UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Secure Software Engineering

Specification and Verification of Security Protocols and their Utilization in Scenario-based Requirements Engineering

PhD Thesis
Submitted in partial fulfillment
of the requirements for the degree of
"Doktor der Naturwissenschaften (Dr. rer. nat.)"

by
THORSTEN KOCH

Supervised by:
Prof. Dr. Eric Bodden

Paderborn, September, 2024

Abstract

The widespread usage of software-intensive systems significantly increases the risk of cyber-attacks. Software-intensive systems must integrate security measures like security protocols to cope with this risk.

The correct specification and application of security protocols are tedious and error-prone. When specifying security protocols, security engineers use symbolic model checking to verify the security of the protocols. However, using more than one symbolic model checker is recommended for a thorough and confident analysis. Thus, security engineers need help transforming a security protocol into the input language of different symbolic model checkers to avoid the time-consuming and error-prone remodeling of the same security protocol. In addition, current requirements-engineering approaches address either functional or security-related requirements. Hence, requirements engineers need help to assess whether the applied security measures are sufficient to secure the system and whether the measures lead to conflicts with other functional requirements.

To cope with these challenges, we propose a systematic model-based approach for specifying and verifying security protocols in the symbolic model and utilizing them in a scenario-based requirements engineering methodology. Based on a UML-compliant modeling language, security engineers can specify security protocols and automatically analyze them using various symbolic model checkers. In addition, requirements engineers can systematically integrate the verified security protocols into requirements specifications. Furthermore, they can specify misuse cases against the system under development and validate whether it is sufficiently secure to mitigate them.

We conduct case studies for all our contributions based on realistic examples and show that our contributions are applicable in practice.

Zusammenfassung

Der weitverbreitete Einsatz von software-intensiven Systemen erhöht das Risiko von Cyberangriffen. Um dieses Risiko zu reduzieren, müssen daher verschiedene absichernde Maßnahmen wie Sicherheitsprotokolle in die Systeme integriert werden.

Die korrekte Spezifikation und Anwendung dieser Protokolle ist jedoch mühsam und fehleranfällig. Bei der Spezifikation verwenden Sicherheitsexperten verschiedene symbolische Model-Checker. Allerdings reicht ein einzelner symbolischer Model-Checker nicht aus, um die Sicherheit vollständig zu verifizieren. Daher benötigen Sicherheitsexperten Hilfe bei der Übersetzung eines Sicherheitsprotokolls in die Eingabesprache verschiedener symbolischer Model-Checker, um die zeitaufwändige und fehleranfällige Modellierung desselben Sicherheitsprotokolls zu vermeiden. Darüber hinaus adressieren die heutigen Ansätze zum Requirements Engineering entweder funktionale oder sicherheits-bezogene Anforderungen. Anforderungsmanagern fällt es daher schwer zu beurteilen, ob die spezifizierten Maßnahmen ausreichen das System abzusichern oder ob sie zu Konflikten mit anderen funktionalen Anforderungen führen.

In dieser Arbeit wird ein modellbasierter Ansatz zur Spezifikation und Verifikation von Sicherheitsprotokollen sowie deren Verwendung im szenario-basierten Requirements Engineering vorgestellt. Auf Basis einer UML-konformen Modellierungssprache können Sicherheitsprotokolle spezifiziert und automatisch in die Eingabesprache verschiedener symbolischer Model-Checker überführt werden. Zudem können die spezifizierten Protokolle systematisch in eine Anforderungsspezifikation integriert werden.

Die Evaluierung anhand von Fallstudien auf Basis realistischer Beispiele zeigt, dass die Beiträge dieser Arbeit in der Praxis anwendbar sind.

Danksagung

Ich möchte all den Menschen danken, die mich auf meinem Weg zur Promotion begleitet, ermutigt und unterstützt haben.

Zunächst gilt mein Dank meinem Doktorvater Eric Bodden für die Möglichkeit diese Arbeit zu schreiben. Eric Bodden gab mir wertvolles Feedback und war immer erreichbar, wenn ich Unterstützung brauchte. Ich danke Eric Bodden und Tibor Jager für die Erstellung ihrer Gutachten und den weiteren Mitgliedern meiner Prüfungskommission Matthias Meyer, Stefan Sauer und Juraj Somorovsky für ihre Bereitschaft an meiner Verteidigung teilzunehmen.

Ich möchte mich auch bei all meinen ehemaligen und aktuellen Kollegen am Fraunhofer IEM und in der Fachgruppe Secure Software Engineering bedanken. Hervorheben möchte ich Faruk Pasic für die gemeinsame Zeit im Büro, die angenehme Arbeitsatmosphäre und die anregenden Gespräche. Ich möchte auch Jörg Holtmann, Markus Fockel und David Schmelter für die vielen Diskussionen und den wissenschaftlichen Austausch im Rahmen der RE-Expertengruppe zu Beginn meiner Zeit am Fraunhofer IEM danken. Für das Lösen von organisatorischen und technischen Herausforderungen bedanke ich mich bei der Verwaltung und der IT am Fraunhofer IEM und in der Fachgruppe, insbesondere bei Vera Meyer und Nicole Graskamp.

Ich danke meinen Co-Autoren Jörg Holtmann, Stefan Dziwok, Sascha Trippel und Eric Bodden für die gute Zusammenarbeit und den wissenschaftlichen Austausch bei den für diese Arbeit relevanten Publikationen. Außerdem danke ich allen, die Teile dieser Arbeit oder Teile meiner Veröffentlichungen Korrektur gelesen haben.

Weiter danke ich Bahar Jazayeri, Lucas Briese, Richard Hochhalter, Alexander Kaiser und Sascha Trippel, die mich durch ihre Abschlussarbeit oder ihre SHK-Tätigkeit bei dieser Arbeit unterstützt haben.

Schließlich möchte ich mich bei meiner Familie bedanken. Meine Eltern und mein Bruder haben mich immer auf meinem Weg unterstützt. Ich möchte meiner Frau Ricarda dafür danken, dass sie in den letzten Jahren alle Höhen und Tiefen mit mir geteilt hat.

Contents

1

Introduction

In recent years, software-intensive systems (e.g., embedded [HS07], mechatronic [Aus96; VDI04], or cyber-physical systems [Poo10; SW07]) have become prevalent in our daily lives and are widely used in private and business environments.

However, the widespread usage of software-intensive systems also significantly increases the risk of cyber-attacks. Thus, security has become one of the most crucial technology risks for the world's population [Wor23]. While attacks in the private environment primarily pursue collecting personal information, attacks on business systems vary from spying on sensitive data to obtaining secret product and production knowledge to manipulate and disrupt operations. In particular, attacks on technical systems (vehicles, plants, or medical devices) pose a high risk since malfunctions caused by security incidents can lead to life-threatening accidents [Com19; Bun21]. For example, security researchers conducted an attack on the Jeep Cherokee and were able to remotely control the vehicle and manipulate the brakes and the motor control [MV15].

A closer look at software-intensive systems reveals that the message-based communication between and within these systems is often vulnerable to cyber-attacks. To counter this risk, security protocols [Sch96] can be used to ensure security within communication networks [Bla01]. Security protocols rely on executing security-related functions and apply cryptographic operations like encryption or digital signatures. Thus, security protocols must be correctly designed and specified. However, this is a tedious and error-prone task due to complex security requirements and their dependencies on the attacker model [Bla01; DT19]. Consequently, many flaws in security protocols were only discovered after years of productive usage [Low96b; JV96; ABD^{+}15; CKM20; BST21]. For example, although TLS is widely used in practice, flaws have been found repeatedly [MS13; BLF^{+}14; BBD^{+}17; BBK17].

Security protocols must be adequately integrated into the subsequent application. If the protocol is integrated incorrectly or parts of the message-based communication remain unprotected, attacks can still be successful. Furthermore, it must be ensured that the security requirements do not conflict with other requirements and do not lead to an inconsistent requirements specification. For example, integrating a security protocol usually leads to more messages exchanged between the communication participants. Moreover, the message delay increases since executing cryptographic operations requires additional time. Both examples can lead to an unintended timing behavior of the final system and, thus, can cause several malfunctions, particularly in safety-critical systems.

Consequently, security must be addressed from the beginning and continuously in the development process when developing software-intensive systems. In particular, security engineers need a systematic approach that supports the specification and analysis of security protocols. Furthermore, requirements engineers need a systematic approach to reuse existing security protocols and apply them in subsequent applications. The resulting requirements specification provides an intuitive representation of all relevant requirements through the combined consideration of functional and security requirements on message-based communication. Moreover, analysis techniques can be executed to find inconsistencies in the specification. Thus, the quality of requirements on software-intensive systems' communication behavior improves, and the risk of security flaws during productive usage is reduced.

1.1 Problem Description

In this section, we sketch challenges that apply in the specification and analysis of security protocols (cf. Section 1.1.1) and in the specification and analysis of functional and security requirements (cf. Section 1.1.2). All problems are currently insufficiently solved by related approaches.

1.1.1 Specification and Analysis of Security Protocols

During the specification and analysis of security protocols, security engineers typically apply approaches from the computer-aided cryptography [BBB$^+$19] area like computational and symbolic model checking to verify whether a security protocol fulfills certain security properties such as secrecy and authentication. The symbolic model is relatively simple and assumes that cryptography is perfect, i.e. an adversary cannot break the algorithm [DY83]. The simplicity of the symbolic model enables the application of automated tools like PROVERIF [Bla01], TAMARIN [Mei13]. In contrast, the computational model is more powerful than the symbolic model and treats cryptographic primitives as probabilistic algorithms on bitstrings and adversaries as probabilistic Turning machines [BBB$^+$19]. These assumptions make it much more difficult to build tools automated tools. Thus, computational model checkers like CryptoVerif [Bla23] and EasyCrypt [BGH$^+$11] need guidance from the user to complete their proofs. However, many practical attacks have been found in both models and thus the analysis based on the symbolic model has been proven to be of great value [LZK20].

The challenges outlined in this section apply to both computational and symbolic model checking. However, in the remainder of this thesis, we focus only on symbolic model checkers and leave the consideration of computational model checkers for future work.

When applying symbolic model checking to verify the security of a protocol, PROVERIF [Bla01] and TAMARIN [Mei13] are typically used by security engineers. Both tools have been used to analyze different real-world protocols [BBB$^+$19]. However, each model checker has different capabilities and restrictions. For instance, PROVERIF aims to provide a fully automatic analysis that always terminates and does not need any interaction with the user. Thus, PROVERIF makes some strong assumptions to fulfill this aim [Bla01]. In contrast, TAMARIN is less restrictive but relies on the user's guidance to proceed with the analysis in some cases [Mei13]. Although the mentioned model checkers are sound under their assumptions, i.e., if they do not find an attack during an analysis, then none exists [Bla01; Mei13], and most of the restrictions do not apply for real-world security protocols [KNT19], a thorough and confident analysis of security protocols typically requires multiple model checkers with different analysis capabilities to ensure that no (critical) security flaws remain undetected.

While specifying and analyzing security protocols using these two model checkers, we identified three challenges that lead to inefficient development and critical security flaws if they are insufficiently handled:

1. "[Both model checkers] have their own textual modeling and query languages, which are fundamentally different. Moreover, each [...] input language requires the security engineer to have deep knowledge and experience. However, as stated above, [to improve the confidence of the analysis], the security engineer [should] use multiple model checkers for a thorough analysis. [Therefore, the security engineer] has to re-model the security protocol, including its queries in other languages repeated times. This is time-consuming and error-prone. Moreover, in our experience, the model checkers' textual input languages are generally rather hard to comprehend, which adds a burden to the engineer" [*KDH+20].

2. "Choosing the set of queries that the model checker must verify is highly important, as the query results ultimately decide whether the protocol is accepted as secure. If important queries are missing or are specified incorrectly, existing flaws remain undetected in the security protocol. This may lead to critical security flaws. In existing models, the knowledge to choose and specify this set of queries is typically hidden, distributed over several papers and websites, or within the brains of experts. Therefore, it is not easily accessible for a common security engineer" [*KDH+20].

3. Comprehensible analysis results are another essential part of the verification of security protocols. Usually, model checkers summarize all properties that have been analyzed and provide a counterexample for the properties that the security protocol does not fulfill. However, these counterexamples are primarily specified in a language similar to the model checker's internal representation and, thus, hard to understand for the security engineer. Furthermore, there is no guidance for the security engineer that highlights the problematic parts of the security protocol. Hence, the security engineer must manually map the counterexample to the specified security protocol.

Related work only marginally addresses these challenges. "FANG ET AL. [FLH+16] propose an extension to Unified Modeling Language (UML) Interactions [Obj17b] to enable the specification of security protocols. Furthermore, they provide an automatic transition to PROVERIF to analyze the specified protocols. However, their modeling and query language is very PROVERIF-specific. Other model checkers are not supported, and their support would require significant changes to the modeling language. Moreover, the security engineer receives no support for choosing a suitable set of queries" [*KDH+20]. Furthermore, they do not provide a translation of the analysis back to the modeling approach. Hence, the security engineer must use the modeling tool for the specification and PROVERIF for the analysis of security protocols. Furthermore, the security engineer must investigate the analysis results and map them manually to the origin input. Consequently, the security engineer still has to perform time-consuming and error-prone tasks and learn the PROVERIF modeling and query language.

"AMEUR-BOULIFA ET AL. [ALA19] present a modeling approach based on the Systems Modeling Language (SysML) [Obj17a] to enable the specification of security and safety aspects using SysML State Machines, including security protocols and their queries. For the security analysis, they translate their models to PROVERIF and generate queries concerning the confidentiality of the protocol. However, they only support PROVERIF as the single model checker for security protocols. In addition, they do not generate all sufficient queries concerning the authenticity of the protocol. Finally, scenario-based models are more appropriate than state-based models for the specification of requirements on message-based interactions in terms of efficient comprehensibility [LT15]. Particularly, scenario-based notations have an intuitive representation [HRD10] and improve the comprehension of interaction requirements for people experienced in modeling [AGI+13]" [*KDH+20].

To summarize this section, the mentioned approaches enable the specification and verification of security protocols. However, they support only one model checker to analyze security protocols and do not allow a thorough analysis. Consequently, to solve the three challenges, we have to conceive a systematic approach for the specification of security protocols and an automatic transformation approach to different model checkers to enable a thorough analysis.

1.1.2 Specification and Analysis of Functional and Security Requirements

Today's software-intensive systems' growing functionality and complexity require rigorous development processes. This is especially true for the requirements engineering phase since the detection and fixing of defects in subsequent development phases cause costly iterations [Poh10]. During the requirements engineering phase, the requirements engineer must consider functional and security requirements.

In the functional requirements engineering of software-intensive systems, models are considered beneficial for documenting functional requirements [STP12] as they increase the understanding [NT09] and enable automatic analysis techniques. In particular, scenario-based approaches are well suited for documenting requirements on the message-based communication behavior. In a scenario-based approach, requirements engineers use scenarios to describe sequences of events the system has to accomplish [HRD10]. The resulting requirements specification provides an intuitive representation of the system's behavior [HRD10] and is easy to understand for people with modeling experience [AGI+13].

GREENYER [Gre11] developed a requirements engineering approach based on Modal Sequence Diagrams (MSDs) [HM08], a recent Live Sequence Charts (LSC) [Har00; Har01] variant compliant to the Unified Modeling Language (UML) [Obj17b]. Furthermore, GREENYER applied and extended two complementary automatic analysis techniques, enabling the early detection of requirements defects. Based on this MSD dialect, we defined a scenario-based requirements engineering methodology for the specification and analysis of requirements on the message-based communication behavior of software-intensive systems [*HFK+16b]. We implemented tool support for the specification and analysis of MSDs as a set of plugins for Eclipse as part of the SCENARIOTOOLS-MSD tool suite. However, although security is a major concern in developing software-intensive systems, our scenario-based requirements engineering methodology and related approaches only address functional and safety requirements.

In the security requirements engineering of software-intensive systems, many approaches exist that enable to specify security requirements. Instead of describing what the system should do, many approaches utilize misuse cases [SO05] or abuse cases [PX05] as negative scenarios to specify what is not allowed to happen during the system's execution. Furthermore, several other approaches worked on eliciting and specifying security requirements, for example, security use cases [Fir03; Fir07]; UMLsec [Jür02; Jür05], a framework for security requirements engineering [HLM+08]; SQUARE [MS05]; security requirements methods based on i* framework [LYM03]; Secure Tropos [GMZ06]. Moreover, WHITTLE ET AL. [WWH08] developed an approach to formalize use and misuse cases using extended interaction overview diagrams (EIODs). Based on the EIOD specification, they provided an analysis technique that executes misuse cases against scenario-based specifications to investigate whether an attack is successful.

However, none of the mentioned approaches provide sufficient analysis techniques to validate whether the specified security requirements are fulfilled and whether a potential attacker can execute the specified attack. In addition, they do not provide any integration into existing functional requirements engineering methodologies or development processes. Therefore, a requirements engineer would have to model large parts of the system twice to analyze both types of requirements. This is an error-prone and time-consuming task. Furthermore, interactions between the two types might only become apparent late in the development.

To summarize this section, the idea of scenarios is beneficial for eliciting and documenting functional and security requirements. However, the mentioned approaches either focus on functional or security requirements. No approach systematically covers both aspects. Consequently, we have to conceive a scenario-based requirements engineering approach for secure software-intensive systems that enables the specification and analysis of functional and security requirements and their interplay.

1.2 Contributions

The goal of this thesis is a systematic approach for the specification and verification of security protocols and their utilization in a scenario-based requirements engineering methodology. Therefore, we propose a modeling approach that enables the specification of security protocols (C1). Based on our modeling approach, we present a model-checking approach to automatically transfer a specified security protocol to the input model of various symbolic model checkers and analyze whether the security protocol is secure in the symbolic model (C2). Finally, we introduce an approach to incorporate security requirements into scenario-based functional requirements specifications (C3).

This section presents the three contributions in further detail. We implemented all concepts as extensions to the tool suite SCENARIOTOOLS-MSD and evaluated the concepts through different case studies.

C1 We propose the SECURITY MODELING PROFILE, a new UML-compliant profile based on Modal Sequence Diagrams [*HFK+16b] that provides a set of widely used security primitives. Security engineers can use the SECURITY MODELING PROFILE for the scenario-based specification of security protocols. Furthermore, requirements engineers can use the SECURITY MODELING PROFILE to integrate security measures into the requirements specification of their application to ensure the security of the message-based communication. The proposed profile is comprehensive for engineers familiar with the UML and basic security concepts. Moreover, the profile defines certain constraints on the final model and, thus, avoids specification errors. In addition, we extend the validation technique Play-out [HM03] to enable the simulation of the specified security protocols and the security-enhanced requirements specification.

