FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Evaluation of Model-Driven Security Approaches

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Kresimir Kasal
Matrikelnummer 0026127

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Univ.-Prof. Dipl.-Ing. Dr. A Min Tjoa
Mitwirkung: Dipl.-Ing. Mag. Dr. Thomas Neubauer

Wien, 03.09.2010

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschliesslich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, am 03.09.2010

Abstract

Since our modern society is critically dependent on software systems, software security is rapidly becoming an important issue. For example, companies depend on applications to administer customer data, payment information and inventory tracking. Threats from security breaches range from defeats of copy protection mechanisms to harassments like malicious intrusions into systems that control crucial infrastructure. Software vulnerabilities, arising from deficiencies in the software design or its implementation due to increasing complexity, are one of the main reasons for security incidents. Although the object-oriented paradigm is mostly employed nowadays, principles like encapsulation, polymorphism and inheritance are insufficient and a paradigm change is necessary. Because of its good characteristics in tackling software complexity, model-driven engineering was utilized to develop secure information systems. Many proposals dealing with integrating security and modeling languages followed and were summarized under the term Model-Driven Security. Due to the large amount of available modeling and specification approaches for describing secure information systems, the question arises which method to use for which problem. When intending to apply model-driven security, or at least to analyze a model of a system, it is fundamental to know which security mechanisms and which security requirements can be modeled by a certain technique, and whether an appropriate tool-chain exists. Due to a multitude of available modeling approaches, it can become tedious to identify the most suitable method for solving the problem at hand. There is no common comparison framework to oppose the different methods to each other with regard to security and to indicate the promising approach. This thesis contributes to the research area of software engineering by defining a taxonomy for model-driven security. It evaluates eleven state-of-the-art approaches and classifies them according to the provided taxonomy. Thereby it answers the question which approaches are applicable for solving which development problems, and what specific characteristics these techniques feature. Furthermore, two evaluated methods are applied and security properties which are required for our case study system PIPE are evaluated. In addition, after analysing and validating the system and its security requirements, a solution is provided in case a required security property has been violated. The benefit of the work is a framework for classifying model-driven security approaches and formal specification methods, and an analysis of the PIPE system and its security requirements, including a fix in case a security violation has been found.

Kurzfassung

In unserer modernen, inzwischen von Softwaresystemen abhängig gewordenen Gesellschaft, gewinnt die Sicherheit und Korrektheit der eingesetzten Informationssysteme immer mehr an Bedeutung. Beispielsweise benötigen Unternehmen Applikationen um den Lagerbestand, Kundendaten sowie Zahlungsinformation zu verwalten. Der Erfolg oder Misserfolg eines ganzen Unternehmens kann vom adäquaten Schutz dieser Daten abhängen. Softwaredefekte, welche durch einen fehlerhaften Entwurf oder eine fehlerhafte Implementierung verursacht werden, sind der Hauptgrund für Sicherheitsvorfälle. Entwurfsprinzipien wie Kapselung, Polymorphismus und Vererbung reduzieren die Systemkomplexität, und helfen dabei die Fehleranzahl zu minimieren, jedoch sind diese auf Grund des kontinuierlich wachsenden Funktionalitätsumfangs nicht mehr ausreichend. Aus diesem Grund wurde modellgetriebene Entwicklung als ein vielversprechender, komplexitätsreduzierender und für die Entwicklung sicherer Informationssysteme geeigneter Ansatz (Model-Driven Security) vorgeschlagen. Mittlerweile ist eine Vielzahl an Modellierungs- und Spezifikationsmethoden verfügbar, und es stellt sich die Frage welcher Ansatz bei welchem Problem anzuwenden ist. Um sicherheitsrelevante Informationssysteme mittels modellgetriebener Methoden zu entwickeln oder zu analysieren, ist es notwendig zu wissen welche Ansätze für die Beschreibung welcher Sicherheitsmechanismen und Sicherheitsanforderungen geeignet sind. In der Literatur wurde bisher keine Methode vorgestellt, welche zu einem solchen Vergleich hätte herangezogen werden können. Die vorliegende Arbeit trägt diesem Umstand Rechnung, und stellt eine Taxonomie für Model-Driven Security vor. Des weiteren werden elf dem gegenwärtigen Stand der Forschung entsprechende Methoden evaluiert, und damit die Frage beantwortet, welche der untersuchten Methoden zur Analyse welcher sicherheitsrelevanter Probleme herangezogen werden kann. Das Resultat ist ein Rahmenwerk zur Klassifizierung von Ansätzen aus dem Bereich der Model-Driven Security. In weiterer Folge werden zwei am besten geeignete Methoden (der insgesamt elf evaluierten), auf die Fallstudie PIPE angewendet, um diese im Hinblick auf geforderte Sicherheitseigenschaften zu untersuchen. Wird eine Sicherheitslücke gefunden, so wird gezeigt wie diese behoben werden kann. Das Resultat ist eine formale Analyse des PIPE Systems und dessen Sicherheitseigenschaften.

Danksagung

Ich möchte mich an dieser Stelle bei all jenen bedanken, die mich während des Studiums und der Anfertigung meiner Diplomarbeit unterstützt haben. Ganz besonderer Dank gilt hierbei meinen Eltern, die immer für mich da waren und mir den Rücken gestärkt haben. Zudem möchte ich Dr. Thomas Neubauer für die hilfreichen Anregungen und die enorme Geduld danken, ohne die diese Arbeit nicht möglich gewesen wäre. Auch bei Mag. Johannes Heurix möchte ich mich für die vielen Verbesserungsvorschläge herzlichst bedanken.

Contents

CHAPTER 1

Introduction

Although our society is critically dependent on software systems, these systems are mainly secured by protection mechanisms during operation instead of considering security issues during software design. Deficiencies in the design of software are the main reason for security incidents that result in severe economic consequences for organizations using the software and the development companies. As formal methods have been used by computer scientists for specifying and verifying correct behavior of computer programs in software engineering, formal methods have also been applied to the field of information security. Significant results have been achieved in verifying security properties of cryptographic communication protocols [1]. However, formal methods are rarely used in industry because of its complexity and expensiveness [2]. Thus, model-driven development (MDD) has been proposed in order to increase the quality and thereby the security of software systems [3]. Model-driven development tries to increase the abstraction level of the implementation, in order to make it suitable for formal analysis and to extract the source code from the model after a successful verification. Recently, a combination of model-driven engineering and security has been proposed [3]. In this thesis we present a taxonomy for model-driven security, and thus provide a framework for classifying model-driven security approaches and formal specification methods. Based upon the taxonomy, we evaluate current efforts that position security as a fundamental element in Model-Driven Development, highlight their deficiencies and identify current research challenges. By this way we answer the question which modeling approaches are applicable for solving which problems, and we select suitable methods for the analysis of our case study system PIPE (Pseudonymization of Information for Privacy in e-Health). In a subsequent step, we apply the previously selected analysis methods and validate whether the required security properties are fulfilled by the case study system. Furthermore, we show how to ensure the security requirements in case they are not fulfilled.

1.1 Motivation

Since our modern society is critically dependent on a wide range of software systems, computer systems security is rapidly becoming an important issue [4]. For example, companies depend on applications to administer customer data, payment information and inventory tracking. Not only companies feature a need for secure computer systems but also consumers use software to communicate with friends or families, to check their banking accounts and to search for resources available on the Internet. However, threats from security breaches range from defeats of copy protection mechanisms to harassments like malicious intrusions into systems that control crucial infrastructure. Software vulnerabilities, arising from deficiencies in the software design or its implementation, are one of the main reasons for security incidents [5]. In order to tackle the problem that software systems are getting more complex, general principles like abstraction, modularization and separation of concerns are widely used [6]. Although the object-oriented paradigm is mostly employed nowadays, principles like encapsulation, polymorphism and inheritance are insufficient to tackle the ever increasing complexity of modern information systems, and a paradigm change is necessary [7].

For this reason, as a successor of the Computer-Aided Software Engineering (CASE) approach, Model-Driven Development (MDD) has been suggested to improve the quality of complex software systems [7, 8]. MDD is used to design abstractions, i.e. platform independent concepts which are then translated into more accurate ones which are adjusted to a particular platform. In a further step, such platform specific models are transformed into production code. In such a development process, models and mappings between them have to be maintained, not the generated code. As a further approach to tackle complexity and increase quality, Aspect-Oriented Software Development (AOSD) has been proposed. It is an emerging approach with the goal of promoting advanced separation of concerns (cf. [9, 10]), and allows multiple concerns (e.g., security, logging, persistance) to be expressed separately and unifies them into a complete system in an automated way. Because of good characteristics in tackling software complexity, model-driven engineering was utilised to develop secure information systems. Juerjens (cf. [11]) firstly proposed a combination of model-driven development and security using UMLsec. Subsequently, many proposals dealing with integrating security and modeling languages followed and were summarized under the term Model-Driven Security (cf. [12]). It represents an approach where security is applied together with model-driven architecture [7], and it indicates building secure software systems by specifying models together with their security requirements. In addition to Model-Driven Security, researchers have proposed formal languages, called specification languages, to represent policies, models and system descriptions. Such languages are based on mathematical logic systems, and have as well been applied to the field of information security, for instance for specifying formal security policies and for analysing cryptographic security protocols [13].

A large amount of modeling and specification approaches for describing secure infromation systems are available, and the question arises which method to use for which problem. When intending to apply model-driven security, or at least to analyse a model of a system, it is fun-

damental to know which security mechanisms and which security requirements can be modeled by a certain technique, and whether an appropriate tool-chain exists. As a multitude of available modeling approaches exists, it can be rather tricky to find the most suitable method to solve the problem. In addition, there is no common comparison framework to oppose the different methods to each other with regard to security and to indicate the promising approach. Thus, despite a multitude of available methods and tools, the developer is left alone with the problem of selecting a suitable method.

1.2 Goals and Contributions

This thesis contributes to the research area of software engineering by defining a taxonomy for model-driven security, which we have based upon the comparison framework developed by Khwaja and Urban (cf. [14]). In particular, we have extended the existent framework by adding supplementary aspects such as security mechanisms or verification methods. Furthermore, this work evaluates eleven state-of-the-art approaches and classifies them according to the provided taxonomy. Thereby we answer the question which approaches are applicable for solving which development problems, and what specific characteristics these techniques feature. The benefit of the work is a framework for classifying model-driven security approaches and formal specification methods, applicable by interested researchers and practitioners.

Furthermore, we apply two evaluated methods and validate the identified security properties which are required for our case study system PIPE. In order to identify the required security properties, we apply the so-called 'Threat Modeling' approach (cf. [15]), which is actually a collection of identification patterns that are used to identify threats for information systems. In a subsequent step we determine the required security properties which we derive from the threats. Thereafter, we analyse and validate the system and its security requirements, and we provide a solution in case a required security property has been violated. The benefit is an analysis of the system and its security requirements, and a fix in case a security violation has been found.

Thereafter, the research questions of this work are:

- *Which modeling approaches are applicable for solving which development problems?* In order to answer this question, we evaluate the selected model-driven security and formal-methods techniques by applying a combination of a testing programs and objectives-based evaluation approach (cf. [16]).
- *Which methods can be used to analyse the PIPE system with regards to its security properties?* To answer this question, we consider evaluation objectives which we take from [14] and from security relevant literature [17].
- *Which security requirements are fulfilled by the PIPE system, and which are not?* Here we apply the method which has been selected as the most appropriate one, provide the system model and the corresponding formal security requirements, and let the tool compute the result.

1.3 Outline

This thesis is structured as follows: Chapter 2 presents the background knowledge, necessary for further presentation. In particular, we give a short overview of information security, model-driven development, formal methods and security protocols. Chapter 3 introduces a taxonomy, which we use in Chapter 4 to classify and evaluate model-based security approaches. In Chapter 5, we describe the problem and the system to be analysed. Furthermore, we identify the required security properties which have to be validated. Chapter 6 selects an appropriate analysis method, suitable to verify the identified requirements. In Chapter 7, our validation results are presented. Chapter 8 concludes the work, discusses its outcome and opens questions and possibilities for further work.

CHAPTER 2

Background

This chapter provides the necessary background information for the following presentation of our work. At first, an introduction into information security is given, and some basic terms like security properties, cryptographic algebra and security threats are defined. Afterwards, model-driven engineering in general and its specializations in information security like model-driven security, aspect-oriented modeling and model-based testing are examined. Finally, formal verification techniques and their application in verifying communication protocols are illustrated.

2.1 Information Security

According to [18], information security indicates *preservation of confidentiality, integrity and availability of information*. As additional requirements, *authenticity, accountability, non-repudiation and reliability* may as well be needed. However, the interpretations of these properties vary, as do the contexts in which they arise.

Security Requirements

This section defines common security requirements. However, as interpretations of these requirements can depend on the context in which they arise, we provide definitions which are sufficient for our purposes.

Confidentiality

This security property represents the concealment of information or resources [17]. It is a requirement *that information is not made available or disclosed to unauthorized individuals, entities, or processes* [18]. In our context, confidentiality of data means that the data should only be read by legitimate parties.

Integrity

Integrity demands that the accuracy and completeness of assets is protected [18]. It refers to the trustworthiness of data and resources, and it requires a protection against improper or unauthorized changes [17]. In our context, integrity is the property of preventing unauthorized modification.

Authenticity

This security requirement demands that in case some information claims to be from a certain party, it was indeed originated by that party. Message authenticity means that a piece of data can be traced back to its original source [3]. Entity authenticity, on the other hand, means that a participant in a protocol can be identified. In our context, entity authenticity is the appropriate interpretation of this property.

Non-repudiation

This property requires that a party in a dispute can not deny the validity of a statement or contract successfully. That is, a proof for the action is given. We distinguish between the two properties non-repudiation of origin (NRO), and non-repudiation of receipt (NRR). NRO demands for evidence that the sender has sent the message. Analogous, the NRR property requires evidence that the recepient has received the transmitted message.

Availability

This property represents the ability to use the designated resource or information [17]. It is the requirement of *being accessible and usable upon demand by an authorized entity* [18]. The aspect of availability that is relevant to security is the fact that an adversary could intentionally enforce denial of access to desired resources by making them unavailable. This definition is sufficient for our purposes.

Cryptography

Methods of information security heavily rely on cryptography, which can be seen as a science of hiding information. If a person (called sender), wants to send a message containing sensitive information to another person (called receiver), then the sender wants to send the message in such a way that an eavesdropper cannot read it. Thus, in order to make the information unreadable for everyone except the receiver, the message has to be converted into unintelligible strings. A message is plaintext if it can be read by anyone. The process of transforming a plaintext message into unintelligible strings is called **encryption**. An encrypted message is called ciphertext, and the process of transforming ciphertext back into plaintext is called **decryption** [19]. This is shown in figure 2.1.

Figure 2.1: Encryption and decryption [19].

In general, there are two associated mathematical operations, one for encrypting the plaintext and the other for decrypting the ciphertext. In modern cryptography, both the encryption and decryption functions require a key, see figure 2.2. This key might be any value, usually bitstrings (e.g. 256 bit length in the case of AES). In modern cryptography, security is based on the key (or several keys), and not on the specific details of the algorithm [20].

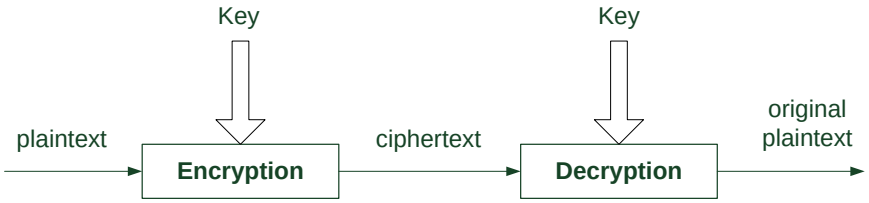

Figure 2.2: Encryption and decryption with a symmetric key [19].

In general, two types of cryptographic algorithms are used, **symmetric** and **asymmetric** key algorithms. Symmetric algorithms use keys where both of them, the encryption key and the decryption key, can be derived from the other [19]. Asymmetric algorithms, in turn, are designed in such a way that both keys cannot be calculated or derived from each other (at least in a reasonable amount of time). Asymmetric algorithms are often called 'public key' algorithms. The reason is that the encryption key does not need to be protected and can be made public. Only the person who is in possession of the decryption key will be able to read the message. Therefore, the encryption key is called **public key**, and the decryption key is called **private key** [19].

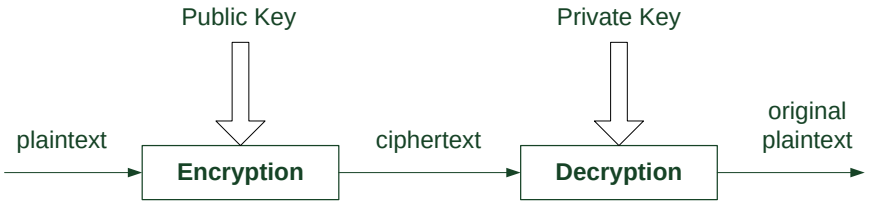

Figure 2.3: Encryption and decryption with different keys [19].

Threats to Security

Howard [15] defines a threat as the attacker's objective. It is the action which the adversary might undertake in order to do harm to a system. Microsoft uses a threat taxonomy called STRIDE [15] to identify various threat types, which are considered from the attackers perspective. These are grouped into six categories, described in [15]:

Spoofing Identity

Spoofing threats allow the attacker to pretend to be something or somebody else. In case an adversary illegally accesses and uses authentication information of another user, username and password for example, then identity spoofing occurs. Entity authenticity is required to prevent from spoofing threats.

Tampering

These threats involve malicious modifications of data or code. An example of tampering is unauthorized change of data, such as relational tables stored in a database. Mechanisms that ensure the integrity property are required in order to prevent tampering.

Repudiation

In case an attacker denies having performed a certain action without other parties having evidence to prove otherwise, a repudiation threat occurs. A system is resistant to repudiation threats if it satisfies the non-repudiation requirement.

Information Disclosure

This threat occurs in case information is exposed to unauthorized persons or processes. Information disclosure can be prevented by mechanisms that ensure the confidentiality requirement.

Denial of Service

Denial of service (DoS) attacks deny or degrade the availability of a service to authorized users.

Elevation of Privilege

Elevation of privilege threats occur when a user gains increased capabilities. For instance, this threat occurs in case an unprivileged user gains superuser access and, in consequence, is able to compromise or destroy the entire system.

2.2 Model-Driven Development

Progress in programming languages and development platforms has increased the level of abstraction available to developers [8]. Neverthless, although the level of abstraction was raised, the complexity of developed systems grew faster. For this reason, methods were required that

provide a higher level of abstraction than object-oriented methods do. In MDD, models are considered as the most important elements for software development. Instead of applying too expressive general-purpose notations, formulating domain concepts and design intent lies in the focus of the MDD approach. On the one hand, Model-Driven Development is based on applying domain-specific modeling languages (DSMLs) which formalize the underlying data model, the behavior and the requirements of a particular problem domain [8]. Examples for such domains are embedded real-time systems, financial services and information systems. DSMLs are described by using metamodels, which in fact are models of models, and define the concepts in the problem domain, associations among these concepts and the corresponding constraints. On the other hand, transformation engines and generators build the second branch that MDD is based on [8]. These are needed to analyse the models and to transform them into various artifacts such as source code, test cases or documentation. The ability to generate artifacts from models is very helpful in practise since it assures that the generated artifacts and the models remain consistent. In general, among the principle of abstraction, the main goal of model-driven development is to generate implementations in a preferably automated way. In literature, this software development process is often characterised as 'correct-by-construction' [8].

Model-Driven Architecture

Model-Driven Architecture (MDA) is an approach for software development, in fact a variation of MDD, which has been proposed by the Object Management Group (OMG) in [21]. MDA defines three viewpoints of a system, the Computation Independent Model (CIM), the Platform Independent Model (PIM) and the Platform Specific Model (PSM). The Computation Independent Model (CIM) indicates an abstraction level which is focused on the system's requirements and its context, without considering how the information is processed [7]. On the next level, the Platform Independent Model (PIM) abstracts from details concerning the programming language and the corresponding environment. It models business functionality, but without including the implementation specific details. The PIM is used by software architects and designers in order to specify the operational capabilities of the system, and to split the application into a number of cooperating components [7]. For instance, the PIM does not specify whether the system is implemented for the .NET or the J2EE platform. This platform specific information is added to the Platform Specific Model (PSM), which is used to generate the production code. Model transformation is crucial for the MDA approach [7], since a CIM has to be transformed into a PIM, and a PIM has to be tranlated into a PSM, which is then transformed into code. In order to make the transformation possible, metamodels are needed which formally describe the structure and composition of models, and thus permit definitions of matchings and correspondencies between the entities and concepts described in different metamodels. Simplified, metamodels allow definitions of transformation rules between different levels of abstraction (e.g., transformation from PIM to PSM).

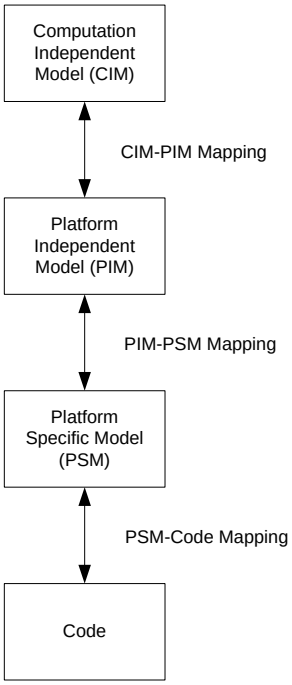

Figure 2.4: The principle of MDA [7].

Model-Based Testing

Model-Based Testing [22] is *software testing where test cases are derived from a model that describes the functional aspects of the system under test (SUT)*. The model is usually an abstract representation of the SUT's desired behavior. The derived functional test suite resides on the same level of abstraction as the model. In case the model is not executable, an executable test suite must be generated, which can be achieved by deriving it from the abstract test suite. For this purpose, a transformation ruleset is needed. Thus, models should be formal enough to allow algorithms to derive test cases from them [22]. We do not consider testing non-functional properties.

Utting et al. describe the following process of model-based testing [23]. At first, the model of the SUT is extracted from the requirement documentation. This model specifies the system's required behaviour, and the step is performed manually by a developer. Then the test selection criteria are defined, which describe characteristics of a test suite. They can be associated with a system's functionality (requirements-based test selection criteria), with the structure of the model (state or transition coverage) or with stochastic characteristics (i.e., randomness) [23]. Afterwards, test case specifications are derived from test selection criteria, since they state the test selection criteria precisely and make them operational. For instance, if state coverage has been chosen as test selection criteria, then a test suite can be generated automatically since enough information has been provided to specify the particular test cases. However, the gener-

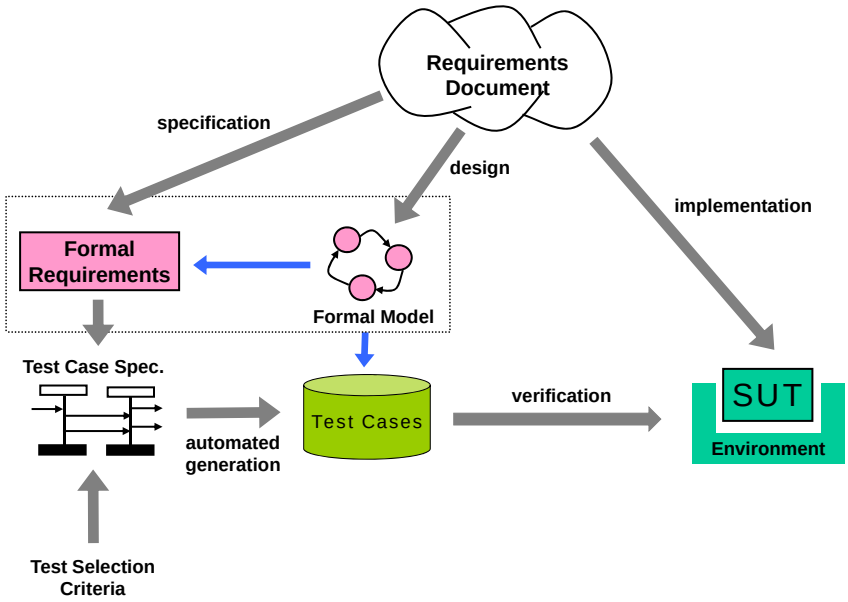

Figure 2.5: The principle of model based testing [23].

ated test suite may not be executable since the model and the SUT may not reside at the same abstraction level. In such a case, the generated test cases have to be transformed and made executable. Afterwards, the test suite is executed. This step involves two stages. At first, the SUT is supplied with test data, and the output is recorded. Then, the actual result is compared with the expected result. Depending on the outcome of the comparison, the test case either succeeded or failed. The principle of model based testing is depicted in figure 2.5.

Aspect-Oriented Modeling

Aspect-Oriented Modeling (AOM) [24] is a generalization of the idea of Aspect-Oriented Programming (AOP). AOP was firstly proposed in [9] in order to clearly capture all important aspects of system's behaviour. As the authors based their work on the belief that a single abstraction framework (e.g., procedures, constraints) would be inadequate for expressing different aspects of system's behaviour (since each aspect has it's own 'natural form'), they concluded that each aspect should be programmed in its most natural domain-specific language. For example, the way error handling, logging, or security is handled is often identical over multiple domains. According to the AOP paradigm, these aspects should be contained in different modules. Subsequently, after the aspects and the main system's functionality have been programmed, these separated programs would be woven together in order to produce executable code. In general, the essence of AOP are modularization and separation of concerns. Separation of concerns is a

design pattern which considers a problem from different perspectives, involving different views on a system which should be separated in order to tackle the complexity. Actually, this principle states that each concern of a given design problem should be mapped to one module in the system. Unfortunately, this can not be achieved with all system views. Concerns exist that can not be easily separated and therefore have to be mapped to many modules. These concerns are called *crosscutting concerns*. They are not a result of bad design, it is a characteristic of such views that they cannot be cleanly decomposed from the rest of the system, in both the design and implementation. Aspect orientation provides explicit abstractions for representing crosscutting concerns, referred to as aspects. That is, a given problem is decomposed into concerns that can be localized into separate modules and concerns that are crosscut over a set of different modules. Afterwards, in a so called 'Pointcut-specification' the points that the aspect crosscuts, are specified. During weaving, core functionality modules and aspects are composed in order to obtain the complete system. Analogue to AOP, AOM consists of several aspects and a single primary model. An aspect model describes the design of the aspect, while the primary model addresses the core functionality of the system. Also here, weaving rules are used to compose the separated aspect models and the primary model. The composition is done before the implementation or code generation is started.

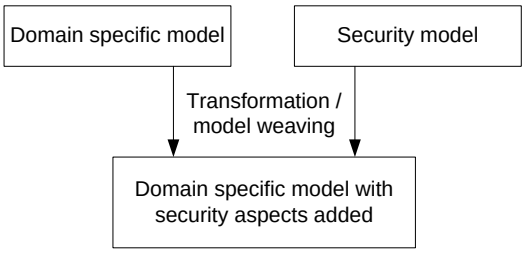

Figure 2.6: The principle of AOM [25].

2.3 Formal Methods

[26] describes formal methods as *mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems*. The phrase 'mathematically rigorous' describes the fact that the used specifications have well-defined syntax and semantics and are based on a mathematical logic system. Thus, formal verifications are deductions in that logic system, which implies that each deduction is a consequence from an inference rule and can automatically be checked by a computer. However, this is rarely done in practice because of the huge complexity of real-world systems [26]. Several approaches are considered useful in order to overcome the large state spaces associated with real-world systems. Butler [26] enumerates the following:

- *Apply formal methods to requirements and high-level designs where most of the details are abstracted away*

- *Apply formal methods to only the most critical components*
- *Analyze models of software and hardware where variables are discretized and ranges drastically reduced.*
- *Analyze system models in a hierarchical manner that enables 'divide and conquer'*
- *Automate as much of the verification as possible*

In general, formal verification techniques comprise three parts (cf. [27]). The first part is made up of a framework for modelling systems, typically a description language. The second part is a specification language in which the properties to be verified are expressed. And third, a method is required which allows to verify whether the given specification is satisfied by the description of a system.

Although all formal methods rely on the underlying structure of some mathematical logic system and the proof theory of that logic, there is no single best 'formal method'. Butler [26] writes that *each application domain requires different modeling methods and different proof approaches*. Current trends have divided these techniques into proof-based and model-based techniques [17]. Proof-based approaches define two sets of formulas. The first set describes the system (premises), while the second set represents the properties or requirements that have to be proved (conclusion). Proof-based techniques rely on finding a set of intermediate formulas that allow the verifier to reach the desired conclusion beginning from the premises [17]. In model-based approaches, the system is represented by a model for an appropriate logic. The properties are represented by formulas, and the verification method consists of validating whether the required properties are satisfied by the given model. Thus, model-based approaches are potentially simpler than the proof-based ones, since only a single model is verified and not a potentially infinite class of them [27].

Proof-based Approaches

In theorem proving, both the model of the system and its requirements are expressed as formulas in an underlying mathematical logic calculus. This calculus is defined by a formal system which comprises axioms and inference rules. Clarke et al. [28] define theorem proving as *the process of finding a proof of a required property from the axioms of the system*. In contrast to model checking, theorem proving can be used on systems with infinite state spaces, since it relies on techniques such as structural induction which are used to generate proofs over infinite domains [28]. In general, theorem proving is an interactive procedure. This implies that, as interaction with a human user is required, theorem proving can be a very slow and error prone process. On the other hand, whilst searching for the proof, valuable insight into the system and its required properties can be gained due to interaction [28].

