1

Introduction

Modern software engineering increasingly involves formal methods to implement security properties. These have to be integrated with traditional software engineering methods, tools, and models, such as process models, programming paradigms (e. g. object-oriented programming (OOP) or aspect-oriented programming (AOP)), or modeling languages (e. g. Unified Modeling Language (UML)). While these techniques are also used to engineer systems in security-critical application domains [Ray et al., 2004; Nguyen et al., 2014, 2015], their major goal is still to ensure the correctness of implementation with respect to functional and non-functional requirements. As a consequence, software specification to some degree always relates to concepts of the implementation environment, be it the type of hardware, the pro-

gramming language or the communication interfaces. Specification of security requirements, on the other hand, is based on a semantically different class of abstractions, such as accesses, information flows, isolated domains or authentic messages. To concentrate such security requirements as a specific family of non-functional properties, the term security policy is normally used. In its most general sense, it describes a set of rules that stipulate how security mechanisms should be implemented and configured in the final system.

While the concept of security policies is long since in use, inconsistent terminology often leads to a rather wide range of possible meanings: It may denote security-related organization and operation of an IT system (security management), a network security policy, or rules for the design and configuration of mechanisms in applications and operating systems, which enforce security properties at runtime. For the scope of this work, we use the latter definition.

Even in this narrower sense, there are two possible dimensions of the term: A security policy may describe the functionality of security mechanisms, but also their individual configuration for a particular application domain and a particular system.

Just as with non-security related requirements, security policies must be gradually refined in a manner of increasing formalism to finally yield an unambiguous software specification to implement, a process which is referred to as modeling. As already mentioned, general-purpose modeling paradigms and tools (such as those of UML) do not generally correspond to the semantical concepts of security requirements – which leads to semantical gaps, which again introduce room for human interpretation. This in turn leads to a departure from specified security properties, crucial for security-critical application domains such as public infrastructure, financial and health service providers, or state institutions. To close such gaps, *security engineering* provides a process orthogonal to software engineering, including specialized formal methods, tools, and models [Sandhu, 1988; Sandhu et al., 2000; Li and Winsborough, 2003; Li et al., 2009; Hicks et al., 2010; Stoller et al., 2011; Ray et al., 2013; Ranise et al., 2014; Shahen et al., 2015], whose goals are to (1.) gradually formalize a security

policy on the semantical level of the application-specific security requirements it should enforce, (2.) verify a security policy against formal security properties. To achieve both goals, formal *security models* are used. In this work, we will use the term *model* for a formalism describing the functionality of security mechanisms, and *model instance* for a description of their configuration (based on a model).

We have illustrated the security engineering process in Fig. 1.1: It starts with security requirements that have resulted from a portion of common software requirements engineering, which we have called security requirements engineering. This results in an informal set of rules stipulated to meet these requirements, the informal security policy. Based on this, the actual modeling steps are conducted: first, a formal representation of security policy semantics is created (a *security model*), which is then analyzed using a plethora of formal methods [Harrison and Ruzzo, 1978; Sandhu et al., 2000; Li and Winsborough, 2003; Stoller et al., 2007; Naldurg and Raghavendra, 2011; Stoller et al., 2011; Ray et al., 2013; Ranise et al., 2014; Shahen et al., 2015; Jha et al., 2008; Jayaraman et al., 2011]. Goal of the model analysis step is to verify the security policy against security properties which define its correctness (in an application-specific sense of, e. g., security goals such as confidentiality or integrity).

We call this phase, which represents the closer context of this work, *model-based security engineering*. Its result is a specification of security-related software functionality in some specification language, such as *Z* [International Organization for Standardization, 2002], *B* [Abrial, 1988, 2006], or *Event-B* [Abrial, 2010] (Pölck [2014] shows an example). This software specification is then, in a traditional software engineering process, foundation for the actual implementation of the security policy.

