Development of Secure Software with GRaViTY

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

***Abstract*—Software systems are entering huge parts of our lives and have to deal with a higher amount of sensitive data than ever before. At the same time, these software systems get more complex and have to be maintained over long periods, including re-engineering. One approach to deal with the issues arising from these trends is model-driven software development. Our tool GRaViTY supports developers in the model-driven development, maintenance, and re-engineering of secure software systems.**

I. Motivation

Software systems tend to be used on a long-term basis in environments prone to changes and often process security-critical data [1], [2]. These trends complicate the re-engineering of a system to keep up with ever-changing security precautions, attacks, and mitigations, which is vital to preserve their security. While model-driven development enables us to address security issues in the early phases of the software design, such as in UML models defined at design time [1], the specification of a system is often inconsistent with its implementation [3]. To allow a secure re-engineering of a system, changes in the security assumptions and the design of the system have to be reflected in both the system's models (e.g., UML models) and its implementation (including program models used, e.g., for static analysis or verification).

Deciding which change is necessary at which location and on which of the many artifacts has currently to be performed manually by developers. The effort for the creation of such mappings after the fact is still high even if this process is guided by tool support [3]. For this reason, we have to maintain mappings between different artifacts used in the different phases of development from the very early beginning and to automate them as much as possible.

To tackle these challenges, we started to develop the GRaViTY framework [4]. This framework allows us to automatically create and maintain trace links between different artifacts, such as UML models, Java source code, and program models for analyses, created during the development of a system. Starting from early design-time models until the creation and maintenance of the code, this framework is intended

1) to reverse-engineer UML models from existing systems,
2) to maintain trace links between these artifacts,
3) keep all artifacts up to date if one artifact is changed,
4) to specify security requirements on the most suitable representation of a system,
5) to continuously check all system representations for security violations and design-flaws, and
6) to assist developers in resolving them, e.g., by applying refactorings or manual re-engineering.

II. Research Questions

For providing the described tool support, the following research questions have been studied.

RQ1: Which Formal Methods are Suitable to Specify Design-Flaws, (Security-)Anti-Patterns, and Refactorings?

One of the biggest issues with the current design- and security-analysis approaches, as well as refactoring approaches, is their informal specification [5], [2]. Anti-patterns or Design-Flaws are mostly specified in a textual manner. Moreover, security checks or refactorings are often only available implemented in an ad-hoc manner. These complicate the study of the side effects and make it, even more, harder to apply identical changes to all system representations.

RQ2: How can Security Properties be Traced between Different System Representations?

While entirely reverse-engineered UML models can be easily synchronized when changes, e.g., as part of a re-engineering, occur this is challenging for elements not present in all system representations. Models that are modeled by developers in early phases have a different granularity than models reverse-engineered automatically. Nevertheless, we have to be able to apply our synchronization approach also to those manually defined models to support secure re-engineering.

Furthermore, if new security properties are specified or updated on one system representation, trace links have to be created to all relevant element in a system's representation. On every single system representation, different security properties have to be handled but they have to be represented in each representation in an appropriate granularity.

RQ3: Which Effect do Changes have on Security Properties?

If we want to guide to developers at implementation and re-engineering, we have to inform them if changes, automatically performed by our tool support or manually by them, affect security properties. This is of special interest in the scenario of certified software [6], [7] where it has to be ensured that no (security) property is violated by a change. While correctly implemented refactorings preserve behavior by definition, they might affect security properties, e.g., due to an increase of the visibility of an attribute. More challenging than refactorings are arbitrary changes of developers. Since we cannot check them in advance, we have to check every security property on every system representation again. Using the trace links established before, we can calculate which security properties we have to check more than one and which security property we don't have to check again.

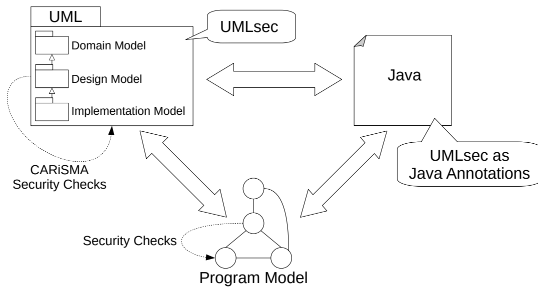

Fig. 1: Concept of the Framework

III. GRaViTY Framework

Our proposed framework, called GRaViTY [4], supports developers in the development and maintenance of secure long-living systems. As shown in Fig. 1, design-time UML models, source code (e.g. written in Java), and a program model for performing sophisticated analyses, e.g., security checks, are continuously synchronized for covering the different phases of software development. To keep the artifacts consistent, we employ triple graph grammars [8] for a bidirectional synchronization between the source code and the program model [9] as well as UML models.

a) Design-time Models: In GRaViTY, design-time models can be defined with different levels of abstraction. While models with different abstractions are often created separately, we encourage developers to model the information of refinement explicitly by the use of inheritance relations.