Model-based Approaches

Model-based techniques do not prove a formula from a set of axioms, but check whether a model satisfies a formula. Typically, model-based verification tools are automatic, with almost no user

interaction. In this section, two model-based approaches are presented. The first of the methods is called model checking. It is based on temporal logic, and intended to be used for concurrent, reactive systems. The second of the two approaches is represented by languages like OCL (Object Constraint Language), Z or Alloy and the appropriate tools required to analyse the specified models. This set of languages and corresponding tools we will call model validation techniques. We explicitly point out that in this context no theorem provers are meant, but model verifier tools which are available for the aforementioned languages. The way models are used in the latter approach is slightly different from model checking. Systems are modeled as collections of variables, with invariants that the system has to satisfy and operations performed by the system, which are declared by pre- and post-conditions. In addition, the approach allows for first-order logic expressions, whereas model checking focuses on verifying temporal properties only.

Model Checking

The concept of model checking refers to a set of techniques for automatic analysis of finite-state reactive systems. It is the process of verifying whether a specified logical formula is satisfied by a given model. That is, the system is tested for conformance with a specified requirement or property. In general, model checking can be applied to all kinds of logics and their models. For instance, one could validate whether a given propositional logic model satisfies a required propositional logic formula. However, great efforts have been undertaken in order to algorithmically verify finite-state concurrent systems. In this context, the system to be verified is presented as a Labeled Transition System, which is also known as a Kripke Structure. A Kripke Structure is basically a graph, representing the system's states and the transitions between these states. For each state, a certain set of logical formulas or properties hold true. Huth and Ryan [27] define a labeled transition system as

> *a set of states S, endowed with a transition relation $\rightarrow$ (a binary relation on S), such that every $s \in S$ has some $s' \in S$ with $s \rightarrow s'$, and a labeling function $L : S \rightarrow \mathcal{P}(Atoms)$*

Modern model checkers support higher level description and specification languages like Promela or the SMV language. Such representations are comfortable for the engineer who uses the system, but in fact the model checker transforms the high-level description into a transition system, an example is depicted in figure 2.7.

First, system requirements are expressed as temporal logic formulas (LTL or CTL for example, see [27]). Then, the Kripke structure corresponding to the system's model is traversed by efficient algorithms in order to perform the analysis and to verify whether the specification holds or not. In [27], Huth and Ryan give a detailed description on both kinds of temporal logic applied in model checking:

> *The idea of temporal logic is that a formula is not statically true or false in a model, as it is in propositional and predicate logic. Instead, the models of temporal logic contain several states and a formula can be true in some states and false in others.*

```
VAR
  request : boolean
  status  : {ready, busy}

ASSIGN
  init(status) := ready;
  next(status) := case
                    request : busy;
                    1       : {ready, busy};
                  esac;
```

Listing 1: Example model in SMV.

> *Thus, the static notion of truth is replaced by a dynamic one, in which the formulas may change their truth values as the system evolves from state to state. In model checking, the models M are transition systems and the properties ϕ are formulas in temporal logic. The model checker outputs the answer 'yes' if $M \models \phi$ and 'no' otherwise.*

As already mentioned, a set of atomic propositions is associated with each node. Each node represents a state, and each edge represents a transition between these states in the corresponding system. Atomic propositions which are mapped to a certain state represent the properties which hold in that state. According to [27], the problem can be stated formally as follows: *given a desired property, expressed as a temporal logic formula p, and a model M with initial state s, decide if $M, s \models p$.* Hardware verification by the means of model checking is already widely used while the verification of software still has not expirienced a breakthrough, mainly because of the state explosion problem. As in model checking the proof is made by an exhaustive exploration of the state space of the finite-state automaton, the technique is limited by the size of the models. For the reason of large state spaces in non-trivial software systems, formal verification of software by the means of model checking is still not widely used. However, model checking has gained popularity because it generates a counterexample for a property that is not satisifed. The counterexample pinpoints to the source of error, and thus answers the question why the model does not satisfy the specification. Usually, the model is given as a source code description in an automata description language. An example taken from [27] is given in listing 1.

The program stated above has two variables, `request` of type `boolean` and `status` of enumeration type `{ready, busy}`, which are defined in the `VAR` part of the program. State transitions, contrariwise, are described in the `ASSIGN` part of the program. In the example code listed above, the initial and subsequent values of the variable `request` are not determined, which models that these values are determined by an external environment. This under-specification of `request` implies that the value of the variable `status` is partially determined: initially it is `ready`, and it becomes busy whenever `request` is true. If `request` is false, the next value of `status` is not determined. The phrase `'case 1'` signifies the default

case. A finite automaton which is corresponding to the source code listed above is depicted in figure 2.7.

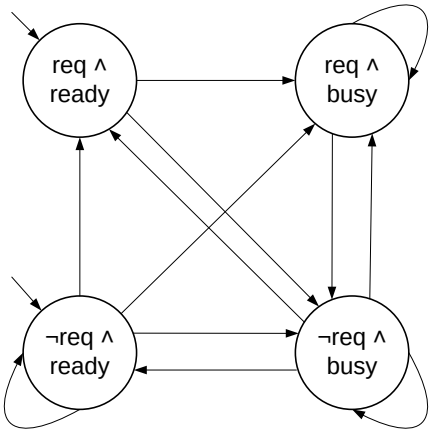

Figure 2.7: Model corresponding to the SMV program.

Model Validation Techniques

Methods belonging to this category include declarative modeling languages based on first-order logic, supported by appropriate analysis tools. Languages like OCL (Object Constraint Language), Z and Alloy belong to this category. Although the methods seem to have similar targets, they differ in terms of the syntax of the language and the analysis [29]. For instance, a combination of UML and OCL can be used to precisely describe the system's static structure and its dynamic behavior, whereas Z (which is a formal specification language based on set theory, first-order predicate logic and lambda calculus) is used for modelling and reasoning about computing systems. Alloy, on the other hand, which is based on first-order relational logic and similar to OCL, is designed for expressing structural constraints and behavior of software systems. Furthermore, Alloy models are amenable to a completely automated analysis, which is not the case with OCL and Z.

2.4 Security Protocols

Security protocols (or cryptographic protocols) can be described as distributed algorithms which ensure security properties in a hostile environment. They represent rules or conventions which define the exchange of messages between communication participants. For example, security protocols can be used for key agreement or entity authentication. They rely on cryptographic primitives, such as symmetric and asymmetric encryption. However, designing correct security protocols does not end with the correct implementation of cryptographic primitives. Given the wide range of operations which an attacker can use in order to compose an attack, it is difficult for the designer to reason intuitively about possible vulnerabilities. Probably the best known

example is the Needham-Schroeder public-key protocol [30], for which an attack was found about 17 years after its publication [31]. Thus, both formal and informal approaches have been proposed in order to support researchers and engineers during protocol design. In [32], the authors presented prudent engineering principles for the design of cryptographic protocols. For instance, here is one of the principles:

> *If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name in the message.*

In the initial proposal of the public-key Needham-Schreoder protocol, this principle was ignored, which has led to a security vulnerability allowing for a Man-In-the-Middle attack [31]. Even if they are very useful, the proposed principles are neither necessary nor sufficient for designing correct protocols.

In [33], for first time formal methods have been suggested in order to validate and verify security protocols before these are put to use. The authors contributed the so called Dolev-Yao model, which comprises two main ideas:

1. The attacker has complete control over the network. This means that he can insert, delete and intercept transmitted messages.
2. The cryptographic primitives are perfect.

Shortly later, researchers applied general-purpose formal methods and state exploration techniques to tackle the problem. In addition to these 'conventional' methods, Burrows et al. applied a logic of knowledge and belief to the problem [34]. The so-called BAN logic comprises possible beliefs and inference rules which are used to derive new beliefs from the old ones [1]. The authors showed in their work that it was possible to apply the logic in order to find vulnerabilities in security protocols. However, in general, belief logics are weaker than state exploration techniques, since they operate on a higher abstraction level [35]. As state exploration systems have improved over the last years, researchers have focused their attention to applying both model checking and theorem proving techniques to tackle the problem [1].

CHAPTER 3

A Taxonomy for Model-Driven Security

This section identifies and describes several dimensions in model-based security. The identified dimensions are orthogonal, but still can affect each other. For example, if the modeling paradigm is aspect-oriented, one of the artifacts that the method requires is a set of transformation rules that are needed for model weaving. The proposed taxonomy is influenced by the comparison framework developed by Khwaja and Urban (cf. [14]). The framework was intended for the evaluation of specification techniques and was already used by Villarroel et al. (cf. [36]) to evaluate development methods for secure information systems. Nevertheless, Khwaja's and Urban's comparison framework did not cover aspects like security mechanisms that can be modeled by a specific technique, it did not classify the distribution of modeled systems, the artifacts that have to be provided by the modeler or the applied modeling paradigm. Furthermore, there was no differentiation between several possible dimension instantiations that can occur in system verification. Therefore, these issues are handled in the proposed taxonomy. The taxonomy will provide a classification method that can easily be applied by a practitioner when comparing model-driven security methods in order to choose the appropriate one. In particular, we differentiate model-driven security by the means of modeling paradigm which tells us how the system is modeled, by the means of artifacts which the modeler has to provide, the formality of the modeling method, the verification method which is used to verify the modeled system, as well as the executability of the provided models. Furthermore, we distinguish the capability to model distribution, the granularity of the modeled system, and the security mechanisms which can be described by using the method in question. As well, we consider the application domain as well as the available tool support.

3.1 Modeling Paradigm

This dimension is concerned about the modeling paradigm. In case of model-driven security, two alternatives are possible. On the one hand, there are *single, possibly hierarchical models* (e.g., a hierarchical model is composed of submodels that in turn are composed of sub-submodels and so on). In this scenario, crosscutting concerns are modeled in each place where they are needed. For example, in each module where access control is required, access control mechanisms have to be modeled. For this reason, there is a significant amount of redundancy in the overall model. On the other hand, there are models conforming to the aspect-oriented development paradigm. In these models, there is almost no redundancy. Crosscutting concerns are described in a separate model and are subsequently weaved (in other words, integrated) with the primary model. In aspect-oriented modeling, weaving rules are needed that determine the manner in which models are integrated together to build a complete system.

3.2 Artifacts

There are three forms of artifacts: (i) *Static models* describe the static structure of a system. For instance, a class diagram consisting of classes, associations between those classes and constraints on them is an example of a system's static structure. (ii) *Dynamic models* describe the behavior of a system. Examples for such models would be interaction diagrams, collaboration diagrams, state charts, or petri nets. (iii) *Transformation rules* are either required for transformating models in the MDA approach, or for aspect-weaving when applying aspect-oriented modeling. Transformation rules are an important artifact in model-driven development and model-driven security. Possible instantiations in this dimension are subsets of the three mentioned forms of artifacts, e.g., class diagrams modeled in the UML modeling language and transformation rules specified in the ATL transformation language.

3.3 Formality

A formal system description is preferable, since a high level of formality allows for a definition of accurate, unambiguous and verifiable models and specifications. Higher levels of formality are more eligible, but also specifications of lower formality levels are of great practical value. Design patterns, for instance, can be seen as a semi-formal approach to specify the system's functionality since they precisely describe how certain entities have to be assembled or how they have to interact in order to form the required system. Metamodels, on the other hand, show how system's valid models are built. They can be seen as grammars that precisely describe how a system is put together. Automata, state machines, several logics and calculus systems are examples for highly formal modeling and specification techniques.

3.4 Distribution

Here we consider whether an application is distributed. If so, we distinguish how these components are distributed and how they work together. The modeled system can be categorized

as a single process or a multiple process (thread) system. In case we deal with multiple processes, these can be distributed over multiple machines, and can act as autonomous, possibly mobile agents. Furthermore, we differentiate whether systems consist of distributed objects and procedures (e.g., CORBA, RMI). Client and server architectures, peer-to-peer architectures, space-based architectures or multiple agent systems are examples for possible instantiations in this dimension.

3.5 Granularity

A system can be viewed from several levels of abstraction. When lowering the level of abstraction, the amount of details to cope with increases, and the complexity grows. Possible instantiations for this dimension are, for example, components and classes. In this context, a component is a part of the system from the architectural point of view. As the system can be seen as a composition of interacting subsystems (where each subsystem consists of other parts, for example classes, functions etc.), a component can be described as such a subsystem. In general, a component can consist of other components, classes or functions. Nevertheless, the system can also be described on a finer granularity level. For instance, the system can also be defined as a set of classes, or even more detailed and more complex, as a set of interacting methods and functions.

3.6 Executability

Here we differentiate whether the system's model is executable or not. In case the model is executable, there is no need for enriching the model with additional information. There is enough information contained in the model, and the system can be verified without execution (in case the state space of the provided model can be handled). If the model is not executable, additional information has to be provided during the transformation process. In such a scenario, generating test cases from the models and making them executable is a plausible technique for validating the system.

3.7 Verification

In this dimension, we consider several methods for validating, verifying and testing the system. Manual testing is one such method. As it is error prone and tedious, automated test case generation is affordable. When dealing with models, model-based testing can be applied. We differentiate two scenarios when generating test cases from the models [23]: The first of the two scenarios considers a single model which is used to generate both, production code and the corresponding test cases. The second scenario, on the other hand, comprises a testing-specific model which is built upon the specification documents, while the system under test is built manually. A further method for verifying the system is model checking. It is a process of verifying whether the model satisfies a specific requirement. In model checking, a model is a finite automaton and a requirement is a temporal logic formula. By applying efficient searching strategies, the model

is checked in a brute-force manner. Therefore, the run-time depends on the size of the provided automaton, which can be very large in real-world systems. A further method for verification is theorem proving. In automated theorem proving a computer program verifies whether a theory (system specification) entails a logic formula (requirement). In other words, the program verifies whether a requirement (logic formula) is satisfied by a system specification (theory).

3.8 Tool-Support

Here, techniques for modeling secure systems are categorized in terms of tool-support. The user can be supported in modeling the system, generating code, verifying whether the system specification is syntactically correct and consistent etc. For example, the tool could check whether the specified model is consistent with its metamodel. A developer could be assisted in validating and verifying the system, e.g. a tool could check whether a subsystem is consistent with its specification (verification), and also whether the whole system fulfills the user requirements (validation). Instantiations in this dimension are combinations of mentioned tool-support possibilities.

3.9 Applicability

In this dimension we differentiate between several application domains for which systems can be specified by applying a particular technique. Examples for application domains are information systems, web-applications, e-commerce systems, embedded systems etc.

3.10 Security-Mechanisms

In this section, security modeling techniques are categorized according to security mechanisms (and thus indirectly, the security requirements) that can be represented and modeled by a particular method. Possible instantiations for security aspects are access control, security protocols, intrusion detection mechanisms etc.

CHAPTER 4

Comparison of Model-Driven Security Approaches

This chapter presents an excerpt from several model-driven security and formal method approaches. In particular, we start with evolved approaches like UMLsec and SecureUML, which we have selected because of their level of popularity within the model-driven security research community. Then we present the interesting idea of specifying secure systems in an aspect-oriented manner, which is in our opinion a very promising concept for modularizing crosscutting concerns. Afterwards, some general and special-purpose techniques in formal methods are presented. In particular, we have selected the Symbolic Model Verifier since it is a well-known model checking tool, Alloy because it offers a simple but powerful modeling language, and we have selected the AVISPA protocol analyser since it has gained huge popularity within the security protocol verification community during recent years. Then we classify the presented approaches according to the taxonomy which we have introduced in the last chapter. By providing the classification, on the one hand, we give an overview for the interested practitioner and we pinpoint to what has to be considered when applying a particular method. On the other hand, we intend to show advantages and disadvantages of the presented techniques and to show where further work is needed.

4.1 UMLsec

UMLsec considers a UML extension in order to enhance the language with security relevant aspects. It is a very generic and powerful technique. Its specification and modeling capabilities are based on UML. UMLsec enhances UML's expressiveness by applying security related stereotypes, tags and security constraints. According to the author, these (stereotypes, tags and constraints) *are used to encapsulate knowledge on prudent security engineering and make it available to developers which may not be specialized in the security field* [3]. UMLsec is a methodology that allows specifying requirements regarding confidentiality, integrity, non-repudiation,

secure information flow and access control. The majority of UML diagrams are used in order to model the mentioned system's security properties. In general, the approach conforms to the single, hierarchical model paradigm. That is, a system is composed of subsystems, these are composed of further subsystems or components that can be modeled by class diagrams, state charts and so on. Nevertheless, the technique could also be applied in order to model aspects and system's crosscutting concerns separately. In such a case, the modeler would have to provide transformation (weaving) rules that specify and determine how specific models have to be integrated [37]. By applying UMLsec, the system can be described on several levels of granularity. Even if there is no consensus whether UML is an Architecture Description Language (ADL) or not [38], in general it can be used to model system architecture in most scenarios. Furthermore, because a formal semantics was provided for UMLsec [3], behaviour of interacting components can formally be analysed. Distributed systems can also be specified [39], and as an example for analysing security properties of a distributed system the author analysed the TLS (Transport Layer security) security protocol [3].

In general, *use case diagrams* show system functions and interactions between users and the computer system. In UMLsec, they are as well used to express security requirements [3]. That is, they can be annotated with stereotypes that represent certain security requirements. *Activity diagrams* are usually used to model workflows and to describe use cases with a higher precision. In UMLsec, they can as well be utilised to express the security requirements more accurate. For instance, the control flow which is modeled by activity diagrams could be dependent on security requirements, which a user could model by associating adequate constraints with the edges in the diagram. *Deployment diagrams*, on the other hand, describe the system's physical layer. In UMLsec, deployment diagrams are also used to validate whether the logical level security requirements are enforced and satisfied on the physical layer, or whether supplementary security measures and mechanisms (i.e. encryption) are needed. *Statechart diagrams*, which describe an object's behavior and the state changes throughout its life, are used to specify security requirements on the resultant state sequences. *Sequence diagrams*, on the one hand, can be used to ensure the correctness of security-critical interactions between objects, like security protocols for example (authenticity, confidentiality). *Class diagrams*, on the other hand, which describe the static structure of the system, are used to ensure that exchange of data obeys certain security levels (secure information flow). In summary, the modeler has to provide static and dynamic models in order to describe the required system.

By applying mentioned diagrams, UMLsec can be used to express standard mechanisms and concepts from multi-level secure systems and security protocols. In fact, the proposed language is able to express many of security properties needed in real-world applications. Secrecy (confidentiality), integrity, secure information flow, non-repudiation, data authenticity and entity authenticity are security requirements that can be modeled by applying UMLsec [3]. In order to be able to specify complete system specifications, Juerjens provided a formal foundation for UML subsystems that incorporates a formal semantics of diagrams contained in the subsystem. That is, the author provided message-passing between components specified in different diagrams, which enables the modeler to compose systems from sets of subsystems and allow them

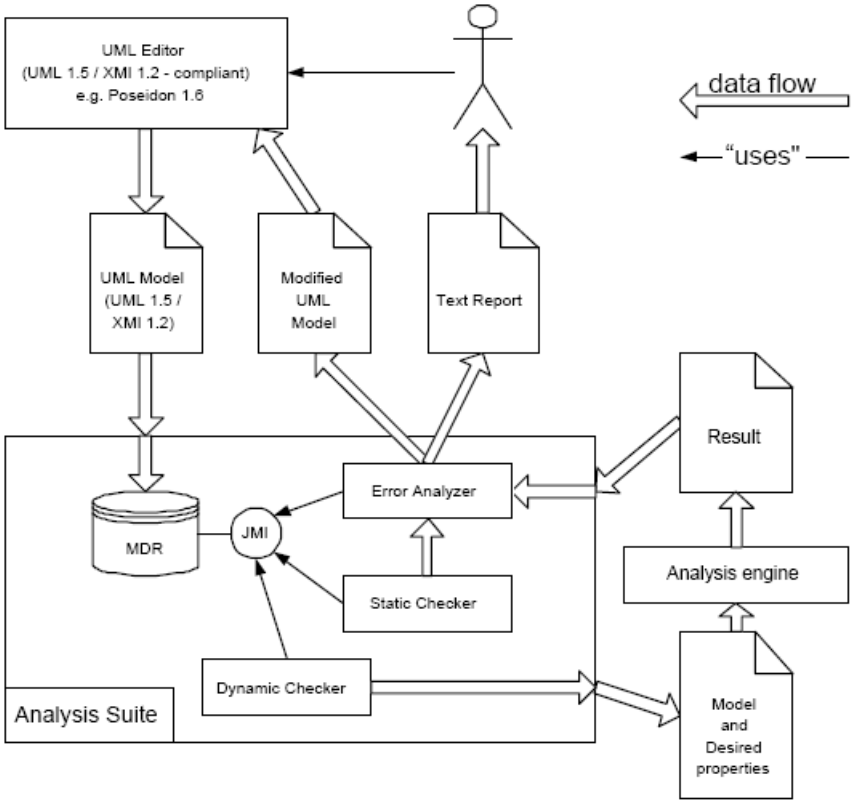

Figure 4.1: UML tool suite [40].

to interact. Furthermore, by providing a formal basis, the author layed a foundation for executable UML modeling, allowing simulations of whole systems. In the meanwhile, a toolset is available as well which makes it possible to verify the models formaly [41]. The main principle is to model the system by using a freely-available open-source UML modeling tool, to export the model in XML Metadata Interchange (XMI) format, and to analyze the model subsequently by using state-of-the art model checkers or theorem provers, see figure 4.1. The author used both, model checking and theorem proving, to verify security properties. Furthermore, Juerjens discussed generation of test-cases from the models as well as generating code from class diagrams and state charts. Although UMLsec was initially proposed in order to tackle the problem of designing secure information systems, the author recently addressed the development of secure embedded systems in [42]. As the approach evolves, further application areas may come along.

4.2 SecureUML

In [43], the authors describe SecureUML as a *modeling language which defines a vocabulary for annotating UML-based models with information considering access control concerns*. It is

based on a model for role based access control (RBAC), see figure 4.2, with additional support for specifying authorization constraints. In general, the RBAC model consists of five data types [44]: *users (USERS), roles (ROLES), objects (OBS), operations (OPS) and permissions (PRMS)*. A user is defined as a person or a process, while a role is defined as a position in an organization. A role unifies all privileges required to fulfill the dedicated job or function. One or many roles can be assigned to a user. This circumstance is described by the relation **User Assignment**. Privileges denote **permissions** assigned to a role, and the relation **Permission Assignment** describes the assignments of privileges to roles [43]. As defined in [44], *a permission is an authorization of a role to execute an operation on one or more protected objects or resources*. Thus, an **object** in this context is a system resource that is protected by a security mechanism [44]. A system entity can initiate actions on protected objects. Such actions are called **operations**.

SecureUML was designed for integrating specification of access control policies into application models, and as such the language is used as a part of the problem domain language, the host language. According to [44], by this way *different models at different abstraction levels can be annotated with access control information using the same syntax and a compatible semantics*.

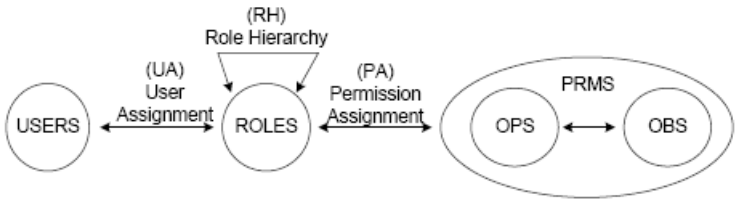

Figure 4.2: Role based access control [44].

The structure of SecureUML conforms to the reference model for model-driven systems, which means that the structure of a model is defined by a metamodel. In figure 4.3, the metamodel of SecureUML is depicted. It is defined as an extension of the UML metamodel [45]. In general, the metamodel is based on RBAC, but it extends RBAC in several directions. On the left side of the diagram, RBAC is formalized, where *Users* are extended by *Groups* and assigned to roles by using their common supertype **Subject** [45]. On the other side of the diagram, *permissions* are represented as the ability to execute *actions* on *resources*. Furthermore, supplementary constraints can be added to permissions such that these hold only in certain system states (e.g. time). In addition to role hierarchies (which are common for RBAC), hierarchies on actions are introduced. In [45], the authors write that *the types **Resource** and **Action** formalize a generic resource model that serves as a foundation for combining SecureUML with different system modeling languages*. Thus, SecureUML is intended to be embedded into a host language. Therefore, the purpose of integrating two languages is to create a language that can express both, access control policies and the required system. In order to combine the languages, their metamodels are merged. By this way, the resulting language is made security aware. Furthermore, in addition to syntax-merging, protected resources have to be identified and resource actions have

to be defined. That is, model elements of the system modeling language have to be represented as resources that have to be protected. Such resources can be processes, files, methods, objects and so on. Also, actions that can be executed on these resources have to be defined, for example read, write, execute, call etc.

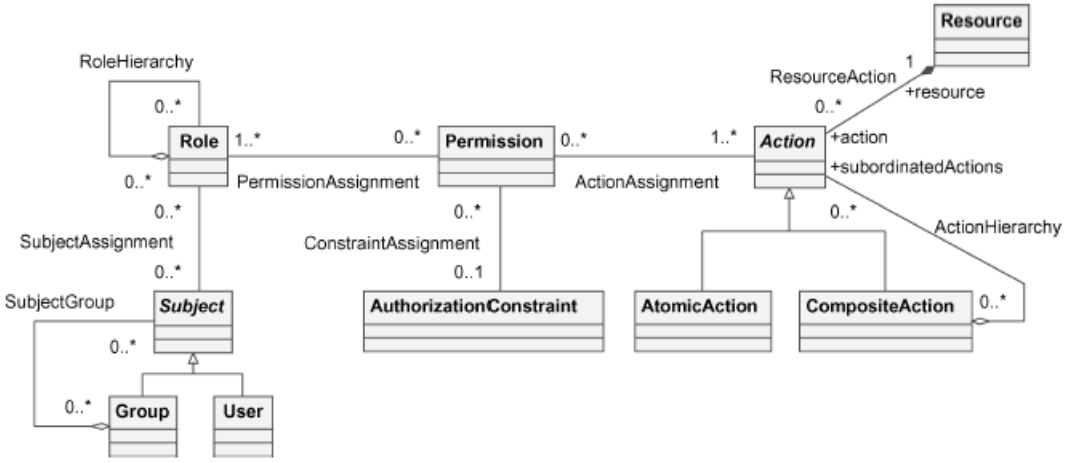

Figure 4.3: SecureUML Metamodel [45].

In order to generate the application code, the authors provided transformation rules to map SecureUML constructs to the vocabulary used by the target security platform of the host language (e.g. EJB). As the system is specified by combining SecureUML with a host language, the modeling paradigm is that of single, hierarchical models. Furthermore, the method does not focus on modeling security of distributed systems nor on modeling the system on different granularity levels. As the method is focused only on modeling role based access control (RBAC) and as the language is intended to be plugged into a host language, there is no point on covering granularity and distribution aspects. Becasue the RBAC models have to comply to a certain metamodel, the level of formality is high. Nevertheless, SecureUML models are not executable. A reason for this is the fact that SecureUML handles static security aspects only. There are no parts that cover dynamic aspects, like interaction diagrams or state charts for example. Therefore, there is not enough information for a model to be executable. Thus, static artifacts are the only ones that have to be provided by the modeler.

Recently, the authors showed how to automate the analysis of SecureUML models [46]. They formalized the models together with scenarios that represent possible run-time instances. Security properties, that is security policies, were expressed as Object Constraint Language (OCL) formulas. Basin et al. showed how the approach can been implemented in the SecureMOVA tool, which allows for evaluation of security policies in an autmated manner. Considering the applicability of the approach, it is not limited since the language was designed to be pluggable and embeddable into arbitrary design modeling languages.

4.3 Using Aspects to Design a Secure system

A big advantage of applying aspect-oriented modeling is its ability to separate crosscutting concerns. Therefore, Georg et al. proposed aspect-oriented modeling as a reasonable approach for designing secure systems [47]. The basic idea of their proposal is to treat aspects as security patterns, to reuse these proved designs and to integrate them into systems where such security mechanisms are required. In other words, additionaly to the primary model that is describing the system's core functionality, design patterns are used to describe the required security mechanisms (e.g. RBAC). As the approach is very general, it is not limited to a single application domain. Furthermore, the method is also not limited to model one single security mechanism. Rather, the proposed technique is applicable to a very broad range of security concerns.

The used design patterns are called Role Models, and they define properties that have to be satisfied by the concrete realizations of these design patterns. The authors focused on two aspect views: the static view and the interaction view. The Role Model type that describes the static view is called the Static Role Model (SRM) and it defines the aspect's structural properties. In fact, such a static model consists of entities (classes) and relationships (associations) between these entities. Furthermore, associations between entities can be described by multiplicity constraints. The interaction view, on the other side, describes the interactions between system parts. The Role Model type that describes this kind of view is called the Interaction Role Model (IRM). An IRM consists of collaboration entities and messages sent between these entities. Instantiating such a template results in an Interaction diagram.

Usually, an aspect definition comprises a single SRM and one or more IRMs. After the aspects are modeled, they have to be integrated with the primary model in order to obtain the whole system. This indicates that the approach conforms to the aspect-oriented modeling paradigm. Therefore, in addition to static and dynamic artifacts that the modeler has to provide (e.g. SRMs and IRMs), weaving rules are also required and have to be developed. These weaving rules, the authors call them weaving strategies, determine the way in which aspects are integrated with the primary model. In [47], the authors write that '*weaving strategies need to be developed from the kinds of threats that can be expected in a system*'. The presented approach differs from work decribed in previous sections (e.g. UMLsec, SecureUML) in that there is no focus on a new notation but on how to describe the different concerns and how to integrate them into the whole system. Nevertheless, the problem of weaving the aspects into the primary model is completely left to the developer. The authors write that '*the weaving strategies are intended to be reusable forms of experiences that can be used to assess the threats to a particular system*'. This means that weaving strategies were intended to be a sort of proved merging, that is, weaving patterns. However, the authors did not propose a way how to obtain adequate weaving strategies.