C2 We present VICE (VIsual Cryptography vErifier), a model-checking approach for verifying security protocols addressing the three challenges mentioned in Section 1.1.1. Figure 1.1 depicts an overview of VICE's workflow. The security engineer manually specifies a security protocol by means of the Security Modeling Profile. Then, the security engineer applies the automatic workflow for the security model checking. This encompasses an automatic and systematic technique for the transition from the Security Modeling Profile to the two symbolic model checkers ProVerif and Tamarin (cf. 2.1 Automatic Translation of Security Protocols). Furthermore, VICE encapsulates the knowledge formerly hidden within models, documents, and experts to specify and choose the necessary set of queries the model checker shall verify to decide whether the protocol is secure concerning secrecy and authentication. This is realized by an automatic generation of all necessary queries (cf. 2.2 Automatic Generation of Analysis Queries). After the termination of the symbolic model checkers ProVerif and Tamarin, VICE translates the analysis results back to the level of the Security Modeling Profile (cf. 2.3 Visualization of Analysis Results). Thereby, we preserve the security engineers from the manual and awkward understanding of the analysis results. Finally, if the analysis shows that the security protocol is secure, the security engineer can store it in a data store.

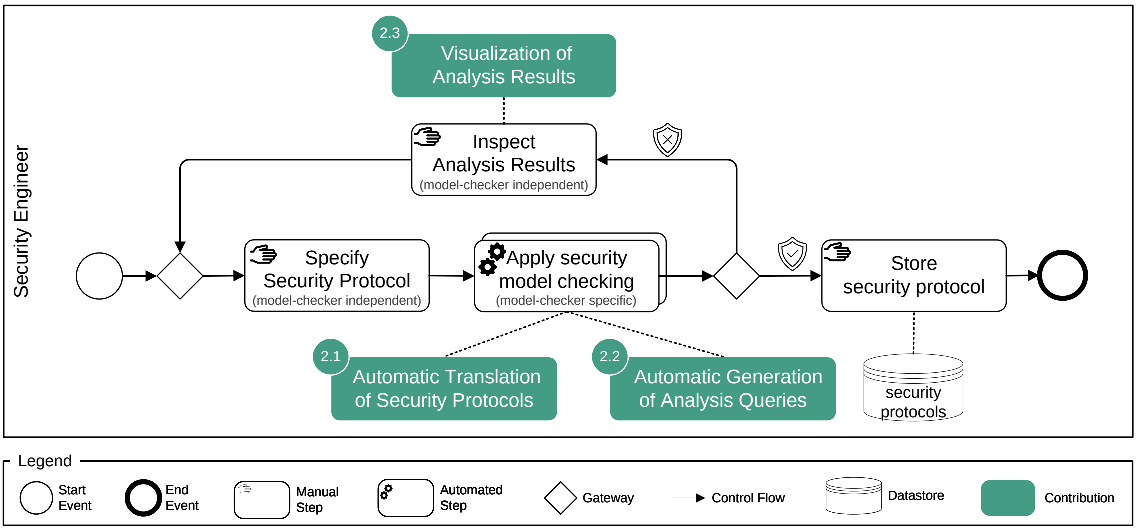

Security Engineer
2.3
Visualization of Analysis Results
Inspect Analysis Results (model-checker independent)
Specify Security Protocol (model-checker independent)
Apply security model checking (model-checker specific)
Store security protocol
security protocols
2.1
Automatic Translation of Security Protocols
2.2
Automatic Generation of Analysis Queries
Legend
Start Event
End Event
Manual Step
Automated Step
Gateway
Control Flow
Datastore
Contribution

Figure 1.1: Overview of VICE's workflow

C3 We propose an extension to our requirements engineering methodology based on Modal Sequence Diagrams [*HFK$^+$16b] to incorporate security requirements into scenario-based functional requirements specifications. Figure 1.2 depicts an overview of the extended requirements engineering methodology. As in the original methodology, the requirements engineer specifies functional requirements on the communication behavior by creating an MSD specification. Therefore, he/she specifies the structural basis for the MSD specification (cf. Specify Structure). Afterward, he/she creates use cases to structure the MSD specification (cf. Specify Use Cases). Each use case encapsulates requirements on the message-based communication behavior of the system under development and encompasses a set of MSDs specifying the actual requirements on the communication behavior (cf. Specify Scenarios).

As an extension to this workflow, we introduce the two steps Specify Misuse Cases and Add Security Measures. To realize the step Specify Misuse Cases, we present the MISUSE CASE MODELING PROFILE enabling the specification of misuse cases. A misuse case describes behavior that is either maliciously exploited by attackers or accidentally triggered through incorrect use (cf. (3.1) Specification of Misuse Cases). To mitigate the specified misuse cases, the requirements engineer must add security measures to the scenarios. Therefore, he/she can either use cryptographic primitives of the SECURITY MODELING PROFILE or reuse existing and verified security protocols stored in a data store. To enable the reuse of security protocols, we introduce the SECURITY PROTOCOL TEMPLATE PROFILE, a technique to systematically reference security protocols defined as UML Interactions through our SECURITY MODELING PROFILE within other MSDs (cf. (3.2) Template-based Utilization of Security Protocols). Finally, we extend the Play-out algorithm to support our extensions and enable requirements engineers to find inconsistencies in the requirements specification early in the development (cf. (3.3) Enhancement of the Play-out Algorithm).

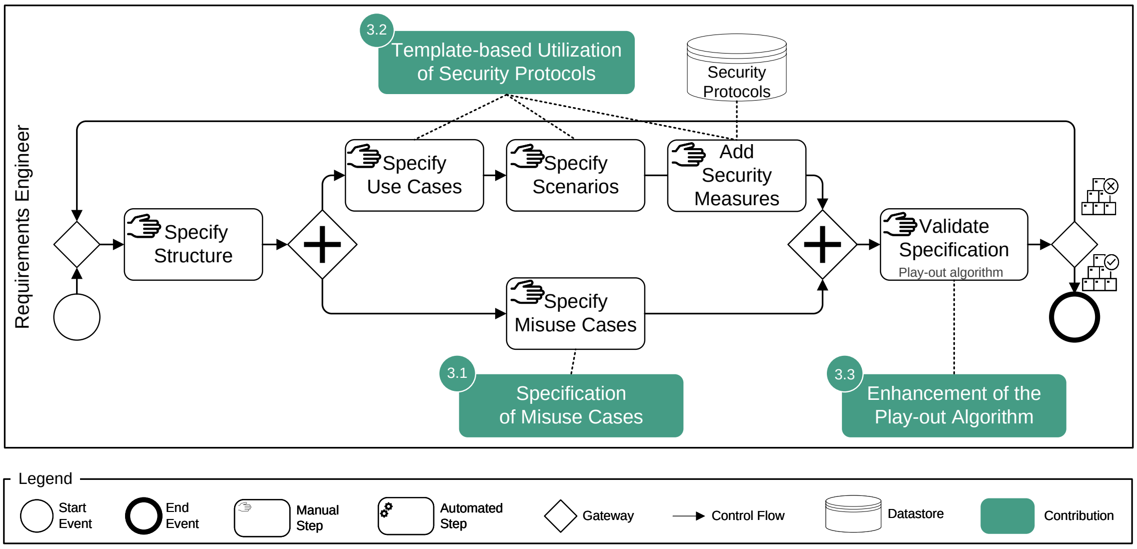


Figure 1.2: Overview of security extensions to our requirements engineering methodology based on Modal Sequence Diagrams [*HFK$^+$16b]

1.3 Thesis Structure

This thesis is structured as follows. Chapter 2 introduces the foundations of this thesis. Chapter 3 presents our SECURITY MODELING PROFILE for the specification of security protocols. Afterward, Chapter 4 introduces VICE our model-checking approach for the analysis of security by means of the two symbolic model checkers PROVERIF and TAMARIN. Chapter 5 explains our approach for the incorporation of functional and security requirements by means of MSDs. Finally, Chapter 6 concludes the thesis with a summary and an outlook on future work.

2

Foundations

This chapter introduces the foundations necessary to understand the concepts of this thesis. Section 2.1 introduces foundations on Modal Sequence Diagrams, and Section 2.2 introduces foundations on the specification and analysis of security protocols.

2.1 Modal Sequence Diagrams (MSDs)

This section presents foundations on Modal Sequence Diagrams (MSDs), a formal, model- and scenario-based behavior specification language based on UML sequence diagrams [HM08]. We follow the definition of the MSD requirements language from the technical report by HOLTMANN ET AL. [*HFK$^+$16b]. The presented concepts have been implemented in the tool suite SCENARIOTOOLSMSD [ST-MSD].

Section 2.1.1 introduces the overall structure of an MSD specification. Afterward, Section 2.1.2 presents the MSD semantics. Then, Section 2.1.3 describes the Play-out algorithm, an automatic validation technique for MSDs. Finally, Section 2.1.4 introduces the Modal profile.

2.1.1 Structure of MSD Specifications

An MSD specification is structured by means of *MSD use cases*. Each use case encapsulates requirements on the message-based communication behavior of the system under development in a self-contained situation. An MSD use case encompasses the participants involved in the situation and a set of MSDs specifying the actual requirements on the communication behavior.

Figure 2.1 depicts an excerpt of an MSD specification for the *Emergency Braking & Evasion Assistance System (EBEAS)*. The EBEAS is an advanced driver assistance system that reduces the risk of rear-end collisions in case of obstacles in front of a vehicle. Therefore, the EBEAS combines autonomous braking and emergency steering systems with vehicle-to-vehicle communication technology [*HFK$^+$16b].

The excerpt describes a self-contained situation in which the vehicle leading detects an obstacle in front and has to perform an emerging brake. The vehicle leading checks if the last point to brake has already been exceeded. If this is not the case, it informs the vehicle ego about the upcoming emergency brake. Then, the vehicle ego checks if it can perform an emergency brake and communicates with the vehicle rear to further coordinate the actions to the dangerous situation.

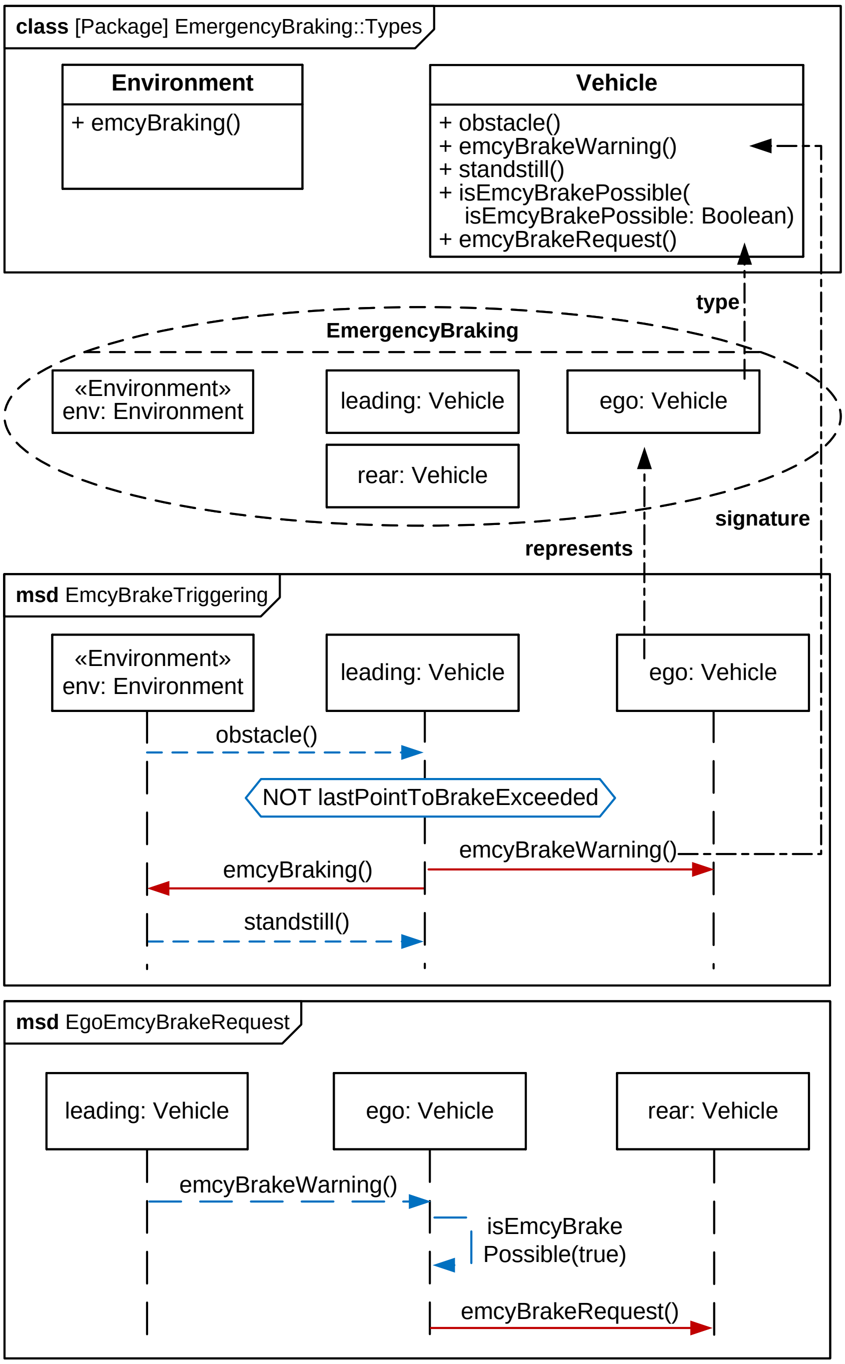


Figure 2.1: MSD Specification Excerpt

UML classes provide reusable types for all MSD use cases. These types are used to define the structure of the system under development and its environment. Moreover, the UML classes define operations used as message signatures and define which messages can be received by a participant in an actual MSD. For example, the class diagram in Figure 2.1 depicts the two classes Environment and Vehicle. The class Environment encompasses the operation emcyBraking().

Based on the UML classes, a UML collaboration (dashed ellipse in Figure 2.1) specifies the roles participating in a particular MSD use case. The roles are typed by the UML classes and used as lifelines in an MSD. For example, the UML collaboration in Figure 2.1 encompasses the role ego: Vehicle which has an abstract syntax link type to the class Vehicle. We distinguish between environment and system objects. Environment objects are annotated with the stereotype «Environment» (e.g., Environment in Figure 2.1).

Based on the UML classes and the collaboration, a set of MSDs specifies the requirements on the communication behavior between the roles involved in the MSD use case. An MSD encompasses lifelines and messages. Lifelines correspond to a role defined by the containing collaboration, i.e., depicted by the abstract syntax link represents. Messages are associated with a sending and a receiving lifeline and an operation signature. For example, as depicted in Figure 2.1, the message emcyBrakeWarning() is associated with the equally named operation of the class Vehicle by means of the abstract syntax link signature. MSD messages sent from environment objects are called environment messages, whereas messages sent from system objects are called system messages.

The operations associated with the messages can have parameters of certain types. For example, the operation associated with the message isEmcyBrakePossible() encompasses the Boolean parameter isEmcyBrakePossible. The argument for the parameters can be concrete literal values like in conventional UML or refer to variables. For example, the parameter isEmcyBrakePossible is specified to carry the literal value “true” as an argument.

Assignments enable storing temporally valid intermediate values. They are represented by rectangles covering one or multiple lifelines. An assignment contains an expression of the form <var> = <expr>, where <var> is the name of a typed *diagram variable* declared for the time the MSD is active, and <expr> is a value expression specifying a literal or an expression specified by means of the Object Constraint Language (OCL) [Obj14]. The diagram variables can be referenced by message arguments and conditions.

To specify conditional behavior, MSDs can contain *conditions* represented as hexagons covering one or more lifelines. Conditions contain OCL expressions that evaluate to a Boolean value, typically involving one diagram variable. For example, the MSD in Figure 2.1 encompasses a condition covering the lifeline leading: Vehicle. This condition contains the expression NOT lastPointToBrakeExceeded.

2.1.2 Semantics of MSD Specifications

Intuitively, an MSD progresses as *message events* occur in a real system at runtime or in an object system during the simulative validation by means of the Play-out algorithm [HM03].

Each message in an MSD has a temperature and an execution kind, represented by its color and line style. The temperature of a message represents its modality and can be hot (red color) or cold (blue color). The semantics of a hot message is that other messages specified by the MSD are not allowed to occur. For example, the MSD messages emcyBrakeWarning(), emcyBraking(), and emcyBrakeRequest() are hot, whereas the others are cold. The execution kind of a message can either be executed (solid line) or monitored (dashed line). Executed messages must be sent eventually when the execution of the MSD reaches them. Monitored messages never need to occur, i.e., their occurrence is optional. For example, the MSD messages obstacle(), standstill(), emcyBrakeWarning(), and isEmcyBrakePossible() are monitored, whereas the others are executed.

A message event is unifiable with an MSD message if and only if the event name equals the message name and the sending and receiving lifelines of the message are bound to the sending and receiving objects. In the case of a parameterized message, the message event is unifiable if, additionally, the message event specifies for each parameter either "the same literal value, a variable bound to the same value, or an unbound variable" [Gre11]. "When a message event occurs in the object system that is unifiable with the first message in an MSD, an active MSD is created. Such a first message in an MSD is also called a minimal message and is always cold and monitored. As further message events occur that can be unified with the subsequent MSD messages, the active MSD progresses. This progress is captured by the cut, which marks for every lifeline the locations of the MSD messages that are unifiable with the message events. If the cut reaches the end of an active MSD, the active MSD is terminated" [Hol19].

Each condition in an MSD has a temperature. A cold condition is colored blue, and a hot condition is colored red, respectively. If the cut is in front of a condition, the expression is evaluated immediately. If the expression evaluates to true, the cut progresses beyond the condition. If the expression evaluates to false, the subsequent message event sequence is not expected to occur anymore (the MSD "terminates"). An assignment in an MSD does not have a temperature or an execution kind. Thus, if the cut is in front of an assignment, the expression is executed immediately, and the cut progresses beyond the assignment. "Executing an assignment [defining a diagram variable] or progressing beyond a condition has no effect that is visible within the object system. [...] Therefore, executing an assignment and progressing beyond a condition is also called a *hidden event*. By contrast, the sending of a message in the object system is visible in the object system and is therefore also called a *visible event*" [Gre11].

A violation occurs in an MSD if a message event occurs that is unifiable with a specified message but not enabled in this MSD or if a condition evaluates to false. If a violation occurs, the active MSD is terminated. We define different types of violations based on the execution kind and temperature of the cut:

- If a violation occurs in a cold cut (i.e., all enabled messages are cold), it is a cold violation, which represents a legal trace and does not violate the requirements.
- If a violation occurs in a hot cut (i.e., at least one enabled message is hot), it is a hot violation, which violates the requirements.
- If a violation occurs in an executed cut (i.e., at least one enabled message is executed), it is a liveness violation, which violates the requirements.

2.1.3 Analysis Techniques for MSD Specifications

The Play-out algorithm [HM03] enables the validation of an MSD specification. While simulating selected execution paths of the system under development, the Play-out algorithm finds defects concerning the consistency and correctness of the requirements specification.

