During the development of a software system, various artifacts are produced, such as models or source code. Following security by design [14, 38], security requirements are already planned and validated on the early design artifacts. These security requirements specified on model elements have to be addressed on later models by planning concrete security measures or their concrete realization in the implementation. To ensure the security of a software system, we need to trace the specified security requirements through all artifacts created. In doing so, we have to take into account continuous changes to the software system, e.g., due to ongoing development activities or maintenance, under which we have to preserve the validity of the created trace links. Furthermore, we need to identify an appropriate granularity of trace links to support security requirements on design-time models and code. Early design-time models are at a different level of abstraction than

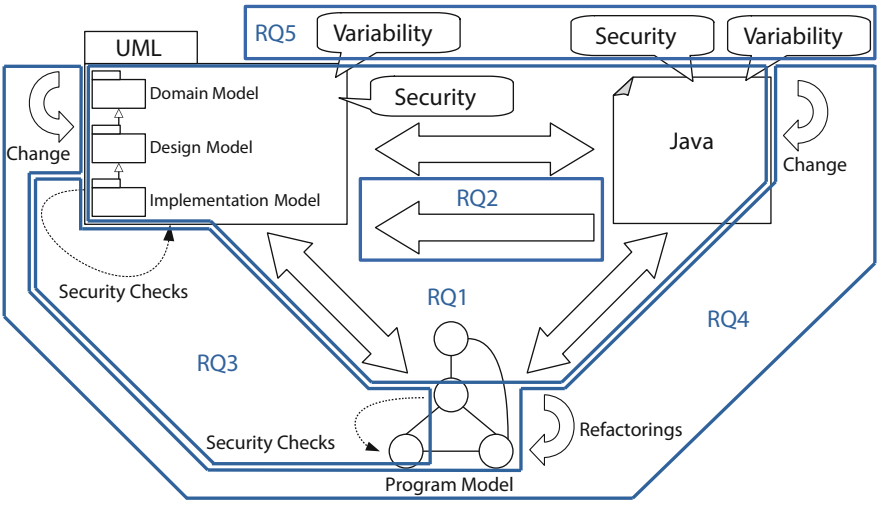

Fig. 1 Concept of GRaViTY with RQs

the final implementation of the software system, where individual methods or statements may be security critical, e.g., a security requirement on a communication link in a deployment diagram that is reflected by a call to a cryptographic API.

3.2 RQ2: How Can We Apply Model-Based Security Engineering to Legacy Projects That Have No or Disconnected Design Models?

Many software systems that were developed decades ago are still in use and are more or less actively maintained. For such legacy systems, often, no models are available, or the existing models have been created in the early phases of system development and are disconnected from the implementation. As most legacy software systems have not been developed using the approach presented in the thesis, the question is how these legacy systems can switch to using the introduced model-based security engineering approach for further development and maintenance. Since tracing between design-time models and implementation is essential, we need efficient and effective means, automated as much as possible, to reverse-engineer these trace links for legacy projects. Thereby, we distinguish between two kinds of legacy projects: projects that do not have design-time models and projects for which early models were initially created but no traces have been maintained.

3.3 RQ3: How Can Developers Be Supported in Realizing, Preserving, and Enforcing Design-Time Security Requirements?

Various approaches have been developed to plan the required security mechanisms in the early stages of software design. However, when it comes to verifying the implementation of security requirements in a software system, most checks have to be performed in manual code reviews. This is due to the local scope of individual security analyses and the lack of automated reuse. To effectively support developers in implementing and verifying design-time security, automated reuse of security specifications and appropriate checks for verifying security properties on other system representations are required. The most relevant question is what we need to check and where we have to check it to show that the specified security requirements are met. For example, a fundamental security requirement for a medical system is that no personal or medical data is accessible to unauthorized entities. In the design models, we can identify what we consider to be such sensitive data and plan appropriate measures at an abstract level, e.g., that only certain parts in a security core of the application are allowed to access this information and that it must be encrypted when it leaves that core. Verification of compliance involves

a variety of checks, including dependency and taint analysis, which must be configured according to the specific requirements, as well as verification that the information is actually encrypted when required. In addition, it may not be sufficient to only statically check the code, as a exchanged library at deployment or a newly discovered attack vector may cause security violations in a software system that has passed all static security checks.

3.4 RQ4: How Do Changes Affect a System's Security Compliance, and How Can These Effects Be Handled?

The development of a software system consists not only of adding new elements but also of modifying existing elements. Both changes require the continuous update of the traces studied in RQ1. However, as part of RQ1, we do not look at how such changes might affect security requirements. Suppose we want to guide developers. In that case, we have to inform them if some changes, which have automatically been performed by our tool support or manually by them, affect security requirements. For example, this is of particular interest in the certified software scenario [27, 31], where it has to be ensured that a change violates no security requirement.

3.5 RQ5: How Can We Verify and Preserve Security Compliance in Variant-Rich Software Systems?

Often, software systems come in many variants that share huge parts in common. Thereby, the number of possible variants can quickly reach an astronomical scale, making the security analysis of every single product infeasible [22]. Nevertheless, for every single variant or product, we have to ensure that it does not contain any security violation. Furthermore, we have to preserve security compliance also in case of changes, e.g., in case of applied restructuring operations. Here, the goal is to find means to apply the developed security engineering approach also to variant-rich software systems.

4 Research Methodology

To answer the presented research questions and provide a solution to the outlined problems, we followed the design science research methodology [6, 13, 24]. The goal of this research approach is to develop artifacts that overcome current boundaries. Thereby, new knowledge is achieved by building and investigating

the application of the developed artifact. Accordingly, this approach requires that, initially, a general solution concept is developed, which is afterward implemented and evaluated. If necessary, the developed solution concept is adapted based on the observations during application and evaluation until the desired goals are met. We divided the topics of the thesis into small sub-problems with individual research questions that can be investigated separately for solving the identified problems and incorporated them into one approach afterward.

5 Approach

To overcome the outlined challenges in developing and maintaining secure software systems, we identified five research questions, focusing on aspects required for improving the model-based development and maintenance of secure variant-rich software systems. To allow continuous model-based security engineering, we mainly focus on the automated tracing of security requirements throughout the whole development process and their continuous verification. In general, the idea of the presented GRaViTY development approach is to automatically create and maintain detailed low-level trace links between design and code artifacts. These trace links are intended to be processed by the tool and not for direct manual use. Developers benefit from the trace links through tool support that uses them for automated navigation between different artifacts. In addition, trace links are used to propagate security-related information between models and the implementation of a software system. Also, the trace links allow to automatically reflect changes on any artifact to all other artifacts. Due to this continuous automated synchronization, which allows changing all artifacts of a software system at any time, the GRaViTY development approach supports both sequential and agile development processes.

In this section, we discuss from a developer's perspective how a secure software system can be developed with GRaViTY to overcome the identified problems. First, we discuss our assumptions on how to allow developers to work efficiently at the development of secure software systems. By doing this, we derive key ideas on which we will build our solution. Afterward, we show the development process for developing secure software systems using GRaViTY. Also, we show the provided tool support and how it is integrated into this process. Finally, we demonstrate the development using our approach from the perspective of a developer.

5.1 Key Ideas of the GRaViTY Approach

Developers play an essential role in the success of a software project. The more developers can focus on their tasks, the more efficient they can be in solving these tasks. The primary goal of GRaViTY is to enable the successful development and

maintenance of secure software systems. To achieve this goal, we identified four key ideas to be realized in GRaViTY.