In general, the proposed method offers a high level of formality, since defined templates for static and dynamic views of security aspects are provided. Because only class and collaboration diagrams are used for modeling, regarding the granularity of design, it seems likely that the approach will suffer from same criticism as UML. The reason is that the approach does not

provide any notation for interfaces. Therefore, surely there are systems whose architecture can be described by applying static and interaction diagrams, but for complex and large systems this may be not enough. The same applies for modeling distributed systems. Some can be modeled, but not all, since processes and threads can not be illustrated properly (e.g. race conditions). The modeled system is not executable since there is not enough behavioral information provided in the IRMs (e.g. algorithms can not be expressed). Furthermore, the problem of validating the system's functionality and the correctness of its security properties is completely left to the developer. In fact, Georg et al. write that '*the security provided by mechanisms in the model is only as good as the weaving strategy*'. Because there is no formal way to derive an appropriate weaving strategy, the big issue of validating the developed system remains untackled. Also, no tool-support is provided.

4.4 Secure Software Architectures by Using Aspects

In [48], the authors apply Software Architecture Models (SAM) [49] to define the system's software architecture and the required security aspects. The approach is general and not limited to a specific domain. SAM is a software architecture development framework based on two complementary formalisms: predicate transition nets (also referred to as High-Level Petri Nets) and temporal logic. On the one hand, High-Level Petri nets are used to visualize and describe the static structure of the system. On the other hand, they are used to model the architecture's behavior, while temporal logic formulas (LTL) are used to specify the required security properties. A consequence of expressing security properties in temporal logic is that policies are expressed on a very low abstraction level. Expressible policies include safety and liveness properties, and variations of them [50]. In SAM, a hierarchical set of compositions is used to describe the system. In [48], the authors write that '*each composition consists of a set of components, of a set of connectors and of a set of constraints that have to be satisfied by the interacting components*'. The approach is also well suited for modeling distributed architectures. The authors describe the contribution of their work as *a formal notion for aspect-oriented modeling at an architectural level, and an aspect-oriented approach to design secure software architectures*. Figure 4.4 provides an overview of the approach.

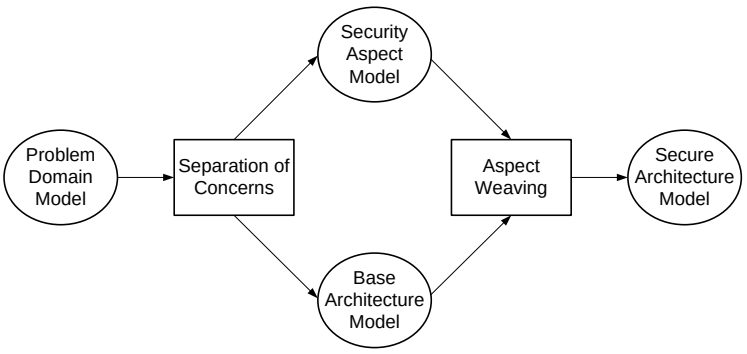

Figure 4.4: Framework for secure software architectures [48].

In the *problem domain model*, a precise description of the system's functionality is given. The *base architecture model* defines the software architecture of the targeted application. In this model, the system's basic functional modules and their connections are provided. In this model, modules are grouped into blocks whereas each block represents an autonomous process. In the *security aspect model*, security requirements are described. Furthermore, vulnerabilities, threats and provided mechanisms that enforce security policies are defined in such a model. That is, the security relevant features of the system are precisely described in this model. A *secure architecture model*, as a resulting artifact after merging, is the model where security policies have been enforced.

At first, after the problem domain model has been established, security requirements have to be specified on the base software architecture in order to construct security mechanisms that can be used for protection and which can be separated as security concerns. Obviously, the according activity is named *Separation of Concerns*, see figure 4.4. After the concerns have been characterized, all the models have to be integrated. Likewise, as depicted in figure 4.4, this second step is called *Aspect Weaving*. In general, in AOSD join points and pointcuts are used to merge the primary model with the aspects. The authors define join points as connectors that satisfy a certain condition, and a pointcut is described as a '*set of join points which have the same security vulnerability and share the common security enforcement mechanism*'. Advices are patterns which prescribe mechanisms enforcing the security for the pointcuts. The authors define the aspect weaving step (that is, merging the aspect models with the base architecture model) as a sequence of following steps:

1. Locating the join point. In this step, the location is appointed where the base architecture model and the security aspect models interact.

2. Constructing advices. Here, the behaviour that will enforce security policies is defined.

3. Weaving aspects. In this last step, the aspect models and the base architecture model are integrated.

Obviously, the presented approach conforms to the aspect-oriented modeling paradigm. The modeler has to provide static and dynamic models (High-Level Petri nets cover both, static and dynamic views of the system), and weaving rules as well (these are represented by pointcuts and advices). The presented method is highly formal, nevertheless the models are not executable since they represent the system's software architecture. The authors show on an example how one can reason about the correctness of aspect weaving. Furthermore, as Petri nets are the basis for the modeling formalism, model checking could also be applied in order to verify security properties (which are represented by LTL formulas). However, the approach lacks any tool support and to our best knowledge, no significant further work has been done in order to enhance the proposed method.

4.5 Aspect-Oriented Modeling of Access Control in Web Applications

In [51], the authors propose an aspect-oriented technique for modeling access control in Web applications. In order to do so, they apply UML state machines to specify navigation rules. The approach is utilised in the context of the UML-based Web engineering method (UWE). In UWE, the Web application is separated in following concerns: the content that is modeled in the conceptual model, the navigation structure that is modeled in the navigation model, the business process that is modeled in the process model and the presentation that is modeled in the presentation model. The authors describe the conceptual model as comprising *entities used in the Web application, which are represented by instances of the* `conceptual class`*, that again is a subclass of the UML metaclass Class* and relationships between these contents which again are modeled by UML associations. Business processes, such as searching and ordering a product for example, are modeled by activity diagrams based on the UML 2.0 specification [52]. The navigation model, on the other hand, models how the content of the Web application can be accessed.

In the presented approach, UWE is extended by associating a state machine to each navigation node. By this way, a detailed behaviour of each navigation node is specifed. That is, when a Web page resource has to be shown, its corresponding state is executed first. As in real applications usually there is more than only a single web page that has to be protected, the access control state-machine has to be replicated. In order to avoid such redundancy, the authors organized the model elements as aspects. For this reason, they extended the UWE metamodel by a package annotated with the `aspect` stereotype. An aspect can contain one or more navigation nodes, and its characterisitic is that access control rules are not defined in single nodes, but in the corresponding aspects. In addition, the authors provided ArgoUWE, a CASE tool to support the design phase of the UWE development process. ArgoUWE is implemented as a plugin module for the open source ArgoUML modeling tool (UML 1.4). Furthermore, the authors provided support for the aspect-oriented modeling extension described above and integrated it in the ArgoUWE modeling tool. In a subsequent work [53], Knapp and Zhang focused on model integration by applying graph transformation rules. By merging the navigation and the business process models, they obtained a UML state machine which included both, a static navigation structure as well as the dynamic behaviour of the Web application. Afterwards, the authors verified the resulted model formally by applying the state-of-the-art model checker SPIN [54]. The modeled system is not distributed and also not executable, since the business process model is containing subtasks that have to be refined in order to be executable (e.g. searching or sorting algorithms).

4.6 An aspect-based approach to modeling access control concerns

In [55], Georg et al. propose the usage of AOM techniques in order to systematically address access control concerns in information system design. The presented work is a consequent application of ideas the authors already proposed in [47]. They model access control from two different perspectives. On the one hand, the structural view of the system is modeled by iden-

tifying domain entities which are constrained by access control policies, and the corresponding associations between these entities. Class diagrams are used to represent this perspective. On the other hand, the dynamic view defines the constraints which are imposed on the system's behaviour. This perspective is represented by interaction and collaboration diagrams. In fact, as already proposed in [47], aspects are templates or patterns that provide static and dynamic views of the system's crosscutting concerns. The method is not intended for modeling distributed systems, and it is also not suitable for modeling architectures (e.g. race conditions). Since the provided models do not contain enough information to express algorithms, the resulting models are not executable. In the proposed approach, the authors compose an aspect with the primary model by doing following steps:

- At first, *they instantiate the aspect in order to obtain a so-called context-specific aspect*. That is, values are bound to aspect's parameters, and a design pattern is concretized to an implementation blueprint.
- Second, *context-specific aspects have to be composed with the primary model*. That is, the views which are described by context-specific aspects are merged with aspects described by the primary model in order to obtain a model which describes the complete system (the so-called woven model). One possibility to merge the models is to weave them in case they are of same syntactic type. In case both models do not have a matching element, which means that they do not have a single element which can be used for merging, a new model element has to be added to the woven model. In case of conflicts, for instance when multiple matches occur, the modeler has to indicate, by using certain rules, which view should dominate.

Merging interaction patterns involves three steps:

- First, aspect participants have to be matched with participants in the primary model.
- Second, aspect participants without a corresponding match have to be included in the woven model.
- Messages specified in the dynamic view have to be merged based on composition directives provided by the modeler.

Nevertheless, even if the proposed methodology leads to composed models that are containing required security aspects, one can not be sure whether the resulting model fullfills the required security properties. In [56], the same authors continue the work and try to handle the problem of assuring security properties when composing aspects and primary models. They present an approach in which verifiable compositions of behaviours are supported. In the mentioned work, the authors use a model of a banking application as the primary model, an RBAC aspect model, and they specify security properties by expressing them in OCL. They illustrate how models can be composed such that compositions are verifiable. That is, they show how one can combine models and reason manually about their security properties. To our best knowledge, until now no tool-support for model composition or verification was provided by the authors.

4.7 A model-based aspect-oriented framework for building intrusion-aware software systems

In [57], the authors propose a model-based, aspect-oriented framework for building intrusion-aware software systems. In the framework, intrusion detection aspects (IDAs) are included which automatically detect intrusions. The authors developed a UML profile with aspect-oriented extensions in order to model attacks, and by this way intrusion detection aspects (IDAs) which are responsible for detecting these attacks. Analogous to work done by G. Georg et al. in [47], static and dynamic views of the system's aspects have to be provided by the modeler. Class diagrams are used to represent the system's static attributes, and state machine diagrams are used to represent the dynamic views of intrusions. That is, by applying state machine diagrams it is represented how the attacker intrudes into the system. After modeling, the attack scenario models are transformed into programs (that is, code is generated for the IDAs) and are woven into the primary program subsequently. After weaving, the aspects work as intrusion detection components and identify attacks automatically. In figure 4.5, the principle of the proposed approach is depicted. The method is not limited to a specific application domain and the framework can also be used to make distributed systems intrusion-aware. Several open source tools, like ArgoUML, AspectJ and Novasoft Metadata Framework are used to build the framework. The level of formality, when modeling intrusions, is high since state machines are used to describe the attack. Nevertheless, when considering the remainig application which is written in an ordinary programming language and for which no model is available, the system is too complex to be verified formally. Therefore, the resulted intrusion-aware application was tested manually by applying a set of attacks from the web security threat classification released by the Web Application Security Consortium (WASC) [58].

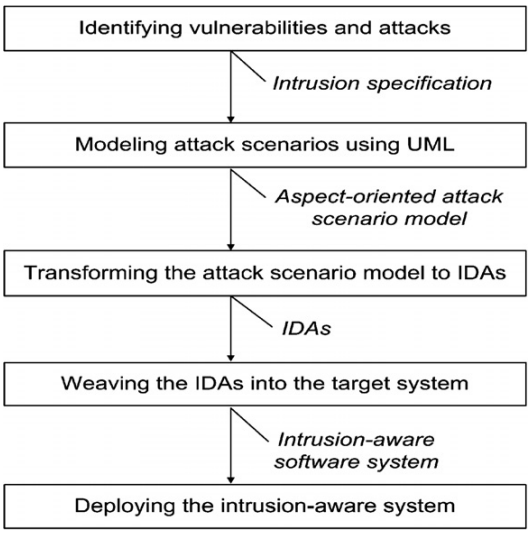

Figure 4.5: AOSD framework for developing intrusion-aware software systems [57].

4.8 A security-aware metamodel for multi-agent systems (MAS)

In [59], Beydoun et al. provided a model based-security approach for development of distributed multi-agent information systems. The proposed method complies to the single model paradigm, since the authors extended the already existent FAME Modeling Language [60] and introduced a novel metamodel that considers security concerns. FAME (in fact a metamodel) defines concepts from which modeling elements can be instantiated in order to construct models or designs of a mult-agent system, but it does not consider security. The resulting artifacts in FAME are models which then can be manually implemented or used as inputs to further model-driven development process. In general, these models are not executable. Beydoun et al. argued that a first step in developing an agent-modelling language, which also takes security issues into account, *is the definition of metamodels that define security concepts together with associated agent development concepts* [7]. Thus, the authors proposed model-based security to ensure considering security requirements throughout the overall development process. The proposed language is not expressive enough to model different security requirements and security mechanisms. The language distinguishes, on a very abstract level, between system specific and agent specific security requirements, between security actions, security tasks and protected resources. The authors provided a case study to illustrate the applicability of the approach and they claimed that an initial verification of the proposed metamodel was started. As the approach was proposed recently, no tool-support is available.

4.9 Automated Validation of Internet Security Protocols and Applications

In [61], the authors propose a tool, called AVISPA, intended to speed up the development of security protocols and to improve their security. The approach provides a language called the High Level Protocol Specification Language (HLPSL), which is used to describe the protocols and their intended security requirements, and a bunch of analysis tools to formally validate them [62]. The authors write that the approach *provides a modular and expressive formal language for specifying security protocols and properties, and integrates several different back-ends that implement a variety of automatic analysis techniques ranging from protocol falsification to abstraction-based verification methods for both finite and infinite numbers of sessions* [61]. The archietcture of the Avispa tool is depicted in figure 4.6. In the HLPSL user manual [62], the authors write that a valid HLPSL specification

> *is translated into the Intermediate Format (IF), using a translator called hlpsl2if. IF is a lower-level language than HLPSL and is read directly by the back-ends to the AVISPA Tool. This intermediate translation step is transparent to the user, as the translator is called automatically. The IF specification of a protocol is then input to the back-ends of the AVISPA Tool in order to analyse whether the security goals are satisfied or violated.*

The language offers an expressive formalism which allows for specifying roles, control flows, data structures as well as security requirements [61]. After providing the specification

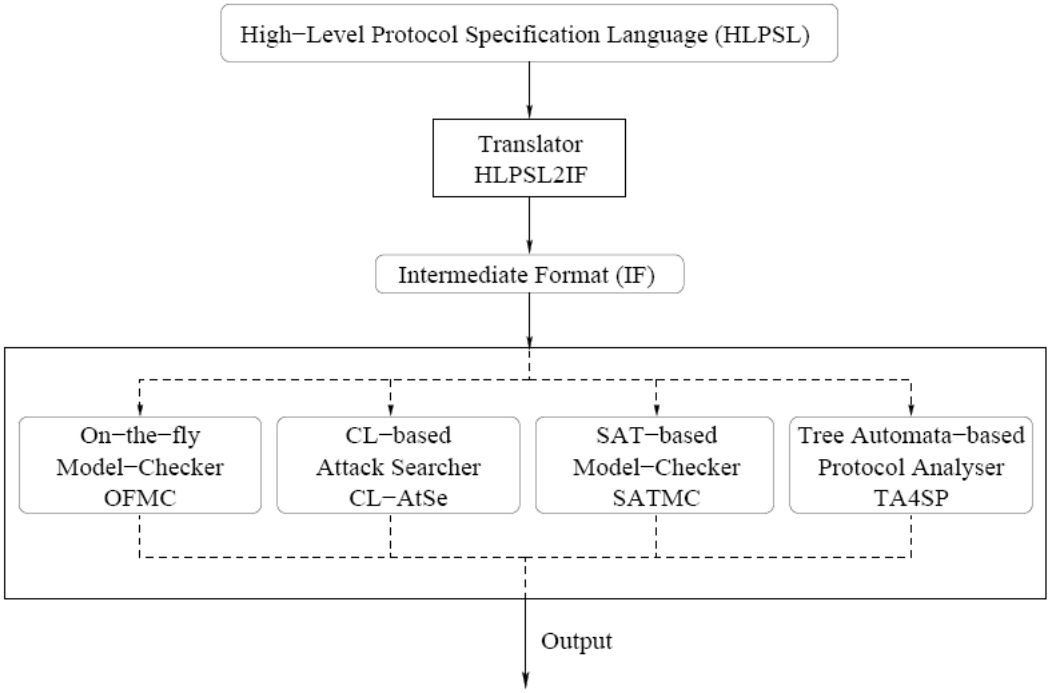

Figure 4.6: Avispa architecture [62].

to the tool which comprises four back-ends, the problem is tackled from different positions with different techniqes. In [13], these are described as follows:

> *The On-the-fly Model-Checker (OFMC) performs protocol falsification and bounded verification by applying symbolic model-checking techniques. It supports the specification of algebraic properties of cryptographic operators, and typed and untyped protocol models. The Constraint-Logic-based Attack Searcher (CL-AtSe) applies constraint solving. CL-AtSe is built in a modular way and is open to extensions for handling algebraic properties of cryptographic operators. It supports type-flaw detection and handles associativity of message concatenation. The SAT-based Model-Checker (SATMC) builds a propositional formula encoding all the possible attacks (of bounded length) on the protocol and feeds the result to a SAT solver. The TA4SP (Tree Automata based on Automatic Approximations for the Analysis of Security Protocols) back-end approximates the intruder knowledge by using regular tree languages and rewriting. For secrecy properties, TA4SP can show whether a protocol is flawed (by under-approximation) or whether it is safe for any number of sessions (by over-approximation).*

In summary, the protocol designer specifies a security problem in the High-Level Protocol Specification Language, and provides the resulting specification to the tool [61]. Then the tool

performs the analysis by utilising four different back-ends. Upon termination, the analysis result is provided by stating whether the problem could be solved, whether the problem could not be solved due to exhausted resources (e.g. memory), or some other reason which prevented the tool from solving the problem.

In general, the method offers a high level of formality, since HLPSL is based on Lamport's Temporal Logic of Actions [63]. The user has to provide a dynamic model of the system's behaviour, which is representsed by a distributed system consisting of interacting processes sending to and receiving messages from each other. The model is not executable, since it is an abstraction of the protocol, and not its implementation. Authenticity, integrity, confidentiality and non-repudiation requirements can be analysed by the approach. However, as with all methods based on state exploration, the size and the complexity of analyzed systems are severly limited by the state explosion problem (cf. [64]).

4.10 Symbolic Model Verifier

The Symbolic Model Verifier (SMV) is a model-checking system applicable for analyzing designs of synchronous and asynchronous process systems. It provides a language for describing finite automata, and it can directly check the validity of temporal logic formulas (that is, LTL and CTL). The tool takes a textual description of the system's dynamic model and the corresponding specification, which is expressed by LTL and CTL terms. On termination, it produces either 'true' if the specification holds, or a trace showing why the required property is violated. SMV programs consist of one or more modules, which can declare variables and assign values to them. Usually, assignments give the initial value of a variable (e.g. `init(var) := 0`), whereas the variable's next value is specified in terms of expressions comprising the current value (e.g. `next(var) := ((var + 1) mod 3)`) [27]. Thus, state transitions are modeled this way. Values can also be non-deterministic, in case the environment is influencing the system. In SMV, processes can be represented by modules which can be composed synchronously or asynchronously. In the asynchronous case, the modules run at different speeds, and they are interleaving arbitrarily. Such asynchronous compositions can be used for description of communication protocols, asynchronous circuits and other systems whose actions are not synchronized to a global clock [27]. In section 2.3, an example was introduced to illustrate the principle of model checking. The sample code illustrated there is written in the SMV modeling language. Below, sample code for a modulo-3 counter is illustrated. The specification (LTL) requires for each state that whenever $y = 3$ holds, then in the next state $y = 0$ is valid.

In general, the proposed method offers a high level of formality, since it is based on temporal logic. It is well suited for modeling distributed systems, and the user has to provide a model of the system's dynamic behavior. The granularity of modeled systems can vary. On the one hand, communicating processes can be modeled, which can describe a view of the system's architecture. On the other hand, the method is applicable for modeling finite state machines, such as Mealy automata. Executable software systems cannot be modeled, but security protocols can. Properties like authenticity, integrity, confidentiality and non-repudiation can be verified. Of

```
MODULE main
VAR
  y : 0..3;

ASSIGN

  /* initial value of y */
  init(y) := 0;

  /* state  transition  */
  next(y) := ((y + 1) mod 3);

LTLSPEC G( (y = 3) -> (X y = 0) );
```

Listing 2: Sample code for the Symbolic Model Verifier.

course, these have to be transformed into temporal logic formula first in order to be analysable. Lastly, as with all model checking methods, the size and the complexity of analyzed systems are severly limited by the state explosion problem (cf. [64]).

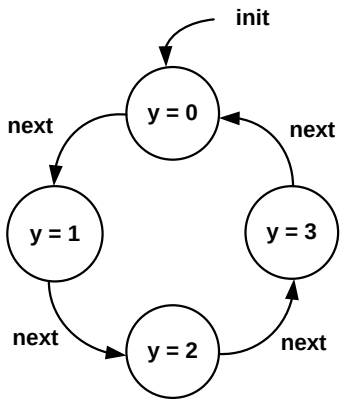

Figure 4.7: Corresponding automaton.

4.11 Alloy

Alloy is a declarative modeling language based on first-order logic, extended with relational logic operators [65]. The language was primarily designed for modeling software designs. Models written in the Alloy language can be analyzed using the so-called Alloy Analyzer, which is a model-finder built on a SAT solver to simulate models and check their properties. Hereafter, we

```
sig Name, Addr{}
sig Book{
  addr: Name -> lone Addr
}
```

Listing 3: Defining sets of objects [65].

use the term Alloy to refer to both the language and the tool. The key elements of the approach are a logic, a language and an analysis, which are introduced below [65].

- In [65], the authors describe Alloy as a first-order relational logic, which provides the building blocks of the language. All logical structures are represented as relations, and all structural properties are expressed with relational operators. States and executions are both described using constraints.

- The language adds a syntax to the underlying first-order relational logic. To support classification, the Alloy language supports typing, sub-typing and compile-time type-checking. Furthermore, the language's module system allows a reuse of generic declarations and constraints [65].

- Literally, the analysis of Alloy models is a form of constraint solving, either by finding an instance of a model or by finding a counterexample for a given property. An instance is an example of the specified model, in which both the facts and the predicate hold. To make instance finding practically feasible, a user-specified scope is defined that limits the size of the analysed instances. Within this bound, the analyzer translates the constraint into a boolean formula and solves it using a commercial SAT (satisfiability problem) solver [65]. The found solution is then presented to the user.

In order to give a better understanding of the language, we will introduce a simple address book example. It is an address book for an email client which maintains a mapping from names to addresses, and it is illustrated in listing 3. In the model, three sets of objects are introduced: *Name*, *Addr* and *Book*. In the Alloy language terminology, these sets are called *signatures*. The *Book* signature has a field *addr*, which maps names to addresses. In other words, *addr* is a three-way mapping associating books, names and addresses. The keyword *lone* indicates the multiplicity - in this case each name is mapped to at most one address. Thus, by using signatures, static objects are modeled, similar to classes in object-oriented programming languages.

Restrictions or constraints that are required to hold can be added by the *pred* keyword. Facts, which are constraints that always hold, can be declared by the keyword *fact*. In listing 4, the predicate *show* defines a model which specifes that the book b contains more than two name-to-address associations. The defined fact, on the other hand, requires that in all models each book contains at least one name-to-address association.

```
pred show(b:Book){
  #b.addr > 2
}
fact{
  all b: Book | b.addr >= 1
}
```

Listing 4: Defining facts and constraints [65].

```
pred add(b, b':Book, n:Name, a:Addr){
  b'.addr = b.addr + (n -> a)
}
```

Listing 5: Defining dynamic behaviour [65].

So far, we have defined a state space by declaring sets of objects and restrictions on them, and by defining facts which hold for each model. Thus, we have shown how to describe static structures. In the next code sample, we introduce dynamic behavior. Both predicates, *add* and *show*, are constraints. However, the predicate *add* represents an operation and thus describes dynamic behavior. The difference lies in the additional parameter b', which is denoting the future state of the book b. That is, a restriction is set on the future state b' which is determined by the actual state b and the newly created mapping from the name n to the address a.

As the Alloy's relational logic is undecidable, the underlying analysis is based on instance finding. The key idea is the specification of a scope, which bounds the sizes of the signatures, and an exhaustive search within the scope for examples or counterexamples. However, as with other formal methods, the size and complexity of analysable systems are strongly limited. The authors write that *with a model containing up to 20 signatures and 20 or 30 fields, an analysis in a scope of 5 to 10 is usually possible*.

In general, the proposed method is suitable for modeling static and dynamic aspects of software systems. Furthermore, it offers a high level of formality, since the language is based on a first-order relational logic. The Alloy language is abstract enough to model the problem domain's specific entities, as well as to model distributed systems since message transmissions can be represented as dynamic operations. The modeled systems are not executable, but are also not bound to a specific application area, since Alloy is expressive enough to capture several problem domains. As the approach is based on first-order logic, security requirements such as authenticity, integrity, non-repudiation and confidentiality can be expressed. These have to be specified by the user as first-order formulas.

Table 4.1: Evaluation results.

Dimension	Juerjens(2002)	Basin et al.(2002)	Georg et al.(2002)	H. Yu et al.(2005)	Zhang et al.(2005)
Paradigm	hierarchical	hierarchical	aspect-oriented	aspect-oriented	aspect-oriented
Artifacts	static and dynamic models	static models	static and dynamic models weaving rules	static and dynamic models weaving rules	static and dynamic models weaving rules
Formality	metamodels constraints	metamodels constraints	design patterns	high-level Petri nets temporal logic	metamodels state machines
Distribution	yes	no	yes	yes	no
Granularity	packages, classes	classes	classes	components and connector	classes
Executability	no	no	no	no	no
Verification	model checking theorem proving	theorem proving	no	model checking	model checking
Tool-Support	yes	yes	no	no	yes
Applicability	information systems embedded systems	widely applicable	widely applicable	widely applicable	web applications
Security mechanisms and requirements	confidentiality integrity non-repudiation non-interference authenticity access control	RBAC policies	no specific mechanisms or requirements	safety liveness	RBAC

Table 4.2: Evaluation results.

Dimension	Zhu et al.(2008)	Georg et al. (2004)	Avispa	Symbolic Model Verifier	Alloy
Paradigm	aspect-oriented	aspect-oriented	hierarchical	hierarchical	hierarchical
Artifacts	static and dynamic models weaving rules	static and dynamic models weaving rules	dynamic models	dynamic models	static and dynamic models
Formality	metamodels state machines	metamodels	temporal logic of actions	temporal logic	first order logic
Distribution	no	no	yes	yes	yes
Granularity	classes	classes	processes	processes	classes
Executability	no	no	no	no	no
Verification	no	theorem proving	model checking theorem proving	model checking	model finding
Tool-Support	no	no	yes	yes	yes
Applicability	widely applicable	information systems	security protocols	synchronous and asynchronous systems	widely applicable
Security mechanisms and requirements	intrusion detection	RBAC	confidentiality integrity authenticity	safety liveness	confidentiality integrity authenticity non-repudiation

CHAPTER 5

Pseudonymization of Information for Privacy in e-Health

This chapter describes the system to be analysed, PIPE. At first, we give a general introduction into the system and its principles. Afterwards, we provide a more detailed view on the system and the offered workflows. Then we describe the problem to be solved in detail.

5.1 General description

Pseudonymization refers to a method where identification data is transformed into a specifier, and afterwards replaced by it. The mentioned specifier, also called a pseudonym, cannot be linked to the corresponding identification data without knowing a certain secret [66]. In order to overcome the drawbacks of existing approaches, Neubauer et al. proposed a new system, PIPE, for the pseudonymization of health data records which comprises methods for data sharing, authorization, and recovery. In [20], Neubauer and Riedl write that

> *the system is based on a hull-architecture, where each hull consists of one or more secrets (encrypted keys), which are only accessible with the unveiled secrets from the next outer hull. For instance, patients' inner private key (e') in the inner hull is encrypted with the outer public key (d) on their smartcards, which represents the outer hull or authentication layer.*

Thus, the authorization mechanism of the system is represented by a layered hull-model, which comprises of at least three secruity-hulls, as depicted in figure 5.1. In [20], Neubauer and Riedl describe the authentication mechanism of the system such that *a key* K_N *of a certain hull* H_N *is encrypted with a key* $K_N + 1$ *of the hull* $H_N + 1$ *envelopinh hull* H_N. Hence, the PIPE system does not depend on a list where the patient's identity is associated with medical data, or a list where all patient's psedonyms are kept. Instead, the pseudonyms are encrypted and stored in a database, and only the patient him/herself and persons authorized by the patient are able to

decrypt the pseudonyms and retrieve the medical data. In order to be able to retrieve the medical data, each user has to be authenticated. In case a successful authentication has been established, a symmetric session key is generated which is used to secure the subsequent communication between the client and the server. Furthermore, all encrypted keys which are owned by the user and stored in the database are transmitted to the client for further use. The security of the system is mainly based on public-key cryptography, which assures security properties like confidentiality, authenticity, integrity and non-repudiation.