At the beginning, the algorithm waits for environment events to occur. If an environmental event occurs, MSDs whose initial event is unifiable with the environment event are activated. Next, the Play-out algorithm chooses and executes one of the enabled system events non-deterministically. The process repeats until no active MSDs with executed cuts are left. The system then waits for the following environment event. In case of a hot violation, the algorithm terminates.

2.1.4 The Modal Profile

The language for an MSD specification is specified by means of the UML language [UML] and the Modal profile [Gre11; ST-MSD]. The Modal profile is realized as a UML profile. A UML profile is an extension mechanism for specifying domain-specific modeling languages based on UML metaclasses. A UML profile encompasses several stereotypes defining the domain-specific language constructs and extending UML metaclasses. A stereotype may encompass several properties stored as tagged values.

Figure 2.2 depicts the Modal profile, which introduces MSD-specific language constructs, such as the temperature and execution kind of MSD messages. Moreover, the Modal profile provides modeling constructs for temporal variables and references to them.

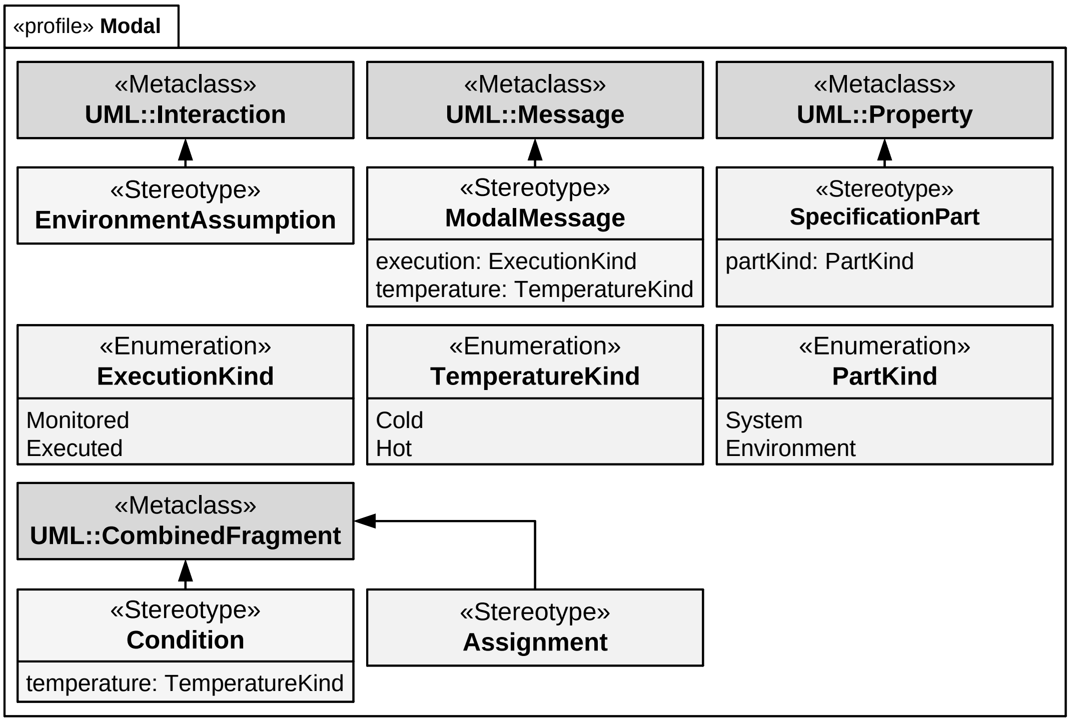

«profile» Modal
«Metaclass» UML::Interaction
«Metaclass» UML::Message
«Metaclass» UML::Property
«Stereotype» EnvironmentAssumption
«Stereotype» ModalMessage
execution: ExecutionKind
temperature: TemperatureKind
«Stereotype» SpecificationPart
partKind: PartKind
«Enumeration» ExecutionKind
Monitored
Executed
«Enumeration» TemperatureKind
Cold
Hot
«Enumeration» PartKind
System
Environment
«Metaclass» UML::CombinedFragment
«Stereotype» Condition
temperature: TemperatureKind
«Stereotype» Assignment

Figure 2.2: The Modal Profile

2.2 Specification and Analysis of Security Protocols

This section introduces basic concepts for the specification and analysis of security protocols. Section 2.2.1 introduces the Alice & Bob notation. Afterward, Section 2.2.2 discusses the foundations for analyzing security protocols in the symbolic model. Finally, Section 2.2.3 and Section 2.2.4 present the two symbolic model checkers PROVERIF and TAMARIN, respectively.

2.2.1 Alice & Bob Notation

The Alice & Bob notation provides an intuitive way to describe security protocols in a textual manner. As in message-based notations (e.g., Message Sequence Charts [Tel96], Live Sequence Charts (LSC) [Har00; Har01]), the Alice & Bob notation describes the communication between entities as a sequence of messages. However, the description is often incomplete and only describes actions taken in a complete protocol run between the participants [CVB06].

We use the *Needham-Schroeder Public Key* protocol [NS78] to illustrate the Alice & Bob notation. The goal of this security protocol is the mutual authentication of the two participants Alice and Bob using asymmetric encryption and a trusted third party called Sue. Listing 2.1 depicts the Alice & Bob notation for the *Needham-Schroeder Public Key*. We slightly modified the Alice & Bob notation syntax defined by MÖDERSHEIM [Möd09] and CALEIRO [CVB06] to improve the readability. A security protocol defined in the Alice & Bob notation encompasses three parts *Types*, *Knowledge*, and *Actions* explained in the following.

The first part of an Alice & Bob notation defines the *Types* used throughout the security protocol, e.g., participants, data types or functions. For example, the *Needham-Schroeder Public Key* security protocol encompasses the three participants Alice, Bob, and Sue (cf. lines 3–4). Furthermore, the security protocol encompasses nonces (cf. lines 5–6) and cryptographic key pairs consisting of a private and a public key (cf. lines 7–12).

The second part of an Alice & Bob notation defines each participant's initial knowledge before the execution. For example, in the *Needham-Schroeder Public Key* security protocol, the participant Alice knows her cryptographic key pair and Sue's identifier and public key (cf. line 14).

The third part of Alice & Bob notation defines the communication between participants. Every message exchange has the form A → B: M, meaning that participant A sends the message M to participant B. The message's receiver must be the next sender [CVB06]. For example, in the *Needham-Schroeder Public Key* security protocol, Alice sends a message to Sue encompassing her identifier and Bob's identifier (cf. line 18).

2.2.2 Analyzing Security Protocols in the Symbolic Model

The symbolic model, also called the *Dolev-Yao model* [DY83], is a simple model in which the adversary is a specific, non-deterministic state machine and cryptographic primitives are symbolic functions. In particular, the symbolic model assumes that cryptography is perfect that is, an adversary cannot break the algorithm, e.g., he/she cannot learn anything from an encrypted message except if he/she has the corresponding key [Bla01; CDL06]. In addition, the symbolic model assumes that the adversary controls the network, e.g., he/she can discard/delay messages or send additional messages over the network [Bla12].

```
Protocol: Needham-Schroder Public Key
Types:
  Participant:
    Alice, Bob, Sue
  Nonce:
    Na, Nb
  PrivateKey:
    KSa, KSb, KSs
  PublicKey:
    KPa, KPb, KPs
  KeyPair:
    (KSa, KPa), (KSb, KPb), (KSs, KPs)
Knowledge:
  Alice: Alice, Sue, (KSa, KPa), KPs
  Bob: Bob, Sue, (KSb, KPb), KPs
  Sue: Alice, Bob, (KSs, KPs), KPa, KPb
Actions:
  1. Alice → Sue: Alice, Bob
  2. Sue → Alice: {KPb, Bob}KSs
  3. Alice → Bob: {Na, Alice}KPb
  4. Bob → Sue: Bob, Alice
  5. Sue → Bob: {KPa, Alice}KSs
  6. Bob → Alice: {Na, Nb, Bob}KPa
  7. Alice → Bob: {Nb}KPb
```

Listing 2.1: Alice & Bob notation of the *Needham-Schroder Public Key* security protocol

Security Properties

Security protocols try to achieve various security properties. These properties can be categorized into two groups: trace properties and equivalence properties. Trace properties can be defined "on each execution trace (each run) of the protocol. The protocol satisfies such a property when it holds for all traces in the symbolic model [...]. For example, the fact that some states are unreachable is a trace property. Equivalence or indistinguishability properties mean that the adversary cannot distinguish two processes. For instance, one of these processes can be the protocol under study, and the other one can be its specification. Then, the equivalence means that the protocol satisfies its specification" [Bla12].

In the following, we explain the two security properties *secrecy* and *authentication*.

Secrecy Secrecy means that the adversary cannot obtain any information from the data exchanged between the participants. In the symbolic model, secrecy can be formalized in two ways:

1. Syntactic secrecy means that the adversary cannot compute the exact data under consideration. However, this does not prevent the adversary from knowing parts of this data. Moreover, this notion cannot be used to express the secrecy of a term chosen from a (small) set of constants [Bla12]. In the symbolic model, syntactic secrecy is formalized by trace properties.

2. Strong secrecy means that the adversary cannot detect a change in the value of the secret information. This notion of secrecy is much stronger than syntactic secrecy. However, it

is not a trace property but an equivalence property and, thus, more difficult to verify using automated tools [Bla12].

Authentication "Authentication means that, if a participant A runs the protocol apparently with a participant B, then B runs the protocol apparently with A, and conversely" [Bla12].

In the symbolic model, authentication is formalized by correspondence properties [WL93; Low97] of the form: "if A executes a certain event e_1 (for instance, A terminates the protocol with B), then B has executed a certain event e_2 (for instance, B started a session of the protocol with A)" [Bla12].

LOWE [Low97] defined a hierarchy of authentication properties for the symbolic model to relate the different definitions of authentication to each other. In the following, we informally describe these properties for the two participants A and B:

Aliveness: "A security protocol guarantees *aliveness* to a [participant] A with another [participant] B if, whenever A completes a run of the protocol, apparently with B, then B has previously been running the protocol" [Low97].

Weak Agreement: "A security protocol guarantees *weak agreement* to a [participant] A with another [participant] B if, A completes a run of the protocol, apparently with B, then B has previously run the protocol, apparently with A" [Low97].

Non-injective Agreement: "A security protocol guarantees *non-injective agreement* to a [participant] A with another [participant] B if, A completes a run of the protocol, apparently with B and some data values $\vec{v}$, then Bob has previously been running the protocol, apparently with A and $\vec{v}$ " [Low97].

Injective Agreement: "A security protocol guarantees *injective agreement* to a [participant] A with another [participant] B if, A completes a run of the protocol, apparently with B and some data values $\vec{v}$, then Bob has previously been running the protocol, apparently with A and $\vec{v}$. In addition, each run of A has to correspond to a unique run of B" [Low97].

2.2.3 PROVERIF

PROVERIF is a model checker for the verification of security protocols presented by BLANCHET ET AL. [Bla01; Bla16] that can prove security properties such as secrecy and authentication in the symbolic model.

PROVERIF takes as input a plain text model of a security protocol specified by means of the *applied pi calculus* [AF01; ABF16]. This model is automatically translated into an internal presentation to execute the analysis and to verify if the desired security properties hold. If they do not hold, PROVERIF tries to construct a counterexample encompassing a trace that falsifies the security properties.

Structure of a PROVERIF model

We explain the three different parts of a PROVERIF input model using the *Needham-Schroeder Public Key* security protocol [Low96b].

The first part of a PROVERIF input model defines terms, e.g., functions and types, used within the security protocol. Functions, denoted by the keywords fun and reduc, specify cryptographic primitives. Since all cryptographic primitives are treated as black boxes, functions only specify the signature, not the behavior.

For example, Listing 2.2 depicts the types and functions necessary to define asymmetric encryption and decryption. Lines 1–2 define the two types privateKey and publicKey. The function genPubKey takes a privateKey as input and returns the publicKey (cf. line 4). The function fun aEnc specifies asymmetric encryption and takes as input an argument of type bitstring and an argument of type publicKey and returns a bitstring (cf. line 6). The function reduc aDec specifies the asymmetric decryption, i.e., aDec(aEnc(m, genPubKey(sk)), sk) is reduced to m (cf. lines 7–8).

```
1  type privateKey.
2  type publicKey.
3
4  fun genPubKey(privateKey): publicKey.
5
6  fun aEnc(bitstring, publicKey): bitstring.
7  reduc forall m: bitstring, k: privateKey;
8    aDec(aEnc(m, genPubKey(k)), k) = m.
```

Listing 2.2: Exemplary PROVERIF input model defining functions and types for asymmetric encryption

Variables describe communication channels (e.g., free c: channel in line 1 of Listing 2.3) or other terms shared by the participants (e.g., free A: host in line 4 of Listing 2.3). Variables are typed either by means of predefined types (e.g., channel in line 1 of Listing 2.3) or by means of user-defined types (e.g., host in line 3–4 of Listing 2.3). Functions and variables are, by default, public, and thus accessible by the attacker. If this is not intended, they can be declared as private (cf. line 7 of Listing 2.3).

```
1  free c: channel.
2  type host.
3
4  free A: host.
5  free B: host.
6
7  free secret: bitstring [private].
```

Listing 2.3: Exemplary PROVERIF input model defining channels and variables

The second part of a PROVERIF input model defines the behavior of participants by so-called sub-processes (denoted by the keyword let). A sub-process may encompass input parameters. An input parameter has a name and is typed. For example, the process depicted in Listing 2.4 encompasses the two input parameters aPrivateKey: privateKey and aPublicKey: publicKey. The behavior described within a sub-process encompasses the declaration of variables (e.g., new nonce: bitstring in line 13), the sending and receiving of messages over a communication channel (e.g., out(c, (A, B)); in line 5 and in(c, msg: bitstring); in line 7) as well as the conditional execution of a sub-process.

```
let processA(
  aPrivateKey: privateKey, aPublicKey: publicKey
)=
  out(c, (A, B));

  in(c, message: bitstring);

  let( bPublicKey: publicKey, =B)
    = aDec(message, aPrivateKey) in

  new nonce: bitstring;
  out(c, aEnc((A, B), bPublicKey));
.
```

Listing 2.4: Exemplary PROVERIF input model defining a sub-process

Finally, the third part of a PROVERIF input model defines the main process, denoted by the keyword process. The main process is the entry point of the security protocol. It can reference any sub-process by calling the name of the sub-process. Listing 2.5 depicts an excerpt of the main process of the *Needham-Schroeder Public Key* protocol. In lines 9–11, two sub-processes are instantiated to be run in parallel (denoted by |) in an unbounded number of sessions (denoted by !).

```
1  process
2    new aPrivateKey: privateKey;
3    let aPublicKey = genPubKey(aPrivateKey) in
4
5    new bPrivateKey: privateKey;
6    let bPublicKey = genPubKey(bPrivateKey) in
7
8    (
9      ( !processA(aPrivateKey, aPublicKey) )
10      |
11     ( !processB(bPrivateKey, bPublicKey) )
12   )
```

Listing 2.5: Exemplary PROVERIF input model defining a main process

Security properties

This section introduces the specification of secrecy and authentication queries.

Secrecy PROVERIF can prove reachability properties, and thus allows the investigation of which terms are kept secret and which are available to an attacker during the execution of a security protocol. To analyze the secrecy of a term M, a query of the form *query attack*(M) is included in the PROVERIF input model [Bla16].

Authentication ProVerif can prove authentication properties based on correspondence assertions. As mentioned in Section 2.2.2, a correspondence assertion captures the relationship between events that occur in the execution of the security protocol. It can be expressed as "if an event e_1 has been executed, then the event e_2 has been previously executed" [Bla16, p. 19].

In ProVerif, sub-processes and the main process can be annotated with events to mark important steps in the execution of the security protocol. These events must be related to each other using a query of the form query: event $e_1()$ ==> event $e_2()$ [Bla16]. Listing 2.6 depicts an exemplary authentication query. Lines 1–2 declare the events used in the query. In ProVerif, the events may contain several parameters. Line 4 specifies the query defining the correspondence assertion. Finally, line 12 annotates the sub-process A with the event $e_1()$.

```
1   event e1(host, host).
2   event e2(host, host).
3
4   query x: host, y: host; event(e1(x,y)) ==> event(e2(y,x)).
5
6   free host A.
7   free host B.
8
9   let processA(
10    aPrivateKey: privateKey, aPublicKey: publicKey
11  )=
12    event e1(A,B);
13  .
```

Listing 2.6: Exemplary ProVerif input model defining an authentication query

2.2.4 Tamarin

Tamarin is a model checker for the verification of security protocols presented by Basin et al. [BCD$^+$17; Tamarin] that can prove security properties such as secrecy and authentication in the symbolic model.

Tamarin takes as input a plain text model of a security protocol, specifying the actions taken by participants (typically called agents in Tamarin) running the protocol and the desired security properties. This model is then automatically translated into an internal presentation to execute the analysis and to verify whether the desired security properties hold. Tamarin provides two ways of constructing proofs. Tamarin has a fully automated mode. "If the tool's automated proof search terminates, it returns either a proof of correctness (for an unbounded number of role instances and fresh values) or a counterexample, representing an attack that violates the stated property. However, since the correctness of security protocols is an undecidable problem, the tool may not terminate on a given verification problem. Hence, users may need to resort to Tamarin's interactive mode to explore the proof states, inspect attack graphs, and seamlessly combine manual proof guidance with automated proof search" [Tamarin].

Structure of a Tamarin model

We explain the different parts of a Tamarin input model using the *Needham-Schroeder Public Key* security protocol [Low96b]. The Tamarin input model starts with the keyword theory followed by the name of the theory, e.g., Needham-Schroeder-Public-Key for the running example depicted in Listing 2.7.

Tamarin provides a set of built-in functions encompassing the most common functions needed to model security protocols, e.g., encryption. The first part of a Tamarin input model imports the built-in functions the security protocol uses. For example, in Listing 2.7, the built-in functions for digital signatures and asymmetric encryption are used (cf. builtins: digital-signature, asymmetric-encryption). In particular, the built-ins for the asymmetric encryption define a binary function aenc denoting the asymmetric encryption, a binary function adec denoting the asymmetric encryption, and a unary function pk denoting the public key corresponding to a private key. “Moreover, the built-in also specifies that the decryption of a ciphertext using the correct private key returns the initial plain text, i.e., adec(aenc(m, pk(sk)), sk) is reduced to m” [Tamarin].

```
theory Needham-Schroeder-Public-Key
begin
   builtins: asymmetric-encryption, signing
end
```

Listing 2.7: Exemplary Tamarin input model defining a theory and importing build-ins

Tamarin uses multiset rewriting rules to specify the behavior of the participants of a security protocol. These rules operate on the system’s state which is expressed as a multiset of facts. As depicted in Listing 2.8, a rewriting rule has a name and encompasses three parts: the left-hand side, the labeling of the transition, and the right-hand side. Labels are used to define certain facts about the state. “A rule can be applied to a state if it can be instantiated such that the left-hand side is contained in the current state. If this is the case, the left-hand side facts are removed from the state and replaced by the right-hand side” [Tamarin].

```
rule Name:
  let m = aenc(~ni)pkR in
  [
    Fr(~ni), !Pk($R, pkR),
  ]
  --[]->
  [
    Out(m), State($I, $R, m)
  ]
```

Listing 2.8: Exemplary Tamarin input model defining a multiset rewriting rule

The left-hand side of the rule, depicted in Listing 2.8, defines that a fresh value n is generated (cf. Fr(∼n)). The fact Fr() can only occur on the left-hand side of the rewriting rule and generates a fresh random value. The ∼ indicates the freshness of a value. Moreover, the left-hand side of the rule specifies the public key pkR for the agent R (cf. !Pk($R, pkR)). The ! denotes that the fact Pk is persistent, i.e., the fact is never removed from the state.