5.1.1 Suitable Views

The first key idea is that developers should work on the most suitable view for their task. For every task, there is a view in which this task can be carried out most effectively. For example, when a security expert is planning or updating the general security requirements of a software system, an abstract view of the software system, such as a thread model or an architectural model, is more likely to be appropriate than the source code with all its details. However, due to circumstances from the used development process or tooling, all the required information might not be available in this view, or the view cannot easily be created. For example, while a software system has initially been designed using means to specify security requirements and measures on abstract design-time models, such as UMLsec [14] or SecDFD [43], due to missing trace links, changes in the security requirements have to be specified on the implementation level. Such situations should be avoided by the design of our approach and proper tool support. Tool support must ensure that software developers and experts, such as security experts or software architects, can always work in the view of the system best suited to their task.

5.1.2 Side Effects

When working on their task, developers should only focus on their tasks and should not have to care about potential side effects. Nearly every task a developer performs comes with side effects she has to think about. Accordingly, these side effects draw attention from the main task and hinder the development. In the thesis, we explicitly consider two kinds of side effects.

Local side effects: First, side effects within the artifact that a developer is changing, e.g., replacing a cryptographic library to better fit the needs at a particular location in the source code, requires also updating the other locations where the previous library was used. Handling such side effects is essential for maintaining the correct behavior of a software system. Automated tool support as part of a development approach can help identify such side effects. For example, compilers can detect calls to non-existent APIs, and UMLsec checks can detect side effects of model-level changes that affect design-time security requirements.

Global side effects: Second, in addition to local side effects, there might be side effects on other artifacts. If these artifacts do not immediately relate to the correct function of the software system, developers should not have to care about side effects on these. For example, consider a developer optimizing a software system's implementation-level design quality. Most changes might not affect the architecture of the software system, since they are too fine-grained and do not

affect the borders of components. In this case, the developer should not have to care about the effects on the architecture during his or her task.
However, coming back to the suitability of views, an architect should also not have to review the local restructurings at the implementation level of the software system. Side effects that occurred and changed the architectural level should be propagated to the architectural level.
Furthermore, refactorings might have side effects regarding a software system's security requirements, e.g., by making sensitive information accessible. Here, the developer should still be able to focus on the code quality, and tool support should take care of preventing changes with such side effects.

To this end, with GRaViTY, we want to get one step closer to the point where developers do not have to think about such side effects. The ultimate goal is to automatically propagate all changes made by a developer to all other artifacts and then present the propagated changes to an appropriate expert for review. In addition, tool support should reduce the risk of changes leading to violations in other artifacts.

5.1.3 Synchronization

To avoid the individual artifacts of a software system, such as design-time models and code, to diverge, a continuous synchronization in the sense of reflecting every change on all artifacts is necessary. Keeping all artifacts synchronized in case of changes usually requires a significant manual effort, even when tool support is used, and is likely to give rise to inconsistencies. Using GRaViTY, developers should be able to change artifacts in arbitrary order, and their changes will be automatically propagated by GRaViTY for keeping all artifacts synchronized. Furthermore, this step is a prerequisite for allowing developers, architects, and security experts to work on the most suitable view of the software system as depicted in the previous two ideas. Accordingly, the synchronization of the artifacts should happen as far as possible in the background with as few user interactions as possible.

5.1.4 Continuous Security

To ensure the security of a software system under development, it is essential to check every change for its security implications at some point of time. This can be either aggregated before a release, when a commit is pushed to a repository, or, in the best case, continuously during the development as part of the live checks provided in an IDE. Similar to bugs, the earlier a security violation is discovered, the easier and cheaper it is to fix. For this reason, in GRaViTY, developers are continuously assisted by automated security compliance checks helping to preserve the security of the software system. Due to the needed runtime, which can vary depending on the size of the system and the properties selected to be checked from a few seconds to multiple hours, security compliance checks of the entire system are provided

as batch checks but should be also integrated into continuous integration pipelines in the future. For live support in an IDE, we discuss how checks can be executed incrementally to only consider the changed parts.

Such continuous automated security checks are also an essential concept in other approaches, e.g., SecDevOps [20]. We consider these in GRaViTY, but the goal is to go even one step further. Usually, when talking about continuous automated security checks, low-level security checks with a limited scopes are meant. In our approach, we target the security compliance of the implementation with the specification in design-time models. Nevertheless, security checks with limited scopes, such as UMLsec that only targets the model-level, are essential to ensure the consistency of the security specifications with which we check the compliance. However, these automated security checks should not replace manual reviews but support these. Also, continuous automated security checks allow to review changes quicker and study their effects. This eases incremental reviews.

To summarize, we need a development process that allows developers to focus on their tasks and allows them to perform the tasks on the most suitable view on the software system. In addition, such an approach might also assist in performing the tasks themselves. The consideration of tool support can be a fundamental part of such an approach. However, in the intended GRaViTY approach, tool support is not meant to replace developers, security experts, or software architects but to assist them. While the desired tool support might not be easy to implement from a technical perspective, the main challenges lie in the design of a development approach supporting the outlined key ideas and in the underlying challenges that have to be solved for realizing the approach.

5.2 The GRaViTY Development Approach

Next, we show the general development process using the GRaViTY approach and the automatically executed tasks within this sequence. Figure 2 shows a conceptual overview of the development using the GRaViTY development approach.

The artifacts that will be created are shown on the left side of Fig. 2. We assume that three levels of design models are used in addition to the concrete implementation of the software system. At the most abstract level, a domain model captures the essential elements and relationships of a domain, usually in the form of a UML class diagram. These elements are then detailed according to the planned system and related to the coarse-grained element of the planned system, e.g., in an early class model or threat model. In the implementation model, the coarse-grained elements such as components, classes, or processes from the system model are detailed to the point where they can be implemented in source code.

As soon as a model is created, it is denoted by a circle representing an instance of the model or the software system's source code. Following the figure, we assume, that all models are created in the order of their abstraction level, and none is temporarily skipped. However, we do not assume that any of these models is

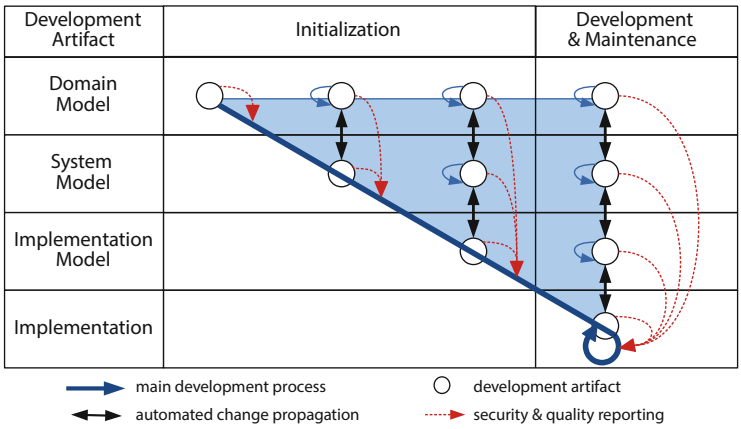

Fig. 2 Development process of the GRaViTY development approach

completed before the next one is created. Incrementally, developing the models in iterations is explicitly possible and allows the usage of GRaViTY in agile development processes.

In agile development, the main development process has three initialization steps in which initial versions of all models are created. In the fourth step, the development and maintenance phase is reached, in which we iterate until the software system has been developed. If we want to consider the maintenance of the software system, we stay in this step and iterate until the software system's end of life.

The blue area above the main development process arrow contains all artifacts available in the current step of the main development process. Whenever a change is applied to any of the artifacts, this change is propagated to all other artifacts that have been developed automatically. The corresponding development activities are denoted in the figure by blue arrows.