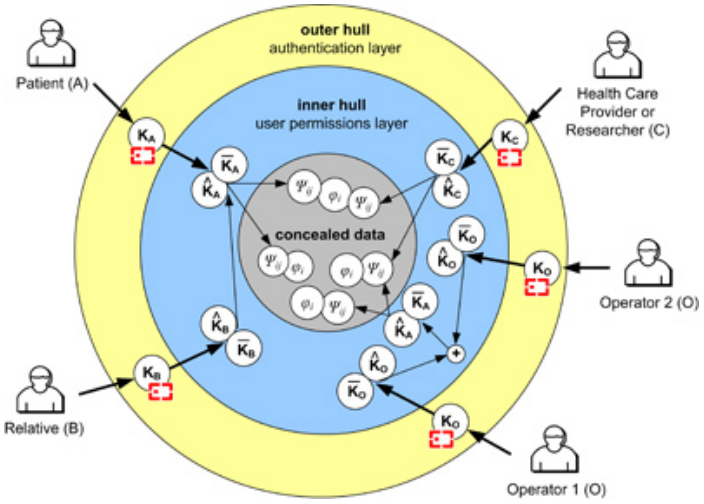

Figure 5.1: Layered securty-hull model [20].

The architecture of the system is depicted in figure 5.2. As illustrated, the system can be divided into client and server components.

In [67], Heurix and Neubauer describe the client as consisting of an application part (which is executed on the host), and a security token. The user-owned security token stores authentication credentials and performs cryptographic operations such as encryption and decryption by applying the user's public and private keys. The remaining PIPE toolbox represents a library that provides functionality for authentication and authorization, backup, emergency, data retrieval and storage [67]. The keys and the operations located at the security token are, according to the concept of two-factor-authentication, only accessible by providing a secret PIN. During the authentication process, cryptographic user keys are transferred to the token, decrypted there within a secured environment, and are then available for further operations. The client application contains functionality which cannot be stored or executed within the security token, due to performance or storage space reasons. As the toolbox logic is located on the client side, the server contains only a minimal logic, and its tasks are limited for providing database operations (searching, retrieving and returning data back to the client) and handling the server-side authentication part [67].

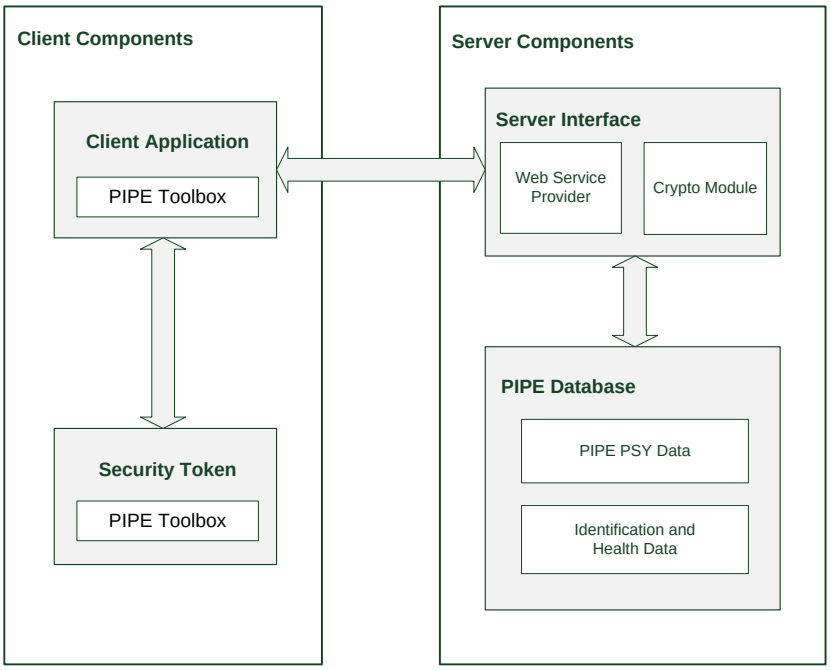

Figure 5.2: Architecture [67].

5.2 Workflows

In this section we describe the system's static data model, the workflows operating on that data and the system users, which can be classified in two categories. At the one hand, there are the data owners. This user class represents patients which are in possession of their medical data records. On the other hand, there are the authorized users. This user class represents health professionals which need access to the patient's health data records and thus can be authorized by the data owners in order to grant the required access. Data owners can

- authenticate against the system,
- insert medical data into the system,
- retrive added health data records,
- authorize other users to hava access to these data records,
- and pseudonymize medical data.

Authorized users as well can perform all these operations, except pseudonymizing data. As a first step, all users have to authenticate against the system. That is, they have to provide a correct PIN in order to access the key pair stored on the smart card. This key pair represents the outer key pair from the layered securtiy hull-model described in the previous section. User

authentication is a precondition for all further workflows. After a successful authentication, the user can add new health records to the system or retrive already stored ones. Use cases 'Add Data' and 'Retrieve Data' describe these two alternatives. The 'Authorize Instance' use case refers to granting access to specific data records to a trusted user. For example, a patient could grant access to his or her health data to a medical doctor. 'Pseudonymize Data' refers to the procedure of pseudonymizing health data records already stored on the server and can be performed by the data owner only [67].

Data model

In order to illustrate and clarify the underlying principles of the PIPE system, we will base our description upon the simplified static data model depicted in figure 5.3, which is an excerpt from the original PIPE data model described in [67]. The model exhibits a conventional relational data structure, where the *UserInstance* table stores data regarding the users, including their identifier as well as the users' cryptographic symmetric key. The inner private and public keys are not modeled, since all the workflows except the 'Authentication' workflow do not rely on public key cryptography. User instances include both user categories, that is patients and health care providers. Pseudonyms can as well be split in two categories. On the one hand, root pseudonyms denote pseudonyms which are not shared and are known to the data owner only. On the other hand, shared pseudonyms represent pseudonyms which are known by authorized users, e.g., health care providers. Both categories can further be split into identification and health pseudonyms. Identification pseudonyms denote pseudonyms which associate a user (data owner or health care provider) with an identification record, whereas health pseudonyms associate users with health records. Figure 5.4 depicts the aforementioned classification. Thus, the *RootIdPseudonyms* table associates the data owner's identification record with his/her identifier, whereas the *RootHealthPseudonyms* table associates the data owner's non-shared health records with his/her identifier. The *SharedIdPseudonyms* table relates the data owner's identification record and the authorized health care provider, whereas the *SharedHealthPseudonyms* table denotes the relation of shared health records and the authorized health care provider. The *PseudonymRecordsMapping* stores a cleartext mapping of pseudonyms and record identifiers, which identify the medical data records [67].

User instances comprise an internal user identifier ($IUID$) and an internal symmetric key (ISK). Shared pseudonyms comprise encrypted internal user identifiers of the data owner and of the authorized user. More precisely, $ISKow(IUIDow)$ represents the internal user identifier of the data owner, encrypted with the data owner's inner symmetric key. Analogous, $ISKau(IUIDow)$ represents the internal user identifier of the data owner, encrypted with the authorized user's inner symmetric key. Furthermore, the pseudonym itself (that is, the plaintext value of the pseudonym) is encrypted with the authorized user's and the data owner's inner symmetric key ($ISKow(PSN)$ and $ISKau(PSN)$). Root pseudonyms, on the other hand, comprise the encrypted data owner's internal identifier and the encrypted pseudonym only ($ISKow(IUIDow)$ and $ISKow(PSN)$). Records contain a medical description and a record identifier (RID).

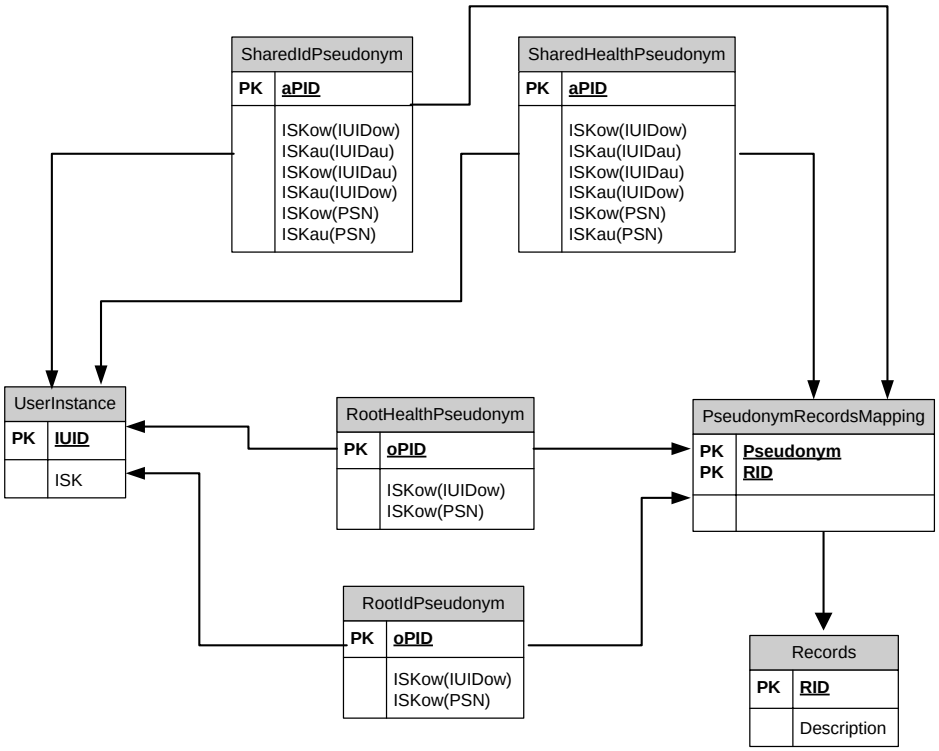

Figure 5.3: An excerpt from the static data model [67].

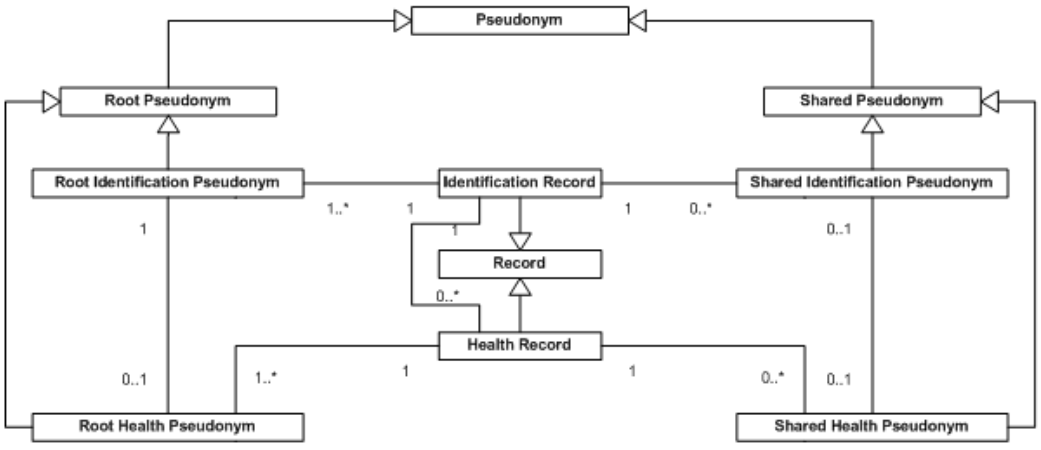

Figure 5.4: An excerpt from the logical data model [67].

Authentication

Authentication is required as a precondition for each session and is based on the challenge-response prionciple [67]. During the authentication process, the user keys stored in the database (that is, the inner private key and the inner symmetric key) are transferred to the security token, where they can be used for further operations. In addition, a symmetric session key is generated which is used for encryption and decryption of each message transmitted between the client and the server. The precondition for this workflow is a created user instance available in the database and the fact that the user is in possession of a personal security token. The workflow comprises five steps, which are depicted in figure 5.5 and described more extensively in [67]:

1. First, the user generates a random value as a challenge and encrypts his internal identifier and the challenge with the server's outer public key. Then, the user sends these items to the server.

2. The server decrypts the received message with its outer private key and uses the retrieved user identifier in order to query the database. If such an identifier exists in the database, the server retrieves the user's outer public key, it generates a random value as the server challenge, and sends these items encrypted with the user's outer public key to the user.

3. Then the user decrypts both random values with his outer private key. In case the user challenge is correct (that is, the original value and the server-returned value are identical), he returns the server challenge back to the server encrypted with the server's outer public key.

4. The server decrypts the received message and checks whether the server challenge has been answered correctly. If so, both the user and the server are successfully authenticated, and the server retrieves and forwards the user's encrypted inner private key and the inner symmetric key which are both encrypted with the newly generated session key, as well as the session key itself encrypted with the user's outer public key, to the user.

5. Then the user decrypts the session key with his outer private key and the remaining items with the retrieved session key. Subsequently, he decrypts the inner private key with his outer private key and the inner symmetric key with his inner private key. From now on, each message exchanged between client and server is encrypted with the session key.

Get Pseudonyms

This 'Get Pseudonyms' workflow describes how to retrieve the pseudonyms associated with a user. Depending on the user's role, these include all root or shared pseudonyms. The workflow consists of the following steps [67], which are also depicted in 5.7:

1. The user encrypts his user identifier with his inner symmetric key and queries the database for all associated pseudonyms.

2. The database engine returns the list of pseudonyms and identifiers.

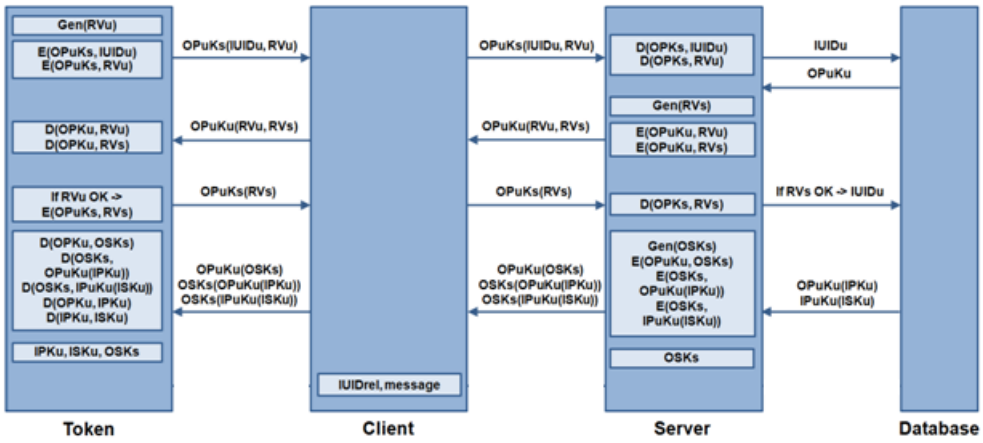

Figure 5.5: The authentication workflow [67].

3. The user decrypts the pseudonyms (with his inner symmetric key) which are then displayed by the client application.

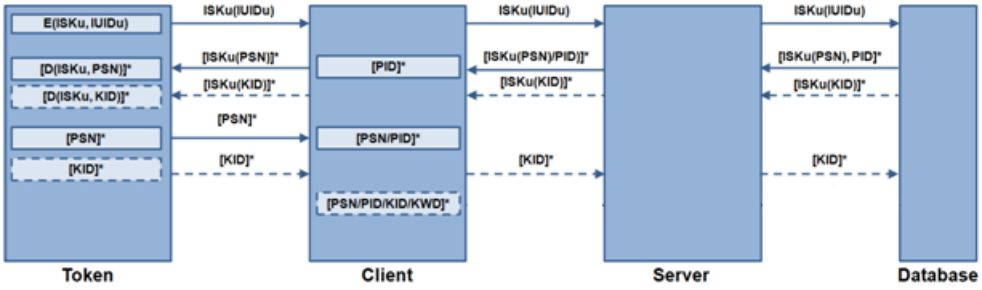

Figure 5.6: Get Pseudonyms workflow [67].

Authorize Instance

The 'Authorize Instance' workflow refers to granting access to specific health data records to a trusted user instance [67]. That is, the data owner allows access to his or her health data records to a health care provider, which we will call the authorized for short. This workflow describes the synchronous authorization scenario where both the data owner and the authorized are present and provide the client application with their identifiers. The workflow requires that both data owner and the authorized are present and authenticated, and that the health data record to be shared is stored in the database. The workflow consists of the following steps [67], which are also depicted in 5.7:

1. The data owner retrieves his or her root pseudonyms and selects the one referenced with the data record he or she intends to share with the authorized.

2. With the selected root pseudonym, the owner retrieves the corresponding record identifier.

3. The client application generates the shared pseudonym and references it with the record identifier if the shared pseudonym is confirmed unique.

4. The owner encrypts the pseudonym and both user identifiers (owner and authorized) with his inner symmetric key.

5. Analogously, the authorized encrypts the pseudonym and identifiers with his inner symmetric key.

6. The encrypted pseudonym and identifiers are stored in the `SharedPseudonyms` table in the database.

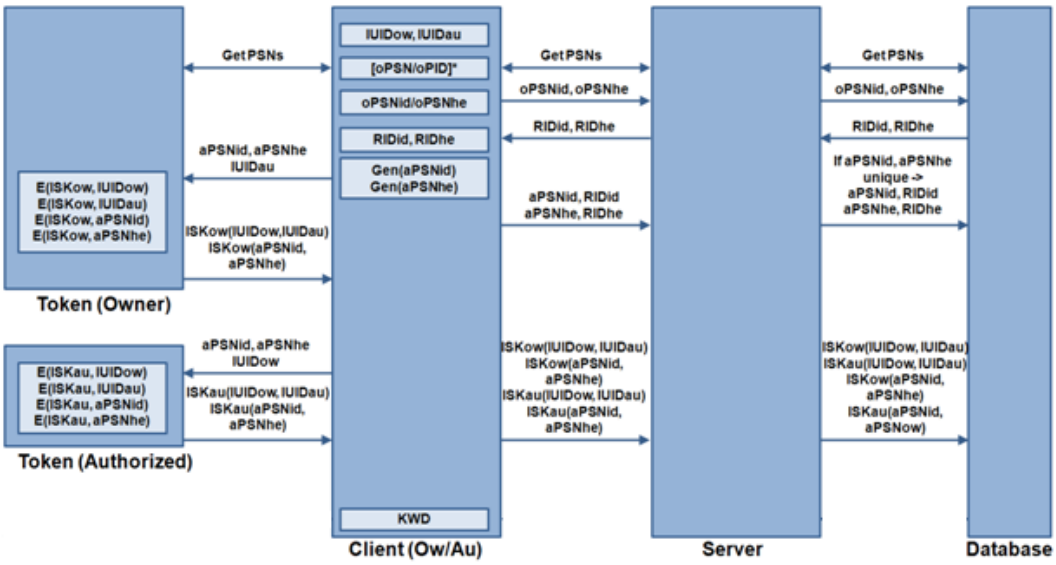

Figure 5.7: The authorization workflow [67].

Data Insertion

The workflow describes how the data owner can add new (health) data into the system. The precondition is that the user (data owner) has to be authenticated. Below we enumerate the steps comprised in the workflow:

1. The data owner retrieves the record identifier of his identification record.

2. The data owner enters the health data to be added (either the medical document itself or a reference to the record's physical location).
3. The client application generates new root identification and health pseudonyms which are referenced with the identification record identifier and the health record identifier provided by the database engine after storing the new health data if confirmed unique.
4. The pseudonyms and the use identifier are encrypted and stored in the database.

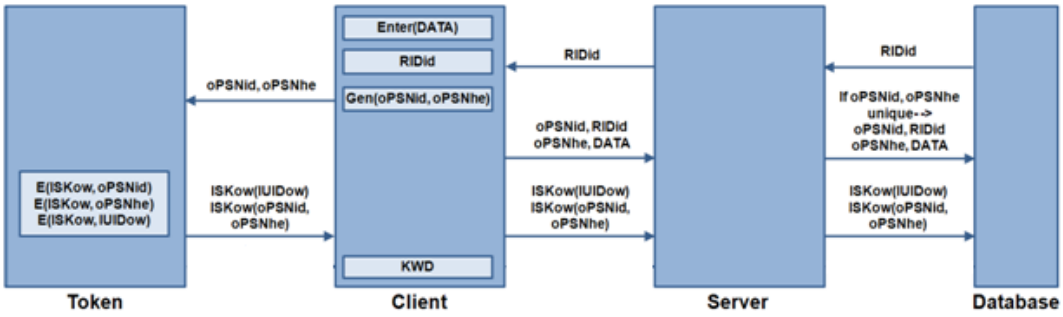

Figure 5.8: The data insertion workflow [67].

Data Retrieval

Data Retrieval involves selecting the respective pseudonym and retrieving the referenced health data record. This workflow is applicable for both data owner and authorized. The only difference is that the owner uses primarily the root pseudonyms for data retrieval, while the authorized relies on the shared pseudonyms. The precondition for this workflow is that the user is authenticatd and that at least one health data record is stored in the database. The workflow comprises following steps, as described in [67], which are also depicted in figure 5.9:

1. The data requestor retrieves legally accessible pseudonyms from the database and selects the pseudonym referenced with the desired health data record.
2. The data requestor then transfers the health pseudonym to the database engine (and the identification pseudonym to the database engine if required) which returns the requested data.

Data Pseudonymization

This workflow describes the procedure of pseudonymizing (health) data which already exist in the database. The workflow assumes that the data records are already depersonalized and that data records are separated into one identification record and multiple health data records. The workflow comprises following steps:

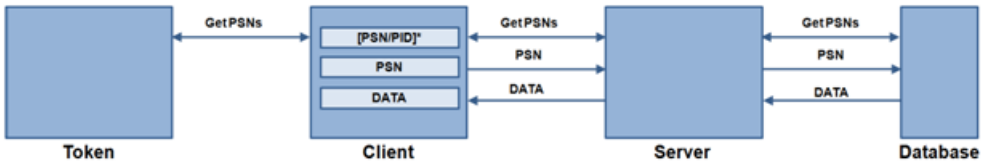

Figure 5.9: Data retrieval [67].

1. As a first step, record identifiers are retrieved from the database.

2. Then, for each health record an identification and a health pseudonym are created. Furthermore, the data owner encrypts his or her user identifier.

3. For each health record, the identification pseudonym is referenced with the identification record and the health pseudonym with the health record.

4. For each helth record, the owner encrypts both pseuondyms with his inner symmetric key and stores them along with his user identifier as a relation in the database.

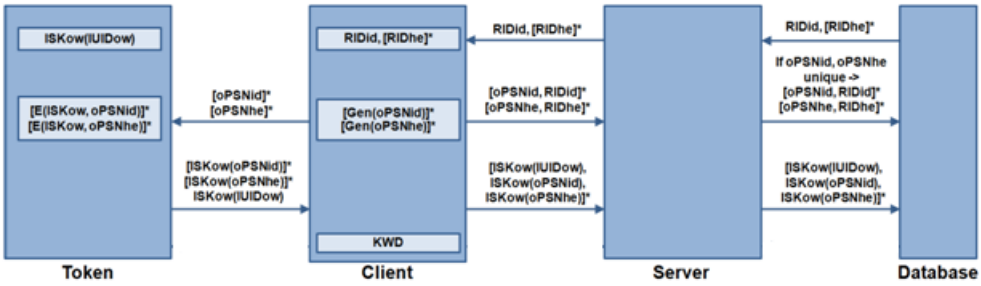

Figure 5.10: Data pseudonymization [67].

5.3 Problem analysis

In this section we apply the so called 'Threat Modeling' approach (described in [15]) in order to identify the system's security requirements. First, we create a data flow diagram (DFD) of the system. In this diagram the system's components, communication links between these components and external entities which interact with the system are represented. Then, based upon the data-flow diagram, we identify potential threats that could menace the system's security. Finally, on the basis of determined threats, we elicitate the required security properties. These properties, also called security requirements, are enforced by mechanisms utilized by the system, such as digital signatures or personal identification numbers (PIN).

Creating the data-flow diagram

In general, when using data flow diagrams, circles represent processes which perform discrete tasks. Processes can be located on a single or on distributed machines. Rectangles represent external entities which interact with the system and which drive the application. The system or the application itself cannot control the external entities. Parallel lines represent persistent data storage such as files and databases. In our case, the data storage is realized by a database. The data flow, or communication in other words, is represented by arrowed lines, while dotted lines stand for trust boundaries. There is a trust boundary between the security token and the remaining components of the system, and there is also a trust boundary between the users of the system and the system itself. The data flow diagram, which is depicted in figure 5.11, represents the basis for further analysis.

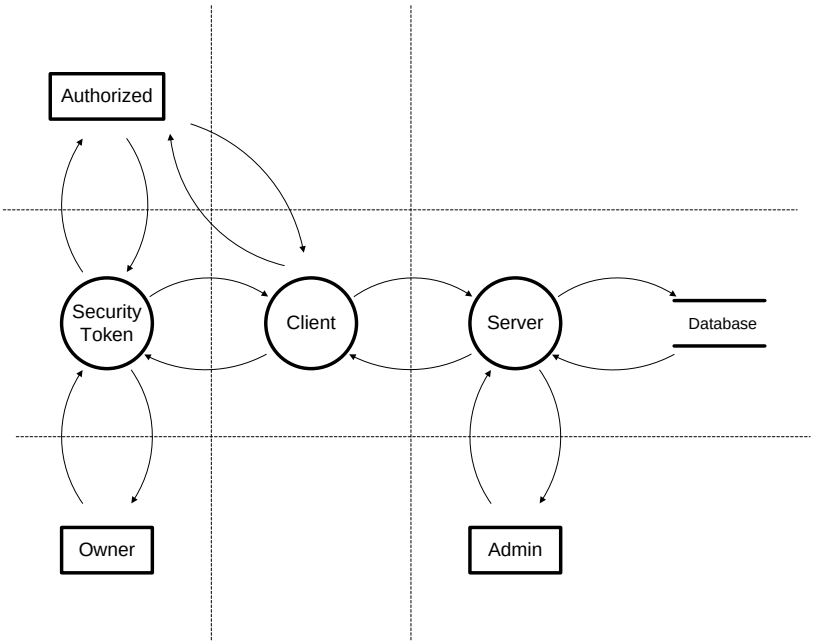

Figure 5.11: Context diagram for PIPE.

Table 5.1: Mapping threats to DFD elemenes. [15]

DFD Element Type	S	T	R	I	D	E
External Entity	x		x			
Data Flow		x		x	x	
Data Store	x	x	x	x	x	x
Process	x	x	x	x	x	x

Identifying threats to system's security

Within the PIPE system, valuable assets such as health data records, corresponding pseudonyms and encrypted keys are stored in a database and need protection. Furthermore, all communication channels, processes and external entities need protection as well. That is, each element of the system can be subject to an attack. However, not all attacks can be applied to all elements in the data-flow diagram. The nature of the attack is determined by the type of the DFD element. For instance, data flow elements can be subject to tampering, information disclosure and denial of service attacks [15]. Thus, in order to identify all threats that could menace the system's security, we applied the mapping depicted in table 5.1 which is based upon a threat mapping pattern proposed and described by the authors in [15]. We slightly modified the original pattern in a way such that data stores are treated in the same way as processes, as databases are in general realized by processes and not by simple files. The difference to the original mapping pattern is that the data store can as well be subject to spoofing and elevation of privileges threats. For the resulting security threats, see table 5.3 and figure 5.12.

Table 5.2: Mapping threats to security properties

Threat Type	**Security Property**
Spoofing	Authentication
Tampering	Integrity
Repudiation	Non-Repudiation
Information Disclosure	Confidentiality
Denial of Service	Availability
Escalation of Privilege	Authorization

Identifying security requirements

In order to mitigate a certain threat, an appropriate security property has to be enforced. In table 5.2, a mitigating security property is given for each threat. Thus, we have to map each threat from table 5.3 and figure 5.12 to the corresponding security property. The result is depicted in table 5.4 and illustrates the security requirements needed to secure the PIPE system. For instance, each external entity has to fulfill the authentication (AEN) and the non-repudiation (NON) security property. Thus, users interacting with the PIPE system have to be authenticated first, and the claims they make during the interaction with the system must not be deniable. As well, each data flow has to be secured by complying to the confidentiality (CON), integrity (INT) and the availability (AVA) security requirements. In security related literature, these properties are also renowned as the CIA properties for short. A further component that has to be protected is the data store, which is implemented as a database in our case. The database has to be protected by the aforementioned CIA security requirements as well as by the non-repudiation (NON) security property. Clearly, as sensitive information is stored and retrieved from the database, the access to it has to be non-deniable. Lastly, processes have to be protected as well. According to table

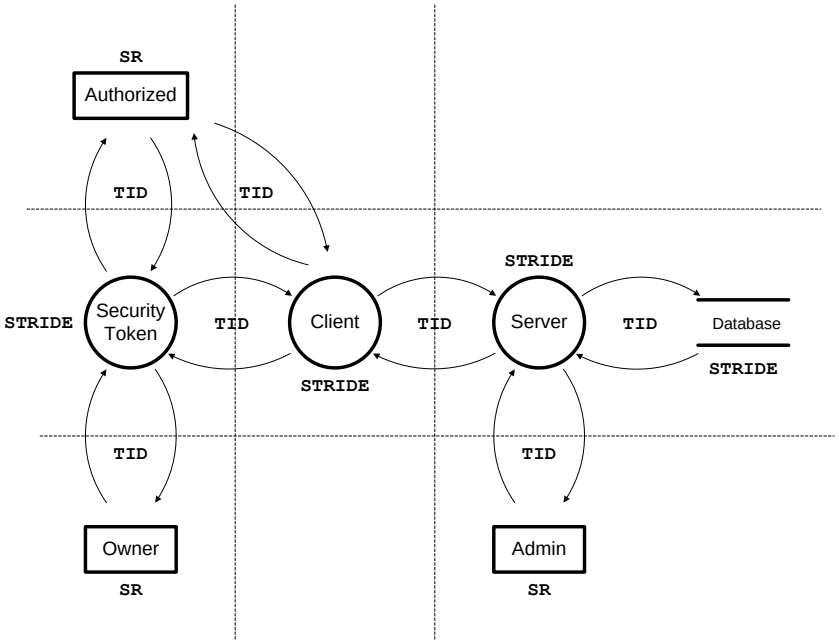

Figure 5.12: Security threats for the PIPE system.