The right-hand side of the rule defines that the message m is sent to the untrusted network (cf. Out(m)). In line 2, the message m has been defined by a let binding as the ciphertext of the asymmetric encryption of n with key pkR. In general, a let-binding can be used to specify local terms within the context of a rule. Finally, the right-hand side of the rule defines the state fact State($I, $R, m) to indicate the progress of a protocol. A state fact is used on the left-hand side to model a precondition that must hold for the rule to fire and on the right-hand side to express the postcondition after the rule has fired.

Security properties

Tamarin uses trace properties to specify security properties. A trace property is given as a guarded first-order logic formula over action facts and timepoints. In the following, we present the basic concepts for verifying secrecy and authentication.

Secrecy In Tamarin, the action fact Secret(x) is used to indicate that the message x is supposed to be secret. Additionally, the action fact Honest(B) is used to indicate that the agent B is assumed to be honest, i.e., the keys of the agent B have not been compromised. Based on these two action facts, we can specify a trace property to verify the secrecy as shown in Listing 2.9. "The lemma states that whenever a secret action fact Secret(x) occurs at timepoint i, the adversary does not know x or an agent claimed to be honest at timepoint i has been compromised at a timepoint r" [Tamarin].

```
lemma secrecy:
  "All x #i. Secret(x) @i ==>
    not (Ex #j. K(x)@j) | (Ex B #r. Reveal(B)@r & Honest(B)@i)"
```

Listing 2.9: Exemplary Tamarin input model defining a secrecy lemma

Authentication Tamarin can prove authentication properties based on correspondence assertions. As mentioned in Section 2.2.2, a correspondence assertion captures the relationship between events that occur in the execution of the security protocol. So, "if an agent A believes that a message m was sent by an agent B, then m was indeed sent by B. To specify A's belief, [...] an appropriate rule in A's role specification [is labeled] with Authentic(b, m)" [Tamarin]. In addition, the action fact Send(b, m) is used in an appropriate rule in B's role specification to indicate that B has sent the message m. Based on these two action facts, we can specify a trace property to verify the authentication as shown in Listing 2.10. The lemma states that whenever an action fact Authentic(b, m) occurs at timepoint j, an action fact Send(b, m) occurred before.

```
lemma authentication:
  "All b m #j. Authentic(b,m) @j ==> Ex #i. Send(b,m) @i &i<j"
```

Listing 2.10: Exemplary Tamarin input model defining an authentication lemma

3

Specification of Security MSDs

This chapter introduces our UML-compliant modeling language called Security Modeling Profile as an extension to Modal Sequence Diagrams (MSDs). In particular, the modeling language provides a set of symbolic cryptographic primitives like (a)symmetric encryption, digital signatures, and cryptographic hash functions. Additionally, the modeling language provides elements to specify the conditional behavior of a security protocol. Thereby, our Security Modeling Profile enables security engineers to specify security protocols intuitively by applying scenarios for message-based interactions. Moreover, requirements engineers can use the Security Modeling Profile to specify and simulatively validate security requirements on the communication behavior of software-intensive systems. Existing modeling languages enabling the specification of security aspects focus either on high-level security requirements but not on security protocols [Jür02; LBD02], need to be more appropriate for the intuitive specification of requirements on the message-based communication [ALA19], or lack generalization [FLH+16].

This chapter is structured as follows: We provide a list of our contributions in Section 3.1. Afterward, we collect and explain the requirements on the Security Modeling Profile in Section 3.2. Subsequently, we apply the profile in Section 3.3 and present the concepts of our Security Modeling Profile in Section 3.4. Next, we present necessary extensions to the runtime semantics of MSDs to handle the concepts of our Security Modeling Profile in Section 3.5. Then, we provide information about the implementation in Section 3.6 and evaluate the Security Modeling Profile by means of a case study in Section 3.7. We investigate related work in Section 3.8. Finally, we summarize this chapter in Section 3.9.

We published contents of this chapter in two papers ([*KDH+20; *Koc18]). In addition, parts of this chapter have been contributed by the master's thesis of Jazayeri [+Jaz15] and the bachelor's theses of Kaiser [+Kai20] and Trippel [+Tri21]. Jazayeri investigated the feasibility of specifying security protocols by means of MSDs but did not formalize modeling elements in a modeling language. Kaiser collected an initial list of requirements on the Security Modeling Profile and made some initial contributions toward the modeling language. Trippel developed an initial concept for the extension of the Play-out algorithm to handle the modeling elements of the Security Modeling Profile correctly.

3.1 Contributions

The contributions of this chapter can be summarized as follows:

- We conduct a literature study and collect the description of 54 security protocols from academia and industry. For each security protocol, we analyzed which building blocks are used to specify it.
- We derive 19 requirements based on the identified building blocks that our modeling language should satisfy.
- We define a UML profile called SECURITY MODELING PROFILE, which extends UML Interactions and Modal Sequence Diagrams for specifying security protocols. This profile enables the utilization of symbolic cryptographic primitives (e.g., symmetric and asymmetric encryption or digital signature) on messages and the specification of conditional behavior. Thus, the SECURITY MODELING PROFILE fulfills the 19 requirements.
- We extend the Play-out algorithm to enable the simulation of security protocols specified by means of the newly created SECURITY MODELING PROFILE.
- We implement a prototype based on SCENARIOTOOLS MSD and show in an evaluation that the SECURITY MODELING PROFILE is applicable in practice.

3.2 Requirements on the SECURITY MODELING PROFILE

In this section, we identify typical building blocks used to specify security protocols and derive requirements that our SECURITY MODELING PROFILE shall satisfy. Unfortunately, none of the existing related works (e.g., [CDL06; Bla01; MSC^{+}13; SLF^{+}14; FLH^{+}16; ALA19; LLA^{+}16]) provided a list of typical building blocks suited for annotating message-based communication with security-related aspects. Hence, we conduct a literature study to answer the research question *Which typical building blocks are used to specify security protocols?*.

We collect the description of 54 security protocols to answer our research question. The primary resource for our study is the *Security Protocol Open Repository (SPORE)* [CJ02], encompassing the description of 49 security protocols. Apart from SPORE, we used additional sources for our study. For example, we investigated the evaluation examples of the two model checkers PROVERIF [Bla01; Bla16] and TAMARIN [Mei13; DHR^{+}18]. Finally, we searched for commonly used security protocols (e.g., IPSec [SS11], Kerberos [BM90; NT94; Low00], TLS [Res18]) and added them to our data collection. The complete list of security protocols is presented in Appendix A.

The selected protocols have in common that they have a fixed number of participants. Furthermore, the selected security protocols rely on unicast-based communication, i.e., a sender only sends a message to exactly one recipient. This is not a limitation for the elicitation of requirements, as the cryptographic primitives used are independent of the underlying communication model.

For each security protocol from our data collection, we investigate the following questions:

- Which building blocks are used to specify the structure of the security protocol?
  - Who are the participants of the security protocol?
  - Which kinds of participants exist?
  - Which properties do participants possess?
  - Which properties do participants know?
- Which building blocks are used to specify the behavior of the security protocol?
  - Which cryptographic primitives are used during the execution?
  - Which variables are created and modified during the execution?
  - Which conditions are used during the execution?

Next, Section 3.2.1 shows how we analyzed the 54 security protocols using two examples. Section 3.2.2 presents the analysis results and concludes with a summary of the requirements that the SECURITY MODELING PROFILE has to satisfy.

3.2.1 Analyzing Exemplary Security Protocols

In this section, we sketch the analysis of security protocols using the two security protocols *Andrew Secure RPC* and *Needham-Schroeder Public Key* as examples. The two security protocols use different cryptographic primitives and vary in the number of participants. Thus, they represent different types of security protocols and are well-suited to be used as running examples in this chapter. For both security protocols, we structure the analysis using the questions mentioned at the end of the last section.

Andrew Secure RPC

The security protocol *Andrew Secure RPC* [Sat89] aims to exchange a new symmetric encryption key between the two participants Alice and Bob. The communication between Alice and Bob is symmetrically encrypted using a pre-shared key. Different versions of the security protocol exist. Listing 3.1 depicts the original version of the *Andrew Secure RPC* [Sat89] protocol, which is vulnerable to replay attacks.

The security protocol *Andrew Secure RPC* encompasses the two participants Alice and Bob (cf. line 4). Both participants own the symmetric encryption key K_{ab} (cf. lines 12–13). At the beginning of the execution, Alice creates a nonce N_a, symmetrically encrypts it with the key K_{ab}, and sends the ciphertext with its identifier Alice to Bob (cf. line 15). Next, Bob decrypts the message, increments the received nonce N_a by 1, and creates a new nonce N_b. Bob symmetrically encrypts both nonces with K_{ab} and sends this message to Alice (cf. line 16). Alice decrypts the message and validates that the received nonce $succ(N_a)$ equals the incremented nonce N_a. If this is the case, the execution of the security protocol continues, and Alice increments the nonce N_b by 1, symmetrically encrypts it with K_{ab}, and sends this message to Bob (cf. line 17). Finally, Bob decrypts the message and

```
1  Protocol: Andrew Secure RPC
2  Types:
3    Participant:
4      Alice, Bob
5    SymmetricKey:
6      Kab, K'ab
7    Nonce:
8      Na, Nb, N'b
9    Function:
10     succ: Nonce → Nonce
11 Knowledge:
12   Alice: Alice, Bob, Kab
13   Bob: Bob, Kab
14 Actions:
15   1. Alice → Bob: Alice, {Na}Kab
16   2. Bob → Alice: {succ(Na), Nb}Kab
17   3. Alice → Bob: {succ(Nb)}Kab
18   4. Bob → Alice: {K'ab, N'b}Kab
```

Listing 3.1: Alice & Bob notation of the *Andrew Secure RPC* security protocol

validates that the received nonce equals the incremented nonce N_b. If this is the case, Bob creates a new symmetric encryption key K'_{ab} and a new nonce N'_b. Both are symmetrically encrypted with K_{ab} and sent to Alice (cf. line 18). At the end of the security protocol, Alice and Bob can use the new symmetric encryption key K'_{ab} for further communication.

As shown in Listing 3.1, the *Andrew Secure RPC* security protocol relies on different building blocks. First, the security protocol uses symmetric encryption. This includes symmetric encryption keys that are exchanged before the execution and created during the execution of the protocol. Symmetric encryption is applied to some or all parameters of a message. Second, the *Andrew Secure RPC* security protocol relies on cryptographic nonces that are created, modified, and exchanged during the execution of the protocol. Finally, the comparison of nonces is used to describe the conditional behavior of the protocol.

Based on the analysis of the security protocol *Andrew Secure RPC*, we elicit the following requirements on the Security Modeling Profile:

The Security Modeling Profile must . . .

- . . . enable the specification of participants that exchange messages to execute the protocol.
- . . . enable the identification and reference of a participant with an identifier.
- . . . enable the specification of symmetric encryption keys for each participant.
- . . . enable the specification of symmetrically encrypted messages and parameters.
- . . . enable the creation of symmetric encryption keys during the execution of the security protocol.
- . . . enable the creation of nonces during the execution of the security protocol.

- . . . enable the modification of nonces (e.g., increment) during the execution of the security protocol.
- . . . enable the comparison of nonces (e.g., equality) during the execution of the security protocol.

Needham-Schroeder Public Key

The goal of the security protocol *Needham-Schroeder Public Key* [NS78] is the mutual authentication of the two participants Alice and Bob using asymmetric encryption and a trusted third party called Sue. Different versions of the security protocol exist. Listing 3.2 depicts Lowe's version of the protocol [Low95].

```
1  Protocol: Needham-Schroder Public Key
2  Types:
3    Participant:
4      Alice, Bob, Sue
5    Nonce:
6      Na, Nb
7    PrivateKey:
8      KSa, KSb, KSs
9    PublicKey:
10     KPa, KPb, KPs
11   KeyPair:
12     (KSa, KPa), (KSb, KPb), (KSs, KPs)
13 Knowledge:
14   Alice: Alice, Sue, (KSa, KPa), KPs
15   Bob: Bob, Sue, (KSb, KPb), KPs
16   Sue: Alice, Bob, (KSs, KPs), KPa, KPb
17 Actions:
18   1. Alice → Sue: Alice, Bob
19   2. Sue → Alice: {KPb, Bob}KSs
20   3. Alice → Bob: {Na, Alice}KPb
21   4. Bob → Sue: Bob, Alice
22   5. Sue → Bob: {KPa, Alice}KSs
23   6. Bob → Alice: {Na, Nb, Bob}KPa
24   7. Alice → Bob: {Nb}KPb
```

Listing 3.2: Alice & Bob notation of the *Needham-Schroder Public Key* security protocol

The security protocol *Needham-Schroeder Public Key* encompasses the two participants Alice and Bob, and the trusted third party Sue (cf. line 4). All participants possess a cryptographic key pair KP_x and KS_x, where KS_x denotes the private key of participant x and KP_x the corresponding public key (cf. lines 7–12). Moreover, Alice and Bob know the public key KP_s of Sue (cf. lines 14–15) and Sue knows the public keys KP_a of Alice and KP_b of Bob (cf. line 16). At the beginning of the execution, Alice sends a message to Sue containing the two identifiers Alice and Bob (cf. line 18). Next, Sue looks up the key for the identifier Bob and sends a message to Alice containing the identifier Bob and his public key KP_b. The message is secured by a digital signature created with the private key KS_s (cf. line 19). Alice validates the signature with the public key KP_s. If the validation is successful, the execution continues and Alice creates a nonce N_a, encrypts the nonce and its identifier with the public key KP_b, and sends the message to Bob (cf. line 20). Bob decrypts the message and requests the public key for the identifier Alice from Sue by sending a message containing the two identifiers Bob and Alice (cf. line 21). Next, Sue looks up the key for the identifier Alice and sends a message to Bob containing the identifier Alice and her public key PK_a. As before, the message is secured by a digital signature created with the private key KS_s (cf. line 22). After successfully validating the signature, Bob creates a message encompassing the nonces N_a, a newly generated nonce N_b and his identifier Bob. The message is encrypted with the received public key KP_a and sent to Alice (cf. line 23). Finally, Alice checks whether the received nonce N_a equals the nonce previously sent to Bob. If this is the case, Alice sends the nonce N_b encrypted with KP_b back to Bob (cf. line 24). At the end of the security protocol, Alice and Bob are mutually authenticated.

As shown in Listing 3.2, the *Needham-Schroeder Public Key* security protocol relies on different building blocks. First, the security protocol uses asymmetric encryption and digital signatures. This includes corresponding cryptographic key pairs that are exchanged before the execution of the protocol. Asymmetric encryption and digital signatures are applied to some or all parameters of a message. Second, the *Needham-Schroeder Public Key* security protocol relies on cryptographic nonces that are created, modified, and exchanged during the execution of the protocol. Finally, the comparison of nonces is used to describe the conditional behavior of the protocol.

Based on the analysis of the security protocol *Needham-Schroeder Public Key*, we elicit the following requirements on the Security Modeling Profile:

The Security Modeling Profile must …

- … enable the specification of participants that exchange messages to execute the protocol.
- … enable the identification and reference of a participant with an identifier.
- … enable the distinction between participants of the protocol and specialized participants like trusted third parties.
- … enable the specification of properties like cryptographic keys owned by each participant.
- … enable the specification of properties like cryptographic keys known to individual participants.
- … provide an element to search the known properties for a specific element.
- … enable the specification of asymmetric encrypted parameters and messages.
- … enable the specification of digitally signed parameters and messages.
- … enable the creation of nonces during the execution of the security protocol.
- … enable the comparison of nonces (e.g., equality) during the execution of the security protocol.

3.2.2 Analyzing the Results

We analyzed 54 security protocols to identify typical building blocks. First, we present the analysis results grouped into five categories: *participants*, *data types*, *cryptographic primitives*, *assignments*, and *conditions*. Second, based on the analysis results, we derive 19 requirements that the SECURITY MODELING PROFILE shall satisfy. Appendix A provides supplementing materials for the analysis.

Participants

Participants execute a security protocol to achieve the purpose intended by the security protocol. Therefore, they might possess and know some properties before executing the protocol. Typically, these properties relate to cryptographic keys. In addition, the participants exchange messages with each other. Most security protocols encompass only two participants. The participant sending the first message is called the *initiator*, while the other is called the *responder*. However, some security protocols contain more than these two participants. For example, in the *Needham-Schroeder Public Key* security protocol, the third participant acts as a trusted third party providing some core security functionalities. Moreover, some security protocols include more than one responder.

To fully support the identified building blocks, the SECURITY MODELING PROFILE should satisfy the following requirements:

The SECURITY MODELING PROFILE must ...

(Req-1) ... enable the specification of participants that exchange messages to execute the protocol.

(Req-2) ... enable the identification and reference of a participant with an identifier.

(Req-3) ... enable the distinction between participants of the protocol and more specialized participants like trusted third parties.

(Req-4) ... enable the specification of properties like cryptographic keys owned by each participant.

(Req-5) ... enable the specification of properties like cryptographic keys known to individual participants.

Data Types

We identified twelve data types (cf. Table A.3). Six of these data types refer to cryptographic keys used in the cryptographic operations (a)symmetric encryption, digital signatures, and hash-based message authentication codes. Moreover, in the execution of a security protocol, the participants create random numbers or nonces. In addition, they use timestamps to share information about the current time or the lifetime of certain information. Finally, the participants may exchange arbitrary data.

However, Strings and Numbers usually represent most of the mentioned data types. Thus, our profile does not need to provide new data types; instead, it can use the existing primitive data types but must provide operations that initialize variables with the specialized form of these primitive data types.

To fully support the identified building blocks, the SECURITY MODELING PROFILE should satisfy the following requirements:

The SECURITY MODELING PROFILE must ...

(Req-6) ... enable the creation of cryptographic keys used in the cryptographic operations (a)symmetric encryption, digital signatures, and hash-based message authentication codes.

(Req-7) ... enable the creation of random numbers and nonces.

(Req-8) ... enable the creation of timestamps.

Cryptographic Primitives

In a security protocol, participants exchange information by sending/receiving messages. The messages contain one or more parameters and are either transmitted in plain text or protected by cryptographic primitives. As shown in Table A.1, five different cryptographic primitives are used to secure the communication: asymmetric encryption, symmetric encryption, digital signatures, message authentication codes (MACs), and hashing.