A software system's development is supported by security and quality reports covering all artifacts that have been developed. While working in the way as presented above, trace links are created and maintained continuously that will be leveraged for selecting and executing security compliance checks. The security and quality aspects derived this way are centrally reported into the main development process, which is denoted by red, dotted arrows.

5.3 Developer Perspective on Using GRaViTY

In Fig. 3, we show the interaction of a developer with the software system under development while using GRaViTY. The software system under development is depicted in the center of the figure. Thereby, the software system consists out of

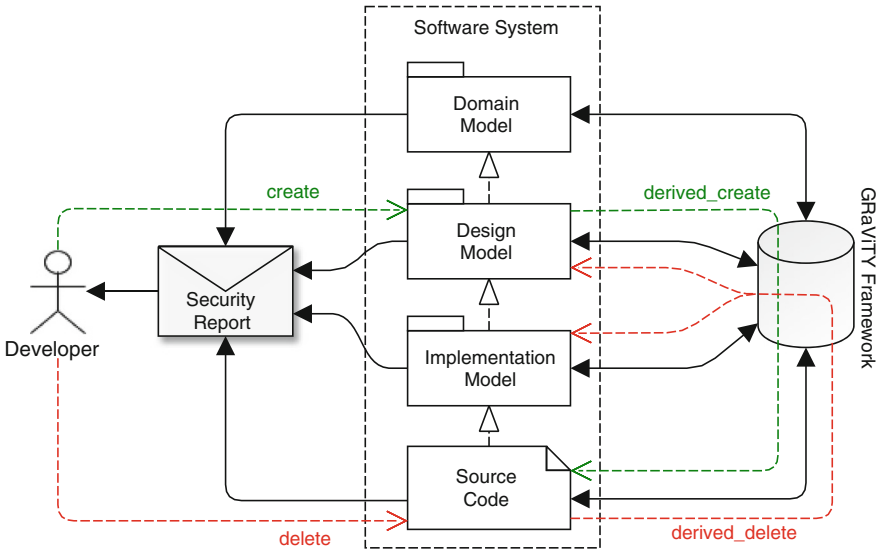

Fig. 3 A developer performing changes using GRaViTY

the discussed development artifacts, namely, different design models and the source code of the software system. These artifacts as well as their relations are shown in the center of the figure.

The GRaViTY framework is indicated by a cylindrical shape on the figure's right side. This shape connects all development artifacts and operates invisibly for a developer in the background. It takes care of synchronizing all artifacts in case of changes, the propagation of security requirements, and security checks.

On the left of the figure, a developer is shown that can directly interact with the development artifacts of the software system. In our case, interaction means that the single artifacts of the software system can directly be edited by the developer, using an IDE into which GRaViTY is integrated. This integration comprises user interfaces allowing developers to make use of the GRaViTY tool support, e.g., by using refactorings for restructuring the implementation. Currently, only Java in the Eclipse IDE in combination with the Papyrus model editor [16, 42] for UML models and data flow diagrams is supported.

Within this IDE, GRaViTY continuously provides reports to developers. For example, this reporting comprises details on security violations currently present in the software system in the form of error markers on the models and code but also more detailed reports via an integration with the UMLsec tooling. For other cases, such as details on the effects of planned refactoring operations, the information is immediately provided as part of the refactoring UI. Based on the reports, developers and experts can plan improvements to the software system. For the generation of reports, GRaViTY considers all artifacts present in the software system.

Whenever a developer edits a development artifact, e.g., by deleting and adding elements in models or source code, these changes are propagated to all other artifacts by GRaViTY. For example, the developer's addition to the design model leads to a derived addition in the source code, and a deletion of elements in the source code leads to deletions in the implementation model and design model. After every change, an updated report is created and presented to the developer. This report can then be used for estimating the impact of the change but also be shared with experts, e.g., software architects or security experts.

While working with GRaViTY, there should be no difference between working on a single product or a variant-rich software system. A developer can still change the software product line in his or her preferred way. Also, security and quality reports are continuously provided but now consider the whole software product line.

6 Research Outcomes

In the thesis, we present GRaViTY, an integrated approach for continuous security compliance checks at model-driven development. While answering the research questions, the approach addresses the challenges identified at problem discussion.

6.1 Inconsistency and Missing Traceability

While we use standard UML technologies for tracing among UML models with different levels of abstraction, we employ Triple Graph Grammars (TGG) [39], a bidirectional graph transformation technology, for tracing between models and code. Based on transformation rules, TGGs build a correspondence model and allow changes to be synchronized between models and code. While the TGG rules allow us to abstract details from the statement level, we still end up with very detailed models that need to be connected to more abstract, manually created instances, for which we provide tool support. However, in combination, this approach allows us to automatically prevent inconsistencies throughout software development (RQ1) and allows developers to work on the more abstract instances [25, 28, 29]. We also discuss semi-automated traceability recovery and reverse engineering of UML models (RQ2) [33, 44].

6.1.1 Continuous Tracing

In the thesis, we have shown that we can propagate arbitrary security requirements within UML models of different abstraction but also between UML models and the implementation and an implementation-level program model. For this purpose, we investigated two different mechanisms for tracing security requirements. First,

we extended the TGG transformation to create corresponding security requirements in the implementation as Java annotations. Second, we looked at dynamic tracing using the correspondence model. In both variants, the TGGs allow to automatically propagate changes to keep all artifacts synchronized.

The dynamic tracing avoids enriching the implementation with additional annotations, but it can have the disadvantage of being inefficient. Since the metamodels of the considered models are given, the trace links contained in the correspondence model can point to elements from the different models, but are not directly accessible from them, resulting in a search for all trace links pointing to an element of interest. If only a few traces are required across the correspondence model or an efficient cache has been created, dynamic tracing should be used to avoid distracting developers. However, if many annotations are required for analysis, the propagation is more likely to be efficient. Also, the created annotations are available at runtime. Altogether, small local lookups should be realized using dynamic tracing, while for full compliance checks or at deployment, the UMLsec security requirements should be propagated into the implementation using additional TGG rules.

To conclude, we provide an automated mechanism to preserve consistency between different program representations for managing evolving Java programs. As a result, we obtain a model-based framework for arbitrarily interleaving program evolution and maintenance steps while ensuring consistency. Furthermore, we can use this approach to also translate and synchronize security requirements of model elements between different system representations, thereby providing traceability of security requirements. Our evaluation on real-world software projects up to 200k LOC shows that our approach allows efficient synchronization between code and models after changes with a speedup of 95% compared to extracting the models after the change.

6.1.2 Restoring Traceability

For legacy projects, we discussed the application of GRaViTY considering two different scenarios. First, we considered software projects in which no design-time models exist. Here, we discussed how the required models and correspondence models between the design-time models and the implementation can be reverse-engineered using GRaViTY's synchronization mechanism [25]. Second, we considered legacy projects in which early design models are available but are disconnected from the implementation. To restore this connection in terms of a correspondence model, we introduced a semi-automated mapping approach [33, 44] that provides the user with suggestions for correspondences and learns from its decisions.

The two approaches can be used complementary in projects containing early design models. First, developers can reverse-engineer UML class diagrams using the TGGs and, afterward, reconstruct the correspondence model between early data flow diagrams (DFD) and the implementation. These correspondence models can then be used to create trace links between the DFDs and reverse-engineered UML class diagrams, which is again supported by suggestions that are provided

by tooling. This allows to transfer security requirements from security annotations on the DFDs (using the SecDFD notation [43]) into the class diagrams and avoids specifying these again, preventing potential errors.