5.1, each process in a data flow diagram has to be protected from all threats. This results in the requirement that for each process all security properties have to be fulfilled.

Selecting relevant security requirements

Even if all security requirements depicted in table 5.4 are needed to prevent the system from security threats, we do not intend to check and to verify them all. The reason is that the PIPE system relies on trusted components, such as the security token or the underlying operating system, and the security properties enforced by these trusted components are assumed to be assured. For instance, the interface between the security token and the end user is considered to be secure, since both the design and implementation of the security token are not part of the PIPE system. Furthermore, as external entities do not participate in the described workflows (only processes do), they require no protection. In turn, communication links between the client and the server components are part of the PIPE system. Hence, security threats resulting from these communication links have to be mitigated by enforcing adequate security properties. This implies that confidentiality and integrity of transported medical data are not assured and need to be verified.

For the security token, spoofing, elevation of privileges and repudiation threats are those which have to be mitigated by security mechanisms provided by the PIPE system. Tampering (i.e. reprogramming the token), denial of service (i.e. overloading the token) and information disclosure (i.e. reading out cryptographic keys, personal identification numbers etc.) should be

Table 5.3: Security threats for the PIPE system

DFD Elements in PIPE	S	T	R	I	D	E
External Entity Owner	x		x			
External Entity Authorized	x		x			
External Entity Administrator	x		x			
Data Flow Owner ↔ Security Token		x		x	x	
Data Flow Authorized ↔ Security Token		x		x	x	
Data Flow Authorized ↔ Client		x		x	x	
Data Flow Server ↔ Administrator		x		x	x	
Data Flow Security Token ↔ Client		x		x	x	
Data Flow Client ↔ Server		x		x	x	
Data Flow Server ↔ Database		x		x	x	
Data Store	x	x	x	x	x	x
Process Security Token	x	x	x	x	x	x
Process Client	x	x	x	x	x	x
Process Server	x	x	x	x	x	x

Table 5.4: Security requirements for the PIPE system

DFD Elements in PIPE	AEN	INT	NON	CON	AVA	AOR
External Entity Owner	x		x			
External Entity Authorized	x		x			
External Entity Administrator	x		x			
Data Flow Owner ↔ Security Token		x		x	x	
Data Flow Authorized ↔ Security Token		x		x	x	
Data Flow Authorized ↔ Client		x		x	x	
Data Flow Server ↔ Administrator		x		x	x	
Data Flow Security Token ↔ Client		x		x	x	
Data Flow Client ↔ Server		x		x	x	
Data Flow Server ↔ Database		x		x	x	
Data Store	x	x	x	x	x	x
Process Security Token	x	x	x	x	x	x
Process Client	x	x	x	x	x	x
Process Server	x	x	x	x	x	x

Table 5.5: Selected security requirements for the PIPE system

DFD Elements in PIPE	AEN	INT	NON	CON	AVA	AOR
Data Flow `Security Token` ↔ `Client`		x		x		
Data Flow `Client` ↔ `Server`		x		x		
Data Flow `Server` ↔ `Database`		x		x		
Process `Security Token`	x		x			x
Process `Client`	x		x			x
Process `Server`	x		x			x

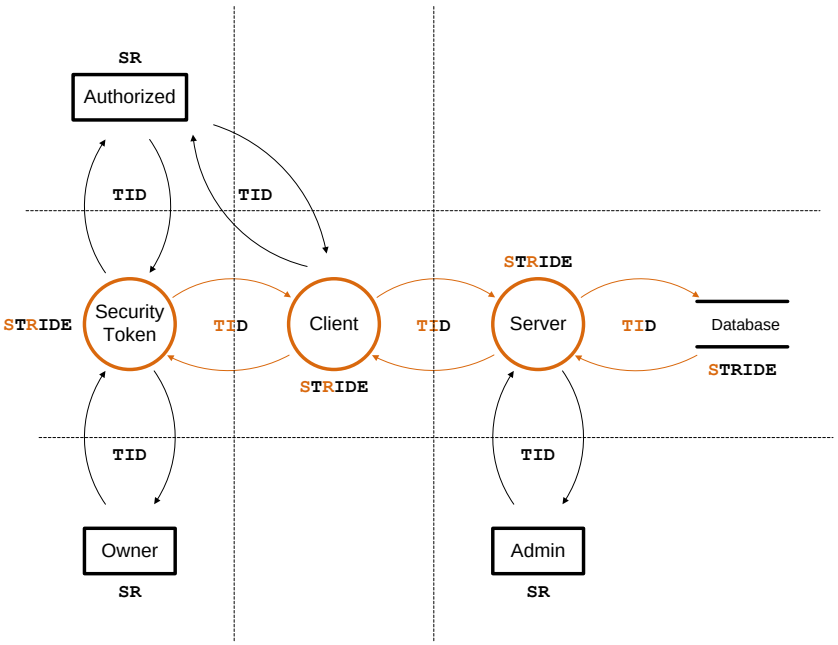

Figure 5.13: Selected security requirements for the PIPE system.

mitigated by the manufacturer, since the token is a trusted security device. The PIPE system has to ensure that the security token is authenticated, that actions performed by the user who possesses the token cannot be denied, and it has to ensure that authenticated users are indeed authorized to perform these actions. Thus, authentication, authorisation and non-repudiation security properties are required.

Authenticity and non-repudiation security properties of the client application are required, since it participates in the session which is established by the security token and the server. Authorisation, availability, confidentiality and integrity, on the other hand, should be provided by the utilized operating system, since only medical practitioners and system administrators should be allowed to use the client workstation (e.g. by providing a valid password). Furthermore, organisational policies should enforce correct usage of the original, unaltered version of the client application, and not a malicious copy provided by an adversary (e.g. by enforcing the usage of signed executables). The same reasoning which has been provided for the client application applies to the server and the database as well. Lastly, communication channels between the involved parties should be secured i.e. by using a secured transmission protocol.

In summary, the security analysis of the PIPE system is strongly focused on workflows and communication links between the processes participating in and realizing these workflows. In table 5.5 security properties handled by the PIPE system are depicted. In figure 5.13, entities and data flows that need protection as well as security requirements which have to be verified, are highlighted.

CHAPTER 6

Selecting the Appropriate Method

In order to analyze the PIPE system regarding its security properties, both kinds of models, dynamic and static models, have to be analyzed. Static models describe domain entities and relations between these entities, and are usually represented as class or EER diagrams. Within such models, security requirements are specified as object constraints, i.e., expressed in OCL. With dynamic models, which describe the worklflows presented in chapter 5, protocols are described which have to preserve secrecy and integrity of exchanged information. In order to analyze both kinds of models, appropriate methods have to be selected. For instance, dynamic models can be seen as interacting automata, and thus can be analyzed by applying model checking techniques. In turn, static models can be analyzed by applying first order logic analysis methods. Below, appropriate approaches are selected and justifications are provided.

6.1 Static Model Analysis

The static PIPE model illustrated in section 5.2 describes the system's database schema which contains domain entities such as users, keys, pseudonyms and data records. This analysis answers the question whether unauthorised users are able to connect patient identities to corresponding pseudonyms without knowing the required secret, i.e., without knowing the patient's private key. In other words, in this section we intend to find out whether users who have access to the database, such as system administrators for example, are able to connect separated pieces of information (such as a patient's identity and the corresponding patient's pseudonyms) in order to reconstruct the patient's anamnesis without being authorised (i.e. without knowing the patient's private key). According to comparison results given in chapter 4, the following methods are suitable for static model analysis and, at the same time, offer an appropriate tool support:

- *SecureUML*
- *UMLsec*
- *Alloy*

However, even if all quoted methods can be applied to static models analysis, the expressivity of the methods and the context in which they can be applied are different. In UMLsec, for instance, classes are annotated with stereotypes which represent security requirements such as secrecy and integrity. These annotated security requirements are verified when the corresponding activity diagrams are analysed. For instance, let the class *SecretData* be annotated with the *«secrecy»* stereotype. Furthermore, in the corresponding activity diagram, let a message containing an instance of *SecretData* be sent unencrypted. The obvious result of this simple example is a detected security violation. However, in case the message would not be sent over the unsecured channel, no security breach would emerge. Thus, in UMLsec, static models are analysed together with the respective dynamic models. This implies that for our particular case there is no benefit in applying UMLsec, simply because of our different application context. SecureUML, on the other hand, was designed to model and express RBAC (role-based access control) policies. Thus, this method as well is not suitable to analyse our particular model.

On the contrary, Alloy offers a powerful and an an expressive language suitable for modeling software designs. Furthermore, it provides a first-order relational logic which allows for specification of constraints which restrict the model and facts which define rules valid for the model. However, even if the method is appropriate for modelling domains and the corresponding properties, the approach is not necessarily adequate for analysing our particular problem. Below, we will present our model specified in the Alloy specification language, which problems arised during the analysis and what needs to be done in order to obtain more information from the model and thus make the analysis more valuable.

In its basic form, the verification of the PIPE system's data model can be reduced to simple type checking, since no dynamic behavior is described and no inference rules are specified in the data model. Type checking in this particular context can be described as verifying whether the information of interest is available in plaintext or ciphertext. Thus, if the information is available in ciphertext, the confidentiality remains preserved. Allowedly, this is trivial and no tool support is needed to check this property. Hence, the data model has to be improved and made more realistic. For this reason, in the first instance we have modeled entities from the problem domain, depicted in figure 6.1, in the Alloy specification language (see appendix I). The corresponding domain facts are modeled in the code snippet illustrated in appendix J. An example for such a fact is that for each shared pseudonym (*SharedPseudonym*), the encrypted data owner identities (IUIDow) are identical, when decrypted. In figure 6.2, the resulting domain's metamodel is depicted.

Subsequently, we have added cryptographic concepts to the model, such as symmetric and asymmetric cryptographic keys, and the corresponding operations like the encryption and decryption functions. The snippet of code depicted in appendix H illustrates how we modeled cryptography. In particular, we modeled concrete values as either readable (*PlainText*) or non-readable (*CipherText*, that is encrypted plaintext). Keys are readable, and can either be public, private or symmetric. Each key is associated with a decrypting and an encrypting function, whereas the first maps ciphertext to plaintext and the second vice versa.

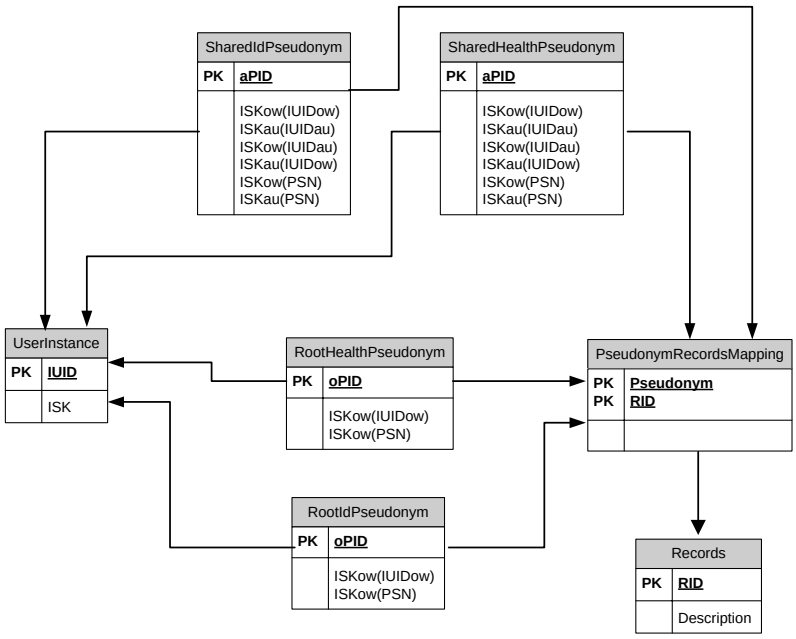

Figure 6.1: An excerpt from the PIPE static data model [67].

In oder to verify whether the modeled system's security can be violated, we formulated a constraint which describes an unsecure state. In particular, we characterised such a state as a situation which occurs when a user obtains an encrypted pseudonym of some other user by encrypting a plaintext pseudonym which he or she has obtained by retrieving it from the *PseudonymRecordsMapping* table with a key that the attacker has access to. In other words, this situation occurs when an attacker tries to brute force the plaintext pseudonyms (contained in the *PseudonymRecordsMapping* table) by encrypting them with the available keys (e.g. all public keys which stored in the database), and comparing them subsequently with the encrypted pseudonyms contained in the *Pseudonyms* table. This constraint is violated in case the keys are not unique and a public key available to the attacker can decrypt a pseudonym or a symmetric key which then can be used to decrypt the patient's pseudonyms. A constraint which formulates the described situation is given below in the Alloy specification language.

```
pred securityViolation()
{
  some p : PlainText, c : CipherText, u1, u2  : User, psn :
     RootPseudonym | (c == psn.enc_ISKow_PSNid) and
       (p==getISK[u1].dec[psn.enc_ISKow_PSNid]) and
       (c == u2.OPuK.enc[p])
}
```

The constraint states that, if there is a ciphertext *c* which is used as a pseudonym, and it can be reconstructed by encrypting a plaintext pseudonym *p* by applying the attackers public key *u2.OPuK* (or any other key which is available to the attacker), then the security of the system is violated. Unfortunately, although we have modeled only an excerpt of the existing static data model, the size of the model exceeded the analyser's capabilities, which resulted in a crash after 10 minutes of computation. We could have simplified the model further, but this would lead to trivial models, in which the problem would be reduced to simple type checking.

Another form of analysis would be to formulate axioms and inference rules, something like a program written in a logic programming language like Prolog, which would allow the attacker to make implicit knowledge explicit. It would be possible to write such specifications in Alloy, since Alloy is capable of expressing first-order logic formulas. However, within this thesis we do not focus on expressing data mining rules as deduction formulas. Nevertheless, the question is interesting and we may try to answer it in our future work.

6.2 Dynamic Model Analysis

According to comparison results illustrated in chapter 4, following methods are suitable for verification of dynamic behavior and, at the same time, offer an appropriate tool support:

- *UMLsec*
- *Avispa*
- *Symbolic Model Verifier*
- *Alloy*

All enumerated methods are expressive enough to describe dynamic behavior of software systems. Thus, all these methods are capable of fulfilling our needs. However, there are differences concerning tool usage and the level of abstraction between the different methods. For instance, even if UMLsec has been designed to deal especially with security, the method and the toolchain are, in our opinion, too complex and too less intuitive to use. Furthermore, no solid tutorial is provided which would be a great help to learn to use the tool chain. Therefore, we decided not to use the UMLsec methodoogy.

In contrast, other methods provide very usable and well documented tools. Amongst the other methods, the main difference lies in the abstraction level. For instance, the Avispa tool takes a high-level description of a protocol as input, whilst when using the Symbolic Model Verifier all states and transitions between them have to be modeled explicitly, and on a lower level of abstraction, which is more error prone. We will take the original Needham-Schreoder public-key authentication protocol [30] as an example.

1. $A \rightarrow B$: $\{N_A\ , \mathrm{A}\}_{K_B}$

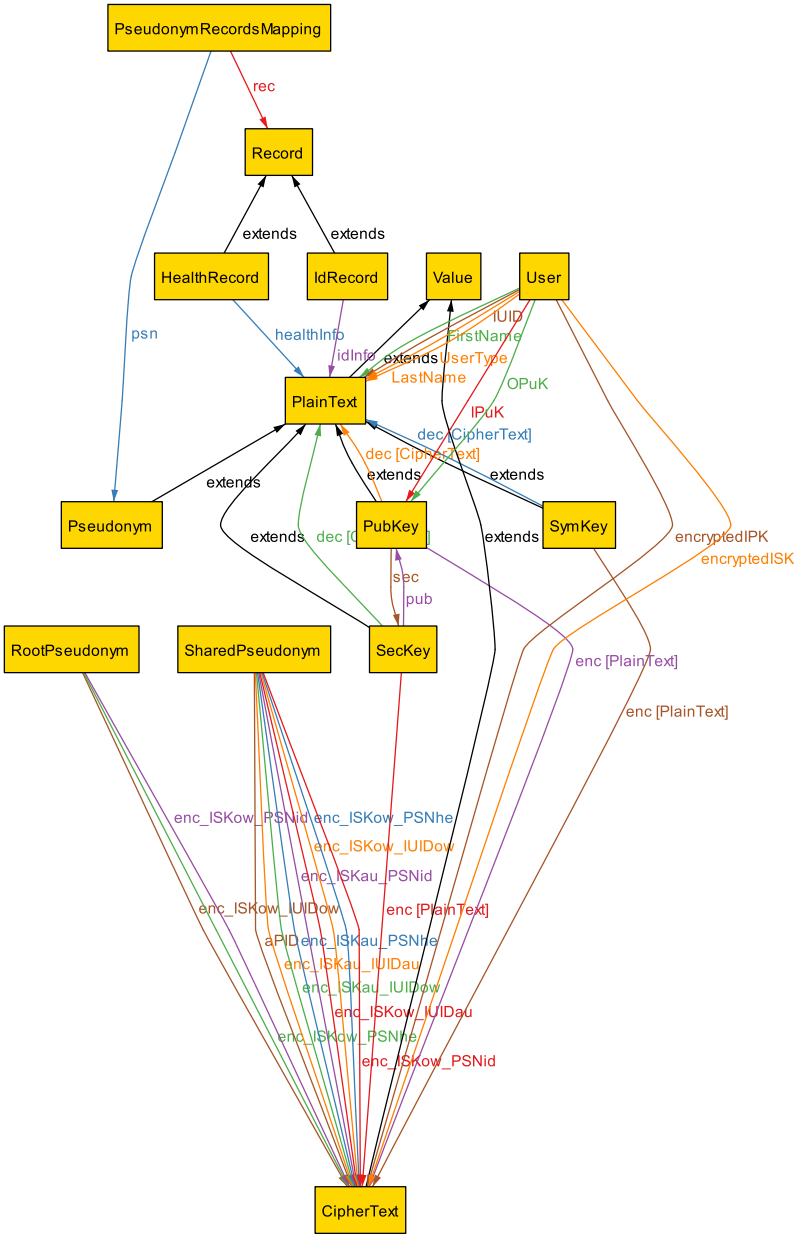

Figure 6.2: Domain metamodel.

2. $B \rightarrow A$: $\{N_A\ , N_B\}_{K_A}$
3. $A \rightarrow B$: $\{N_B\}_{K_B}$

The protocol assumes the use of a public-key encryption algorithm. We use key labels such as K_A to denote the A's public key, and nonce labels such as N_A to denote a nonce generated by participant A. The exchange begins with A transmitting a nonce and his identity, encrypted with B's public key. Then, B answers by transmitting the received nonce N_A and a newly created one, N_B. By this way, B confirms his identity since only B is able to read the message. At the same time, he forces A to authenticate. In the last step, A authenticates himself, since only A knows the nonce N_B.

The protocol is rather simple, and it seems to be correct. However, despite its simplicity, the protocol is flawed. In [31], the flaw and the corresponding attack are described. Though, we will not try to prove the protocol wrong here, but to show what needs to be done in order to transform a protocol as simple as this into a state machine which subsequently can be verified. In order to model the protocol as a set of states and transitions among them, a global view of the system based on a suitable abstraction is needed. A common approach is to model the global system state, which is based on communicated messages. In our case, the global state depends on the agents' and the intruder's knowledge. Initially, communication participants A and B both know their corresponding public keys and their nonces, while the intruder doesn't know anything. After step 1, A's knowledge remains the same. Participant B, in turn, now knows A's identity and A's nonce. And the network, which can be seen as the intruder, knows $\{N_A\ , \mathrm{A}\}_{K_B}$. Thus, after each protocol step, the global state changes because the participants and the intruder consequently gain new knowledge. Figure 6.3 illustrates this correlation.

In order to represent all possible scenarios, all possible states of knowledge and the transitions between them need to be modeled. Modeling security protocols by this way is hard and error prone. Thus, tools and techniques were proposed in order to make designing security protocols accessible to a broader audience. Amongst the techniques examined in chapter 4, the Avispa tool is a promising candidate for verifying security protocols. It provides a high level protocol specification language (HLSPL), which allows for high-level protocol descriptions without having to think about all the possible knowledge states and the transitions amongst them. Example code for the Needham-Schreoeder protocol is illustrated in listing 7.

Similarly, the Alloy language allows as well to describe security protocols on a higher level of abstraction, since the method is based on first-order logic. In [68], the authors present an approach which they call *knowledge flow analysis*, and use the Alloy methodology in order to model and analyse security protocols. They write that

> *the key idea behind knowledge flow analysis is the observation that, at the most basic level, the purpose of a security protocol is to distribute knowledge among its legitimate participants. A protocol is flawed if it allows an intruder to learn a value that is intended to remain strictly within the legitimate principals' pool of knowledge.*

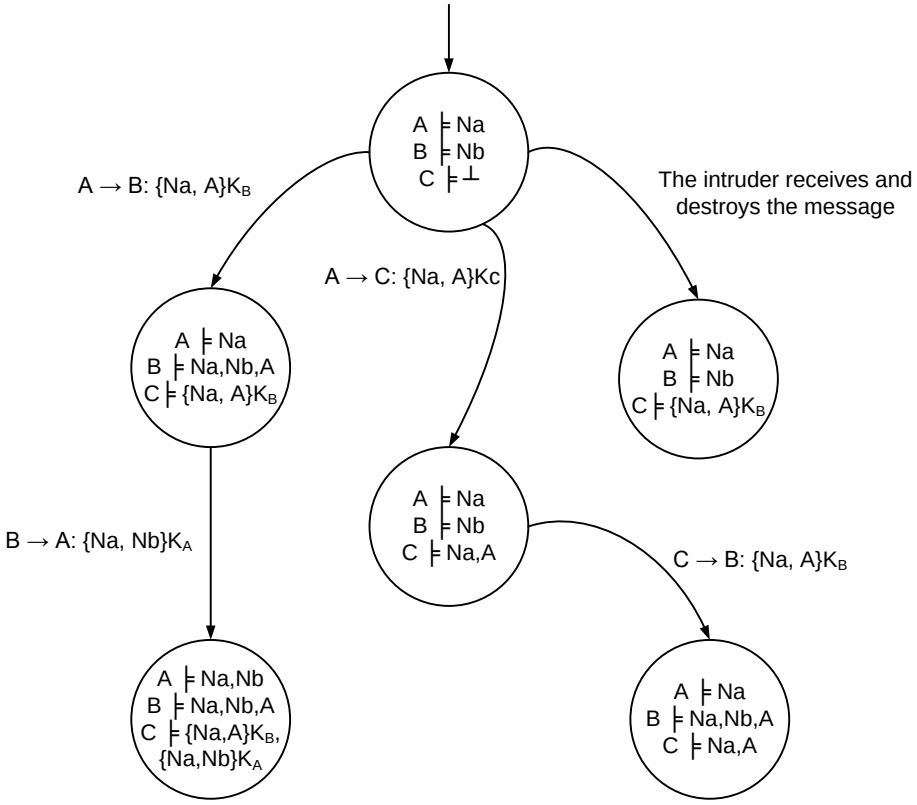

Figure 6.3: State diagram.

They provide a uniform framework suitable for expressing the actions of principals and assumptions on intruders. In listing 6, an excerpt from the model describing the Needham-Schroeder protocol, based upon the mentioned framework, is given. A description of the same protocol, but in contrast written in a low-level automata description language intended for a general purpose model-checker, is much less intuitive. Furthermore, it is more complex, it requires more lines of code and is more error prone. For an examplified description of the Needham-Schreoder protocol written in the SMV modeling language, see appendix A. Because of the enumerated disadvantages, we have rejected the general purpose model-checker approach and have chosen the Avispa method. Furthermore, we decided against Alloy since the language used for protocol description in Avispa, HLPSL, is much closer to the intuitive protocol description (also known as the 'Alice-Bob' notation) and thus makes the modeling task easier and less error prone for the developer. A detailed descriptioin of the HLPSL syntax and semantics is given in [69].

```
sig Identity extends AtomicValue {}
pred IdentitiesAreKeys(){
  all p : Principal | some p.owns & Identity &&
  Ciphertext.key in Identity
}

pred PrimitiveRules(x : set Value, v : Value){
  Encryptor(x,v) || Decryptor(x,v) ||
  NonceGenerator(x,v)
}

pred ProtocolRules(x : set Value, v : Value){
  v in Ciphertext && {
  (x : some Oscar.draws &&
    let text = v.plaintext, n = text & Nonce |
      #text = 2 && one n && n.seed in AtomicValue &&
      n.id = text & Identity) ||
      (x : one Ciphertext && (some n : seed.x |
        #x.plaintext = 2 && v.key in x.plaintext &&
        n.id = x.key &&
        v.plaintext = (x.plaintext - v.key) + n)) ||
      (x : one Ciphertext &&
        (some n : id.(x.key) & Nonce |
        #x.plaintext = 2 && n in x.plaintext &&
        v.plaintext = x.plaintext - n)
      )
   }
}

pred ApplyRules(){
  all v : Value | let x = Oscar.learns.v |
  some x <=> PrimitiveRules(x, v) ||
  ProtocolRules(x, v)
}
```

Listing 6: Excerpt from the model for the Needham-Schreoder protocol in Alloy [68].

```
role Alice(A,B:agent, Ka,Kb:public_key, SND,RCV:channel(dy))
    played_by A def =
      local State : nat, Na: text (fresh), Nb: text
      init State = 0
      transition
        0. State = 0 /\ RCV(start) =|>
           State'= 2 /\ SND({Na'.A}_Kb)
                     /\ witness(A,B,na,Na')
        2. State = 2 /\ RCV({Na.Nb'}_Ka) =|>
           State'= 4 /\ SND({Nb'}_Kb)
                     /\ request(A,B,nb,Nb')
end role

role Bob(A,B: agent, Ka,Kb: public_key, SND,RCV: channel(dy))
    played_by B def =
      local State : nat, Na: text, Nb: text (fresh)
      init State = 1
      transition
        1. State = 1 /\ RCV({Na'.A}_Kb) =|>
           State'= 3 /\ SND({Na'.Nb'}_Ka)
                     /\ witness(B,A,nb,Nb')
        3. State = 3 /\ RCV({Nb}_Kb) =|>
           State'= 5 /\ request(B,A,na,Na)
end role

role Session(A, B: agent, Ka, Kb: public_key) def =
    local SA, RA, SB, RB: channel (dy)
    composition
      Alice(A,B,Ka,Kb,SA,RA) /\ Bob (A,B,Ka,Kb,SB,RB)
end role

role Environment() def =
    const a, b: agent, ka, kb, ki: public_key
    knowledge(i) = {a, b, ka, kb, ki, inv(ki)}
    composition
      Session(a,b,ka,kb) /\
      Session(a,i,ka,ki) /\
      Session(i,b,ki,kb)
end role

goal
    Alice authenticates Bob on nb
    Bob authenticates Alice on na
end goal

Environment()
```


Listing 7: Example code for the Needham-Schreoder protocol in HLSPL [70].

CHAPTER 7

Results of the Evaluation

Simplified, PIPE consists of two phases. During the first phase, users are authenticated and a symmetric session key is generated for further communication. Thus, after successful authentication, in the second phase, all communication is secured by the previously generated symmetric session key. Within the second phase, authenticated users can modify or retrieve data, and authorize other users to access their medical records. Data retrieval and modification is performed by the client application, which implicates that the client application as well has to participate in the session, and that it has to know the session key. Thus, our approach to verify the PIPE system comprises analysing the authentication phase, analysing how the client application gets the session key, and whether mechanisms utilized by workflows belonging to the second phase provide security even in case the attacker knows the session key. In order to verify all the workflows, we use the AVISPA validation tool, which analyses specifications of security protocol. We have already described it in section 4.

As a first step, we verify whether the authentication phase is performed correctly. That is, we check whether the involved parties are authenticated against each other, as well whether the generated session key remains undisclosed. Then, we validate the workflows belonging to the second phase. We check how the client application has obtained the session key, and we validate whether the workflow is still secure in case an adversary can observe the communication (e.g. the session key has been broken). At the end of the chapter, we summarize the results.

7.1 Authentication

Based upon the recognised security threats which we have described in chapter 5.3, we have identified several security requirements the system has to provide. Thus, for each required security property, we will state a reason why the system satisfies the property or why it doesn't. In this workflow, the server and the user have to authenticate against each other in order to establish a secure channel. Thus, after the authentication, both parties are in possession of a symmetric session key which is then used to secure the subsequent communication. The session key is

a shared secret between the server and the user, and it should be made available to the client application for subsequent workflows (i.e. workflows which belong to the second phase).

In the PIPE design specification [67], it is not stated explicitly that the database is installed on the server machine. Thus, it would not be contrary to the specification to run the database system on a dedicated machine. Therefore, in order to be able to analyse the workflow properly, we decided to model and to verify both scenarios.