When using cryptographic primitives to protect the communication, several cases must be considered: First, all parameters of the message are protected by the same number of cryptographic primitives (e.g., encryption, or encryption and digital signature) and the receiver of the message has the appropriate keys for all used primitives. Second, some parameters of the message are protected by none, one, or more cryptographic primitives (e.g., some are sent in plain text, while others are encrypted) and the receiver of the message has the appropriate keys for all used primitives. Finally, some parameters of the message are protected by an arbitrary number of cryptographic primitives and the receiver does not have the appropriate keys.

To fully support the identified building blocks, the SECURITY MODELING PROFILE should satisfy the following requirements:

The SECURITY MODELING PROFILE must ...

(Req-9) ... enable the specification of parameters and messages secured by (a)symmetric encryption.

(Req-10) ... enable the specification of parameters and messages secured by a digital signature.

(Req-11) ... enable the specification of parameters and messages secured by a hash-based message authentication code.

(Req-12) ... enable the specification of hashed parameters and messages.

Assignments

In the description of security protocols, assignments are usually not represented. Instead, they are implicitly assumed. When executing security protocols, assignments are used to initialize new variables or modify existing variables. Variables can have any of the previously mentioned data types. In addition, many operations can be used in assignments (cf. Table A.2), for example, mathematical operations (addition, subtraction, multiplication, and modular exponentiation). Often, addition and subtraction only increase or decrease the value by one. Moreover, as explained in the previous paragraph, in some security protocols, cryptographic primitives are applied to variables and the receiver does not have the appropriate keys. Thus, these parts of the message must be created outside the message by means of assignments.

To fully support the identified building blocks, the SECURITY MODELING PROFILE should satisfy the following requirements:

The SECURITY MODELING PROFILE must . . .

(Req-13) . . . provide an operation to create an instance of a datatype and to assign this instance to a variable.

(Req-14) . . . enable the modification of data types (e.g., increment nonces) during the execution of the security protocol.

(Req-15) . . . enable the specification of variables secured by (a)symmetric encryption.

(Req-16) . . . enable the specification of variables secured by a digital signature.

(Req-17) . . . enable the specification of variables secured by a hash-based message authentication code.

(Req-18) . . . enable the specification of hashed variables.

Conditions

Similar to assignments, conditions are not represented in the description of the security protocol. However, when executing security protocols, conditions define conditional behavior. For example, in the *Needham-Schroeder Public Key* security protocol, there is an implicit condition validating whether the received nonce is the same as the one previously sent (cf. line 23 Listing 3.2). If the validation evaluates to true, the execution continues; otherwise, the run of the protocol is terminated. Mostly, the comparison operator "=" is used in security protocols. However, there are also rare cases where other operators are used.

To fully support the identified building blocks, the SECURITY MODELING PROFILE should satisfy the following requirement:

(Req-19) The SECURITY MODELING PROFILE must enable the specification of conditions with the common set of comparison operators.

3.3 Exemplary Application of the SECURITY MODELING PROFILE

In this section, we apply the SECURITY MODELING PROFILE to model the security protocols *Andrew Secure RPC* (cf. Section 3.3.1) and *Needham-Schroeder Public Key* (cf. Section 3.3.2). Afterward, in Section 3.3.3, we summarize the application of the SECURITY MODELING PROFILE.

3.3.1 Modeling the *Andrew Secure RPC* Security Protocol

Figure 3.1 depicts the UML classes for the security protocol *Andrew Secure RPC*. As explained in Section 3.2.1, the purpose of the security protocol is to exchange a new symmetric encryption key between the two participants Alice and Bob.

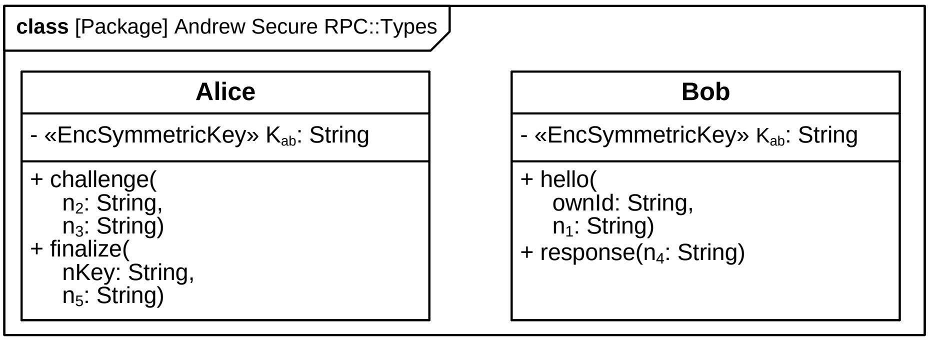

class [Package] Andrew Secure RPC::Types
Alice
- «EncSymmetricKey» K_{ab}: String
+ challenge(
n_2: String,
n_3: String)
+ finalize(
nKey: String,
n_5: String)
Bob
- «EncSymmetricKey» K_{ab}: String
+ hello(
ownId: String,
n_1: String)
+ response(n_4: String)

Figure 3.1: UML class diagram for the *Andrew Secure RPC* security protocol

We define the two classes Alice and Bob. Each class encompasses operations that are used as message signatures as part of the actual MSD. For example, the class Alice encompasses the two operations challenge(n_n: String, n_3: String) and finalize(newKey: String, n_5: String). Moreover, each class contains a property K_{ab} of type String. We apply the stereotype «EncSymmetricKey» to the property K_{ab}, indicating that the property can be used as a symmetric encryption key.

Based on the classes, the UML collaboration, depicted in Figure 3.2, specifies the two roles alice: Alice and bob: Bob typed by the UML classes Alice and Bob that participate in the *Andrew Secure RPC* security protocol. In addition, we use the dependency with the applied stereotype «sharesKeysWith» to specify that the two keys K_{ab} of alice: Alice and bob: Bob are considered to be the same.

Figure 3.3 depicts the MSD for the *Andrew Secure RPC* security protocol. The MSD encompasses the two lifelines alice: Alice and bob: Bob representing the roles defined in the UML collaboration. At the beginning of the protocol, participant alice: Alice creates a new nonce N_a. In the MSD, the nonce creation is specified by the security assignment N_a = createNonce(). Security assignments are an extension of assignments (cf. Section 2.1.1) and provide means to create and modify security-related data types like cryptographic keys or nonces. Visually, security assignments contain a small lock at the upper right corner of the rectangle. Moreover, as assignments, they have the form <var> = <expr>, where <var> is the name of a diagram or lifeline variable, and <expr> can be any expression evaluating to a value of the type of <var>. Next, alice: Alice sends the message hello() to bob: Bob. The message contains the two parameters ownId with value "alice" and n_1 with value N_a. Moreover, the parameter n_1 is symmetrically encrypted with K_{ab}.

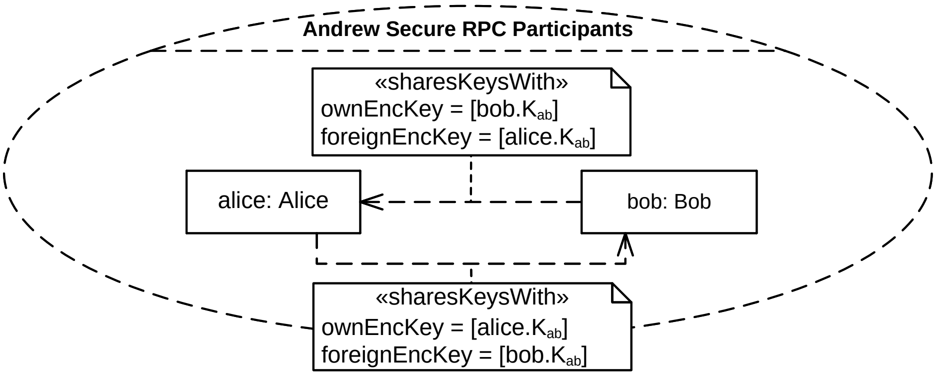


Figure 3.2: UML collaboration diagram for the *Andrew Secure RPC* security protocol

In the MSD, the symmetric encryption of the parameter is specified by the stereotype «SymmetricEncrypted» applied to the parameter. The cryptographic key needed to perform the parameter encryption is specified by the property symmetricKey of the stereotype «SymmetricEncrypted». In our example, we assign K_{ab} to the property. After receiving the message, bob: Bob increments the received nonce by one (cf. $succNa = inc(n_1)$). inc(param) is a unary operation that increments param by one. Next, bob: Bob creates a nonce N_b. Then, bob: Bob sends the message challenge to alice: Alice. This message contains the two parameters n_2 with value succNa and n_3 with value N_b. Moreover, the message is symmetrically encrypted, specified by the applied stereotype «SymmetricEncrypted». After receiving the message, alice: Alice validates whether the received nonce n_2 is equal to the incremented nonce N_a. In the MSD, the condition is specified by the security condition $inc(N_a) = n_2$. A security condition contains an expression that evaluates to a Boolean value. To visually distinguish security conditions from non-security conditions, they contain a lock at the upper right corner of the hexagon. Then, alice: Alice increments the received nonce n_3 by one (cf. $succN_b = inc(n_3)$) and sends it to bob: Bob. After receiving the message, bob: Bob validates the correctness of the received nonce (cf. $inc(N_b = n_4)$). Then, bob: Bob creates a new symmetric encryption key (cf. key = createSymmetricKey()) and a nonce N'_b (cf. N'_b = createNonce()). Finally, bob: Bob sends the symmetrically encrypted message finalize to alice: Alice containing the newly created symmetric encryption key key and nonce N'_b.

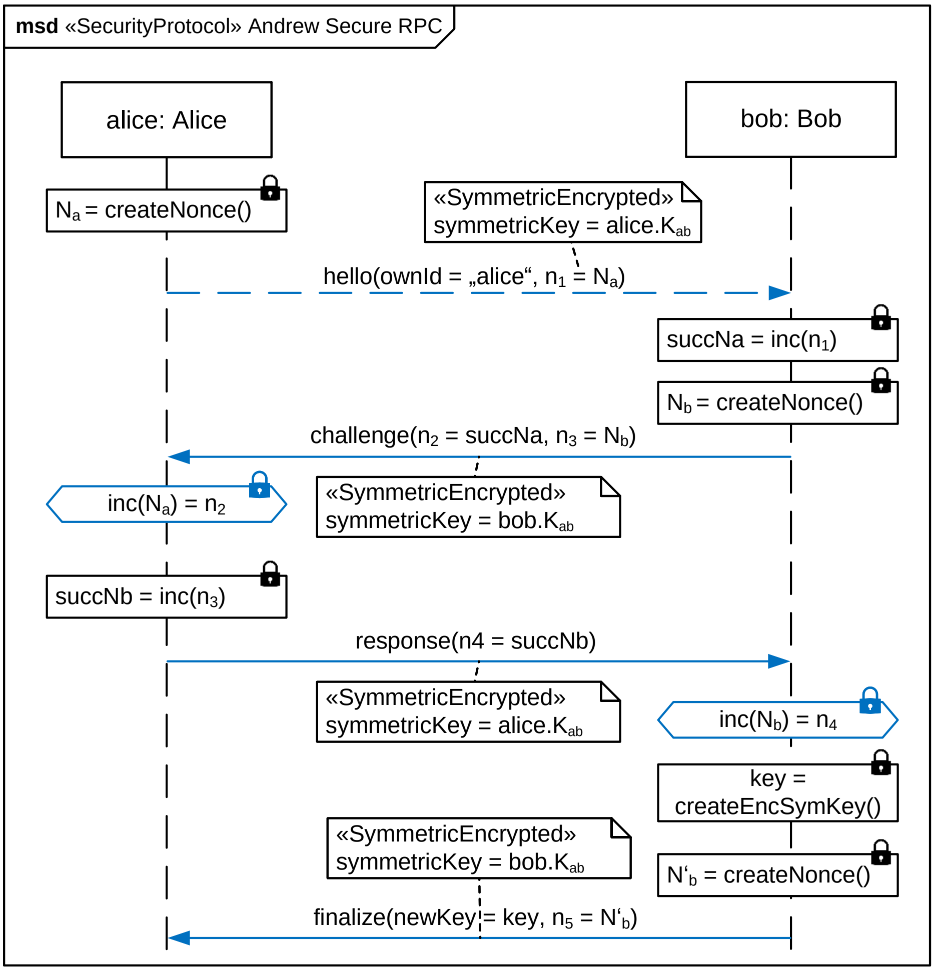


Figure 3.3: MSD for the *Andrew Secure RPC* security protocol

3.3.2 Modeling the *Needham-Schroeder Public Key* Security Protocol

Figure 3.4 depicts the UML classes for the *Needham-Schroeder Public Key* security protocol. As explained in Section 3.2.1, the purpose of the security protocol is the mutual authentication of the two participants Alice and Bob using asymmetric encryption and a trusted third party.

We define three classes Alice, Bob, and Sue. We annotate the class Sue with the stereotype «TrustedThirdParty». Moreover, each class encompasses operations used as message signatures as part of the actual MSD and a cryptographic key pair. Alice and Bob use their key pair for asymmetric encryption. Thus, both classes possess a property sKeyX of type String annotated with the stereotype «EncPrivateKey» to specify private asymmetric encryption keys (cf. sKeyA and sKeyB in Figure 3.4) and a property pKeyX of type String annotated with the stereotype «EncPublicKey» to specify public asymmetric encryption keys (cf. pKeyA and pKeyB in Figure 3.4). The association keyPair models the relationship between a key pair's private and public key. Sue uses her key pair for digital signatures. Thus, the class Sue contains a property sKeyS and pKeyS of type String annotated with the stereotypes «SigPrivateKey» and «SigPublicKey», respectively. The relationship between the two keys is specified by an association with the applied stereotype «keyPair».

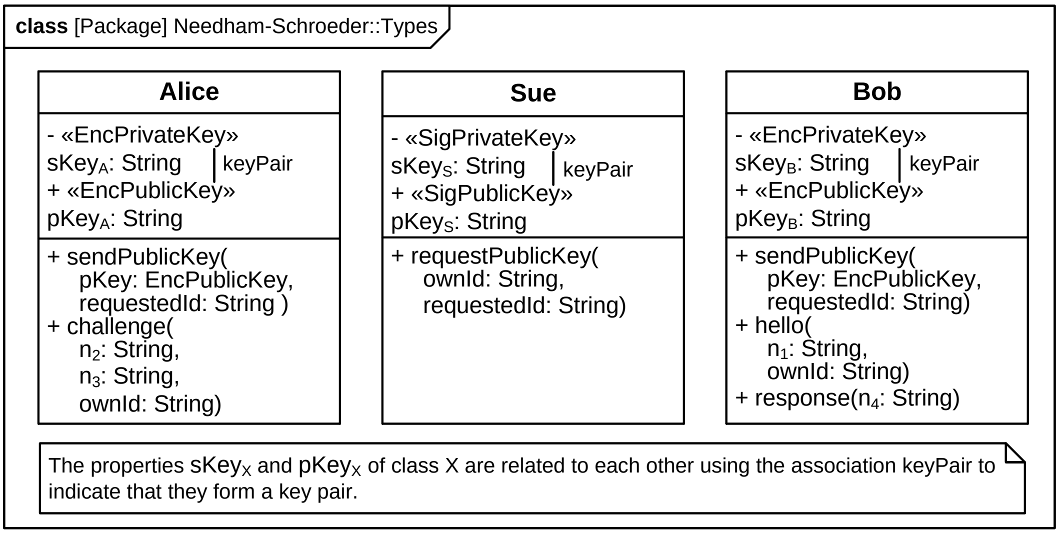

class [Package] Needham-Schroeder::Types
Alice
- «EncPrivateKey» sKey$_A$: String
keyPair
+ «EncPublicKey» pKey$_A$: String
+ sendPublicKey(pKey: EncPublicKey, requestedId: String)
+ challenge(n$_2$: String, n$_3$: String, ownId: String)
Sue
- «SigPrivateKey» sKey$_S$: String
keyPair
+ «SigPublicKey» pKey$_S$: String
+ requestPublicKey(ownId: String, requestedId: String)
Bob
- «EncPrivateKey» sKey$_B$: String
keyPair
+ «EncPublicKey» pKey$_B$: String
+ sendPublicKey(pKey: EncPublicKey, requestedId: String)
+ hello(n$_1$: String, ownId: String)
+ response(n$_4$: String)
The properties sKey$_X$ and pKey$_X$ of class X are related to each other using the association keyPair to indicate that they form a key pair.

Figure 3.4: UML class diagram for the *Needham-Schroeder Public Key* security protocol

Based on the classes, the UML collaboration, depicted in Figure 3.5, specifies the three roles alice: Alice, bob: Bob, and sue: Sue typed by the UML classes Alice, Bob, and Sue. Moreover, the UML collaboration encompasses dependencies annotated with the stereotype «knownsKeysOf» to specify initial knowledge of a particular role. For example, the trusted third party sue: Sue knows the public keys of alice: Alice and bob: Bob. Thus, there is a dependency from sue: Sue to alice: Alice and bob: Bob, respectively. Furthermore, the stereotype «knownsKeysOf» contains the list knowsPublicKeys encompassing the public key of alice: Alice and bob: Bob, respectively.

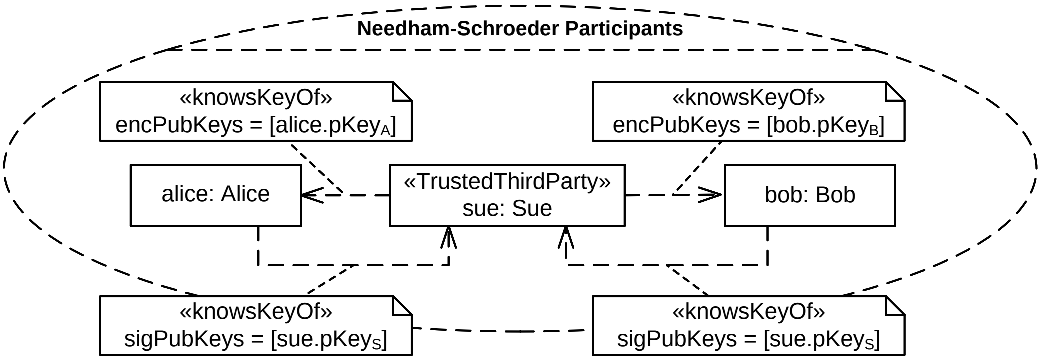

Needham-Schroeder Participants
«knowsKeyOf» encPubKeys = [alice.pKey$_A$]
«knowsKeyOf» encPubKeys = [bob.pKey$_B$]
alice: Alice
«TrustedThirdParty» sue: Sue
bob: Bob
«knowsKeyOf» sigPubKeys = [sue.pKey$_S$]
«knowsKeyOf» sigPubKeys = [sue.pKey$_S$]

Figure 3.5: UML collaboration diagram for the *Needham-Schroeder Public Key* security protocol

Figure 3.6 depicts the MSD for the *Needham-Schroeder Public Key* security protocol. The MSD encompasses the lifelines alice: Alice, bob: Bob, and sue: Sue representing the roles defined in the UML collaboration. At the beginning of the protocol, the participant alice: Alice requests the public encryption key of bob: Bob. Therefore, it sends the message requestPublicKey to sue: Sue containing the two parameters ownId with value "alice" and requestedId with value "bob". After receiving the message, sue: Sue looks up the public key for the requestedId and assigns the value to the diagram variable pkB. In the MSD, the security assignment pkB = lookUpPublicKey(requestedId) is used to specify the search for the public key based on the «knownsKeysOf» dependencies defined in the UML collaboration. Next, sue: Sue sends the public key together with the requestedId back to alice: Alice (cf. message sendPublicKey(pKey = pkB, requestedId = "bob")). The message is secured by a digital signature created with the private signature key of sue: Sue. In the MSD, the digital signature is specified by the stereotype «DigitalSigned» applied to the message. After receiving the message, alice: Alice creates a nonce $\mathsf{N_a}$ and sends this nonce together with its own identifier to bob: Bob (cf. message helloBob($\mathsf{n_1 = N_a}$, ownId = "alice")). The message is asymmetrically encrypted using the public key received from sue: Sue. Next, bob: Bob requests the public key for the received identifier from sue: Sue. After receiving the public key, bob: Bob creates a nonce $\mathsf{N_b}$ and sends an asymmetrically encrypted message containing the new created nonce $\mathsf{N_b}$, the previously received nonce $\mathsf{n_1}$ and the identifier bob (cf. message helloAlice($\mathsf{n_2 = N_a}$, $\mathsf{n_3 = n_1}$, ownId = "bob") annotated with the stereotype «AsymmetricEncrypted»). Next, alice: Alice checks whether the received nonce equals the one previously sent to bob: Bob (cf. $\mathsf{N_a = n_3}$). If that is the case, it sends back the received nonce. Finally, bob: Bob checks whether the received nonce equals the one previously sent to alice: Alice (cf. $\mathsf{N_b = n_4}$).