To conclude on the application of GRaViTY on legacy projects, the proposed reverse-engineering approaches allow reconstructing models and correspondence models that allow the application of GRaViTY. The reverse-engineered UML class diagrams can continuously be synchronized with the implementation using GRaViTY's synchronization mechanism without any adaptions. We evaluated the scalability of the reverse engineering on real-world Java projects up to a size of 200k LOC. The correspondence model created between early design models and the implementation is a snapshot of the current state and cannot be automatically synchronized. However, as outlined, they build a basis for propagating security requirements and reconstructing the model hierarchy used by GRaViTY. For the semi-automated approach, we have shown in our evaluation on five open-source projects that we already reach a precision of 50.5% and recall of 69.8% in the first iteration, reaching 87.2% and 92% after a few iterations. Thereby, the user has on average an impact on the recall of 7.9% and provides new input for the automatization. Notice that on average, 75% of all correct correspondences are suggested to the user and do not have to be manually defined. All in all, the user is not only guided through the implementation by our tool but also assisted in creating the correspondence model between SecDFDs and their implementations.

6.2 Non-integrated Solutions

To overcome non-integrated solutions, for ensuring security compliance, we connect design-time security with implementation-level security. The presented automation allows us to effectively check security at low cost by allowing security experts to only specify security requirements once in combination with an automated propagation based on our tracing mechanism (RQ3) [1, 25, 34, 44]. We leverage design-time security requirements for static and dynamic implementation-level security checks. Besides newly developed checks, specifically tailored for verifying considered design-time security requirements, we also discussed how state-of-the-art taint analysis can be improved by connecting design-time security with a data flow analyzer [2]. Finally, we present a runtime monitor for detecting and mitigating violations of design-time security requirements. Furthermore, we support an adaption of the design models to allow an inspection of observed security violations.

6.2.1 Static Security Checks

We introduce a novel approach for tackling the problem of automating the code-level verification of planned security mechanisms. In particular, we have developed

a solution with tool support for executing security compliance checks between an abstract design model and its implementation (in Java). Once defined, the correspondence model is leveraged for an automated security analysis of the implementation against the security design. Two types of security compliance checks are executed: a check whether cryptographic operations are used at the expected locations and a local data flow check for data processing contracts specified in the model. The results of the compliance checks (convergence, absence, and divergence) are lifted to the attention of the user via the user interface of our tool. Similarly, the mapped design is also leveraged to initialize and execute a state-of-the-art data flow analyzer over the entire Java project. We can optimize and automate taint analysis by automatically identifying sources of sensitive information while improving precision by identifying allowed sinks in the design.

Our approach was evaluated with two studies on open-source Java projects, focused on assessing the performance from different angles. The rule-based security compliance checks are very precise (100%) and rarely overlook implemented cryptographic operations (recall is 94.5%). In addition, the local data flow checks are fairly precise (79.6%) but may overlook some implemented flows (recall is 65.6%), due to the large gap between the design-time SecDFD models and the implementation. Further, our approach enables a project-specific data flow analysis with up to 62% fewer false alarms.

6.2.2 Dynamic Security Checks

To ensure security compliance at runtime, we introduce an approach for coupling model-based security analyses with the code level at runtime and supporting round-trip engineering by providing feedback into the models [35]. We realized support for checking secure call dependencies at runtime, by extending the realization of UMLsec *Secure Dependency*, which could only be checked statically (and thus partly) by now. We provide a runtime monitor that leverages the implementation-level security annotations discussed above for enforcing the design-time secure dependency security property. Reaction to detected security issues is supported by passive reactions like call trace logging or actively by providing modified return values to protect real application data. Round-trip engineering is supported both by feeding additional associations monitored during execution back into the model and automatically generating sequence diagrams of attacks to support developers in investigating attacks with graphical support and related to the model. Thus, software system evolution detection is also tackled.

We evaluated the effectiveness and applicability of the security monitoring against real CWEs and DaCapo benchmark. Results show that we support realistic application scenarios and real-world software systems. Further, a user survey shows that the generated sequence diagrams are useful for investigation security violations that were observed or mitigated.

6.3 Security-Aware Restructuring

To detect security violations after changes, we introduce security violation patterns that encode implementation-level security checks against design-time security requirements as graph patterns (RQ4) [34]. Especially, we discuss their incremental execution for efficiently verifying security compliance instead of full-security compliance checks. In addition, we provide security-preserving refactorings for ensuring security compliance at restructuring (RQ3 & 4). The security-preserving refactorings allow checking security compliance before modifying the implementation [28, 29, 37].

While the refactoring of a software system is already challenging, this challenge even gets greater on security-critical software systems. We have shown how refactorings can be formalized using graph transformation languages [28, 29]. Existing works show that such formalizations allow reasoning about the correctness of the refactorings regarding them not changing a software system's behavior [19]. Also, such formalization allows checking the applicability of the refactorings upfront. However, the correctness of the refactored implementation could not be guaranteed as the refactorings had to be performed manually on the implementation. Here, we show how to overcome this gap using the program model and synchronization mechanism introduced in the thesis. Finally, we have shown how the formalized refactorings can be extended with security constraints, leveraging design-time security requirements.

In summary, the presented solutions allow the restructuring of security-critical software systems as part of the GRaViTY development approach. In our evaluation, we show that the incremental execution of the security violation patterns provides a significant speedup against security compliance checks of the entire system (which did not terminate within a reasonable time). During refactoring, the discussed security extensions allow to automatically prevent security-violating refactorings. Further, we have shown that our refactoring approach also prevents behavior-changing refactorings that are executed by the Eclipse IDE.

6.4 Variant-Rich Systems

Finally, we investigated the application of GRaViTY to variant-rich software systems (RQ5). To verify UMLsec security requirements in model product lines, we have encoded the checks as OCL constraints and applied a template interpretation approach [32]. Developers verifying their product lines can use the our OCL constraints as a black box and do not have to look into the complicated logic. Detected violations are automatically presented on a concrete variant containing the violation, and other affected variants are listed. To apply arbitrary pattern-based checks, such as security violation patterns or security-preserving refactorings, we have extended the Henshin graph transformation engine to support variability within transformation rules and models at the same time [41].

6.4.1 Design Time Variability

We provide a comprehensive methodology for the model-based security analysis of software product lines. We extended our UMLsec to also support variability within the security requirements by adding presence conditions to the security annotations [32]. Users specify security requirements as well as variability information as part of the design-time system models. Furthermore, we investigated how we can detect security violations on the UML product lines without iterating all products. For this purpose, we specified UMLsec checks as OCL constraints and evaluated these using a state-of-the-art template interpretation technique [7]. This way, our analysis addresses the scalability issues encountered in this setting by lifting the analysis to the level of the entire product line rather than individual products. In our evaluation, this solution enables the analysis of realistic product lines where the naive approach terminated without a result; a user study indicates the usefulness of our methodology.

6.4.2 Variability on the Implementation Level

To allow the application of refactorings and security violation patterns to SPLs, we introduce a multivariant model transformation approach allowing applying variability-based transformation rules to software product lines. To be more precise, we propose a methodology for software product line transformations in which not only the input product line but also the transformation system contains variability. At the heart of our methodology, a staged rule application technique exploits reuse potential concerning shared portions of the involved products and rules. We present a formalization of our technique, including an optimization that supports an efficient checking of negative application conditions (an advanced transformation feature). We demonstrated practical benefit by applying our technique to two scenarios from a software evolution context. We observed speedups in all considered cases, in some of them by one order of magnitude. As part of this evaluation, we have shown how our methodology can be used for refactoring software product lines using security-preserving refactorings. The application of security violation patterns to SPLs works analogously. The proposed multivariant transformation approach is not only applicable to our two scenarios but to every variability-based transformation rule and product line. For example, the variability-supporting UMLsec checks, currently expressed by us using OCL constraints, could also be implemented using this technique.