1.1 Motivation

The general motivation of this work is to methodologically support model-based security engineering. This motivation is based on the

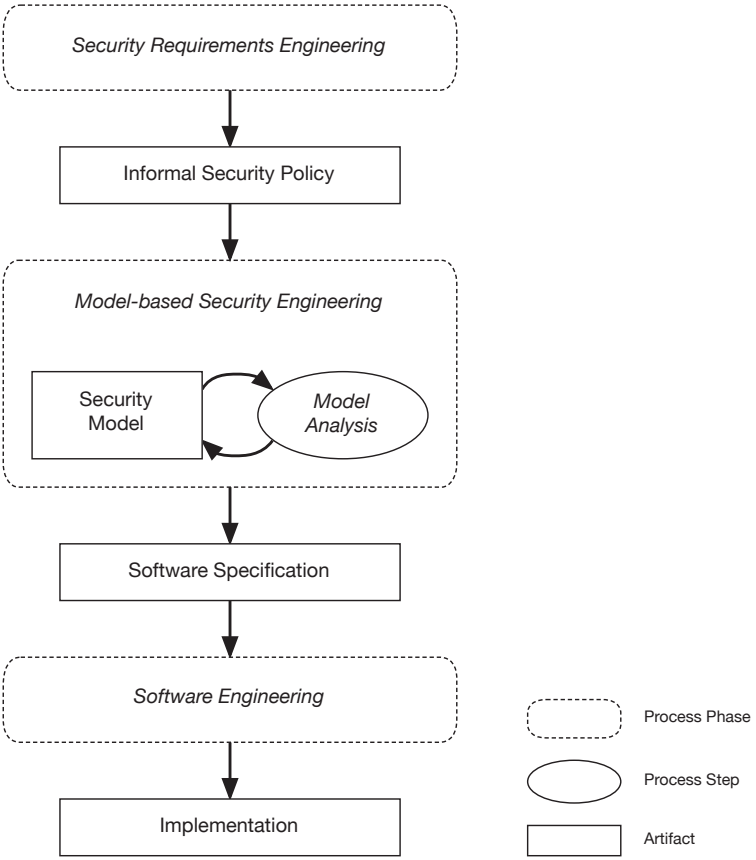

Figure 1.1: General security engineering process.

criticality of the results of model-based security engineering, combined with (very similar to general software engineering) the heterogeneous group of stakeholders involved, all with their own language and understanding of formal modeling: security managers and technology consultants during requirements engineering and policy authoring, model engineers and analysts during policy formalization and verification, security architects during specification engineering and architecture integration, and of course software developers, administrators and future users (clients). In case of security engineering, the different views and languages are critical because of the inherently high potential of human error in the transition between process steps on different levels of abstraction (as an example, take the different meanings of the terms "access control", "security policy", and "safety" in the vocabulary of administrators, model analysts, and security managers).

We are trying to mitigate the inevitable influence of human errors on two paths: First, any formalism to express parts of a security policy should be as precise as necessary, given its use in the current process step it is applied in, and as intuitive as possible, given the level of abstraction of that step. Second, transforming results of one process step to the next should be subject to automatic tool support as far as possible. Both paths have the goal to restrict human engineering decisions in a meaningful way, that helps to recognize and correct faulty or contradictory design decisions on any level of abstraction as early as possible.

1.2 Aspect-oriented Engineering

A strategy to achieve this is to tailor each step in model-based security engineering to either the requirements of a specific family of policy semantics that should be modeled, or of a family of security goals that should be analyzed, which we call *aspects* of security engineering. The point here is that the *process itself* is adapted to a non-functional property, such as representing operating system or

database management system policies, or analyzing consistency or runtime behavior of a security policy. The idea behind such an aspect is to keep each successive step and partial step of model-based security engineering well-defined, small, and monotonic in terms of the degree of formalism.

Fig. 1.2 depicts an aspect-oriented security engineering (AOSE) process. It shows the following steps:

Model Engineering Creating a formal, aspect-oriented model, tailored to the goal of that aspect.

Model Analysis Analyzing the security model, where goal and analysis methods are based on its aspect.

Specification Engineering Creating a formal software specification of the security model, from which a policy implementation may be generated.