3.3.3 Summarizing the Exemplary Application of the Security Modeling Profile

The exemplary application of the Security Modeling Profile has shown that the profile is sufficient to specify security protocols such as the *Andrew Secure RPC* security protocol or the *Needham Schroeder Public Key* security protocol by means of MSDs. Moreover, we can derive two modeling rules based on the two exemplary applications. First, the participants in a security protocol only exchange a few messages. Thus, it is sufficient to model the complete message exchange by only one MSD. Second, the created MSDs contain only cold messages and cold conditions. It is reasonable to use only cold messages and cold conditions since a violation in a protocol run should only lead to a termination of the current run and not to an inconsistent specification.

The two exemplary applications also showed two minor restrictions. First, we modeled all participants as system objects. However, without an environment object sending messages, the security protocol cannot be simulated by means of the Play-out algorithm. To solve this issue, we could define a modeling rule that the initiator of a security protocol, i.e., the participant sending the first message, is always an environment object. Second, in the *Andrew Secure RPC* security protocol, the first event is not a message but a security assignment creating a random nonce. This contradicts the definition of the first event in an MSD as described in Section 2.1. We discuss two opportunities to solve this issue in Section 3.5.

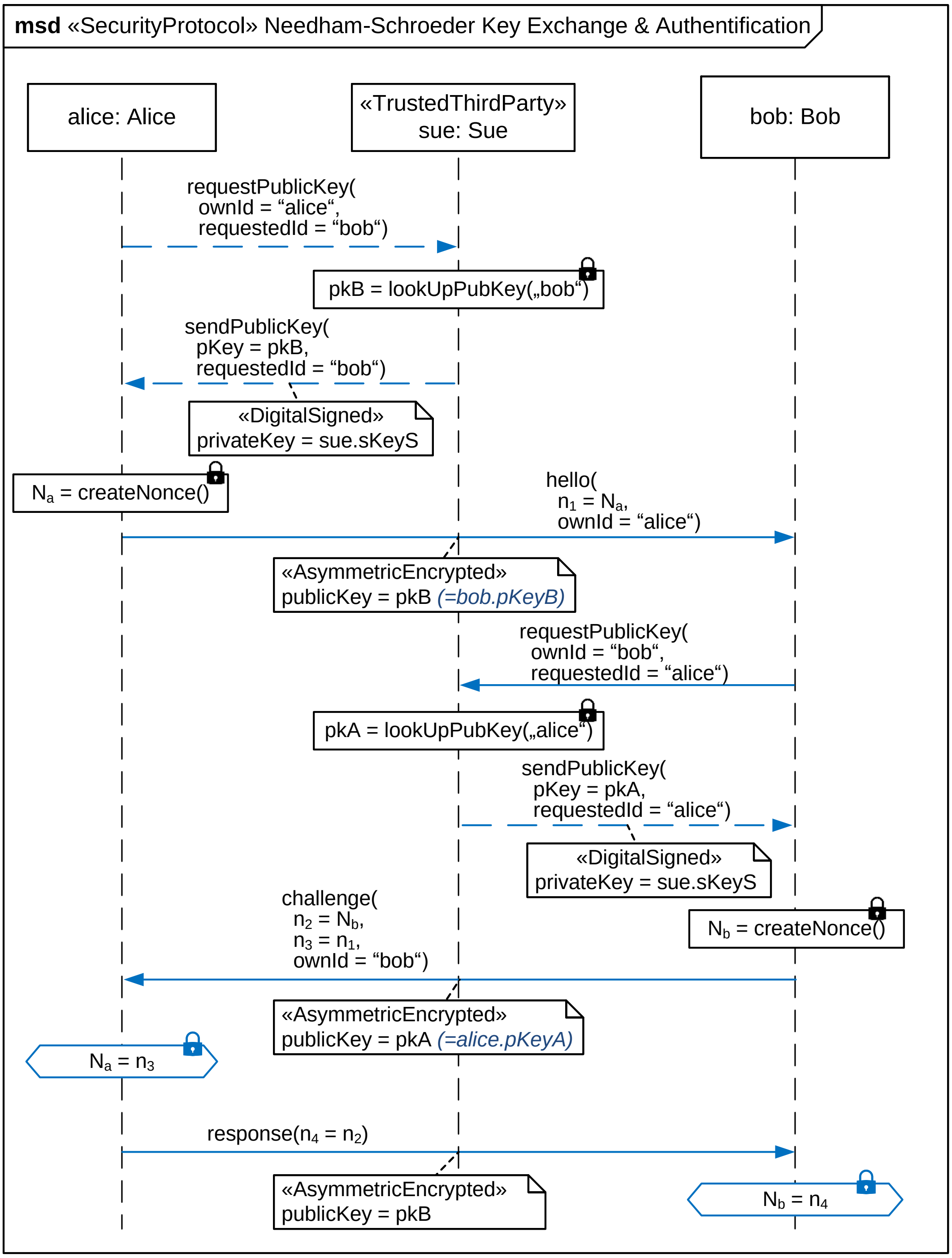

msd «SecurityProtocol» Needham-Schroeder Key Exchange & Authentification
alice: Alice
«TrustedThirdParty» sue: Sue
bob: Bob
requestPublicKey(ownId = “alice“, requestedId = “bob“)
pkB = lookUpPubKey(„bob“)
sendPublicKey(pKey = pkB, requestedId = “bob“)
«DigitalSigned» privateKey = sue.sKeyS
N_a = createNonce()
hello(n_1 = N_a, ownId = “alice“)
«AsymmetricEncrypted» publicKey = pkB (=bob.pKeyB)
requestPublicKey(ownId = “bob“, requestedId = “alice“)
pkA = lookUpPubKey(„alice“)
sendPublicKey(pKey = pkA, requestedId = “alice“)
«DigitalSigned» privateKey = sue.sKeyS
N_b = createNonce()
challenge(n_2 = N_b, n_3 = n_1, ownId = “bob“)
«AsymmetricEncrypted» publicKey = pkA (=alice.pKeyA)
N_a = n_3
response(n_4 = n_2)
«AsymmetricEncrypted» publicKey = pkB
N_b = n_4

Figure 3.6: MSD for the *Needham-Schroeder Public Key* security protocol

3.4 The SECURITY MODELING PROFILE in Detail

This section introduces our SECURITY MODELING PROFILE implementing the requirements described in Section 3.2. Figure 3.7 depicts an overview of the SECURITY MODELING PROFILE. It references the UML metamodel and imports the Modal profile for the specification of variables and references to these variables or parameter values. We divide the SECURITY MODELING PROFILE into three subprofiles: SecurityModelingProfile::ProtocolModeling, SecurityModelingProfile::CryptographicKeyModeling, and SecurityModelingProfile::SecureElementModeling. In addition, we define a domain-specific language for the specification of security assignments and security conditions, respectively. Subsequently, we describe the subprofiles and the domain-specific languages.

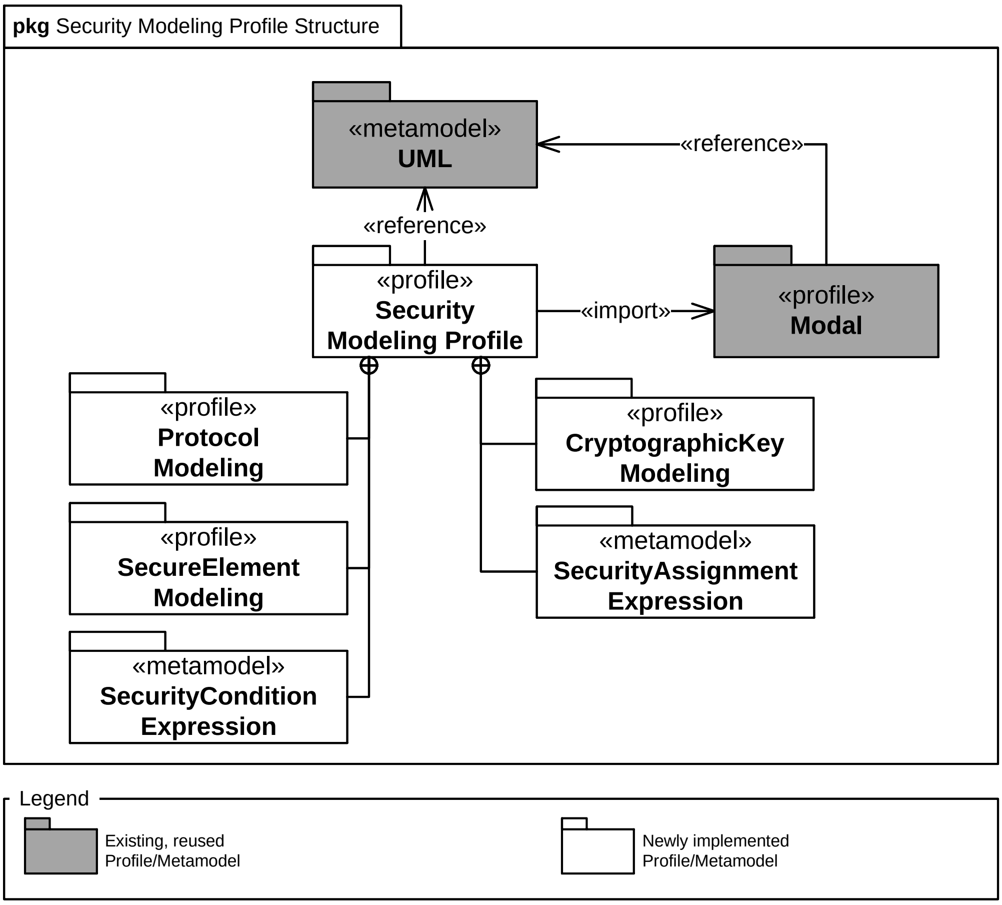

pkg Security Modeling Profile Structure
«metamodel»
UML
«reference»
«reference»
«profile»
Security Modeling Profile
«import»
«profile»
Modal
«profile»
Protocol Modeling
«profile»
CryptographicKey Modeling
«profile»
SecureElement Modeling
«metamodel»
SecurityAssignment Expression
«metamodel»
SecurityCondition Expression
Legend
Existing, reused Profile/Metamodel
Newly implemented Profile/Metamodel

Figure 3.7: Overview of the SECURITY MODELING PROFILE subprofiles

3.4.1 Subprofile SecurityModelingProfile::ProtocolModeling

The subprofile SecurityModelingProfile::ProtocolModeling is depicted in Figure 3.8 and provides means to specify security protocols. As mentioned in Section 2.1, an MSD specification is structured by means of MSD use cases where each use case encapsulates requirements on the communication behavior of the system under development.

The SECURITY MODELING PROFILE follows the structure of a pure MSD specification. However, we define the following modeling rules to enable the intuitive specification of security protocols and their simulative validation by means of the Play-out algorithm:

- An MSD specification for a security protocol describes one use case and shall contain only one MSD.
- An MSD specification for a security protocol describes the security protocol from one or more responders' perspectives. Consequently, the protocol initiator is considered part of the environment, and the responders are considered part of the system under development.
- An MSD specification for a security protocol only contains cold messages and conditions. Moreover, messages sent from environment elements are monitored, and messages sent from system elements are executed [+Jaz15].

The stereotype SecurityProtocol extending the metaclass UML::Interaction enables to distinguish between a security protocol MSD and a requirements/assumption MSD. The constraint C_1 ensures that only one stereotype is applied to the UML::Interaction.

To realize (Req-1) and (Req-2), no specific modeling elements must be created for the SECURITY MODELING PROFILE. The participants of a security protocol do not differ from participants in requirements/assumption MSDs. Both kinds may contain properties and send and receive messages with no, one or more parameters. Thus, security engineers can use the role of an MSD specification to specify the participants of a security protocol (cf. (Req-1)). Moreover, security engineers can refer to a participant by using the name of the role specified by means of the collaboration (cf. (Req-2)). For example, in the *Needham-Schroeder Public Key* security protocol depicted in Figure 3.6, the first message requestPublicKey(ownId = "alice", requestedId = "bob") refers to the roles alice and bob using their name.

However, as shown in our study (cf. Section 3.2), some security protocols encompass specialized participants like trusted third parties. To realize (Req-3), the subprofile provides the stereotype TrustedThirdParty extending the metaclass UML::Property. The stereotype TrustedThirdParty distinguishes a trusted third party from other participants. A participant annotated with the stereotype TrustedThirdparty is part of the environment. The constraint C_2 ensures this.

As described by (Req-4), the participants of a security protocol may possess properties like cryptographic keys. To realize this requirement, no specific modeling elements must be created for the SECURITY MODELING PROFILE. Security engineers can use the properties of the UML::Class to specify properties and annotate these properties with stereotypes from the subprofile SecurityModelingProfile::CryptographicKeyModeling. For example, in the *Needham-Schroeder Public Key* security protocol depicted in Figure 3.4, each class encompasses two properties used as a cryptographic key pair.

Moreover, the participants of a security protocol may have initial knowledge (e.g., knowing the properties of another participant) or share symmetric keys with other participants. For example, in the *Needham-Schroeder Public Key* security protocol, the trusted third party knows the public keys of the other participants before the execution of the protocol. To realize (Req-5),

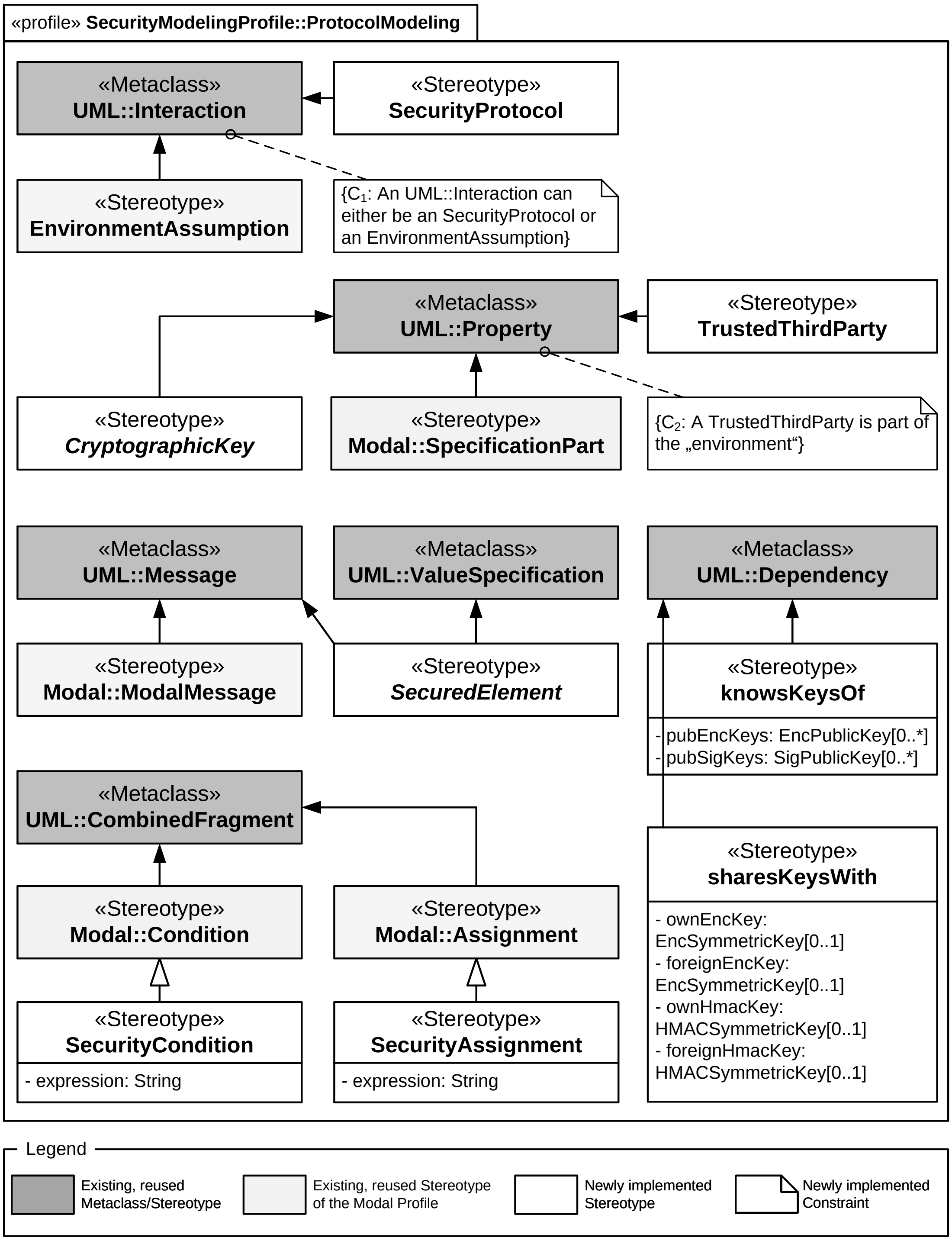

«profile» SecurityModelingProfile::ProtocolModeling
«Metaclass» UML::Interaction
«Stereotype» SecurityProtocol
«Stereotype» EnvironmentAssumption
{C1: An UML::Interaction can either be an SecurityProtocol or an EnvironmentAssumption}
«Metaclass» UML::Property
«Stereotype» TrustedThirdParty
«Stereotype» CryptographicKey
«Stereotype» Modal::SpecificationPart
{C2: A TrustedThirdParty is part of the „environment“}
«Metaclass» UML::Message
«Metaclass» UML::ValueSpecification
«Metaclass» UML::Dependency
«Stereotype» Modal::ModalMessage
«Stereotype» SecuredElement
«Stereotype» knowsKeysOf
- pubEncKeys: EncPublicKey[0..*]
- pubSigKeys: SigPublicKey[0..*]
«Metaclass» UML::CombinedFragment
«Stereotype» sharesKeysWith
- ownEncKey: EncSymmetricKey[0..1]
- foreignEncKey: EncSymmetricKey[0..1]
- ownHmacKey: HMACSymmetricKey[0..1]
- foreignHmacKey: HMACSymmetricKey[0..1]
«Stereotype» Modal::Condition
«Stereotype» Modal::Assignment
«Stereotype» SecurityCondition
- expression: String
«Stereotype» SecurityAssignment
- expression: String
Legend
Existing, reused Metaclass/Stereotype
Existing, reused Stereotype of the Modal Profile
Newly implemented Stereotype
Newly implemented Constraint

Figure 3.8: The subprofile SecurityModelingProfile::ProtocolModeling

the subprofile provides the stereotype knowsKeysOf and sharesKeysWith extending the metaclass UML::Dependency. Both stereotypes shall be used in a UML::Collaboration. The source of the stereotype knowsKeysOf specifies the role with the initial knowledge, while the target specifies the role that owns the knowledge/property. Our literature study showed that initial knowledge primarily encompasses the participants' public keys. Thus, the stereotype knowsKeysOf encompasses a list of public keys used for encryption and a list of public keys used for digital signatures, respectively. The stereotype sharesKeysWith relates the symmetric keys of participants with each other. Thus, the stereotype sharesKeysWith encompasses a property for the key of the source role (cf. ownEncKey and ownHmacKey) and a property for the key of the target role (cf. foreignEncKey and foreignHmacKey).