7 Case Studies

In addition to the individual evaluations, we applied GRaViTY in two case studies to demonstrate that the approach works as a whole. The first case study is the Electronics Health Management System *iTrust*, and the second case study is the

Eclipse Secure Storage of the Eclipse IDE. As the developers of iTrust provide complete documentation and models are available in existing research, we used iTrust to demonstrate the feasibility of the GRaViTY approach for developing a new software system taking security into account. While the Eclipse IDE also provides good documentation of the implementation, there are no requirements or models available. For this reason, we applied the GRaViTY approach to Eclipse Secure Storage to demonstrate its feasibility on legacy projects.

7.1 Case Study 1: iTrust

The iTrust case study comprises a realistic and working *electronic health records* system that has been developed and maintained in university classes over 25 semesters and is compliant with the HIPAA Security and Privacy Rules [12, 18]. The main documentation is provided as requirements describing use cases of the iTrust system. The software system itself has been implemented in Java using Java Server Pages (JSP). This project has been used as a subject in various research projects, resulting in the creation of design-time models in addition to the original source code [4, 5, 12].

In this case study, we simulate the implementation of the iTrust system using GRaViTY from the very beginning, starting with requirements engineering. After the initial development of the software system, we focus on the restructuring of iTrust as part of the maintenance. Finally, we showcase the conversion of iTrust into an SPL. In all steps, we reuse the existing iTrust artifacts and create all required artifacts following the GRaViTY development approach.

7.1.1 Requirements Engineering

Usually, the development of a software system starts with an analysis of the domain as part of the requirements engineering. The knowledge about entities and relations within the software system's domain is captured in a domain model. The domain model elements are then used to specify their realization in the software system. Here, the specification of the software system's intended functionality is one of the first steps of requirements engineering. For this purpose, the UML provides the notation of use case diagrams. To simulate the requirements engineering, we manually recreated iTrust's use case diagram based on iTrust's requirements by redrawing a diagram in less than an hour. Thereby, we took a domain model as given and refined it by specifying the use case diagram. The used domain model shows basic concepts in a hospital such as doctors treating patients. Whenever there was a refinement relation between the use case diagram and the domain model, we explicitly modeled this relation. In the next step, the domain model and use case diagrams are refined further to specify an architecture that allows the implementation of the specified use cases.

7.1.2 Software Architecture and Security Modeling

After requirements engineering, based on the requirements models and the textual requirements, the software system's architecture is specified. Following the principle of security by design, we have to consider security requirements explicitly in this step. Accordingly, we discuss the simulation of the architecture specification for the iTrust system. To this end, we focus on the feasibility of refinements for specifying software architecture and security engineering. Starting from the models developed at requirements engineering, we iteratively refine these models until we reach a detailed specification of the iTrust system.

After every extension step, comprising the addition of a coherent set of model elements, a security engineering step takes place. Here, we considered the security engineering using UMLsec and SecDFDs. As the SecDFD and UMLsec specifications and checks are known from the literature, we do not focus on their usage but the *Secure Realization* security-refinement mechanism introduced in the thesis.

As part of our case study, we simulated these steps by selecting parts of the design and implementation models and iteratively rebuilding the models. Whenever we added a new part to the models, we also created the corresponding refinement relations. We started our simulation with a domain model already containing fundamental security requirements, such as that personal data has to be classified at the security level of secrecy. Based on this model, we simulated three evolution steps:

1. In the first step, we defined classes in the design model refining persons and actors of the domain model and use case diagram.
2. Afterward, we added the data classes for storing medical information about patients to the design model.
3. Finally, we added classes and operations for implementing the functionality of the use cases.

Figure 4 shows on the right-hand side of the figure a corresponding excerpt of the domain model of the healthcare domain in which sensitive information such as the home address of a person is classified using UMLsec. On the left-hand side, the figure shows the design of iTrust and using realization edges how it realizes to the elements from the domain model.

7.1.3 Implementation

After reaching a state in which the design-time models are detailed enough, we have to start implementing the software system. Thereby, tracing is required from the first written line of code for applying the GRaViTY approach. For this reason, we focus on the integration of GRaViTY's tracing approach into software development.

Using the synchronization mechanism of GRaViTY, we generated an early Java class layout from the implementation model. Afterward, we filled this layout manually with functionality. During this step, the implementation model has been

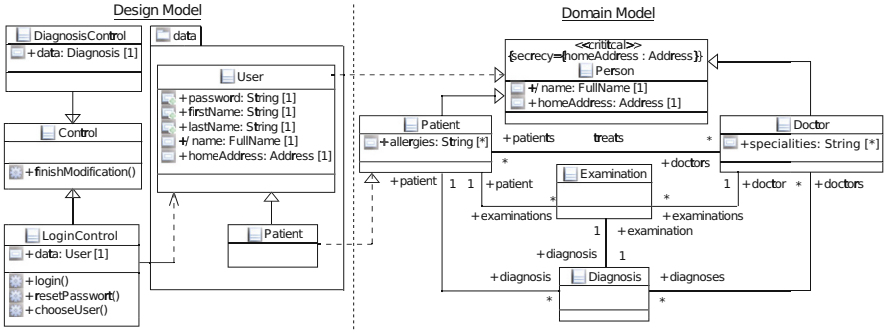

Fig. 4 Refinement relations between the design model of iTrust and a domain model of the healthcare domain

kept synchronized by GRaViTY with the manual changes. We performed this manual extension by copying and pasting implementation fragments of the iTrust implementation into the generated class layout. However, as the MoDisco parser is not incremental, in addition, we had to simulate these changes on the MoDisco model by manually copying the corresponding changes into this model. After every set of source code changes, we generated a MoDisco model and copied the changes into the MoDisco model previously used by GRaViTY, making the changes processable for the used TGG this way.

7.1.4 Security Compliance

The continuous verification of the planned and implemented security is an essential contribution of GRaViTY. As part of this case study, we investigate how these verification steps integrate into the software development process.

Comparable to the incremental specification of the software system's architecture, we also interleaved security verification steps with the implementation steps. These implementation steps have been discussed as the subject of the previous part of this case study. After synchronizing every change made on the implementation with the design models, we manually executed all security compliance checks.