Single Machine

The communication protocol used in the 'single machine' scenario is given below in the so-called Alice-Bob notation. The expression U represents the user and S the server. Expressions N_U and N_S represent nonces, $IUID_U$ represents the system's internal user identifier, and the expression $K_{session}$ represents the generated session key. The protocol of the workflow is given below:

1. $U \rightarrow S$: $\{N_U, IUID_U\}_{K_S}$
2. $S \rightarrow U$: $\{N_U, N_S\}_{K_U}$
3. $U \rightarrow S$: $\{N_S\}_{K_S}$
4. $S \rightarrow U$: $\{K_{session}\}_{K_U}$

A detailed description of the workflow is given in section 5.2. The corresponding HLPSL source code, which is needed for the tool to perform the verification, is provided in appendix B. First, we will verify the authenticity of both communication participants. Then we will validate confidentiality and integrity of the established session key and the transmitted nonces. Lastly, we will check the non-repudiation of origin and non-repudiation of receipt security properties which are required to prove that a session indeeed has been established. In the following table 7.1, the security requirements that we have to verify are outlined.

Table 7.1: Selected security requirements for the Authentication workflow

DFD Elements in PIPE	**AEN**	**INT**	**NON**	**CON**	**AVA**	**AOR**
Data Flow `Security Token` ↔ `Client`		x		x		
Data Flow `Client` ↔ `Server`		x		x		
Process `Security Token`	x		x			x
Process `Server`	x		x			x

Furthermore, both communication parties have to be authorised (in our particular case, authenticated users are authorised users) to perform actions on medical data records. First, we

will model the authenticity and confidentiality security properties. The requirement that the user authenticates the server is modeled by the expressions `witness(U,S,auth_nu,Nu')` and `request(S,U,auth_nu,Nu)`. The `witness` term means that agent `U` is a witness of the message `Nu'`. Supplementary, the `request` term means that agent `S` expects that agent `U` exists and that both agree on `Nu`. Thus, the server's authenticity is given when both predicates hold true. Likewise, the requirement that the server authenticates the user is modeled by expressions `request(S,U,auth_ns,Ns')` and `witness(S,U,auth_ns,Ns')`. Authenticity and thus integrity of the session key $K_{session}$ is modeled by expressions `request(U,S, auth_symkey,K')` and `witness(S,U,auth_symkey, K')`. The confidentiality (or secrecy) of the session key `K` is modeled by the expression `secret(K',sec_symkey,{U,S})`.

In fact, the protocol is a variation of the original public-key Needham-Schroeder protocol [30]. Unfortunately, the protocol adaptation was not carried out carefully since message 2 does not include the sender's identity [31]. This marginal difference allows for man-in-the-middle attacks which occur when an honest user initiates a session with an intruder. In our system, however, the client knows the server's public key, and thus he would not start a session with an adversary, since he would have to encrypt the initial message with the adversary's public key. Thus, even if the protocol is not correct, the fact that the user knows the server's public key prevents from exploiting the vulnerability. Nevertheless, the error shall be corrected. The attack in figure 7.1 shows how the secrecy of a nonce can be circumvented in case an honest user initiates a session with an intruder. Figure 7.2, on the other hand, shows how an intruder i can pretend to be an honest agent. Both attack traces are caused by the missing sender's identity in message 2 of the protocol. The corrected protocol is given below. Now, message 2 contains the sender's identity, and attacks depicted in figures 7.1 and 7.2 are prevented.

1. $U \rightarrow S$: $\{N_U, IUID_U\}_{K_S}$
2. $S \rightarrow U$: $\{N_U, N_S, \mathrm{S}\}_{K_U}$
3. $U \rightarrow S$: $\{N_S\}_{K_S}$
4. $S \rightarrow U$: $\{K_{session}\}_{K_U}$

Unfortunately, even if secrecy of all transmitted messages has been achieved and both communication partners have been successfully authenticated, the protocol is still vulnerable to replay attacks. The reason for the vulnerability is that the last message of the protocol, namely message 4, does not have to be authentic. That is, there is no guarantee that it was the server that has sent the session key, since anybody else could have sent it. Anyone could take the user's public key, encrypt an old and broken session key with it, and send it as the last message of the protocol. In figure 7.3, the attack trace for the replay attack is depicted.

In order to achieve the required authenticity, and thus the integrity of the session key, the last message has to contain, in addition to the session key $K_{session}$, a fresh secret known only to two

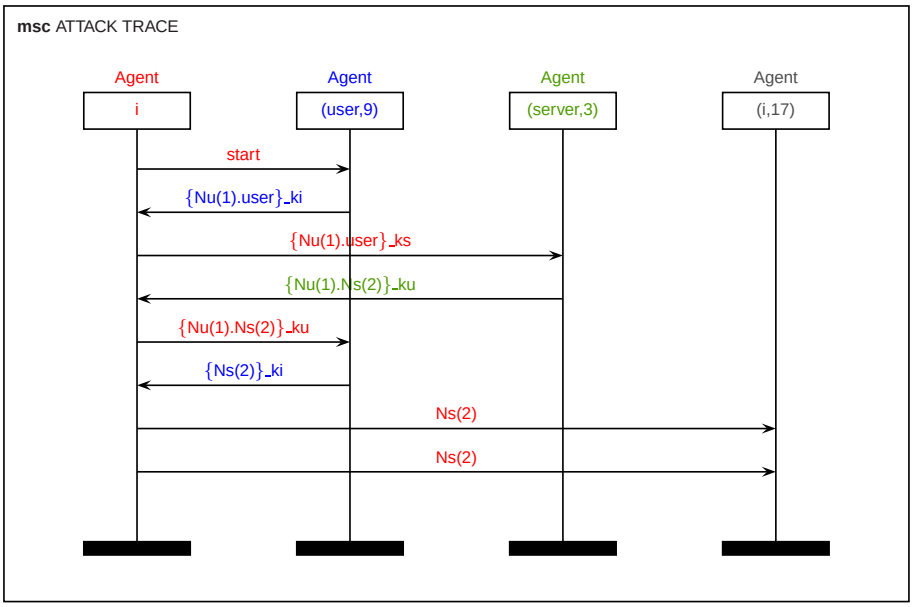

Figure 7.1: Attack trace compromising the secrecy of a nonce.

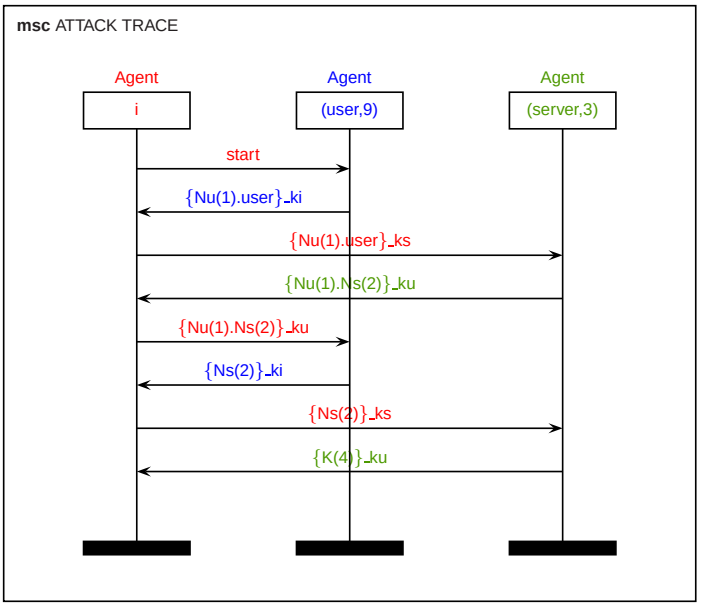

Figure 7.2: Attack trace compromising server's authenticity.

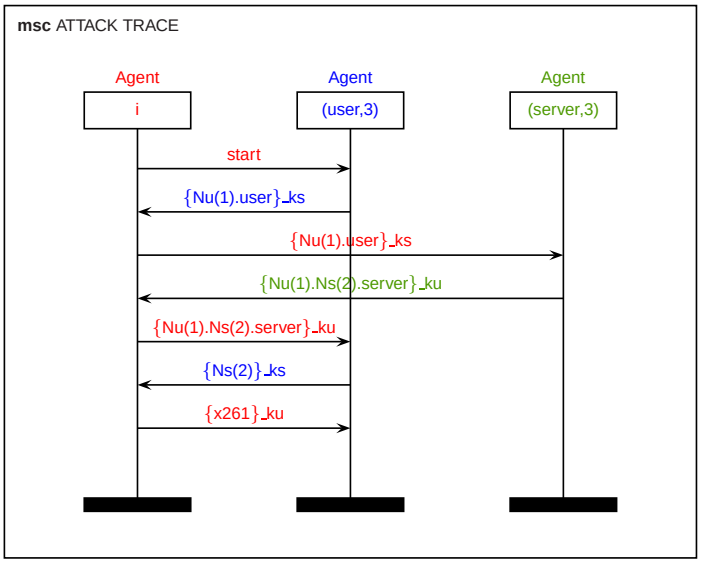

Figure 7.3: Replay attack enforcing an old session key.

communicating parties. The nonce N_U fulfills these requirements. Actually, after inserting N_U into the last protocol message, no further attacks are found by the AVISPA tool. Below is the corrected version of the protocol:

1. $U \rightarrow S$: $\{N_U, IUID_U\}_{K_S}$
2. $S \rightarrow U$: $\{N_U, N_S, \mathrm{S}\}_{K_U}$
3. $U \rightarrow S$: $\{N_S\}_{K_S}$
4. $S \rightarrow U$: $\{K_{session}, N_U\}_{K_U}$

So far, we have verified the confidentiality of exchanged nonces and the established session key, the authenticity and the integrity of the session key, as well as the authenticity of both communication participants. Thus, the non-repudiation security properties still have to be validated. In our particular case, non-repudiation of origin and non-repudiation of receipt properties are required to provide evidence such that no participant is able to deny having established a session. Thus, the non-repudiation of origin property (NRO for short) requires a guarantee that the server has sent the session key, and non-repudiation of receipt (NRR), on the other hand, requires a guarantee that the user has received the session key. The threat scenario for non-repudiation usually comes in the form of an honest agent communicating with a dishonest agent [71], who denies actions or events. For instance, a dishonest user could deny having established a session and thus deny that medical data has been inserted or changed. Certainly, a medical system should not allow such fraud. Thus, in non-repudiation protocols, participants

usually need protection from each other, rather than from an external adversary [71]. In [71], the authors map non-repudiation properties to authentication requirements, which works well for the Fair Zhou Gollmann Non-repudiation Protocol [72], which the authors took as example. They approximated a dishonest agent by an intruder model where the intruder is able to impersonate a dishonest agent, since a trusted third party (TTP) is used for authentication. In such a scenario, modeling a dishonest agent by the Dolev-Yao intruder model is possible, since an intruder can impersonate the dishonest agent. In our particular case, however, things are different, since a dishonest user who knows the secret nonces and has been correctly authenticated could act dishonestly, which does not conform to the Dolev-Yao intruder model. Thus, even if non-repudiation properties can be expressed as authentication problems, as shown in [73], other intruder models would be helpful in order to express scenarios in which agents behave dishonestly. Unfortunately, the AVISPA tool supports the Dolev-Yao intruder model only, and therefore our subsequent reasoning is performed manually and without tool support.

The last message in our protocol solves the problem of NRO, since no adversary could replay or create the message, as nonce N_U is known only to both protocol participants, the server and the user. Thus, in case of a dispute, the server could not deny that he has sent the message, since no other party except the user knows the nonce N_U. Unfortunately, even if the NRO security property is fulfilled (the server cannot deny that he has sent the message), the protocol does not protect against creation of false evidence. In other words, a dishonest user could claim that a session was established, even if it was not, since he as well knows the secret nonce N_U and the encryption key K_U, and is thus able to create the message and falsify evidence without receiving it from the server. In order to solve the problem, we need a message that could have been created by the server only. For this reason, the last message has to be encrypted with the server's private key, which we will denote with $_{inv(K_S)}$. Similar reasoning applies for the non-repudiation of receipt (NRR) security property, which is assured by an additional message that acknowledges the reception of $K_{session}$. The corrected protocol is given below.

1. $U \rightarrow S$: $\{N_U, IUID_U\}_{K_S}$
2. $S \rightarrow U$: $\{N_U, N_S, \mathrm{S}\}_{K_U}$
3. $U \rightarrow S$: $\{N_S\}_{K_S}$
4. $S \rightarrow U$: $\{\{K_{session}, N_U\}_{K_U}\}_{inv(K_S)}$
5. $U \rightarrow S$: $\{\{K_{session}, N_S\}_{K_S}\}_{inv(K_U)}$

Thus, we have shown how to enforce the authenticity, confidentiality, integrity and non-repudiation security requirements for the Authentication workflow in its 'single machine' varinat.

Separated Machines

The communication protocol used in the 'dedicated machine' scenario is given below. The expression U represents the user, S the server, and D the database. Expressions N_U and N_S represent nonces, $IUID_U$ represents the system's internal user identifier, and the expression $K_{session}$ represents the generated session key. A detailed description of the workflow is given in section 5.2. The corresponding HLPSL source code, which is needed for the tool to perform the verification, is provided in appendix C. Security requirements such as authenticity of protocol participants, as well as confidentiality and integrity of the session key are modeled as already shown in the previous section. The protocol of the workflow is given below.

1. $U \rightarrow S$: $\{N_U, IUID_U\}_{K_S}$
2. $S \rightarrow D$: $IUID_U$
3. $D \rightarrow S$: K_U
4. $S \rightarrow U$: $\{N_U, N_S\}_{K_U}$
5. $U \rightarrow S$: $\{N_S\}_{K_S}$
6. $S \rightarrow U$: $\{K_{session}\}_{K_U}$

Unfortunately, both authenticity and confidentiality security properties are violated. In figure 7.4, an attack found by the AVISPA tool is demonstrated and shows how the intruder i can gain access to the session key $K_{session}$. The essence of the problem is that the database is not authenticated. That is, even if the database does not provide confidential knowledge (since the only information released in plaintext is the user's public key), a security violation occurs. The reason is that there is no integrity or authenticity provided such that one can be sure that the information released by the database has not been tampered with, replayed or even completely created by someone else.

Initially, an honest user wishes to establish a communication with the server and initiates the authentication workflow by sending the message $\{N_U, IUID_U\}_{K_S}$. The server receives the message, and requests the user's public key from the database. The intruder, who listens on the wire, responds instead the database, and provides the server with a wrong public key, K_I. Thus, the original answer is overwritten. Then, as a next step in the protocol conversation, the server responds and transmits the message $\{N_U, N_S\}_{K_I}$, because he believes that K_I is the user's correct public key. The intruder intercepts the message, decrypts it, encrypts it again with the user's public key, and then forwards the message to the user. The user thinks the message comes from the server, as only the server should know the secret nonce N_U, upon which the server should be authenticated. The user decrypts the received message, and encrypts the server's nonce N_S in order to authenticate against the server. The encrypted nonce is then sent to the server, and the server believes both parties have been correctly authenticated. Then, the server encrypts the newly generated session key with the intruder's public key, and sends it to the intruder. From

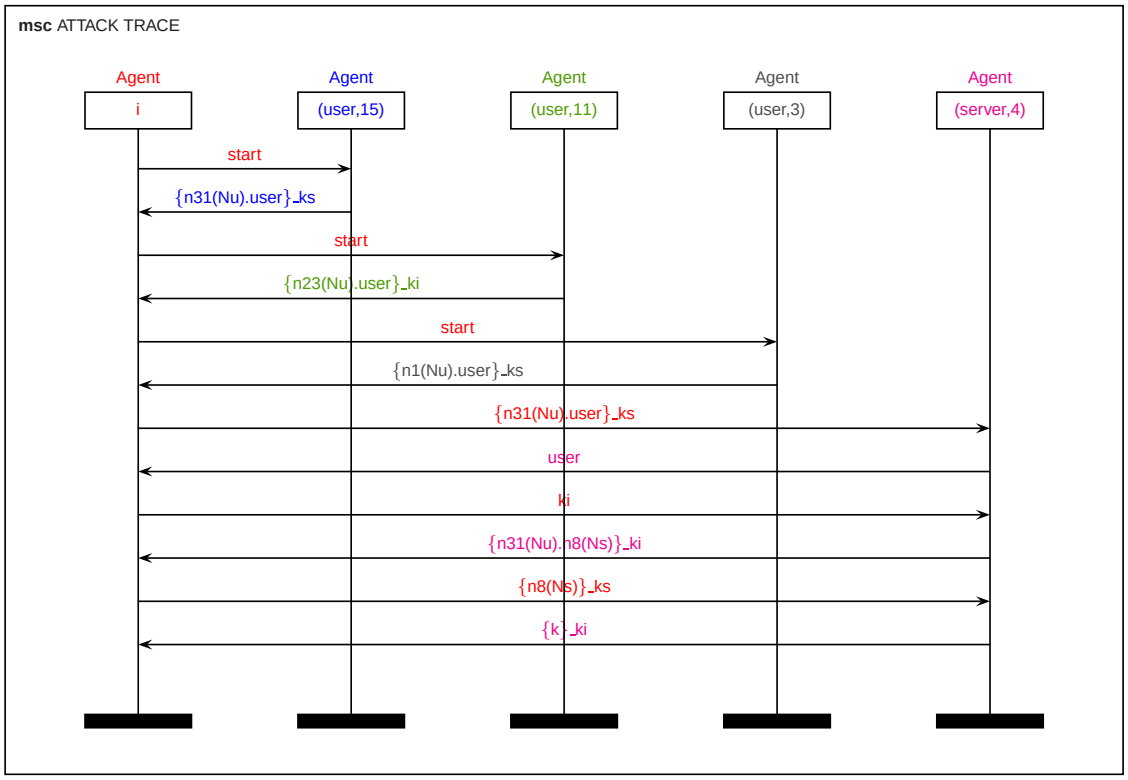

Figure 7.4: Attack trace for the authentication workflow.

that point, the intruder is able to observe the communication between the user and the server, and he is able to reconstruct the connection between user identities and their corresponding medical records.

In order to fix the protocol, we have to assure the integrity of the message sent by the database. For this purpose, the database as well has to hold an individual public and a corresponding private key. Thus, when the user's public key is sent back to the server, it has to be encrypted with the database's private key. By this way, the server can be sure that the received message has indeed been sent by the database. However, even if integrity is assured, an adequate protection against replay attacks is still missing. The reason is that there is no link between the intended user and the received public key, which would ensure that the received public key is indeed the users's public key, and not someone else's. For instance, an intruder could replay an old message, recorded in a previous session, and by this way force the server to encrypt the further messages with a wrong public key. In order to prevent such an attack, the user identifier has to be contained in the returned message. Thus, the user identifier and the user's public key have to be encrypted with the database's private key and sent back to the server. The corrected protocol is given below:

1. $U \rightarrow S$: $\{N_U, IUID_U\}_{K_S}$
2. $S \rightarrow D$: $IUID_U$
3. $D \rightarrow S$: $\{U, K_U\}_{SK_D}$
4. $S \rightarrow U$: $\{N_U, N_S\}_{K_U}$
5. $U \rightarrow S$: $\{N_S\}_{K_S}$
6. $S \rightarrow U$: $\{K_{session}\}_{K_U}$

However, the verification still fails since the Needham-Schreoder protocol has been applied in its original version, which is vulnerable to a man-in-the-middle attack, as already described in previos section. The attack trace generated by the AVISPA tool is depicted in figure 7.5. In the protocol's original version, the vulnerability allows an intruder I to impersonate an agent A and to set up a fake session with another agent B, which requires two simultaneous protocol runs [31]. During the first run, participant A establishes a valid session with the intruder I. In the second run, I impersonates an honest agent A and establishes a falsified session with another honest agent B, who believes to communicate with agent A. In our particular case, agent A corresponds to the security token (user), while the agent B corresponds to the server. In order to fix the protocol, we have to apply the Lowe's patch [31]. That is, instead of sending $\{N_U$, $N_S\}_{K_U}$ to the user, $\{N_U$, N_S, S$\}_{K_U}$ has to be sent. Now, the receipient is able to compare the sender's identity to the one stated in the encrypted message, which prevents from authentication falsification.

So far, we have verified the confidentiality and the authenticity security requirements. In order to tackle the non-repudiation problem, measures described in the previos section have to be applied here as well. That is, in order to fulfill the non-repudiation security properties (NRO and NRR), the protocol has to be enhanced in an analogous manner as already done and described in the previos section. For this reason, the message 6 of the protocol has to be widened in order to contain the nonce N_U, and it has to be encrypted with the server's private key in order to prevent creation of false evidence. Similar reasoning applies for the NRR security property, which is assured by an additional message that acknowledges the reception of the key $K_{session}$. The protocol is given below.

1. $U \rightarrow S$: $\{N_U, U\}_{K_S}$
2. $S \rightarrow D$: U
3. $D \rightarrow S$: $\{U, K_U\}_{inv(K_D)}$
4. $S \rightarrow U$: $\{N_U, N_S, \mathrm{S}\}_{K_U}$
5. $U \rightarrow S$: $\{N_S\}_{K_S}$

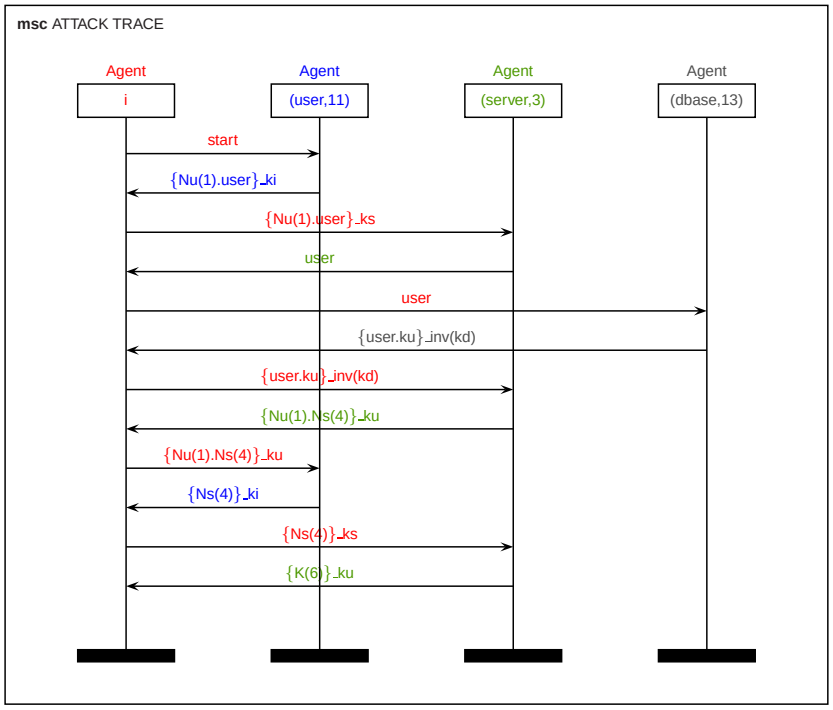

Figure 7.5: Attack trace for the authentication workflow without Lowe's fix.

6. $S \rightarrow U$: $\{\{K_{session}, N_U\}_{K_U}\}_{inv(K_S)}$
7. $U \rightarrow S$: $\{\{K_{session}, N_S\}_{K_S}\}_{inv(K_U)}$

7.2 Get Pseudonyms

This workflow describes how the pseudonyms associated with a user are retrieved. The expression U represents the user and the expression C represents the client application. The precondition for this workflow is an authenticated user. A detailed description of the workflow is given in section 5.2. Unfortunately, the PIPE specification [67] does not define how the client application obtains the session key which has been established between the user and the server during the authentication phase, albeit the session key is required for the client application to participate in the second phase workflows. For our analysis, we will assume that the client application has obtained the session key, and we will assume that the session key has been broken by the attacker. The communication protocol is given below:

1. $U \rightarrow C$: $\{IUID_U\}_{ISK_U}$
2. $C \rightarrow D$: $\{IUID_U\}_{ISK_U}$

3. $D \rightarrow C$: $\{PSN\}_{ISK_U}$
4. $C \rightarrow U$: $\{PSN\}_{ISK_U}$
5. $U \rightarrow C$: PSN

The user transmits his identifier $IUID_U$, which is encrypted with his inner symmetric key ISK_U, to the client. The client then forwards the message to the database. The database, in turn, retrieves the encrypted pseudonyms and transmits these to the client, who is then forwarding these to the user. After reception, the user can decrypt the pseudonyms and select the appropriate one. For the HLPSL implementation of the workflow, see appendix D. In case the session key $K_{session}$ has been broken, the wokflow cannot protect the confidentiality of the user's medical data, since the internal user identifier $IUID_U$ and the corresponding pseudonyms can be linked. The error in the protocol is subtle, but devastating. As the user is waiting to receive the encrypted pseudonym(s), and to decrypt it (them), an attacker who possesses the session key could exploit this circumstance by sending the encrypted user identifier $IUID_U$ to the user (which the user would confuse with a pseudonym), and the user then would decrypt it and send it back to the attacker. Actually, as not only one but all user's encrypted pseudonym are sent from the database to the user, the attcker could add the encrypted $IUID_U$ to the encrypted pseudonyms list. Then, the user would decrypt the whole list and send it to the client for subsequent selection. Thus, the attacker would obtain the user's internal identifier and **all** his pseudonyms. In other words, the described attack allows an insight into the whole patient's medical history. The message sequence corresponding to the attack is given below in figure 7.6, which is an attack trace generated by the AVISPA tool.

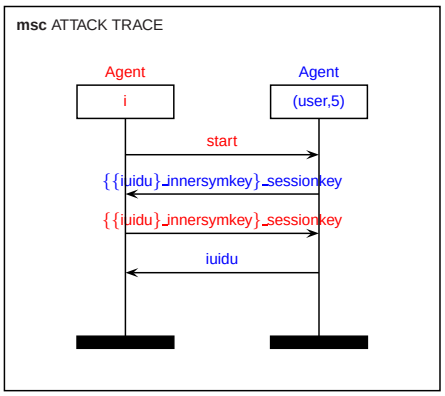

Figure 7.6: Attack trace for the get pseudonyms workflow.

In order to prevent such attacks, the PIPE system provides a mechanism in which the message digest is encrypted with the sender's private key. This as well protects the confidentiality of user's medical data, since otherwise the user would notice that message 4 has either been modified or created by the attacker. In consequence, the user would abort the communication. In

addition to confidentiality, all messages are authentic, unaltered and non-deniable. Nevertheless, despite these advantages, the extended protocol version is computationally more expensive. It is given below.

1. $U \rightarrow C$: $\{IUID_U\}_{ISK_U}$
2. $C \rightarrow D$: $\{IUID_U\}_{ISK_U}$
3. $D \rightarrow C$: $\{\{PSN\}_{ISK}\}.\{hash(\{PSN\}_{ISK})\}_{inv(K_D)}$
4. $C \rightarrow U$: $\{\{PSN\}_{ISK}\}.\{hash(\{PSN\}_{ISK})\}_{inv(K_D)}$
5. $U \rightarrow C$: PSN

In chapter 5.2, we have shown how to secure the authentication phase and how to provide the required security properties. Therefore, in our opinion, applying additional protection mechanisms as shown above is not necessary. We argue that in case the authentication phase has been performed correctly, no other security mechanisms are necessary for the subsequent workflows, since either they are useless (i.e. the user has lost his private key, the used cryptography is not up to date etc.) or they do not improve the security of the system (i.e. the authentication phase provides all the required security properties), but demand more computational power without providing a benefit.

7.3 Authorize Instance

Instance authorization refers to granting a trusted user access to specific health data records. Both users, the data owner and the authorised user, have to authenticate against the system, and they provide the application with their identifiers. The record to be shared is selected by the data owner, and then new pseudonyms are created. The application references the newly generated pseudonyms with the selected data records and makes them available to the authorized user. Both users, the authorized user and the data owner, encrypt the pseudonyms and the identifiers with their inner symmetric keys and send these to the database for storage. This is required due to the PIPE's static data model, in which the pseudonyms and the identifiers are stored encrypted. Preconditions for this workflow are a successful completion of the authentication phase on the one hand, and the assumption that no security vulnerabilities are introduced by workflows referenced by this one (i.e. 'Get Pseudonyms'). Furthermore, we again assume that the client application has obtained the session key, and we assume that the session key has been broken by the attacker. A detailed description of the workflow is given in section 5.2. For the HLPSL implementation of the workflow, see appendix G. The communication protocol for the workflow is given below. The data owner is denoted by O, the authorized user by A, and the 'Get Pseudonyms' workflow is denoted by $GetPSNs$. Terms $auPSN$ and $owPSN$ denote the authorised user's and data owner's pseudonyms. Analogously, terms $IUIDau$ and $IUIDow$

denote the authorised user's and the data owner's identifiers.

1. $O \leftrightarrow C \leftrightarrow D$: $GetPSNs$
2. $C \rightarrow D$: $owPSN$
3. $D \rightarrow C$: RID
4. $C \rightarrow O$: $auPSN, IUIDau$
5. $C \rightarrow D$: $auPSN, RID$
6. $O \rightarrow C$: $\{IUIDow, IUIDau, auPSN\}_{ISKow}$
7. $C \rightarrow A$: $owPSN, IUIDow$
8. $A \rightarrow C$: $\{IUIDow, IUIDau, owPSN\}_{ISKau}$

Assuming that the attacker posseses the session key $K_{session}$, then the confidentiality, integrity, authenticity and non-repudiation security requirements for this workflow are not fulfilled. The attacker can read and modify the internal identifiers of the data owner and the authorized user, as well as pseudonyms and data records, since all these are readable and modifiable for the intruder. In case the digest mechanism is used, the attacker cannot modify the transmitted messages, but he can still read them. However, despite these weaknesses, we are convinced that, based on the security provided by the authentication workflow, no further measures are necessary for this workflow.