Usually, the most abstract model is a domain model. Domain models are used in the earliest phases of software development to capture general properties about a system's domain. Afterward, the domain elements realized in the software are concretized in design models, specifying the design of the system and how the functionality is distributed. This is the first point where we have to start to continuously use design and security analyses to ensure the system's maintainability and security. Precise functionality is specified in an implementation model. The implementation model is usually the first platform-dependent model and contains information about the deployment or languages used to implement the system.

b) Program Model: The program model provides a high-level abstraction from the pure source code [9], e.g., by reducing details from the statement level to access edges between the single members. Besides, easy to query structures are created, such as structuring methods and fields into a tree with names, signatures, and definitions [9]. This allows the easy specification of, e.g., refactorings [5], [9], anti-pattern detection [2] and elimination [10], or compliance checks with models [3], allowing to verify re-engineering actions.

c) Security: For the enforcement of security requirements from the beginning of development until re-engineering, GRaViTY makes use of various kinds of security checks. According to the principle of security by design, security should be considered from the earliest phases of development. The UMLsec [1] approach, integrated into GRaViTY, allows the specification and check of essential security requirements already at design time. In UMLsec, UML models are annotated with security requirements like security levels of class members. These security annotations are checked for their compliance with different security policies. Also, GRaViTY supports the enforcement of these security requirements by reusing the information at static code analysis.

Static code analysis is usually used to detect security issues during software implementation. Many approaches locally analyze calls to critical APIs 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. While those 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. GRaViTY can infer this information from the design-time models and initialize required static analyses with them.

IV. Prerequisites

GRaViTY has been implemented as an Eclipse plugin and can currently only be used from the Eclipse IDE. While the program model is language independent and the synchronization with the models is in principle also language-independent, currently we only support Java projects.

If no UML models of the system exist, only some of the required UML models can be reverse engineered from the source code. The manual extension and creation of more abstract UML models have to be done manually.

V. Conclusion

We presented the GRaViTY approach for model-driven development and maintenance of secure long-living systems. GRaViTY allows synchronizing the different artifacts created at the development of a system as well as the specification and reuse of security requirements in the execution of security checks for ensuring its security even at re-engineering.

References

[1] J. Jürjens, *Secure Systems Development with UML.* Springer, 2005, chinese translation: Tsinghua University Press, Beijing 2009.

[2] S. Peldszus, G. Kulcsár, M. Lochau, and S. Schulze, "Continuous Detection of Design Flaws in Evolving Object-Oriented Programs using Incremental Multi-pattern Matching," in *ASE*, 2016.

[3] S. Peldszus, K. Tuma, D. Strüber, J. Jürjens, and R. Scandariato, "Secure Data-Flow Compliance Checks between Models and Code based on Automated Mappings," in *MODELS*, 2019.

[4] "GRaViTY." [Online]. Available: http://gravity-tool.org

[5] S. Peldszus, G. Kulcsár, and M. Lochau, "A Solution to the Java Refactoring Case Study using eMoflon," in *TTC*, 2015.

[6] S. Peldszus and J. Jürjens, "Werkzeuggestützte Sicherheitszertifizierung – Anwendung auf den Industrial Data Space," in *Software Quality Days*. Software Quality Lab GmbH, Jan. 2017, pp. 10–14.

[7] S. Peldszus, J. Cirullies, and J. Jürjens, "Sicherheitszertifizierung für die Digitale Transformation – Anwendung auf den Industrial Data Space," in *Software-QS-Tag*, Oct. 2017, best Paper Award.

[8] A. Schürr, "Specification of Graph Translators with Triple Graph Grammars," in *WG*, 1995.

[9] S. Peldszus, G. Kulcsár, M. Lochau, and S. Schulze, "Incremental Co-evolution of Java Programs Based on Bidirectional Graph Transformation," in *PPPJ*, 2015.

[10] S. Ruland, G. Kulcsár, E. Leblebici, S. Peldszus, and M. Lochau, "Controlling the Attack Surface of Object-Oriented Refactorings," in *FASE*, 2018.