3.4.2 Subprofile SecurityModelingProfile::CryptographicKeyModeling

The subprofile SecurityModelingProfile::CryptographicKeyModeling is depicted in Figure 3.9 and provides means to specify cryptographic keys used in cryptographic primitives to secure the communication. Thereby, the subprofile realizes (Req-6).

The subprofile provides the abstract stereotype CryptographicKey extending the UML metaclass UML::Property and six stereotypes extending the CryptographicKey. The constraint C_3 ensures that the stereotypes are only applied to a UML::Property owned by a UML::Class and constraint C_4 that only one cryptographic key stereotype is applied to a UML::Property. Thereby, only UML::Properties of a UML::Class can be used as cryptographic keys.

Next, we provide an overview of the stereotypes extending the abstract stereotype CryptographicKey.

- The stereotype EncSymmetricKey provides means to specify a cryptographic key used for symmetric encryption. In symmetric encryption, several communication partners use the same key to encrypt and decrypt the communication. The stereotype sharesKeysWith (cf. Figure 3.8) relates the symmetric keys of participants with each other.

- The stereotype HMACSymmetricKey provides means to specify a cryptographic key used to create a hashed message-authentication code (HMAC). As for the stereotype EncSymmetricKey, the stereotype sharesKeysWith (cf. Figure 3.8) relates symmetric keys of participants with each other.

- The two stereotypes EncPrivateKey and EncPublicKey define a key pair used for asymmetric encryption. The EncPublicKey specifies the public key used for the encryption, while the EncPrivateKey specifies the private key used for the decryption. The UML association keyPair is used to relate the two parts of the key pair.

- The two stereotypes SigPrivateKey and SigPublicKey define a key pair used to create and validate a digital signature. The SigPrivateKey specifies the private key used for creation, while the SigPublicKey specifies the public key used to validate the signature. The UML association keyPair is used to relate the two parts of the key pair.

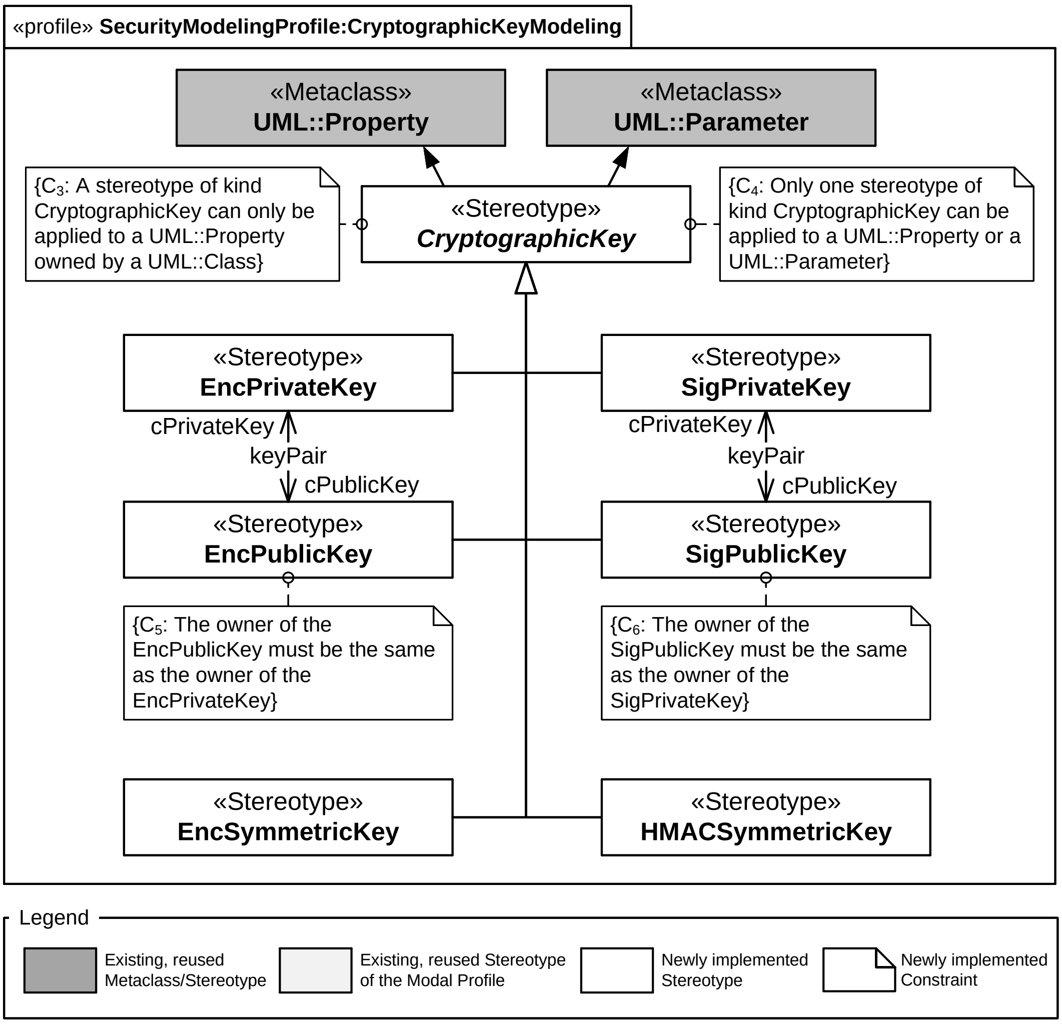

«profile» SecurityModelingProfile:CryptographicKeyModeling
«Metaclass» UML::Property
«Metaclass» UML::Parameter
{C3: A stereotype of kind CryptographicKey can only be applied to a UML::Property owned by a UML::Class}
«Stereotype» CryptographicKey
{C4: Only one stereotype of kind CryptographicKey can be applied to a UML::Property or a UML::Parameter}
«Stereotype» EncPrivateKey
«Stereotype» SigPrivateKey
cPrivateKey
keyPair
cPublicKey
cPrivateKey
keyPair
cPublicKey
«Stereotype» EncPublicKey
«Stereotype» SigPublicKey
{C5: The owner of the EncPublicKey must be the same as the owner of the EncPrivateKey}
{C6: The owner of the SigPublicKey must be the same as the owner of the SigPrivateKey}
«Stereotype» EncSymmetricKey
«Stereotype» HMACSymmetricKey
Legend
Existing, reused Metaclass/Stereotype
Existing, reused Stereotype of the Modal Profile
Newly implemented Stereotype
Newly implemented Constraint

Figure 3.9: The subprofile SecurityModelingProfile::CryptographicKey

3.4.3 Subprofile SecurityModelingProfile::SecureElementModeling

The subprofile SecurityModelingProfile::SecureElementModeling is depicted in Figure 3.10 and provides means to annotate messages and message arguments specified in an MSD with cryptographic primitives. Thereby, the subprofile realizes (Req-9)–(Req-12).

The subprofile provides the abstract stereotype SecuredElement extending the two metaclasses UML::Message and UML::ValueSpecification and five stereotypes extending the abstract stereotype.

Next, we provide an overview of the stereotypes extending the abstract stereotype SecuredElement.

- The stereotype SymmetricEncrypted describes that a message or an argument is symmetrically encrypted. Therefore, the stereotype encompasses the tagged value symmetricKey: EncSymmetricKey specifying the key used for the encryption and decryption.
- The stereotype HMAC describes that a hash-based message authentication code is added to a message and covers all or only part of the message's arguments. Therefore, the stereotype

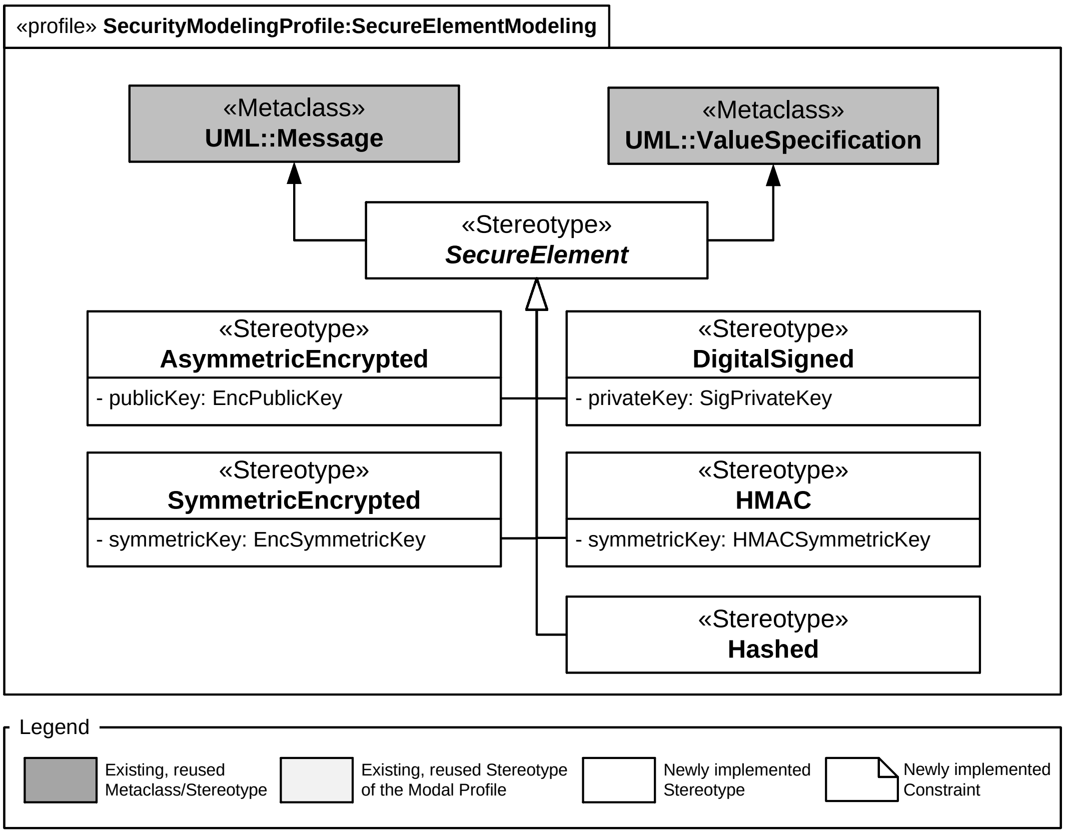

«profile» SecurityModelingProfile:SecureElementModeling
«Metaclass» UML::Message
«Metaclass» UML::ValueSpecification
«Stereotype» SecureElement
«Stereotype» AsymmetricEncrypted
- publicKey: EncPublicKey
«Stereotype» DigitalSigned
- privateKey: SigPrivateKey
«Stereotype» SymmetricEncrypted
- symmetricKey: EncSymmetricKey
«Stereotype» HMAC
- symmetricKey: HMACSymmetricKey
«Stereotype» Hashed
Legend
Existing, reused Metaclass/Stereotype
Existing, reused Stereotype of the Modal Profile
Newly implemented Stereotype
Newly implemented Constraint

Figure 3.10: The subprofile SecurityModelingProfile::SecureElementModeling

encompasses the tagged value symmetricKey: HMACSymmetricKey specifying the key used for creating and validating the hash-based message authentication code. This HMAC is added to the message and sent to the receiver.

- The stereotype AsymmetricEncrypted describes that a message or an argument is asymmetrically encrypted. The stereotype encompasses the tagged value publicKey: EncPublicKey. The tagged value publicKey specifies the EncPublicKey that is used for the encryption of the message. The privateKey specifying the EncPrivateKey used for the decryption can be resolved via the association keyPair.

- The stereotype DigitalSigned describes that a message or an argument is digitally signed. This signature is added to the message and sent to the receiver of the message. The stereotype encompasses the tagged value privateKey: SigPrivateKey. The tagged value privateKey specifies the SigPrivateKey that is used for the creation of the digital signature. The publicKey specifying the SigPublicKey used for the validation of the signature can be resolved via the association keyPair.

- The stereotype Hashed describes that a message or an argument of a message is cryptographically hashed. This hash value is added to the message and sent to the receiver.

3.4.4 Metamodel SecurityModelingProfile::SecurityAssignment

The metamodel SecurityModelingProfile::SecurityAssignment provides means to create and manipulate protocol variables and thereby realizes (Req-6)–(Req-8) and (Req-13)–(Req-18). The stereotype SecurityAssignment specializes the stereotype Modal::Assignment. To visually distinguish SecurityAssignments from non-security Modal::Assignments, we add a lock in the upper right corner of the rectangle.

As a Modal::Assignment, a SecurityAssignment has the form <var> = <expression>, where <var> can be any diagram or lifeline variable and <expression> can be any security assignment expression evaluating to a value of the type of <var>. For example, in the MSD describing the *Needham-Schroeder Public Key* security protocol, we use security assignments to generate random nonces (e.g., N_a = createNonce() in Figure 3.6) or to look up known keys (e.g., pkB = lookUpPubKey("bob") in Figure 3.6).

The domain-specific language to specify an expression is depicted in Listing 3.3 and provides means to textually describe operations for creating and modifying protocol variables.

```
SecurityAssignment = Identifier '=' Expression

Expression = AssignmentExpression | ArithmeticExpression |
    CryptographicExpression

AssignmentExpression = 'createNonce()'
      | 'createPrime()'
      | 'createTimestamp()'
      | 'createEncSymmetricKey()'
      | 'createHMACSymmetricKey()'
      | 'lookUpPublicKey(' Identifier ')'

ArithmeticExpression = ArithmeticExpression ArithmeticOp
    ArithmeticExpression
      | 'inc(' ArithmeticExpression ')'
      | 'dec(' ArithmeticExpression ')'
      | 'exp(' ArithmeticExpression,  ArithmeticExpression  ')'
      | 'mod(' ArithmeticExpression,  ArithmeticExpression  ')'
      | Literal

ArithmeticOp = "+", "-", "*", "/"

CryptographicExpression =
      sEnc((Literal (, Literal)*), [SymmetricEncKey])
      | sDec((Literal (, Literal)*), [SymmetricEncKey])
      | aEnc((Literal (, Literal)*), [PublicEncKey])
      | aDec((Literal (, Literal)*), [PrivateEncKey])
      | cHMAC((Literal (, Literal)*), [SymmetricEncKey])
      | vHMAC((Literal (, Literal)*), [SymmetricEncKey])
      | 'xor(' Literal,  Literal ')'

Literal =   Identifier | INT | DOUBLE | STRING
```

Listing 3.3: Domain-Specific Language to express Security Assignments

The description of the grammar is as follows: A SecurityAssignment has the form Identifier = Expression. The Identifier can refer to any diagram or lifeline variable. If it refers to a lifeline variable, the fully qualified name of this variable must be used. The Expression may resolve to an AssignmentExpression, an ArithmeticExpression or a CryptographicExpression. The AssignmentExpression provides rules to assign a new value of data type String or Integer to the variable (createX(), where X refers to a particular data type). For example, the term createNonce() creates a random string suitable for cryptographic applications (e.g., created with a secure random function). Moreover, the AssignmentExpression provides a rule to look up a public key for a specific lifeline. The ArithmeticExpression provides some mathematical and logical operations. First, the mathematical operations encompass expressions to express the increment and decrement of a variable by one. Second, the mathematical operations encompass expressions to express modular exponentiation. The CryptographicExpression provides cryptographic primitives like encryption. Finally, the Literal may refer to a literal for the primitive data types Integer, Double, and STRING, or to an identifier.

3.4.5 Metamodel SecurityModelingProfile::SecurityCondition

The metamodel SecurityModelingProfile::SecurityCondition provides means to compare protocol variables and, thus, describes the conditional behavior of the protocol. Thereby, the metamodel realizes (Req-19). The stereotype SecurityCondition specializes the stereotype Modal::Condition. To visually distinguish SecurityConditions from non-security Modal::Conditions, we add a lock in the upper right corner of the hexagon.

A SecurityCondition contains an expression that evaluates to a Boolean value. Thus, the semantics of a SecurityCondition equals the semantics of a Modal::Condition. If the expression evaluates to true, the MSD progresses otherwise a violation occurs. For example, in the MSD describing the *Needham-Schroeder Public Key* security protocol, we use security conditions to compare whether the received nonce is the same as the one previously sent (e.g., $N_a = n_3$ in Figure 3.6).

The domain-specific language is depicted in Listing 3.4 and provides means to textually describe the SecurityCondition.

```
SecurityCondition   = ArithmeticExpression Op ArithmeticExpression

Op = ">" | ">=" | ’=’ | "<=" | "<"

ArithmeticExpression = ArithmeticExpression ArithmeticOp
    ArithmeticExpression
        | ’inc(’ ArithmeticExpression ’)’
        | ’dec(’ ArithmeticExpression ’)’
        | ’exp(’ ArithmeticExpression,  ArithmeticExpression  ’)’
        | ’mod(’ ArithmeticExpression,  ArithmeticExpression  ’)’
        | Literal

ArithmeticOp = "+" | "-" | "*"  "/"

Literal =   INT | DOUBLE | STRING
```

Listing 3.4: Domain-Specific Language to express Security Conditions

The description of the grammar is as follows: A SecurityCondition has the form ArithmeticExpression Op ArithmeticExpression. The Op refers to the valid comparison operators. The left-hand side and the right-hand side of the condition refer to an ArithmeticExpression. The rules for the expression are the same as for the ArithmeticExpression discussed in the previous section.