We argue that in case the authentication phase has been performed correctly and all security requirements are fulfilled, no other security mechanisms are necessary for the subsequent workflows, since either they are useless (i.e. the user has lost his private key) or they do not improve the security of the system (i.e. the authentication phase provides all the required security properties) but demand more computational power without providing a benefit.

7.4 Data Insertion

In this workflow, the data owner adds new data records. After the data has been entered and a new medical record created, a new pseudonym is created by the client application and sent to the data owner. Then, both the pseudonym and the user identifier are encrypted and sent to the database, where they are stored. For a detailed description of the workflow, see section 5.2. In appendix E, the model of the workflow, implemented in the HLPSL modeling language, is illustrated. Again, we assume that the client application has obtained the session key, and we assume that the session key has been broken by the attacker.

Although the attacker posseses the session key $K_{session}$, confidentiality is still given, since it is the combination of the user identifier and the pseudonym which reveals confidential information concerning the patient's medical history. In this workflow, only the pseudonym, but not the user identifier, can be gathered by the attacker. However, the integrity, authenticity and non-repudiation security requirements are not fulfilled in case the session key is known to the adversary. In case the digest mechanism is used, the attacker cannot modify the transmitted messages, and he also cannot read them. Thus, by applying the message digest mechanism, we would establish all the required security properties. In chapter 5.2, we have shown how to secure the authentication phase and how to provide the required security properties. Thus, we argue that in case the authentication phase has been performed correctly and all security requirements are fulfilled, no other security mechanisms are necessary for the subsequent workflows.

7.5 Data Retrieval

This workflow involves selecting the respective pseudonym and retrieving the referenced health data record. The workflow is very simple, since the favored pseudonym is selected by the user and sent to the database. Then, the database queries the corresponding health data record and transmits it to the user. Again, we assume that the client application has obtained the session key, and we assume that the session key has been broken by the attacker. However, an attacker who can read the traffic on the wire cannot make the connection between the user identifier and the pseudonym (or the medical data record), since no user identifier is transmitted in this workflow, either in plaintext nor encrypted. Nevertheless, the integrity, authenticity and non-repudiation security requirements are not fulfilled in case the session key is known to the adversary. In case the digest mechanism is used, the attacker cannot modify the transmitted messages, and he also cannot read them. Thus, by applying the message digest mechanism, we would establish all the required security properties. However, in chapter 5.2 we have shown how to secure the authentication phase and how to provide the required security properties. Thus, in our opinion, additional security measures are not necessary. Our arguments for the security of this workflow are the same as already described in previous sections.

7.6 Data Pseudonymization

This workflow describes the procedure of pseudonymizing (health) data which already exist in the database. The workflow assumes that the data records are already depersonalized and that data records are separated into one identification record and multiple health data records. For a detailed description of the workflow, see section 5.2. The workflow model implemented in the HLPSL language is illustrated in appendix F. Again, we assume that the client application has obtained the session key, and we assume that the session key has been broken by the attacker.

Confidentiality of the user identifier $IUID_U$ in the workflow has been verified and proven to be secure, since it is the combination of the user identifier and the pseudonym which reveals confidential information concerning the patient's medical history. In this workflow, only the pseudonym, but not the user identifier, can be gathered by the attacker. However, the integrity,

authenticity and non-repudiation security requirements are not fulfilled in case the session key is known to the adversary. For instance, the attacker could modify the retrieved data, which the user would not notice. In case the digest mechanism is used, the attacker cannot modify the transmitted messages, and he also cannot read them. Thus, by applying the message digest mechanism, we would establish all the required security properties. However, in chapter 5.2 we have shown how to secure the authentication phase and how to provide the required security properties. Thus, in our opinion, additional security measures are not necessary. Our arguments for the security of this workflow are the same as already described in previos sections.

7.7 Summary

In this chapter, we have shown how the authentication phase can be corrected in order to provide the authenticity of involved participants (i.e., user, server, database), confidentiality and integrity of the established session key, as well as non-repudiation for both communication participants such that evidence for having established a session is provided for both, the user and the server. Furthermore, we have argued that, in case the authentication phase fulfills all the enumerated requirements, there is no need for additional cryptographic operations (i.e., encryption, signatures) within workflows which belong to the second phase, as no additional security is provided. In table 7.2, the summary of our validation results is given when workflows in their corrected versions are considered. As we have argued in previous sections, a correct authentication phase provides enough security such that subsequent workflows do not have to fulfill all security requirements.

Table 7.2: Result summary

Workflow	**AEN**	**INT**	**NRO**	**NRR**	**CON**
Authentication	x	x	x	x	x
Get Pseudonyms					x
Authorize Instance					
Data Insertion					x
Data Retrieval					x
Data Pseudonymization					x

In this work we have strongly focused on attackers corresponding to the Dolev-Yao intruder model, which have complete control over the network and can modify, replace and delete messages. We did not consider internal attackers (i.e., corrupt system administrators and medical practitioners) which completely or partially control the server, the database, and the client application. In case an attacker controls the client application (e.g., a backdoor is installed), then he or she would be able to modify the content displayed on the screen and thus show falsified information to the medical practitioner who is using the application. In addition, the attacker would be able to gain insight into the patient's medical history, since during the 'Authorize Instance' workflow pseudonyms and record identifiers are received and decrypted by the client

application. Thus, in such a case the attacker would be able to recover the patient's medical history. In case the attacker has compromised the server or the database, or in case he is able to read the communication between the server and the database (i.e., when both processes run on the same machine and inter-process communication is used), the same reasoning applies since pseudonyms and record identifiers can be associated, and in consequence confidentiality can no longer be assured.

CHAPTER 8

Conclusion

A large amount of modeling and specification approaches for describing secure infromation systems are available, and the question arises which method to use for which problem. In order to answer this question, it is fundamental to know which security mechanisms and which security requirements can be modeled by a certain technique. However, as a multitude of available modeling approaches exists, it can be rather tricky to find the most suitable method to solve a particular problem. In addition, there is no common comparison framework to oppose the different methods to each other with regard to security and to indicate the promising approach. Thus, despite a multitude of available methods and tools, the developer is left alone with the problem of selecting a suitable method.

In this thesis we have provided a taxonomy for model-driven security, we have evaluated several state-of-the-art approaches, and we have classified them according to the proposed taxonomy. Thus, we have answered the question *which approaches are applicable for solving which development problems*, and we have showed which specific characteristics these methods feature. The benefit of the work is a comparison framework for classifying model-driven security approaches and formal specification methods. In addition, we have applied two evaluated methods and have validated the identified security properties which were required for PIPE, our case study system. Thereby we have answered the question *which methods can be used to analyse the PIPE system with regards to its security properties*. We have identified AVISPA as the most appropriate tool for the analysis of protocols and workflows, and we have applied it to validate PIPE. Thereby we have answered the research questions *which security requirements are fulfilled by the PIPE system, and which are not*. We have found vulnerabilities to replay and repudiation attacks, and we have illustrated how the workflows have to be fixed in order to mitigate the identified threats. That is, we have shown how the authentication phase can be corrected in order to provide authenticity of involved participants (i.e. user, server, database), confidentiality and integrity of the established session key, as well as non-repudiation for both communicating parties, such that evidence for having established a session is provided for both, the user and the server. We have experienced that no proper intruder model for the verification of non-repudiation

security properties is suported by the AVISPA tool. Therefore, we have performed some parts of the analysis manually. Furthermore, we have discovered that all the workflows which belong to the second phase are dependent on an authenticated client application which possesses the session key, although the authentication phase does not ensure the authenticity of the client application. As consequence, the problem of incorporating the client application into the session which has been established by the server and the user has not been tackled and remains unsolved.

The conducted classification revealed that UMLsec is the most generally applicable approach focused on security, since all the other methods are either limited to modeling single security mechanisms (e.g. role based access control), or they are generic enough to model security as well, but offer no security-specific language elements. Alloy is an example of such a language, which is indeed very expressive but does not provide established rules of prudent security engineering to make them available for users who may not be experts in security. In such a case, the user has to model all the security aspects of the problem domain, which often requires a deep understanding of security (e.g. cryptographic protocols). Likewise, the Symbolic Model Verifier (SMV) model-checking system is applicable for analysing dynamic behaviour of parallel executing processes, and can be used as well for the analysis of cryptographic security protocols. Though, the level of abstraction which is offered by the SMV modeling language is far lower than the level adequate for describing security protocols. That is, modeling security protocols in the SMV language is more complex than in AVISPA, since there are much more details to care about. Thus, even if generally applicable methods (e.g. UMLsec, Alloy) can be applied to a broader range of security problems than special purpose methods can, this does not imply that they are more adequate. First, it depends on the particular problem which method fits best. And second, we have made the experience that picking the adequate special purpose method and applying it to the particular problem is more efficient and leads to better results, since (i) the problem can be represented on the proper abstraction level, (ii) the user can build on the knowledge of experts, and (iii) the available tools are more efficient and capable.

8.1 Limitations

We have not been able to analyse the system's static data model properly. One reason was that the model which we have provided was too complex for the tool that we have used. Even after simplifying the model, the tool still crashed. However, we are convinced that we have simplified the model sufficiently, since any further reduction would lead to trivial models which would not represent the system which we intended to analyse. Furthermore, we have not been able to analyse non-repudiation properties automatically, since the AVISPA tool supports the Dolev-Yao intruder model only. As well, we did not analyse the resistivity of the system against denial-of-service (DoS) attacks.

8.2 Future Work

In order to simplify the analysis of static data models, it would be possible to formulate axioms and inference rules for particular models to obtain logic programs which would allow us to make

implicit knowledge explicit (that is, to deduce information which is implicitly represented in the model). That is, by formulating rules for obtaining additional information from combinations of information pieces included in the system's static data model, one would be able to analyse the data model during the design phase, which would help to protect the system from data mining attacks during operation.

Availability, which we have identified as a desireable property in section 5.3 but which we did not analyse, could not be verified since it cannot be expressed in the HLPSL language, in which we have implemented and analysed the workflows. In [74], the authors proposed a formal framework for analysis of denial of service attacks. They showed how principles to make protocols more resistant against DoS attacks can be formalized, and they indicated a way how present protocol analysis tools could be modified to support the proposed analysis framework. It is an interesting question how the PIPE workflows behave regarding sensitivity to denial-of-service attacks, and we intend to cover the issue in our future work.

A further insight was that all the aspect-oriented approaches modeled only a single security aspect. It would have been interesting to see how complex the weaving rules might become and how difficult it might be to verify a system consisting of several security aspects. That is, even if the presented techniques worked well for modeling a single security mechanism, it was not shown by anyone how adequate the AOSD principle is for developing secure real-world applications. Therefore, modeling several security aspects and combining them with the primary model is one of the next steps that the modeling community has to pace. Successfully modeling and thus generating a secure system including several security mechanisms, like a security protocol (e.g. Needham-Schroeder) and access control (e.g. RBAC) for example, would provide the necessary confidence that the AOSD paradigm is suitable for development of complex and secure real-world applications. Furthermore, our evaluation revealed that approaches which analyse implementations of modeled systems are still missing. Due to the fact that implementations are not generated automatically from formal specifications, verification of running code is reasonable. However, beside Juerjens who suggested to apply model-based testing in order to test the implementations [3], attention was not paid to automated test case generation.

Bibliography

[1] Catherine Meadows. Formal methods for cryptographic protocol analysis: Emerging issues and trends, 2003.

[2] Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. A survey of industrial applications of formal methods. Technical report, Engineering College of Aarhus, 2008.

[3] Jan Juerjens. *Secure Systems Development with UML*. Springer, 2005.

[4] Premkumar T. Devanabu and Stuart Stubblebine. Software engineering for security: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 227–239. ACM, 2000.

[5] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley, 2006.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison wesley, 1995.

[7] Eduardo Fernandez-Medina, Jan Juerjens, Juan Trujillo, and Sushil Jajodia. Model-driven development for secure information systems. *Information and Software Technology*, 2008.

[8] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), 2006.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. pages 220–242, 1997.

[10] Josh Dehlinger and Nalin Subramanian. Architecting secure software systems using an aspect-oriented approach: A survey of current research. Technical report, Iowa State University, 2006.

[11] Jan Juerjens. Umlsec: Extending uml for secure systems development. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, 2002.

[12] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security for process-oriented systems. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 100–109. ACM, 2003.

[13] Luca Vigano. Automated security protocol analysis with the avispa tool. *Electronic Notes in Theoretical Computer Science*, 155:61–86, 2006.

[14] A. Khwaja and J. Urban. A synthesis of evaluation criteria for software specifications and specification techniques. *International Journal of Software Engineering and Knowledge Engineering*, 2002.

[15] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

[16] House E. R. Assumptions underlying evaluation models. *Educational Researcher*, 1978.

[17] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2002.

[18] International Standardization Organization ISO. Iso/iec 27000 information security management systems: Overview and vocabulary, 2009.

[19] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., 1996.

[20] Bernhard Riedl, Thomas Neubauer, Gernot Goluch, Oswald Boehm, Gert Reinauer, and Alexander Krumboeck. A secure architecture for the pseudonymization of medical data. In *ARES '07: Proceedings of the The Second International Conference on Availability, Reliability and Security*, 2007.

[21] OMG. Model driven architecture guide version 1.0.1, 2003.

[22] Alexander Pretschner. Model-based testing. 2005.

[23] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Technical report, 2006.

[24] Robert France, Indrakshi Ray, Geri Georg, and Sudipto Gosh. An aspect-oriented approach to early design modeling. *IEEE Proceedings*, 151:173–185, 2004.

[25] A. Kleppe and J. Warmer. Explore model-driven architecture and aspect-oriented programming. Web, April 2005. http://www.devx.com/enterprise/Article/27703/0/page/1, Retrieved on 2010-04-17.

[26] R. W. Butler. What is formal methods?, retrieved on 2010-04-15 from http://shemesh.larc.nasa.gov/fm/index.html, 2001.

[27] Michael Huth and Mark Ryan. *Logic in Computer Science*. Cambridge University Press, 2004.

[28] Edmund M. Clarke, Jeannette M. Wing, and Et Al. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28:626–643, 1996.

[29] Emine G. Aydal, Mark Utting, and Jim Woodcock. *Objects, Components, Models and Patterns 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 - July 4, 2008. Proceedings*, chapter A Comparison of State-Based Modelling Tools for Model Validation, pages 278–296. Springer Berlin Heidelberg, 2008.

[30] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21:993–999, 1978.

[31] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.

[32] Martin Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Softw. Eng.*, 22:6–15, 1996.

[33] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions*, 29:198–208, 1983.

[34] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8:19–36, 1990.

[35] Catherine Meadows. Open issues in formal methods for cryptographic protocol analysis. In *In Proceedings of DISCEX 2000*, 2000.

[36] Rodolfo Villarroel, Eduardo Fernandez-Medina, and Mario Piattini. Secure information systems development - a survey and comparison. *Computers & Security*, 2005.

[37] Jorge Fox and Jan Jurjens. Introducing security aspects with model transformations. In *ECBS '05: Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*. IEEE Computer Society, 2005.

[38] Derek Coleman, Grady Booch, David Garlan, Sridhar Iyengar, Cris Kobryn, and Victoria Stavridou. Is uml an architectural description language?, retrieved on 2010-04-14 from http://www.sigplan.org/oopsla/oopsla99/2_ap/tech/2d1a_uml.html, 1999.

[39] Bastian Best, Jan Jurjens, and Bashar Nuseibeh. Model-based security engineering of distributed information systems using umlsec. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, 2007.

[40] Jan Juerjens. Sound methods and effective tools for model-based security engineering with uml. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 322–331, New York, NY, USA, 2005. ACM.

[41] Jan Juerjens. Uml analysis tools, retrieved on 2010-04-13 from http://ls14-www.cs.tu-dortmund.de/main2/jj/umlsectool/index.html, 2008.

[42] Jan Juerjens. Developing secure embedded systems: Pitfalls and how to avoid them, 2007.

[43] A. Bruckner, J. Doser, and B. Wolff. *A Model Transformation Semantics and Analysis Methodology for SecureUML*. Springer, 2006.

[44] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. Secureuml: A uml-based modeling language for model-driven security. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 426–441, London, UK, 2002. Springer-Verlag.

[45] David Basin, Juergen Doser, and Torsten Lodderstedt. Model driven security: from uml models to access control infrastructures, 2005.

[46] David Basin, Manuel Clavel, Juergen Doser, and Marina Egea. Automated analysis of security-design models. *Information and Software Technology*, 2008.

[47] Geri Georg, Indrakshi Ray, and Robert France. Using aspects to design a secure system. *ICECCS02*, 2002.

[48] Huiqun Yu, Dongmei Liu, Xudong He, Li Yang, and Shu Gao. Secure software architectures design by aspect orientation. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 47–55. IEEE Computer Society, 2005.

[49] Xudong He, Huiqun Yu, Tianjun Shi, Junhua Ding, and Yi Deng. Formally analyzing software architectural specifications using sam, 2002.

[50] Bowen Alpern, Bowen Alpera, Fred B. Schneider, and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[51] Gefei Zhang, Hubert Baumeister, Nora Koch, and Alexander Knapp. Aspect-oriented modeling of access control in web applications. 2005.

[52] OMG. Unified modeling language: Superstructure, version 2.0, specification, 2005.

[53] Alexander Knapp and Gefei Zhang. Model transformations for integrating and validating web application models. In *Modellierung 2006*, 2006.

[54] Gerard J. Holzmann. *The SPIN model checker. Primer and Reference Manual.* Addison-Wesley, 2003.

[55] Indrakshi Ray, Robert France, Na Li, and Geri Georg. An aspect-based approach to modeling access control concerns. *Information and Software Technology*, 46:575–587, 2004.

[56] Eunjee Song, Raghu Reddy, Robert France, Indrakshi Ray, Geri Georg, and Roger Alexander. Verifiable composition of access control and application features. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, 2005.

[57] Zhi Jian Zhu and Mohammad Zulkernine. A model-based aspect-oriented framework for building intrusion-aware software systems. *Information and Software Technology*, 2008.

[58] WASC. Threat classification. Technical report, Web Application Security Consortium, 2008.

[59] G. Beydoun, G. Low, H. Mouratidis, and B. Henderson-Sellers. A security-aware metamodel for multi-agent systems (mas). *Information and Software Technology*, 2008.

[60] Ghassan Beydoun, , Cesar Gonzalez-Perez, Brian Henderson-Sellers, and G. Low. Developing and evaluating a generic metamodel for mas work products. In *Software Engineering for Multi-Agent Systems IV*, 2006.

[61] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.C. Heam, O. Kouchnarenko, J. Mantovani, S. Moedersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Vigano, and L. Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In *Lecture Notes in Computer Science 3576*, 2005.

[62] D. Basin et al. *HLPSL Tutorial - A Beginner's Guide to Modelling and Analysing Internet Security Protocols*.

[63] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16:872–923, 1994.

[64] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, 1998. Springer-Verlag.

[65] Daniel Jackson. *Software abstractions: Logic, Lamguage, and Analysis*. The MIT Press, 2006.

[66] Bernhard Riedl, Veronika Grascher, Stefan Fenz, and Thomas Neubauer. Pseudonymization for improving the privacy in e-health applications. In *HICSS '08: Proceedings of the Proceedings of the 41st Annual Hawaii International Conference on System Sciences*, 2008.

[67] Johannes Heurix and Thomas Neubauer. Pseudonymization of information for privacy in e-health: Design specification. Technical report, TU Vienna - Institute of Software Technology and Interactive Systems, 2009.

[68] Emina Torlak, Marten van Dijk, Blaise Gassend, Daniel Jackson, and Srinivas Devadas. Knowledge flow analysis for security protocols. Technical report, Computer Science and Artificial Intelligence Laboratory, Massachussets Institute of Technology, 2005.

[69] D. Basin et al. *AVISPA v1.1 User Manual*, 2006.

[70] The avispa library: Needham schroeder public-key protocol, 2005.

[71] Judson Santiago and Laurent Vigneron. Study for automatically analysing non-repudiation. In *Colloque sur les Risques et la Securite d'Internet et des Systemes - CRiSIS 2005*, 2005.

[72] Jianying Zhou and Dieter Gollmann. A fair non-repudiation protocol. 1996.

[73] Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *Modelling and Analysis of Security Protocols*. Addison Wesley, 2000.

[74] Catherine Meadows. A formal framework and evaluation method for network denial of service. In *In Proceedings of the 1999 IEEE Computer Security Foundations Workshop*, 1999.

List of Figures

List of Tables

Appendix A

```
/*****************************************************************************

Needham-Schroeder public-key protocol model
and intruder model for the SMV formal verification tool
taken from http://www.ics.uci.edu/~isse/proj268/index.html

SMV home page: http://www-cad.eecs.berkeley.edu/~kenmcmil/

*****************************************************************************/

/* Number of protocol instances */
#define n 1

/* Maximum number of messages the intruder remembers */
#define max_messages 4

/* Lowe's protocol fix: 0 - off, 1 - on */
#define FIX 0

#define offs (n+1)
#define max_node_name (n*2+1)
#define intruder_name (n+1)
#define intruder_nonce 0
#define max_nonce (n*4+3)

typedef node_name 1..max_node_name;
typedef node_local_name 1..n;
typedef node_local_dest_name 1..n+1;
typedef mes_state {idle, mes1, mes2, mes3_idle, mes3, finished};
typedef nonces 0..max_nonce;

typedef message struct
{
/* Type of the message: mes1, mes2 or mes3 */
```

```
type: mes_state;

/* Nonce 1 */
nonce1: nonces;

/* Nonce 2 (for mes2) */
nonce2: nonces;

/* Key (node name of the destination) */
key: node_name;

/* Initiator of the message (for mes1 and for fixed mes2) */
initiator: node_name;

/* Index # of the initiator of the message (for mes1) */
source: node_name;

/* Destination node name */
dest: node_name;
}

/* -------------- Protocol description: -------------------

mes1: A -> B: {Na, A}Pk(B)
mes2: B -> A: {Na, Nb}Pk(A)
mes2 fixed: B -> A: {Na, Nb, B}Pk(A)
mes3: A -> B: {Nb}PK(B)

-----------------------------------------------------------*/


/*---------------- Intruder Model -------------------------*/
module intruder(inp_prev, outp_prev, outp_next, inp_next)
{
input inp_prev, inp_next: message;
output outp_prev, outp_next: message;

mess_null: message;

/* connections left-to-right and right-to-left */
l_r_connect, r_l_connect: boolean;

my_name: intruder_name..intruder_name;
```

```
my_nonce: intruder_nonce..intruder_nonce;

/* non-deterministically synthesized message */
my_mes: message;

/* memory for the messages the intruder has snooped */
known_messages: array 0..max_messages-1 of message;

/* which nonces the intruder has decoded and remembered so far */
known_nonces: array nonces of boolean;

/* pointer to the first empty entry in array of snoopped messages */
curr_mess: 0..max_messages-1;

/* the message that is being snooped */
snooping_mes: message;

init(known_messages) := [mess_null : i = 0..max_messages-1];
init(known_nonces) := [(i = my_nonce) : i = 0..max_nonce];
init(curr_mess) := 1;

mess_null.type := idle;
mess_null.nonce1 := 0;
mess_null.nonce2 := 0;
mess_null.key := 1;
mess_null.initiator := 1;
mess_null.dest := 1;
mess_null.source := 1;

/* ---- Connect inputs with outputs or discard the message ---- */
if (l_r_connect)
{
    outp_next := inp_prev;
}
else
{
    outp_next := my_mes;
}

if (r_l_connect)
{
    outp_prev := inp_next;
}
```

```
else
{
    outp_prev := my_mes;
}

/* ---- Snooping the messages/nonces from incoming packets ----*/
default
    snooping_mes := mess_null;
in
{
    if (inp_prev.type ~= idle)
    {
      snooping_mes := inp_prev;
    }
    else if (inp_next.type ~= idle)
    {
      snooping_mes := inp_next;
    }
}

default
    next(known_messages) := known_messages;
in default
    next(known_nonces) := known_nonces;
in default
    next(curr_mess) := curr_mess;
in
{
    if (snooping_mes.type ~= idle)
    {
      next(known_messages[curr_mess]) := snooping_mes.type;
      next(curr_mess) := (curr_mess + 1) mod max_messages;
      if (snooping_mes.key = my_name)
      {
        for (i = 1; i <= max_nonce; i=i+1)
        {
          if ( ( (snooping_mes.type = mes2) &
                 (snooping_mes.nonce2 = i) )
               | (snooping_mes.nonce1 = i) )
            next(known_nonces[i]) := 1;
        }
      }
    }
```

```
}

/* ---- Generating fake messages ----*/

/* non-determenistic value */
ch: boolean;

default
    /* take any of snooped messages */
    my_mes := known_messages[0..max_messages-1];
in
if (ch)
{
    /* change the desination and send snooped message or */
    my_mes.dest := 1..max_node_name;
}
else
{
    /* generate new message from known nonces and random keys */
    my_mes.type := {mes1, mes2, mes3};
    my_mes.nonce1 := {i : i = 0..max_nonce, known_nonces[i]};
    my_mes.nonce2 := {i : i = 0..max_nonce, known_nonces[i]};
    my_mes.key := {1..max_node_name};
    my_mes.dest := {1..max_node_name};
    my_mes.source := my_name;
    my_mes.initiator := {1..max_node_name};
}
}

/*---- Node Model (initiator and receiver) ----*/
module node(index, my_name, offs1,
            dest_name, offs2, my_nonce_init,
            my_nonce_rec, inp_prev, outp_prev,
            outp_next, inp_next)
{
  input my_name: node_local_name;
  dest_name: node_local_dest_name;

  /* constant parameters; these are used
   * to reduce the size of node_local_name
   * type to reduce the size of the BDDs
  **/
  input offs1, offs2: 0..50;
```

```
/* initiator and receiver nonces */
input my_nonce_init, my_nonce_rec: nonces;

input inp_prev, inp_next: message;

output outp_prev, outp_next: message;

state_initiator: mes_state;
state_receiver: mes_state;
initiator_opposite_node: node_local_dest_name;
receiver_opposite_node: node_name;
receiver_opposite_node_index: node_name;
initiator_opposite_nonce: nonces;
receiver_opposite_nonce: nonces;
init_message, rec_message, rec_m_message, mess_null, rec_inp:
message;

/* if the message is consumed by initiator or passed further */
get_inp: boolean;

/* if the message is consumed by receiver or passed further */
get_rec_inp: boolean;

mess_null.type := idle;
mess_null.nonce1 := 0;
mess_null.nonce2 := 0;
mess_null.key := 1;
mess_null.initiator := 1;
mess_null.dest := 1;
mess_null.source := 1;

if (index = intruder_name)
{
  /* instantiate intruder or usual node */
  my_node: node_name;
  intr: intruder(inp_prev, outp_prev, outp_next, inp_next);
  my_node := intr.my_name;
  state_initiator := idle;
  state_receiver := idle;
  initiator_opposite_node := my_node;
  receiver_opposite_node := my_node;
  receiver_opposite_node_index := index;
```

```
    initiator_opposite_nonce := 0;
    receiver_opposite_nonce := 0;
  }
  else
  {
    my_node: node_local_name;
    init(state_initiator) := {idle, mes1};
    init(state_receiver) := idle;
    init(my_node) := my_name;
    next(my_node) := my_node;
    init(initiator_opposite_node) := dest_name;
    next(initiator_opposite_node) := initiator_opposite_node;
    init(receiver_opposite_node) := 1;
    init(receiver_opposite_node_index) := 1;
    init(initiator_opposite_nonce) := 1;
    init(receiver_opposite_nonce) := 1;

/* ------------ Initiator ------------ */
  default
    get_inp := 0;
  in default
    init_message := mess_null;
  in default
    next(initiator_opposite_nonce) := initiator_opposite_nonce;
  in default
    next(state_initiator) := state_initiator;
  in switch(state_initiator)
  {
    idle:
    {
      /* non-determenistic next state selection */
      next(state_initiator) := {idle, mes1};
    }
    mes1:
    {
      init_message.type := mes1;
      init_message.nonce1 := my_nonce_init;
      init_message.key := initiator_opposite_node + offs2;
      init_message.initiator := my_node + offs1;
      init_message.source := index;
      init_message.dest := initiator_opposite_node + offs2;
      if (rec_m_message.type = idle)
        next(state_initiator) := mes2;
```

```
      else
        next(state_initiator) := mes1;
    }
    mes2:
    {
      if ((inp_next.type = mes2) &
          (inp_next.nonce1 = my_nonce_init) &
          (inp_next.key = my_node + offs1) &
          ( (inp_next.initiator = initiator_opposite_node + offs2)
            | ~FIX ) &
          /* protocol fix */
          (inp_next.dest = my_node + offs1)) {
        next(state_initiator) := {mes3_idle, mes3};
        next(initiator_opposite_nonce) := inp_next.nonce2;
        get_inp := 1;
      }
      else
      {
        next(state_initiator) := mes2;
      }
    }
    mes3_idle:
      next(state_initiator) := {mes3_idle, mes3};
    mes3:
    {
      init_message.type := mes3;
      init_message.nonce1 := initiator_opposite_nonce;
      init_message.key := initiator_opposite_node + offs2;
      init_message.source := index;
      init_message.dest := initiator_opposite_node + offs2;
      if (rec_m_message.type = idle)
        next(state_initiator) := finished;
      else
        next(state_initiator) := mes3;
    }
    finished:
      next(state_initiator) := finished;
  }

  if ( (rec_m_message.type = idle)
     & ((state_initiator = mes1) | (state_initiator = mes3)) )
    outp_next := init_message;
  else
```

```
    outp_next := rec_m_message;

  if (get_inp)
    rec_inp := mess_null;
  else
    rec_inp := inp_next;

/* ------------ Receiver ------------ */

  default
    get_rec_inp := 0;
  in default
    rec_message := mess_null;
  in default
    next(receiver_opposite_node) := receiver_opposite_node;
  in default
    next(receiver_opposite_node_index) := receiver_opposite_node_index;
  in default
    next(receiver_opposite_nonce) := receiver_opposite_nonce;
  in default
    next(state_receiver) := state_receiver;
  in switch(state_receiver)
  {
    idle:
    {
      if ((rec_inp.type = mes1) &
          (rec_inp.key = my_node + offs1) &
          (rec_inp.dest = my_node + offs1)) {
        next(state_receiver) := {mes1, mes2};
        next(receiver_opposite_node) := rec_inp.initiator;
        next(receiver_opposite_node_index) := rec_inp.source;
        next(receiver_opposite_nonce) := rec_inp.nonce1;
        get_rec_inp := 1;
      }
      else
      {
        next(state_receiver) := idle;
      }
    }
    mes1:
    {
      next(state_receiver) := {mes1, mes2};
    }
```

```
    mes2:
    {
      rec_message.type := mes2;
      rec_message.nonce1 := receiver_opposite_nonce;
      rec_message.nonce2 := my_nonce_rec;
      rec_message.initiator := my_node + offs1;
      rec_message.key := receiver_opposite_node;
      rec_message.source := index;
      rec_message.dest := receiver_opposite_node;
      if (inp_prev.type = idle)
        next(state_receiver) := mes3;
      else
        next(state_receiver) := mes2;
    }
    mes3:
    {
      if ((rec_inp.type = mes3) &
          (rec_inp.nonce1 = my_nonce_rec) &
          (rec_inp.key = my_node + offs1) &
          (rec_inp.dest = my_node + offs1)) {
        next(state_receiver) := finished;
        get_rec_inp := 1;
      }
      else
      {
        next(state_receiver) := mes3;
      }
    }
    finished:
      next(state_receiver) := finished;
  }

  if ((inp_prev.type = idle) & (state_receiver = mes2))
    rec_m_message := rec_message;
  else
    rec_m_message := inp_prev;

  if (get_rec_inp)
    outp_prev := mess_null;
  else
    outp_prev := rec_inp;
  }
}
```

```
module main()
{
  mess_null: message;
  node_ref: array 1..2*n+1;
  temp_left: array 0..(n*2+3) of message;
  temp_right: array 0..(n*2+1) of message;
  name_set: node_local_name;
  name_dest_set: node_local_dest_name;

  temp_left[0] := mess_null;
  temp_left[n*2+3] := temp_right[n*2];

  for (i = 1; i <= n+1; i=i+1)
  { /* first set of nodes plus intruder */
    node_ref[i]: node(i, name_set, 0,
                      name_dest_set, offs, i*2,
                      i*2+1, temp_left[(i-1)*2], temp_left[(i-1)*2+1],
                      temp_left[i*2], temp_left[i*2+1]);
  }

  temp_right[0] := mess_null;
  temp_right[n*2+1] := temp_left[n*2+2];

  for (i = 1; i <= n; i=i+1)
  { /* second set of nodes */
    node_ref[offs+i]: node(offs+i, name_set, offs,
                           name_dest_set, 0, offs*2+i*2,
                           offs*2+i*2+1, temp_right[(i-1)*2],
                           temp_right[(i-1)*2+1],
                           temp_right[i*2],
                           temp_right[i*2+1]);
  }

  mess_null.type := idle;
  mess_null.nonce1 := 0;
  mess_null.nonce2 := 0;
  mess_null.key := 1;
  mess_null.initiator := 1;
  mess_null.dest := 1;
  mess_null.source := 1;
```

```
  /* Assertions to check: */

  ass: array 0..n of boolean;
  ass[0] := 0;
  for (i = 1; i <= n; i=i+1)
  {
    ass[i] := ass[i-1] | ((node_ref[i].state_receiver = finished) &
     (node_ref[i].receiver_opposite_node ~= intruder_name) &
     /* don't check connections with intruder */
     ((node_ref[node_ref[i].receiver_opposite_node_index].
     state_initiator ~= finished) |
     ((node_ref[node_ref[i].receiver_opposite_node_index].my_node +
       node_ref[node_ref[i].receiver_opposite_node_index].offs1 )
       /* index and name do match */
       ~= node_ref[i].receiver_opposite_node) |
     ((node_ref[node_ref[i].receiver_opposite_node_index].
       initiator_opposite_node +
       node_ref[node_ref[i].receiver_opposite_node_index].offs2)
      ~= (node_ref[i].my_node + node_ref[i].offs1))));
  }

  /* should return TRUE */
  check_if_everything_fine: assert G (~ass[n]);

  ass1: array 0..n of boolean;
  ass1[0] := 0;
  for (i = 1; i <= n; i=i+1)
  {
    ass1[i] := ass1[i-1] | ( (node_ref[i].state_receiver = finished) &
      ( node_ref[node_ref[i].receiver_opposite_node_index].
        state_initiator = finished )&
      ( ( node_ref[node_ref[i].receiver_opposite_node_index].
          initiator_opposite_node +
          node_ref[node_ref[i].receiver_opposite_node_index].offs2 )
        = (node_ref[i].my_node + node_ref[i].offs1)) );
  }

  /* should return FALSE and the trace */
  show_successful_trace: assert G (~ass1[n]);
}
```