As in the generated class design and the first pasted code fragments, no security mechanisms have been contained, and all have been reported as absent. For this reason, initially, we faced a long list of absences regarding the planned security design. However, as we incrementally added more functionality from iTrust's implementation, the size of the lists of absences reduced until we got rid of all absences. Thereby, the absences functioned as a kind of to-do lists for security-related tasks and as selection criteria for the next code fragments to paste. For example, Fig. 5 shows a screenshot of the corresponding tooling with detected issues at the bottom. In the example, the security-critical change password process considered in the design represented as data flow diagram has not been implemented

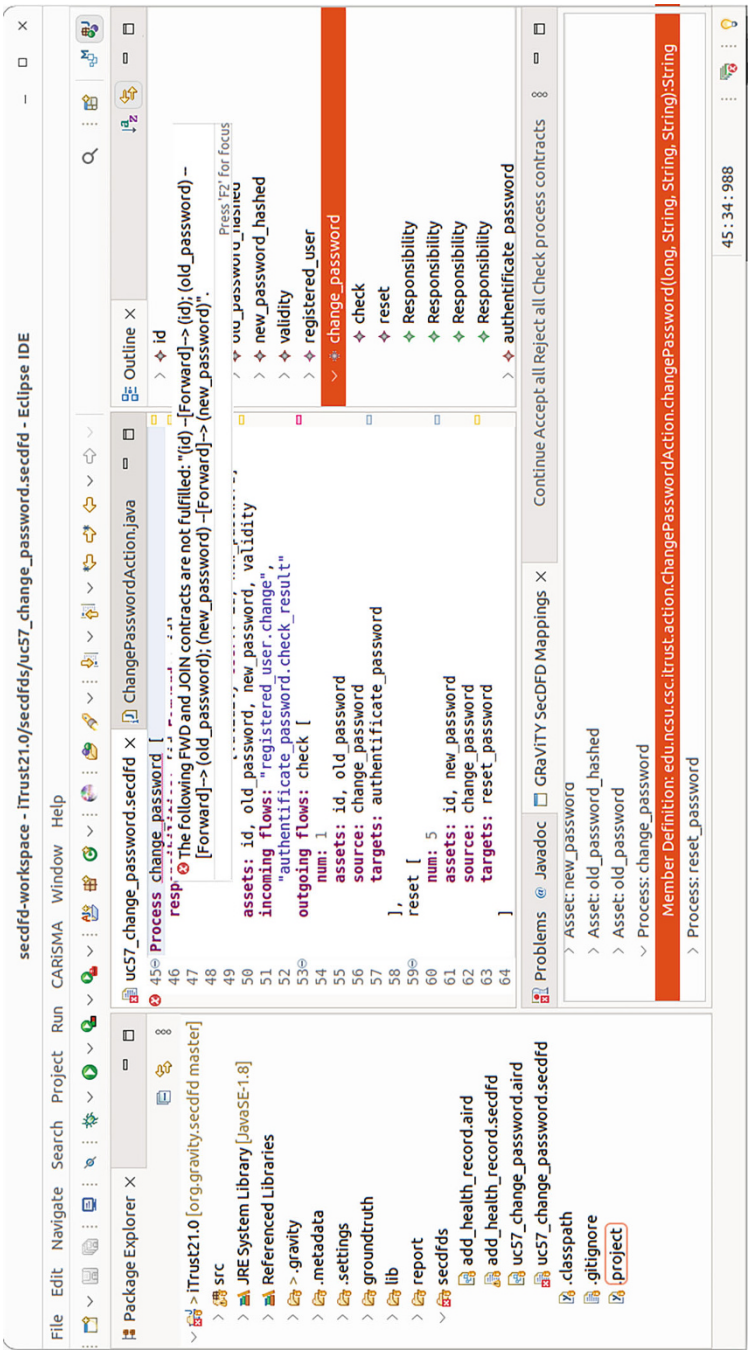

Fig. 5 Security compliance check between a data flow diagram describing an iTrust use case and the implementation

in the implementation yet. As the source code inserted this way was always security compliant, no other violations have been reported.

7.1.5 Restructuring

After reaching the state in which our case study system's implementation was identical to the original iTrust implementation, we investigated this implementation regarding possibilities for restructuring the software system. Thereby, we only focused on restructuring in terms of refactorings.

To find additional refactoring opportunities, we executed the search-based optimization tool GOBLIN [37]. Thereby, we added all three refactorings introduced in the thesis (*Create Superclass*, *Pull-Up Method*, and *Move Method*) to GOBLIN. Besides, the optimization criteria considered in the summarized experiment of Ruland et al. (design-flaws [31], coupling/cohesion, visibilities, and the number of changes), we also added the *Critical Design Proportion* metric discussed in the thesis as an optimization criterion. Due to iTrust's architecture along with the Java server pages, most times, the implemented functionality was already well located, and we only rarely found additional beneficial refactoring opportunities.

7.1.6 Variability Engineering

As the last part of this case study, we considered the re-engineering of iTrust into an SPL. In this case study, we mainly focus on the specification of an SPL in terms of the variability within all artifacts of the software system. However, we also consider the security checks for SPLs.

We started on the use case diagrams with the identification of possible features and ended in assigning individual use cases to features. Afterward, we investigated two different approaches for realizing the identified features in the software system: first, a top-down approach by specifying variability on the models and propagating it to code and, second, a bottom-up approach in which we specified variability on the source code and propagated it into the design-time models. After realizing the variability in the iTrust system, we executed the SecPL checks to verify the security of the iTrust SPL.

7.2 Case Study 2: Eclipse Secure Storage

Our second case study focuses on applying GRaViTY to relatively small (2,900 LLOC) but security-critical part of the Eclipse IDE. *Eclipse Secure Storage* [8] is used by Eclipse plugins such as the Eclipse git client to store confidential data like passwords. The Eclipse Secure Storage is implemented as an Eclipse plugin itself using Java. How exactly the secure storage works is described in the help document

of Eclipse [8]. However, this description is rather high level and complemented by the low-level API documentation. We consider Eclipse Secure Storage due to its security criticality, good documentation, and wide usage in practice.

In this case study, we focused on migrating legacy projects to GRaViTY. In what follows, we first discuss the reverse engineering of the Eclipse Secure Storage to create a state in which the application of the GRaViTY approach is possible. Next, we discuss security engineering, aiming at making security requirements explicit and checking the software system regarding compliance with them. Finally, we discuss the runtime monitoring of the Eclipse Secure Storage based on a fictive malicious Eclipse plugin and the adaption of the reverse-engineered models.

7.2.1 Reverse Engineering of Models

As there are no models available for Eclipse Secure Storage, the first step of this case study was the reverse engineering of models. For the reverse engineering of models, we followed a three-step approach. First, based on the documentation of Eclipse Secure Storage, we manually created data flow diagrams and UML activity diagrams, which are similar to the DFDs but include control flow, for two use cases, accessing a value from the secure storage and resetting a password. As usual in threat modeling, these diagrams are at a high level of abstraction and are limited to the essential elements; in our case, they include only 7 assets per diagram and 7 or 10 nodes, respectively. Afterward, we automatically reverse-engineered a detailed UML class diagram from the source code of Eclipse Secure Storage using GRaViTY. Finally, we used the semi-automated mapping approach to establish refinements between the manually created diagrams, the automatically reverse-engineered class diagram, and the software system's implementation.

7.2.2 Static Security Specification and Checks

One of the two main goals of applying GRaViTY to legacy projects is to create artifacts that allow an easier specification of security requirements, compared to their specification on the implementation, and the security compliance checks with these security requirements. The other main goal is to continue with the continuous verification of the software system's security after the initial state has been proven to be secure. In this part of the case study, we focus on creating such an initial secure state using GRaViTY.

After reverse engineering, we started to annotate the models with security requirements. Here, we started by specifying essential security requirements on the DFDs, which were automatically propagated to the detailed reverse-engineered class diagram. These propagated security requirements served as starting points for the specification of detailed security requirements with annotations according to UMLsec *Secure Dependency*, which was guided by the UMLsec tooling.

Unlike the iTrust case study, there is only one level of inheritance, which still simplified this step, but required us to look into the very detailed UML model more often than it was necessary in the iTrust case, where almost all detailed security requirements were propagated from more abstract models. After this specification, the propagation to the source code worked without problems, and as expected, the compliance checks showed no issues. Technically, we demonstrated the feasibility of the tools for annotating the models and, in particular, of GRaViTY's synchronization mechanism for propagating the security requirements to the implementation. An extension of the tooling with clustering approaches to generate additional more abstract models could be helpful.