Each of these steps may be covered by some aspect of model-based security engineering, to a different extent. In this work we will focus on two examples for such aspects: (1.) the Entity labeling (EL) aspect, that represents typical security policy semantics in the application domains of operating system and middleware systems, and (2.) the model core aspect, that represents model semantics typically needed to analyzed dynamic *safety* properties (which will be discussed in Chapter 3). Note the dashed areas, which include artifacts of process steps outside of model-based security engineering. Given an aspect is tailored to their respective semantics, interpretation and creation of these artifacts, respectively, can be streamlined. We will not cover specification engineering, which we consider out of scope of the two aspects presented and which we leave to future work.

1.3 Contributions

The goal of this work is to substantiate the claim that tailoring of model-based security engineering methods and tools based on as-

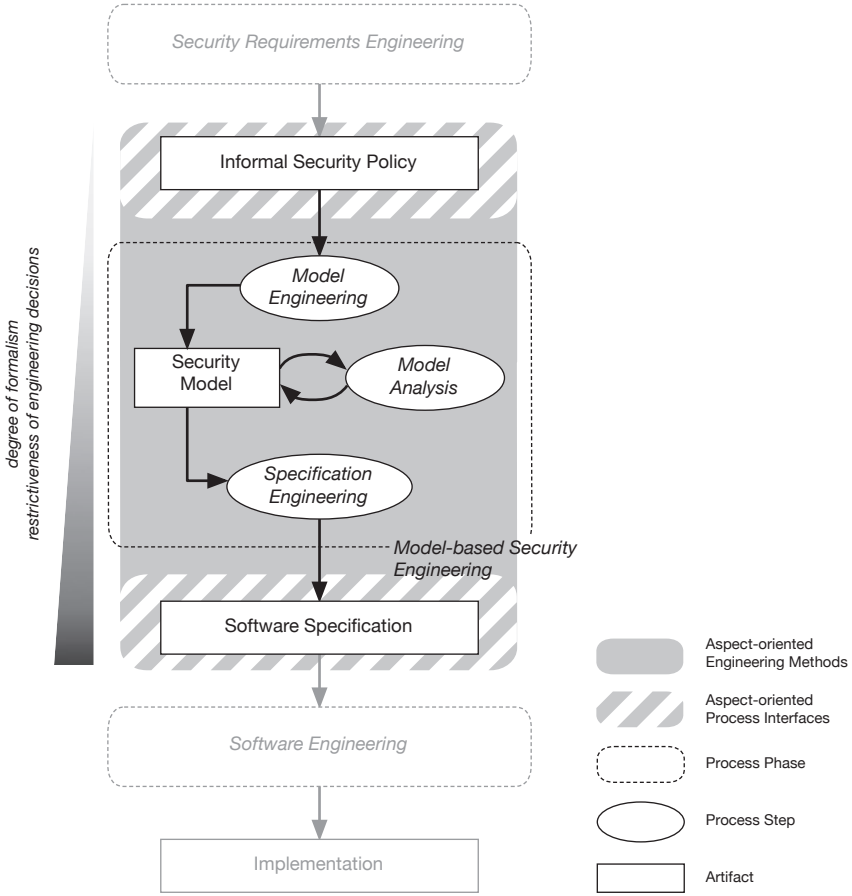

Figure 1.2: Aspect-oriented security engineering process.

pects, which represent non-functional requirements toward the engineering process, reduces the impact of human errors. We try to show this based on two exemplary aspects, covering both classes of non-functional requirements mentioned above, in the following way.

Entity Labeling Aspect To design an aspect that represents a specific family of policy semantics, we have chosen the family of operating system (OS) and middleware (MW) security policies. In contrast to application-level security policies, these system-level policies typically share similar semantical traits that make them predestined for aspect-oriented modeling; as an essential property, they are based on labels on one or more levels of indirection, which are assigned to active or passive entities (such as processes or resources like database tables). These semantics have arisen from deficiencies in traditional OS access control policies, related to their enforcement concept (DAC, cf. Sec. 3.4.1) as well as their policy semantics (no appropriate security-related OS abstractions). As a reaction to this, the paradigm of *policy-controlled operating systems* has become increasingly widespread in all types of application domains [Spencer et al., 1999; Loscocco and Smalley, 2001a; Watson and Vance, 2003; Smalley and Craig, 2013; Russello et al., 2012; Bugiel et al., 2013; Faden, 2007; Grimes and Johansson, 2007], which motivates our choice from a practical standpoint.