Appendix B

```
%%
%%
%% Description of the Authentication workflow for the PIPE model
%% (single machine variant)
%%
%%

role user(U:agent,S:agent,Ku:public_key,Ks:public_key,
          SND,RCV:channel(dy))
played_by U
def=
  local
    State:nat,Nu:text,Ns:text,K:symmetric_key

  init
    State := 1

  transition
    1. State=1 /\ RCV(start) =|> State':=3 /\
                                 Nu':=new() /\
                                 SND({Nu'.U}_Ks) /\
                                 witness(U,S,auth_nu,Nu') /\
                                 secret(Nu',sec_nu,{U,S})

    %% user receives nonces Nu, Ns from the server
    %% and requests authentication on Nu (= check of value Nu),
    %% and at same time sees Ns (is witness) for which
    %% the server will demand a check (request for auth.)

    3. State=3 /\ RCV({Nu.Ns'.S}_Ku) =|> State':=5 /\
                                         SND({Ns'}_Ks) /\
                                         request(U,S,auth_ns,Ns')
```

```
    5. State=5 /\ RCV({K'.Nu}_Ku) =|> State':=7 /\
                                       request(U,S,auth_symkey,K')

end role

role server(U:agent,S:agent,Ku:public_key,Ks:public_key,
            SND,RCV:channel(dy))
played_by S
def=
  local
    State:nat,Nu:text,Ns:text,K:symmetric_key
  init
    State := 0
  transition
    %% the server receives Nu (is witness),
    %% for which the user demands authentication (request for auth.)
    0. State=0 /\ RCV({Nu'.U}_Ks) =|> State':=2 /\
                                       Ns':=new() /\
                                       SND({Nu'.Ns'.S}_Ku) /\
                                       witness(S,U,auth_ns,Ns') /\
                                       secret(Ns',sec_ns,{U,S})

    %% server requests the user to authenticate on Ns
    %% furthermore, the session key K shall not be known
    %% by anyone else
    2. State=2 /\ RCV({Ns}_Ks) =|> State':=4 /\
                                  K':=new() /\
                                  SND({K'.Nu}_Ku) /\
                                  request(S,U,auth_nu,Nu) /\
                                  secret(K',sec_symkey,{U,S}) /\
                                  witness(S,U,auth_symkey,K')
end role

role session(U:agent,S:agent,Ku:public_key,Ks:public_key)
def=
  local SND_U,RCV_U,SND_S,RCV_S:channel(dy)

  composition
    user(U,S,Ku,Ks,SND_U,RCV_U) /\
    server(U,S,Ku,Ks,SND_S,RCV_S)
end role
```

```
role environment()
def=
  const ku,ks,ki:public_key,user,server:agent,
  auth_nu,auth_ns,sec_nu,sec_ns,auth_symkey,sec_symkey:protocol_id

  intruder_knowledge = {user, server, ku, ks, ki, inv(ki)}

  composition
    session(user,server,ku,ks) /\
    session(i,server,ki,ks) /\
    session(user,i,ku,ki)
end role

goal
  authentication_on auth_nu
  authentication_on auth_ns
  secrecy_of sec_ns
  secrecy_of sec_nu
  authentication_on auth_symkey
  secrecy_of sec_symkey
end goal

environment()
```

Appendix C

```
%%
%%
%% Description of the Authentication workflow for the PIPE model
%% (dedicated machine variant)
%%
%%

role user(U:agent,S:agent,D:agent,Ku:public_key,Ks:public_key,
          SND,RCV:channel(dy))
played_by U
def=
  local
    State:nat,Nu:text,Ns:text,K:symmetric_key
  init
    State := 1
  transition
    1. State=1 /\ RCV(start) =|> State':=3 /\
                                 Nu':=new() /\
                                 SND({Nu'.U}_Ks)

    %% user receives nonces Nu, Ns from the server
    %% and requests authentication on Nu (= check of value Nu),
    %% and at same time sees Ns (is witness) for which the server
    %% will demand a check (request for auth.)
    2. State=3 /\ RCV({Nu.Ns'.S}_Ku) =|> State':=5 /\
                                         SND({Ns'}_Ks) /\
                                         request(U,S,nu,Nu) /\
                                         witness(U,S,ns,Ns')

    3. State=5 /\ RCV({K'}_Ku) =|> State':=7
end role

role server(U:agent,S:agent,D:agent,Ks:public_key,Kd:public_key,
            SND,RCV:channel(dy))
```

```
played_by S
def=
  local
    State:nat,Nu:text,Ns:text,Ku:public_key,K:symmetric_key
  init
    State := 0
  transition
    %% the server receives Nu (is witness),
    %% for which the user demands authentication
    %% (request for authentication)
    1. State=0 /\ RCV({Nu'.U}_Ks) =|> State':=2 /\
                                      SND(U) /\
                                      witness(S,U,nu,Nu')

    2. State=2 /\ RCV({U.Ku'}_inv(Kd)) =|> State':=4 /\
                                           Ns':=new() /\
                                           SND({Nu.Ns'.S}_Ku')

    %% server requests the user to authenticate on Ns
    %% furthermore, the session key K shall not be known by
    %% anyone else
    3. State=4 /\ RCV({Ns}_Ks) =|> State':=6 /\
                                   K':=new() /\
                                       SND({K'}_Ku) /\
                                       request(S,U,ns,Ns) /\
                                       secret(K',symkey,{U,S})
end role

role dbase(U:agent,S:agent,D:agent,Ku:public_key,
           Ks:public_key,Kd:public_key,
           SND,RCV:channel(dy),KeyMap:(agent.public_key) set)
played_by D
def=
  local
    State:nat

  init
    State := 0
  transition
    1. State=0 /\
       RCV(U') /\
       in(U'.Ku', KeyMap) =|> State':=1 /\ SND({U'.Ku'}_inv(Kd))
end role
```

```
role session(U:agent,S:agent,D:agent,
             Ku:public_key,Ks:public_key,Kd:public_key,
             KeyMap:(agent.public_key) set)
def=
  local
    SND_U,RCV_U,SND_S,RCV_S,SND_D,RCV_D:channel(dy)
  composition
    user(U,S,D,Ku,Ks,SND_U,RCV_U) /\
    server(U,S,D,Ks,Kd,SND_S,RCV_S) /\
    dbase(U,S,D,Ku,Ks,Kd,SND_D,RCV_D,KeyMap)
end role

role environment()
def=
  local KeyMap: (agent.public_key) set

  const ku,ks,ki,kd:public_key,user:agent,server:agent,dbase:agent,
        nu:protocol_id,ns:protocol_id,symkey:protocol_id

  init KeyMap := {user.ku, server.ks, i.ki}

  intruder_knowledge = {user, server, dbase, ku, ks, kd, ki, inv(ki)}

  composition
    session(user,server,dbase,ku,ks,kd,KeyMap) /\
    session(i,server,dbase,ki,ks,kd,KeyMap) /\
    session(user,i,dbase,ku,ki,kd,KeyMap) /\
    session(user,server,i,ku,ks,kd,KeyMap)
end role

goal
  authentication_on nu
  authentication_on ns
  secrecy_of symkey
end goal

environment()
```

Appendix D

```
%%
%%
%% Description of the Get Pseudonyms workflow for the PIPE model
%%
%%

role user(U:agent,C:agent,D:agent,K:symmetric_key,
          ISKu:symmetric_key,IUIDu:text)
played_by U
def=
   local
      State:nat,PSN:text
   init
      State := 0
   transition
      1. State=0   /\ RCV(start) =|>
         State':=1 /\ SND({{IUIDu}_ISKu}_K)
                   /\ secret(IUIDu,sec_1,{U})
      4. State=1   /\ RCV({{PSN'}_ISKu}_K) =|>
         State':=2 /\ SND(PSN')
end role

role client(U:agent,C:agent,D:agent,K:symmetric_key)
played_by C
def=
   local
      State:nat,IUIDu:text,ISKu:symmetric_key,PSN:text
   init
      State := 0
   transition
      1. State=0   /\ RCV({{IUIDu'}_ISKu'}_K) =|>
         State':=1 /\ SND({{IUIDu'}_ISKu'}_K)
                   /\ secret(IUIDu',sec_1,{U})
      3. State=1   /\ RCV({{PSN'}_ISKu}_K) =|>
```

```
         State':=2 /\ SND({{PSN'}_ISKu}_K)
      5. State=2   /\ RCV(PSN) =|> State':=3
end role

role dbase(U:agent,C:agent,D:agent,K:symmetric_key)
played_by D
def=
   local
      State:nat,IUIDu:text,PSN:text,ISKu:symmetric_key
   init
      State := 0
   transition
      2. State=0   /\ RCV({{IUIDu'}_ISKu'}_K) =|>
         State':=1 /\ PSN':=new()
                   /\ SND({{PSN'}_ISKu'}_K)
                   /\ secret(IUIDu',sec_1,{U})
end role

role session(ISKu:symmetric_key,IUIDu:text,
             U:agent,C:agent,D:agent,K:symmetric_key)
def=
local
SND_D,RCV_D,SND_C,RCV_C,SND_U,RCV_U:channel(dy)
composition
dbase(U,C,D,K,SND_D,RCV_D) /\
client(U,C,D,K,SND_C,RCV_C) /\
user(U,C,D,K,ISKu,IUIDu,SND_U,RCV_U)
end role

role environment()
def=
const
user:agent,client:agent,dbase:agent,
innersymkey:symmetric_key,iuidu:text,
sessionkey:symmetric_key,sec_1:protocol_id
intruder_knowledge = {sessionkey}
composition
session(innersymkey,iuidu,user,client,dbase,sessionkey)
end role

goal
   secrecy_of sec_1
end goal
```

Appendix E

```
%%
%%
%% Description of the Data Insertion workflow for the PIPE model
%%
%%

role user(U:agent,C:agent,D:agent,ISKu:symmetric_key,
          IUIDu:text,SND,RCV:channel(dy))
played_by U
def=
  local
    State:nat,PSN:text
  init
    State := 0
  transition
    1. State=0   /\ RCV(PSN') =|>
       State':=1 /\ SND({IUIDu}_ISKu.{PSN}_ISKu)
                 /\ secret(IUIDu,sec1,{U})
end role

role client(U:agent,C:agent,D:agent,SND,RCV:channel(dy))
played_by C
def=
  local
    State:nat,IUIDu:text,ISKu:symmetric_key,PSN:text,RID:text
  init
    State := 0
  transition
    1. State=0   /\ RCV(RID') =|>
       State':=1 /\ PSN':=new()
                 /\ SND(PSN')
    2. State=1   /\ RCV({IUIDu'}_ISKu.{PSN'}_ISKu) =|>
       State':=2 /\ SND(PSN.RID.{IUIDu'}_ISKu.{PSN'}_ISKu)
end role
```

```
role dbase(U:agent,C:agent,D:agent,SND,RCV:channel(dy))
played_by D
def=
  local
    State:nat,RID:text,PSN:text,ISKu:symmetric_key,IUIDu:text
  init
    State := 0
  transition
    1. State=0   /\ RCV(start) =|>
       State':=1 /\ RID':=new()
                 /\ SND(RID')
    2. State=1 /\ RCV({IUIDu'}_ISKu.{PSN'}_ISKu) =|>
       State':=2
end role

role session(ISKu:symmetric_key,IUIDu:text,U:agent,C:agent,D:agent)
def=
local
SND_D,RCV_D,SND_C,RCV_C,SND_U,RCV_U:channel(dy)
composition
dbase(U,C,D,SND_D,RCV_D) /\
client(U,C,D,SND_C,RCV_C) /\
user(U,C,D,ISKu,IUIDu,SND_U,RCV_U)
end role

role environment()
def=
const
user:agent,client:agent,dbase:agent,innersymkey:symmetric_key,
iuidu:text,sec_1:protocol_id
intruder_knowledge = {user,client,dbase}
composition
session(innersymkey,iuidu,user,client,dbase)
end role

goal
  secrecy_of sec_1
end goal
environment()
```

Appendix F

```
%%
%%
%% Description of the Data Pseudonymization workflow for the PIPE model
%%
%%

role user(U:agent,C:agent,D:agent,ISKu:symmetric_key,IUIDu:text,
          SND,RCV:channel(dy))
played_by U
def=
  local
    State:nat,PSN:text
  init
    State := 0
  transition
    1. State=0   /\ RCV(PSN') =|>
       State':=1 /\ SND({IUIDu}_ISKu.{PSN}_ISKu)
                 /\ secret(IUIDu,sec1,{U})
end role

role client(U:agent,C:agent,D:agent,SND,RCV:channel(dy))
played_by C
def=
  local
    State:nat,IUIDu:text,ISKu:symmetric_key,PSN:text,RID:text
  init
    State := 0
  transition
    1. State=0   /\ RCV(RID') =|>
       State':=1 /\ PSN':=new()
                 /\ SND(PSN')
    2. State=1   /\ RCV({IUIDu'}_ISKu.{PSN'}_ISKu) =|>
       State':=2 /\ SND({IUIDu'}_ISKu.{PSN'}_ISKu)
end role
```

```
role dbase(U:agent,C:agent,D:agent,SND,RCV:channel(dy))
played_by D
def=
  local
    State:nat,RID:text,PSN:text,ISKu:symmetric_key,
    IUIDu:text
  init
    State := 0
  transition
    1. State=0   /\ RCV(start) =|>
       State':=1 /\ RID':=new()
                 /\ SND(RID')
    2. State=1 /\ RCV({IUIDu'}_ISKu.{PSN'}_ISKu) =|>
       State':=2
end role

role session(ISKu:symmetric_key,IUIDu:text,U:agent,C:agent,D:agent)
def=
local
SND_D,RCV_D,SND_C,RCV_C,SND_U,RCV_U:channel(dy)
composition
dbase(U,C,D,SND_D,RCV_D) /\
client(U,C,D,SND_C,RCV_C) /\
user(U,C,D,ISKu,IUIDu,SND_U,RCV_U)
end role

role environment()
def=
const
user:agent,client:agent,dbase:agent,innersymkey:symmetric_key,
iuidu:text,sec_1:protocol_id
intruder_knowledge = {user,client,dbase}
composition
session(innersymkey,iuidu,user,client,dbase)
end role

goal
  secrecy_of sec_1
end goal
environment()
```

Appendix G

```
%%
%%
%% Description of the Authorize-Instance workflow for the PIPE model
%%
%%

role owner(O:agent,A:agent,C:agent,D:agent,IUIDow:text,ISKow:symmetric_key,
           SND,RCV:channel(dy))
played_by O
def=
   local
      State:nat,IUIDau:text,PSNau:text
   init
      State := 0
   transition
      1. State=0 /\ RCV(PSNau'.IUIDau') =|> State':=1
      3. State=1 /\ SND({IUIDow.IUIDau.PSNau}_ISKow) =|> State':=2
end role

role authorized(O:agent,A:agent,C:agent,D:agent,IUIDau:text,
                ISKau:symmetric_key,SND,RCV:channel(dy))
played_by A
def=
   local
      State:nat,PSNow:text,IUIDow:text
   init
      State := 0
   transition
      1. State=0 /\ RCV(PSNow'.IUIDow') =|> State':=1  /\
                                           secret(PSNow',sec_1,{O,A,C,D})
      3. State=1 /\ SND({IUIDow.IUIDau.PSNow}_ISKau) =|> State':=2
end role

role client(O:agent,A:agent,C:agent,D:agent,IUIDow:text,IUIDau:text,
```

```
            PSNow:text,ISKow:symmetric_key,ISKau:symmetric_key,
            SND,RCV:channel(dy))
played_by C
def=
   local
      State:nat,RID:text,PSNau:text
   init
      State := 0
   transition
      1. State=0 /\ RCV(start) =|> State':=1 /\ SND(PSNow)
      3. State=1 /\ RCV(RID')  =|> PSNau':=new() /\
                                   SND(PSNau'.IUIDau) /\
                                   SND(PSNau'.RID) /\
                                   State':=3
      7. State=3 /\ RCV({IUIDow.IUIDau.PSNau}_ISKow) =|> State':=4
      9. State=4 /\ SND(PSNow.IUIDow) =|> State':=5
     11. State=5 /\ RCV({IUIDow.IUIDau.PSNow}_ISKau) =|> State':=6
end role

role dbase(O:agent,A:agent,C:agent,D:agent,SND,RCV:channel(dy))
played_by D
def=
   local
      State:nat,PSNow:text,RID:text,PSNau:text
   init
      State := 0
   transition
      2. State=0 /\ RCV(PSNow') =|> RID' := new() /\
                                    SND(RID') /\
                                    State':=2
      6. State=2 /\ RCV(PSNau'.RID) =|> State':=3
end role

role session(O:agent,A:agent,C:agent,D:agent,IUIDow:text,IUIDau:text,
             PSNow:text,ISKau:symmetric_key,ISKow:symmetric_key)
def=
  local SND_O,RCV_O,SND_A,RCV_A,SND_C,RCV_C,SND_D,RCV_D:channel(dy)

  composition
    owner(O,A,C,D,IUIDow,ISKow,SND_O,RCV_O) /\
    authorized(O,A,C,D,IUIDau,ISKau,SND_A,RCV_A) /\
    client(O,A,C,D,IUIDow,IUIDau,PSNow,ISKow,ISKau,SND_C,RCV_C) /\
    dbase(O,A,C,D,SND_D,RCV_D)
```

```
end role

role environment()
def=
  const
     owner,authorized,client,dbase:agent,iuidow,iuidau,psnow:text,
     iskau,iskow:symmetric_key,sec_1:protocol_id

  intruder_knowledge = {owner,authorized,client,dbase}

  composition
    session(owner,authorized,client,dbase,iuidow,iuidau,psnow,iskau,iskow) /\
    session(i,authorized,client,dbase,iuidow,iuidau,psnow,iskau,iskow) /\
    session(owner,i,client,dbase,iuidow,iuidau,psnow,iskau,iskow) /\
    session(owner,authorized,i,dbase,iuidow,iuidau,psnow,iskau,iskow) /\
    session(owner,authorized,client,i,iuidow,iuidau,psnow,iskau,iskow)
end role

goal
   secrecy_of sec_1
end goal

environment()
```

Appendix H

```
/*
 *
 * Alloy code for modeling cryptography
 *
 */

abstract sig Value {}
sig PlainText extends Value {}
sig CipherText extends Value {}

sig SymKey extends PlainText {
   enc: PlainText one -> one CipherText,
   dec: CipherText one -> one PlainText
}
{
   // (decode(encode(p)) == p) and (encode(decode(c)) == c)
   all p : PlainText, c : CipherText |
      (enc[p] == c implies dec[c] == p) and
      (dec[c] == p implies enc[p] == c)
}
sig PubKey extends PlainText {
   sec: one SecKey,
   enc: PlainText one -> one CipherText,
   dec: CipherText one -> one PlainText
}
{
   // (decode(encode(p)) == p) and (encode(decode(c)) == c)
   all s : SecKey, v : PlainText |
    (sec == s) implies (dec[s.enc[v]] == v and s.dec[enc[v]] == v)
}
sig SecKey extends PlainText {
   pub: one PubKey,
   enc: PlainText one -> one CipherText,
   dec: CipherText one -> one PlainText
```

```
}
{
   // (decode(encode(p)) == p) and (encode(decode(c)) == c)
   all p : PubKey, v : PlainText |
    (pub == p) implies (dec[p.enc[v]] == v and p.dec[enc[v]] == v)
}
fact EncodingFacts {
   // different keys cause different encryptions and decryptions
   all disj k, k' : SymKey, p : PlainText, c : CipherText |
      k.enc[p] != k'.enc[p] and k.dec[c] != k'.dec[c]
   all disj k, k' : PubKey, p : PlainText, c : CipherText |
      k.enc[p] != k'.enc[p] and k.dec[c] != k'.dec[c]
   all disj k, k' : SecKey, p : PlainText, c : CipherText |
      k.enc[p] != k'.enc[p] and k.dec[c] != k'.dec[c]
   all k : PubKey, k' : SecKey, p : PlainText, c : CipherText |
      k.enc[p] != k'.enc[p] and k.dec[c] != k'.dec[c]
   all k : PubKey, k' : SymKey, p : PlainText, c : CipherText |
      k.enc[p] != k'.enc[p] and k.dec[c] != k'.dec[c]
   all k : SecKey, k' : SymKey, p : PlainText, c : CipherText |
      k.enc[p] != k'.enc[p] and k.dec[c] != k'.dec[c]
}
```

Appendix I

```
/*
 *
 * Alloy code for modeling the PIPE domain
 *
 */

abstract sig Record {}
sig Pseudonym extends PlainText {}
sig SharedPseudonym {
   aPID: one CipherText,
   // pseudonyms
   enc_ISKow_PSNid: one CipherText,
   enc_ISKow_PSNhe: one CipherText,
   enc_ISKau_PSNid: one CipherText,
   enc_ISKau_PSNhe: one CipherText,
   // user ids
   enc_ISKow_IUIDow: one CipherText,
   enc_ISKow_IUIDau: one CipherText,
   enc_ISKau_IUIDow: one CipherText,
   enc_ISKau_IUIDau: one CipherText
}

sig RootPseudonym {
   // user id
   enc_ISKow_IUIDow: one CipherText,
   // pseudonyms
   enc_ISKow_PSNid: one CipherText,
   enc_ISKow_PSNhe: one CipherText,
}

sig IdRecord extends Record {
   idInfo: some PlainText
}
```

```
sig HealthRecord extends Record {
   healthInfo: some PlainText
}

sig PseudonymRecordsMapping {
   psn: one Pseudonym,
   rec: one Record
}

sig User {
   IUID: one PlainText,
   OPuK: one PubKey,
   IPuK: one PubKey,
   encryptedIPK: one CipherText,
   encryptedISK: one CipherText,
   LastName: one PlainText,
   FirstName: some PlainText,
   UserType: one PlainText
}
```

Appendix J

```
/*
 *
 * Alloy code for modeling the PIPE domain facts
 *
 */

fact PseudonymFacts {
   // PSNid und PSNhe are, when decrypted, pseudonyms
   all psn : SharedPseudonym | one u : User |
      getIUID[u, psn] == u.IUID
   all psn : RootPseudonym  | one u : User |
      getIUID[u, psn] == u.IUID

   // PSNid und PSNhe are, when decrypted, pseudonyms
   all psn : SharedPseudonym | one ow : User |
      getPSNid[ow, psn] in Pseudonym and getPSNhe[ow, psn]
         in Pseudonym
   all psn : RootPseudonym  | one au : User |
      getPSNid[au, psn] in Pseudonym and getPSNhe[au, psn]
         in Pseudonym

   // decrypt(ISKau, IUIDow) == decrypt(ISKow, IUIDow)
   // decrypt(ISKau, PSNid) == decrypt(ISKow, PSNid) etc.
   all psn : SharedPseudonym | one ow, au : User |
      ( (getISK[ow].dec[psn.enc_ISKow_IUIDow] == ow.IUID)
           and
        (getISK[au].dec[psn.enc_ISKau_IUIDau] == au.IUID)
      )
      implies
      ( ( getISK[ow].dec[psn.enc_ISKow_IUIDow] ==
          getISK[au].dec[psn.enc_ISKau_IUIDow] ) and
        ( getISK[ow].dec[psn.enc_ISKow_IUIDau] ==
          getISK[au].dec[psn.enc_ISKau_IUIDau] ) and
        ( getISK[ow].dec[psn.enc_ISKow_PSNid] ==
```

```
          getISK[au].dec[psn.enc_ISKau_PSNid] ) and
        ( getISK[ow].dec[psn.enc_ISKow_PSNhe] ==
          getISK[au].dec[psn.enc_ISKau_PSNhe] )
      )
}
fact UserFacts {
   all u : User | getIPK[u] in SecKey and getISK[u] in SymKey
}

// helper functions
fun getIUID[u : User, s : SharedPseudonym] : one PlainText {
   getISK[u].dec[s.enc_ISKow_IUIDow]
}
fun getPSNid[u : User, s : SharedPseudonym] : one Pseudonym {
   getISK[u].dec[s.enc_ISKow_PSNid]
}
fun getISK[u : User] : one SymKey {
   u.OPuK.sec.dec[u.encryptedISK]
}
```