7.2.3 Runtime Monitoring

In the last part of this case study, we focused on leveraging the specified security requirements to enforce these at runtime. In the implementation of a software system specified by a UML model, the dependencies stereotyped with «call» are usually implemented as method calls and field accesses. Even if a model does not contain violations, at runtime, it has to be guaranteed that the security requirements specified at design time are not violated. Furthermore, detecting all dependencies which can occur at runtime is statically undecidable, e.g., due to the use of Java reflection [17, 21]. What can also not be foreseen from a static perspective are violations caused by an exchanged library or malicious code.

In Eclipse, for example, every installed plugin can access the password store. Which plugins a developer installs into his or her Eclipse IDE is not predictable. However, considering the discussed security annotations, only plugins that comply with the *secrecy* security level should be allowed to access the password store.

To conduct this part of the case study, we implemented a malicious plugin that attempts to illegally access passwords stored in Eclipse Secure Storage. In addition to the security requirements annotated to the design models discussed above, we extended the Eclipse Secure Storage implementation with countermeasures to actively prevent illegal accesses that violate these security requirements. Based on this, we monitored Eclipse for security violations with respect to UMLsec secure dependency and executed the malicious plugin. The runtime monitoring successfully detected and mitigated the security violations caused by the malicious Eclipse plugin we prepared. In addition, the models were modified as expected, providing details about the operation of our malicious plugin and allowing a detailed investigation of the security violation.

7.3 Observations

In the two case studies, we demonstrated the technical suitability of the developed approach to work as a whole and being technically applicable for developing secure software systems. Although some parts of the case studies required manual

simulations of parts of the approach, our case studies revealed that the current implementation of GRaViTY already provides much support for effectively and efficiently aiding the development of secure software systems. As for most research prototypes, especially the user interface should be improved for practical application and there is room for automation to more seamlessly integrate into the workflow, currently, most tooling that could run automatically has to be triggered manually. Altogether, the case studies demonstrated the technical feasibility of GRaViTY.

Considering the key assumptions on users of the GRaViTY approach, we made the following observations.

Suitable Views: As part of the case studies, we were able to specify security requirements mainly on design models, as we suggest security experts do. While it was necessary to specify some security requirements on a fairly detailed version of these models, it was often possible to specify security requirements on abstract models and propagate them to more detailed models and the implementation. It may be that we have enforced working on the model rather than the code, but this still shows technical feasibility. However, we agree that the usability should be improved, but this is only partly related to our own tools, but mainly to the Papyrus UML editor used.

Side effects: While conducting the case studies, we explicitly tried not to let our actions be influenced by their potential side effects, to inspect if we would be notified about them. As expected due to this behavior, there were some situations where we had to resolve conflicts caused by side effects, but they were always prominently presented to us by the tool support. For example, a dependency added due to an implementation-level change caused a security problem on the design models but was detected by the continuous security checks and displayed as an error marker. Therefore, we believe that this approach allows us to focus more on security engineering or implementation, but it remains to be verified with external developers.

Synchronization: In the case study, we were always able to synchronize our changes without any technical problems. This is not surprising, as our case studies mainly contained changes that resided at the detailed source code level, and TGGs can propagate all changes from a more detailed to a more abstract model. The opposite direction is more challenging, as changes cannot be reflected one to one and sometimes ended up with the change being applied to the source code but requiring manual post-processing. For example, when we deleted a dependency in a design model to fix a security issue due to an illegal access, the source code was adjusted accordingly by deleting a method call, but the variable to which the method's return value was assigned was now unassigned. We considered this to be intentional and not a significant problem, since the method had to be changed as a result of a design decision. Instead of accepting this compilation error, the entire body of the method could be automatically commented out and a to-do added as a more elegant solution. In practice, there may be cases where large changes to the UML models that need to be propagated

to the implementation, or parallel changes, may cause synchronization problems. However, what such cases are is explicitly discussed in the thesis.

Continuous Security: Throughout the case studies, the primary goal of being able to continuously check for compliance with security requirements was possible. After the initial specification of security requirements, we were able to continuously check the software system for security violations and were notified of violations. However, this only demonstrates the technical feasibility of the approach and the ability to tailor implementation-level security checks based on design-time security requirements in all situations we faced. Based on the case study, we can only state that we did not receive any false positives for security violations, but we cannot judge whether the results were always correct or whether security issues were missed.

8 Outlook

In the thesis, we mainly looked at individual software systems that are located in critical domains. In these domains, standards such as ISO/IEC 62304 for medical device software, which was relevant to our first case study, require developers to deliver all of the artifacts that are created when following MDD and are also considered in GRaViTY. However, there are still many domains with individual challenges that need to be addressed. Also, the approach is designed in principle not to be limited to the Java world but any object-oriented language, which remains to be confirmed.

One particularly relevant domain that requires extremely complex software-intensive systems and is utmost security and safety critical toward which we are currently expanding GRaViTY is autonomous driving. While autonomous driving systems are in principle located in a strongly regulated domain that would guarantee the perfect applicability of GRaViTY due to standards such as the IEEE 26262 on functional safety management, most autonomous driving projects, especially the open-source projects such as Autoware.auto or Baidu Apollo, do not seem to follow strict model-based development processes. The same also applies to many other domains such as mobile apps, and we have to identify more lightweight tracing approaches to allow an easy application of GRaViTY to these domains.

Further, autonomous vehicles are extremely distributed systems as well on the individual vehicle as on external systems that are communicating with the autonomous vehicle. Popular robotic middlewares, such as the Robotic Operation System (ROS), which is the basis for Autoware.auto, foster the realization of such systems as a multitude of individually running nodes that are deployed on the vehicle itself or on external servers and communicate through a message API of the middleware with each other. In the thesis, we have already seen that we can extract much information about the borders of the system, e.g., to optimize security checks with information that is not contained in the source code. Leveraging such

information becomes even more important for effective security checks in such massively distributed systems. Also, we have to integrated different kinds of checks on different nodes. Especially in autonomous driving, there is a significant use of machine learning approaches for tasks such as perception of the environment and prediction of behavior. Here, we have to work with assumptions on the machine learning-based nodes when checking other nodes that interact with these. For checking these nodes themselves, we have to create more dynamic verification approaches. However, as already shown in the thesis even for handwritten code, many aspects cannot be verified statically. We have to develop approaches to trace nonfunctional requirements throughout the entire development process to the runtime and to verify them in all phases to provide developers with a holistic picture.

Finally, we are currently extending our approaches to consider not only security but also other nonfunctional requirements such as safety. In most software-intensive systems, the various nonfunctional requirements do not stand alone, but there is significant interaction. For example, in autonomous driving, a successful attack will potentially lead to a malfunction of the car's behavior, e.g., because some nodes of the system do not provide required data. Such a malfunction is obviously also safety critical, and we cannot consider these two aspects completely separately. Our ultimate goal is to provide an approach that allows the selection of relevant domain-specific profiles of nonfunctional requirements, such as safety and security, and provides a holistic verification of whether the system satisfies all these requirements. In combination with the demonstrated change handling and incremental verification, this will form the basis for an incremental certification framework.

9 Summary

In the thesis, we present the GRaViTY approach for continuously supporting developers with automated propagation of changes to avoid security-critical inconsistencies. Based on this synchronization, security experts can specify security requirements on the most suitable system representation. We can verify and enforce these security requirements on all system representations using automated security checks, allowing us to check the implementation's security compliance, as needed in certifications. To preserve this compliance when restructuring the system, we provide semantics-preserving refactorings that are enriched with security-preserving constraints. For both security checks and refactorings, we show their application to variant-rich software systems. To support legacy systems, we show how UML models can be reverse-engineered also for systems with variants and how existing early SecDFD design models can be semi-automatically mapped to the implementation. In addition to an evaluation of the single parts of the approach, the overall approach is demonstrated in two real-world case studies, the iTrust electronics health records system and the Eclipse Secure Storage.