The first result of this work is the EL aspect that represents semantical requirements of the family of OS and MW security policies.

Model Core Aspect As an analysis goal to represent, we focus on model safety, a well-investigated and still highly practical family of security properties [Li and Winsborough, 2003; Naldurg and Raghavendra, 2011; Stoller et al., 2011; Ranise et al., 2014; Shahen et al., 2015; Jha et al., 2008; Jayaraman et al., 2011, 2013]. As a formal basis, we build on previous work by Amthor et al. [2011, 2013, 2014]; Amthor [2016, 2017]; Kühnhauser and Pölck [2011]; Pölck [2014]: a uniform, state-machine-based formal calculus to represent dynamic

access control (AC) models ("security model core") and a heuristic safety analysis strategy, implemented for the classical HRU security model. After rewriting the security model core as an aspect, we define patterns that may be used for a more structured and thus less error-prone specification of (potentially complex) model dynamics. These patterns, despite describing model components in the model core aspect, rely on semantics of a model in the EL aspect – which is why we believe these two aspects demonstrate potential synergies of combining aspects for both non-functional classes. A resulting *core-based model* may then be used for heuristic safety analysis.

The second result of this work is the model core aspect and a pattern for synergetic model specification, using both EL and model core in combination.

Heuristic Safety Analysis As a consequence from the motivation of these two aspects, we also describe their usage in model analysis. To this end, we have generalized a heuristic safety analysis algorithm from our previous work [Fischer and Kühnhauser, 2010; Amthor et al., 2013, 2014; Amthor, 2017] and show how it may be tailored to a particular policy, modeled in both the model core and the EL aspects. Moreover, while using the original algorithm in practice, we made a number of observations regarding efficiency and effectivity of safety analysis, which we incorporated into an optimized version of the general framework.

Our third result is a generic, optimized algorithmic framework for heuristic safety analysis, that may be tailored to a core-based model combined with EL semantics.

Application to Security-Enhanced Linux As described so far, the AOSE process only relates to *a-priori* engineering, i. e. realizing a security-critical system from scratch. As an application of our approach to a practical system, we will demonstrate an alternative use case for AOSE: reverse-security-engineering of an existing, policy-

controlled system, with the goal of analyzing its security policy. We will term this approach *a-posteriori* engineering.

We use the Security-Enhanced Linux (SELinux) OS [Loscocco and Smalley, 2001a,b], as an established modern policy-controlled operating system. We create a formal model of the SELinux AC system, called SELX, and show how it can be used to analyze an actual policy. This is also our practical evaluation of feasibility of both streamlined model engineering by the use of EL, and practical tailoring of heuristic safety analysis. We embed our results in a discussion of tool support for each engineering step, which is also ongoing work in line with an integrated model-based security engineering toolkit (*WorSE* [Amthor et al., 2014]).

Our practical results are an SELinux AC model, a family of meaningful safety definitions for SELinux, a heuristical analysis algorithm tailored to SELinux policies and any of these definitions as a falsification goal.

1.4 Organization

This dissertation is organized in seven chapters:

Following this introduction (Chapter 1), we will discuss the state of the art in model-based security engineering in Chapter 2. We will focus on unified modeling approaches for security policies and the integration and interoperation of the general engineering phases (as depicted in Fig. 1.1) based on a paradigm of rigorous formalization.

After this, Chapter 3 surveys the foundations of a modern, model-based security engineering process: Model classes, implementations of AC systems, and model analysis problems. We give an overview of the most important policy semantics and their different semantic paradigms, which motivates the importance of a uniform formalization approach that is useful throughout the whole engineering process.

Chapter 4 will cover the main idea of this dissertation: an aspect-oriented view on security models and their usage. After introducing