Tools for model-based security engineering: models vs. code

Conference or Workshop Item

How to cite:


For guidance on citations see FAQs.

© The Authors/Owners

Version: Version of Record

Link(s) to article on publisher’s website:
http://www.cse.msu.edu/ase2007/welcome.html

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online’s data policy on reuse of materials please consult the policies page.

oro.open.ac.uk
Tools for Model-based Security Engineering: 
Models vs. Code*

Jan Jürjens and Yijun Yu 
Computing Department, The Open University, GB 
http://www.computing.open.ac.uk/people/{j.jurjens,y.yu}

ABSTRACT 
We present tools to support model-based security engineering on both the model and the code level. In the approach supported by these tools, one firstly specifies the security-critical part of the system (e.g. a crypto protocol) using the UML security extension UMLsec. The models are automatically verified for security properties using automated theorem provers. These are implemented within a framework that supports implementing verification routines, based on XMI output of the diagrams from UML CASE tools. Advanced users can use this open-source framework to implement verification routines for the constraints of self-defined security requirements.

In a second step, one verifies that security-critical parts of the model are correctly implemented in the code (which might be a legacy implementation), and applies security hardening transformations where is that not the case. This is supported by tools that (1) establish traceability through refactoring scripts and (2) modularize security hardening advices through aspect-oriented programming. The proposed method has been applied to an open-source implementation of a cryptographic protocol implementation (JESSIE) in Java to build up traceability mappings and security aspects. In that application, we found a security weakness which could be fixed using our approach. The resulting refactoring scripts and security aspects have found reusability in the Java Secure Socket Extension (JSSE) library.

Categories and Subject Descriptors: D.2.2 Software Engineering: Design Tools and Techniques -Computer Aided Software Engineering (CASE), D.2.4 Software Engineering: Software/Program Verification

General Terms: Security.

Keywords: Security, Model-based Software Engineering, UML, Verification Framework, Code Analysis, Refactoring, Security Hardening.

Model-based Security Engineering
Understanding the security goals provided by software making use of cryptography is one of the major challenges with security-critical systems. Any support to aid secure systems development is thus dearly needed. Towards this goal, the security extension UMLsec for the Unified Modeling Language (UML) [3] allows us to include security requirements as stereotypes with logical constraints. In this paper we present automated tool-support for the analysis of UMLsec models against security requirements by checking the constraints associated with the UMLsec stereotypes. Besides presenting a general, extensible framework for implementing verification routines for the constraints associated with security-critical UML stereotypes, we focus on a plug-in that utilizes an automated theorem-prover (ATP) for first-order logic (FOL) to verify security properties of UMLsec models which make use of cryptography (such as cryptographic protocols), which was explained in [5]. To do so, the analysis routine extracts information from behavioral UML diagrams that may contain additional specific cryptography-related information. If the analysis reveals that there is an attack, an attack generation script written in Prolog generates the attack trace.

To link the model-based security analysis to the code level, we have shown how to insert cryptographic assertions into implementations of cryptographic protocols to ensure that the analyzed cryptographic protocol design is implemented securely [6]. To our experience, it is however non-trivial to insert the right assertions at the right place in the program. As the implementation or the used libraries evolve, the instrumentation may not anymore guarantee the correct link to the protocol design. Moreover, it is not clear whether and how such assertions can be reliably transferred to a different implementation of the protocol. As such assertions tend to crosscut in the code, a good candidate to instrument the program is aspect-oriented programming (AOP). However, aspectJ cannot intercept arbitrary control flow at the statement level (and does not intend to do so either, for good reasons). Our tools therefore maintain traceability between the design and the implementation of a cryptographic protocol through techniques and tools for refactoring scripting and aspect-oriented programming that are supported respectively by the JDT¹ and AJDT² in the Eclipse Integrated Development Environment (IDE). Other Java IDE's (e.g. Netbeans) are applicable as long as both refactoring scripting and aspectJ are supported.

Note that our goal is not to provide an automated full formal verification of Java code but to increase understanding of the security properties enforced by cryptographic imple-
The framework is designed to be extensible: advanced users can define stereotypes, tags, and first-order logic constraints which are then automatically translated to the automated theorem prover for verification on a given UML model. Similarly, new adversary models can be defined.

In particular, the automated translation of UMLsec diagrams to first-order logic (FOL) formulas which allows automated analysis of the diagrams using ATPs for FOL is explained in [5]. In case the result is that there may be an attack, in order to fix the flaw in the code, it would be helpful to retrieve the attack trace. Since theorem provers such as e-SETHEO are highly optimized for performance by using abstract derivations, it is not trivial to extract this information. Therefore, we also implemented a tool which transforms the logical formulas explained above to Prolog. While the analysis in Prolog is not useful to establish whether there is an attack in the first place (because it is in order of magnitudes slower that using e-SETHEO and in general there are termination problems with its depth-first search algorithm), Prolog works fine in the case where one already knows that there is an attack, and it only needs to be shown explicitly (because it explicitly assigned values to variables during its search, which can then be queried).

The user webinterface and the source code of the verification framework is accessible at [2].

### Traceable Security Hardening

Software refactoring actions are changes to the internal structure of the software without changing external behavior. For our tools to (1) establish traceability through refactoring scripts and (2) modularize security hardening advices through aspect-oriented programming, we use scripts to automatically repeat refactoring actions found during program analysis such that these actions become semantics-preserving program transformations aiming at improving the program understanding. Specifically, two virtualization mappings are maintainable by our refactoring scripts. The first mapping is from symbols in the designed message sequence chart to program entities. The second mapping is from program entities to the join point entities that are crosscut by aspect advices. Both mappings, when applied reversely, can trace program entities accurately back to the message sequence charts such that one can reflect implementation changes back to the design. The aim is thus to automate the maintenance of traceability mappings created between the cryptographic protocol in design and their open-source implementations.

### 1. REFERENCES