References

1. Ahmadian, A.S., Peldszus, S., Ramadan, Q., Jürjens, J.: Model-based privacy and security analysis with CARiSMA. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (2017)
2. 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: Proceedings of the 35th Conference on Programming Language Design and Implementation (2014)
3. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice. Synth. Lect. Softw. Eng. **1**(1) (2012)
4. Bürger, J.: Recovering Security in Model-Based Software Engineering by Context-Driven Co-Evolution. PhD thesis, University of Koblenz-Landau (2019)
5. Bürger, J., Gärtner, S., Ruhroth, T., Zweihoff, J., Jürjens, J., Schneider, K.: Restoring security of long-living systems by co-evolution. In: Proceedings of the 39th Annual Computer Software and Applications Conference (COMPSAC) (2015)
6. Crnkovic, G.D.: Constructive research and info-computational knowledge generation. In: Proceedings of the International Conference on Model-based Reasoning in Science and Technology (MBR) (2010)
7. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness OCL constraints. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE) (2006)
8. Eclipse contributors: Workbench User Guide – Secure Storage – How secure storage works. Technical report. The Eclipse Foundation (2013). https://help.eclipse.org/
9. Fowler, M.: Refactoring: Improving the Design of Existing Code. Object Technology Series (1999)
10. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Proceedings of the Conference on the Future of Software Engineering (FOSE) (2007)
11. Gorschek, T., Tempero, E., Angelis, L.: On the use of software design models in software development practice: an empirical investigation. J. Syst. Softw. (JSS) **95**, 176–193 (2014)
12. Heckman, S., Stolee, K.T., Parnin, C.: 10+ years of teaching software engineering with iTrust: the good, the bad, and the ugly. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training (2018)
13. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS Quaterly **28**(1), 75–105 (2004)
14. Jürjens, J.: Secure Systems Development with UML, Springer (2005)
15. 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: Proceedings of the 32nd International Conference of Automated Software Engineering (ASE), pp. 931–936 (2017)
16. Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., Schnekenburger, R., Dubois, H., Terrier, F.: Papyrus UML: an open source toolset for MDA. In: Proceedings of the 5th European Conference on Model-driven Architecture Foundations and Applications (ECMDA-FA), pp. 1–4 (2009)
17. Livshits, B., Whaley, J., Lam, M.S.: Reflection analysis for Java. In: Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS), pp. 139–160 (2005)
18. Meneely, A., Smith, B., Williams, L.: iTrust Electronic Health Care System Case Study. (2021) https://github.com/ncsu-csc326/iTrust2
19. Mens, T., Taentzer, G., Müller, D.: Model-driven software refactoring. In: Model-driven Software Development: Integrating Quality Assurance, pp. 170–203 (2008)
20. Mohan, V., Othmane, L.B.: SecDevOps: is it a marketing buzzword? - mapping research on security in DevOps. In: Proceedings of the 11th International Conference on Availability, Reliability and Security (ARES), pp. 542–547 (2016)

21. Murphy, G.C., Notkin, D., Griswold, W.G., Lan, E.S.: An empirical study of static call graph extractors. Trans. Softw. Eng. Methodol. **7**(2), 158–191 (1998)
22. Oster, S., Markert, F., Ritter, P: Automated incremental pairwise testing of software product lines. In: Proceedings of the 14th International Conference on Software Product Lines (SPLC), pp. 196–210 (2010)
23. Parnas, D.L.: Software aging. In: Proceedings of the 16th International Conference on Software Engineering (ICSE), pp. 279–287 (1994)
24. 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)
25. Peldszus, S.: Model-driven development of evolving secure software systems. In: Collaborative Workshop on Evolution and Maintenance of Long-Living Software Systems (EMLS) (2020)
26. Peldszus, S.: Security Compliance in Model-driven Development of Software Systems in Presence of Long-Term Evolution and Variants. Springer, Berlin (2022)
27. Peldszus, S., Jürjens, J.: Werkzeuggestützte Sicherheitszertifizierung – Anwendung auf den Industrial Data Space. In: Proceedings of the Software Quality Days, Software Quality Lab GmbH, pp. 10–14 (2017)
28. Peldszus, S., Kulcsár, G., Lochau, M.: A Solution to the Java refactoring case study using eMoflon. In: Proceedings of the Transformation Tool Contest (TTC), pp. 118–122 (2015)
29. Peldszus, S., Kulcsár, G., Lochau, M., Schulze, S.: Incremental co-evolution of Java programs based on bidirectional graph transformation. In: Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ), pp. 138–151 (2015)
30. Peldszus, S., Kulcsár, G., Lochau, M., Schulze, S.: Continuous detection of design flaws in evolving object-oriented programs using incremental multi-pattern matching. In: Proceedings of the 31st International Conference on Automated Software Engineering (ASE) (2016)
31. Peldszus, S., Cirullies, J., Jürjens, J.: Sicherheitszertifizierung für die Digitale Transformation – Anwendung auf den Industrial Data Space. In: Software-QS-Tag (2017)
32. Peldszus, S., Strüber, D., Jürjens, J.: Model-based security analysis of feature-oriented software product lines. In: ACM SIGPLAN International Conference on Generative Programming (GPCE), pp. 93–106 (2018)
33. 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, pp. 23–33 (2019)
34. Peldszus, S., Bürger, J., Kehrer, T., Jürjens, J. Ontology-driven evolution of software security. Domain Knowl. Eng. **134**, 1–31 (2021)
35. Peldszus, S., Bürger, J., Jürjens, J.: UMLsecRT: reactive security monitoring of java applications with round-trip engineering. IEEE Trans. Softw. Eng. https://doi.org/10.1109/TSE.2023.3326366
36. Rajlich, V., Gosavi, P.: Incremental change in object-oriented programming. IEEE Softw. **21**(4), 62–69 (2004)
37. Ruland, S., Kulcsár, G., Leblebici, E., Peldszus, S., Lochau, M.: Controlling the attack surface of object-oriented refactorings. In: International Conference on Fundamental Approaches to Software Engineering (FASE), pp. 38–55 (2018)
38. Santos, J.C.S., Tarrit, K., Mirakhorli, M.: A catalog of security architecture weaknesses. In: Proceedings of the International Conference on Software Architecture Workshops (ICSAW), pp. 220–223 (2017)
39. Schürr, A.: Specification of graph translators with triple graph grammars. In: Proceedings of the International Workshop on Graph-theoretic Concepts in Computer Science (WG), pp. 151–163 (1995)
40. Shostack, A.: Threat Modeling: Designing for Security. John Wiley & Sons, New York (2014)
41. Strüber, D., Peldszus, S., Jürjens, J.: Taming multi-variability of software product line transformations. In: International Conference on Fundamental Approaches to Software Engineering (FASE), pp. 337–355 (2018)

42. The Eclipse Foundation: Papyrus Modeling Environment. (2019) https://www.eclipse.org/papyrus/
43. Tuma, K., Scandariato, R., Balliu, M.: Flaws in flows: unveiling design flaws via information flow analysis. In: Proceedings of the International Conference on Software Architecture (ICSA), pp. 191–200 (2019)
44. Tuma, K., Peldszus, S., Strüber, D., Scandariato, R., Jürjens, J.: Checking security compliance between models and code. Softw. Syst. Model. **22**, 273–296 (2022)