3.5 Extension of the Runtime Semantics to Support the SECURITY MODELING PROFILE

This section introduces extensions to the semantics of MSDs (cf. Section 2.1.2) necessary to define the behavior of security protocols specified by means of the SECURITY MODELING PROFILE. To visualize the concepts of the extensions, we use an excerpt of the MSD for the *Andrew Secure RPC* security protocol enriched with information about the cuts used to keep track of the MSD process (cf. Figure 3.11).

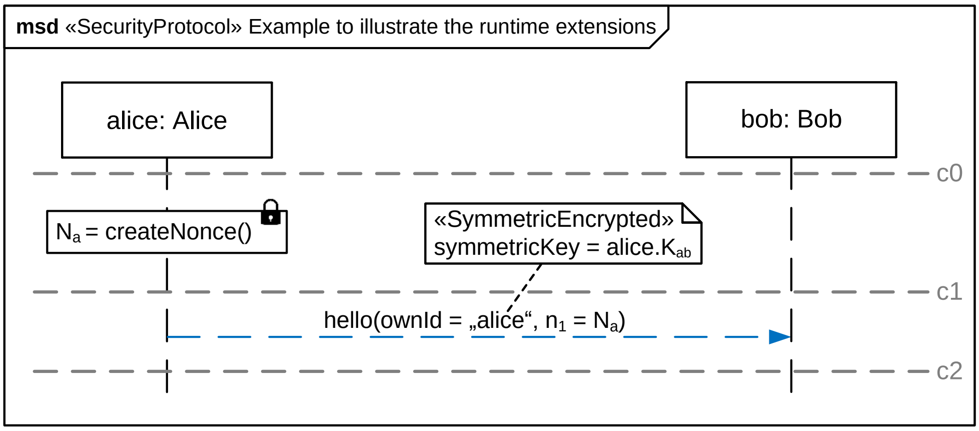

msd «SecurityProtocol» Example to illustrate the runtime extensions
alice: Alice
bob: Bob
c0
N_a = createNonce()
«SymmetricEncrypted»
symmetricKey = alice.K_{ab}
c1
hello(ownId = „alice“, n_1 = N_a)
c2

Figure 3.11: Exemplary MSD to illustrate the Runtime Extensions

3.5.1 Runtime Semantics: Minimal Event

As explained in Section 2.1.2, the minimal event in an MSD is a cold and monitored message, meaning that there are no preceding events (e.g., assignments or conditions). However, as shown for the *Andrew Secure RPC* security protocol, in some security protocols, the participants create random values before they send the first message.

There exist several possible solutions to enable this behavior. Intuitively, the security engineer could add a role to the MSD specification that sends a trigger message to the initiator of the security protocol. This message would be the first in the MSD and could be the minimal event of the actual MSD. However, this would cause some side effects if a requirements engineer wants to reuse an existing specification in the requirements specification of an application.

Instead of adding a trigger message, we decide to relax the rule for the minimal event as follows: In an MSD annotated with the stereotype «SecurityProtocol», the minimal event is still a cold and monitored message but an arbitrary number of security assignments defining diagram variables may occur before the minimal event on the sending lifeline. Thereby, we provide an intuitive solution for the desired behavior without adding additional roles to the specification. Moreover, our solution does not cause any changes to the object system since an MSD specification for a security protocol contains only one MSD (cf. Section 3.4.1) and the definition of diagram variables is considered as hidden events [Gre11].

The MSD in Figure 3.11 shows this situation, the message hello() sent from alice: Alice to bob: Bob is the minimal event and thus the cut c1 is directly before that message. To take the relaxed rule for the minimal event into account, we change the procedure to create an active MSD as follows: If a message event occurs that is message unifiable with the minimal event, an active MSD is created. Then, it is checked whether security assignments defining diagram variables exist before the minimal event on the sending lifeline. If this is the case, the diagram variables are created. Afterward, it is checked whether the message event is also parameter unifiable with the minimal event. If this is the case, the MSD remains active and the cut progresses, otherwise, the MSD is terminated. We split the unification process into two parts since the minimal event might refer to the diagram variables created by the security assignments. Without splitting the unification process into two parts, the message event and the minimal event might not be parameter unifiable, since the minimal event might refer to diagram variables that do not yet exist. After the creation of the active MSD, the cut c2 is enabled and the MSD progresses as described in Section 2.1.2.

3.5.2 Runtime Semantics: Message Unification

As mentioned in Section 3.4, cryptographic primitives can be applied to messages and parameters. This requires an extension to the definition of unification since messages and message events that have different applied cryptographic primitives would not be unifiable using the original definition. Initially, the definition of unification covered the sending and receiving of a message, its signature, and its parameters in the case of a parameterized message [Gre11].

Accordingly, a message and a message event are message unifiable if the following four conditions are fulfilled [Gre11]:

1. The message and the message event both reference the same operation.
2. The sending lifeline of the message must be able to represent the sending object of the message event.
3. The receiving lifeline of the message must be able to represent the receiving object of the message event.
4. The order, number, and types of the parameters of the message and the message event must be compatible.

To account for cryptographic primitives, TRIPPEL [+Tri21] added two conditions to the definition of message unification resulting in the following definition:

5. The order, number, and kind of stereotypes applied to parameters of the message and the message event must be equal if the parameters are at the same position.

6. The properties of stereotypes that are applied to parameters must be equal for the message and the message event.

For a message and a message event to be parameter unifiable, GREENYER [Gre11] extended the definition of message unification with the following two conditions:

1. The message and the message event must be message unifiable.
2. The values of the parameters of the message event must be compatible with the parameters specified for the message.

To account for cryptographic primitives, TRIPPEL [+Tri21] added two conditions to the definition of parameter unification resulting in the following definition:

3. The receiving lifeline of the message must be able to represent the receiving object of the message event.
4. The order, number, and types of the parameters of the message and the message event must be compatible.

However, the extension of TRIPPEL [+Tri21] does not take into account whether a sending or a receiving object possesses or knows the key that is used within a cryptographic primitive. Thus, we extend the last condition of each definition as follows: If a stereotype from the SECURITY MODELING PROFILE is applied to a message or a parameter and this stereotype refers to a cryptographic key, both the sending and receiving object must either possess or know this key. If this is the case, the cut progresses, otherwise a violation occurs.

3.6 Implementation

This section presents an overview of our prototypical implementation to support and evaluate the concepts described throughout this chapter. The implementation is integrated into the Eclipse-based SCENARIOTOOLS MSD tool suite [ST-MSD]. In particular, we present the architecture of our implementation in Section 3.6.1 and the user interface in Section 3.6.2.

3.6.1 Security ScenarioTools (Software Architecture)

Figure 3.12 depicts the software architecture that realizes the concepts described in this chapter. The architecture visualization encompasses the components and UML profiles newly implemented in the course of this thesis, the existing frameworks, tool suites, and UML profiles, as well as the dependencies between these components. The overall implementation is based on the Eclipse Modeling Framework (EMF) [EMF] and Eclipse Papyrus [Papyrus].

The newly implemented tool suite SECURITY SCENARIOTOOLS MSD provides the Security Modeling Profile and the Security Runtime. The Security Modeling Profile extends the UML metamodel as part of the component UML2 and the Modal profile as part of the SCENARIOTOOLS MSD tool suite. Moreover, the domain-specific languages for the specification of security assignments and security conditions are realized by means of Xtext [Xtext]. The component Security Runtime extends the component SCENARIOTOOLS MSD runtime to simulate the security MSD specifications as described in Section 3.5.

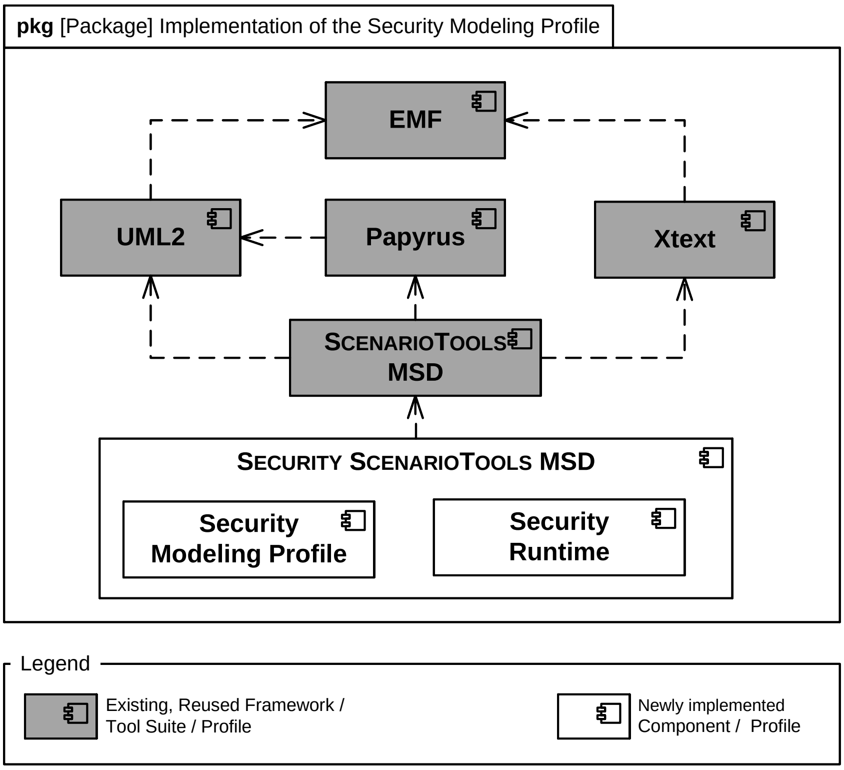

pkg [Package] Implementation of the Security Modeling Profile
EMF
UML2
Papyrus
Xtext
SCENARIOTOOLS MSD
SECURITY SCENARIOTOOLS MSD
Security Modeling Profile
Security Runtime
Legend
Existing, Reused Framework / Tool Suite / Profile
Newly implemented Component / Profile

Figure 3.12: Coarse-grained architecture of the implementation and the reused components

3.6.2 Security ScenarioTools (User Interface)

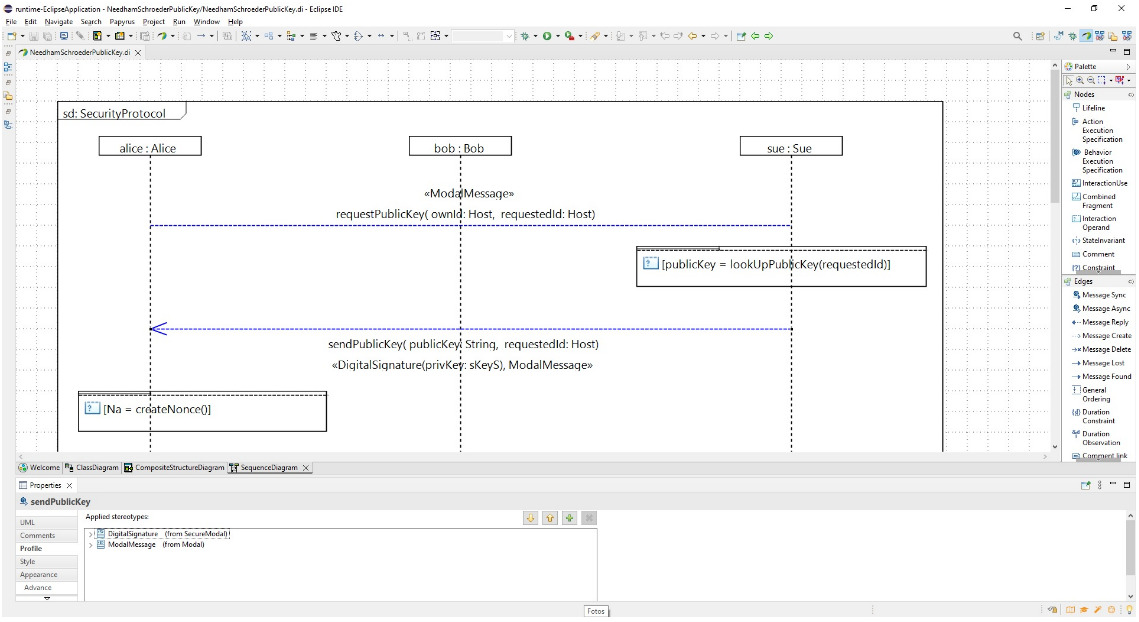
The user interface of the SECURITY SCENARIOTOOLS MSD tool suite provides a modeling and a simulation perspective. Subsequently, we describe both perspectives.

Security ScenarioTools MSD Modeling Perspective

The user can use the Papyrus modeling editors to specify the different parts of the MSD specification. Figure 3.13 depicts an excerpt of the modeling perspective showing the MSD for the *Needham-Schroeder Public Key* security protocol. The modeling editor provides a palette on the right side of the editor such that the user can drag the different model elements and drop them on the drawing plate. We have extended the visualization of messages such that it includes information about the applied cryptographic primitives. For example, the message sendPublicKey(publicKey: EncPublicKey, requestedId: Host) is digitally signed using the private signature key sKeyS. As shown in the bottom of Figure 3.13, the stereotypes of a message can be configured using the properties view.

Security ScenarioTools MSD Simulation Perspective

The simulation perspective provides information necessary for the simulation of an MSD specification. In particular, the simulation perspective provides a set of enabled messages and the user


runtime-EclipseApplication - NeedhamSchroederPublicKey/NeedhamSchroederPublicKey.di - Eclipse IDE
File Edit Navigate Search Papyrus Project Run Window Help
NeedhamSchroederPublicKey.di
sd: SecurityProtocol
alice : Alice
bob : Bob
sue : Sue
«ModalMessage»
requestPublicKey(ownId: Host, requestedId: Host)
[publicKey = lookUpPublicKey(requestedId)]
sendPublicKey(publicKey: String, requestedId: Host)
«DigitalSignature(privKey: sKeyS), ModalMessage»
[Na = createNonce()]
Palette
Nodes
Lifeline
Action Execution Specification
Behavior Execution Specification
InteractionUse
Combined Fragment
Interaction Operand
StateInvariant
Comment
Constraint
Edges
Message Sync
Message Async
Message Reply
Message Create
Message Delete
Message Lost
Message Found
General Ordering
Duration Constraint
Duration Observation
Comment link
Welcome
ClassDiagram
CompositeStructureDiagram
SequenceDiagram
Properties
UML
Comments
Profile
Style
Appearance
Advance
Fotos

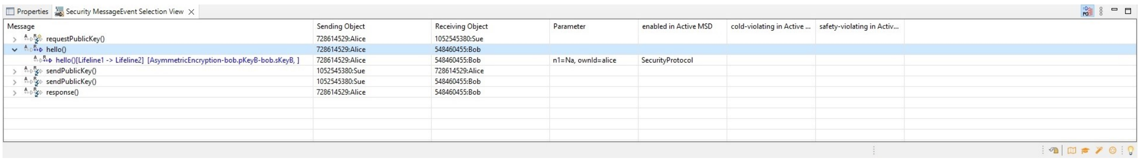
sendPublicKey

Applied stereotypes:

- DigitalSignature (from SecureModal)
- ModalMessage (from Modal)

Figure 3.13: Screenshot of the SECURITY SCENARIOTOOLS MSD modeling perspective

can select the message that should be sent next. As shown in Figure 3.14, we extended the existing view such that the concepts provided by the SECURITY MODELING PROFILE are also shown to the user. In Figure 3.14, the simulation perspective shows five messages to the user. For the message hello(), the simulation perspective also shows that the message is asymmetrically encrypted with the public key owned by Bob.



Message	Sending Object	Receiving Object	Parameter	enabled in Active MSD	cold-violating in Active ...	safety-violating in Activ...
requestPublicKey()	728614529:Alice	1052545380:Sue				
hello()	728614529:Alice	548460455:Bob				
hello()[Lifeline1 -> Lifeline2] [AsymmetricEncryption-bob.pKeyB-bob.sKeyB,]	728614529:Alice	548460455:Bob	n1=Na, ownId=alice	SecurityProtocol		
sendPublicKey()	1052545380:Sue	728614529:Alice				
sendPublicKey()	1052545380:Sue	548460455:Bob				
response()	728614529:Alice	548460455:Bob				

Figure 3.14: Screenshot of the SECURITY SCENARIOTOOLS MSD simulation perspective

3.7 Evaluation

In this section, we conduct a case study based on the guidelines by KITCHENHAM ET AL. [KPP95] and RUNESON ET AL. [RH09; Run12] for evaluating our modeling approach. Our case study investigates the SECURITY MODELING PROFILE's applicability in practice.

3.7.1 Case Study Context

We examine the following evaluation questions:

EQ1 Does the SECURITY MODELING PROFILE enable the specification of real-world security protocols?

EQ2 Does the extension to the runtime semantics enable the simulation of real-world security protocols specified by means of the SECURITY MODELING PROFILE?

To answer the questions, we select 14 security protocols from our data collection. As stated in Section 3.2, our data collection encompasses 54 security protocols from academia and practice. The security protocols that we selected for our case study use different cryptographic primitives and, thus, present a broad range of possible security protocols. Moreover, we ensure that the selected security protocols cover all building blocks that we identified as well as the defined 19 requirements the SECURITY MODELING PROFILE shall satisfy.

3.7.2 Setting the Hypotheses

We define the following hypotheses for this case study.

H1 The different security protocols can be specified using the SECURITY MODELING PROFILE as presented in Section 3.4. For evaluating H1, we model 14 different security protocols from our data collection. We rate H1 as fulfilled if the security protocols can be specified using solely the SECURITY MODELING PROFILE presented in Section 3.4.

H2 The different security protocols can be simulated by means of the Play-out algorithm following the extensions to the runtime semantics defined in Section 3.5. We rate H2 as fulfilled if the 14 modeled security protocols are correctly simulated by means of the Play-out algorithm.

3.7.3 Validating the Hypotheses

In the following, we validate each hypothesis separately using the prototypical implementation of our approach described in Section 3.6.

Validating Hypotheses H1

To validate H1, we created an MSD specification for each of the 14 selected security protocols. During the modeling process, we checked whether the SECURITY MODELING PROFILE is expressive enough to specify the security protocol or whether any language constructs were missing. Afterward, we executed the Papyrus validate function on each MSD specification to check for syntactical problems and unfulfilled OCL constraints. Papyrus returns that no problem exists in any MSD specification.

However, we noticed two problems while using Papyrus. First, moving UML::CombinedFragments in the diagram crashed the editor. The reason for this was a known issue in Papyrus that we could fix. Second, in some cases, the message ordering was corrupted after we closed and opened the editor. The corrupted models could not be repaired via Papyrus; instead, we had to edit the XML file storing the model or re-model the MSD specification. To check whether this issue was caused by our SECURITY MODELING PROFILE, we modeled various sequence diagrams without our profile and could reproduce the behavior of Papyrus without finding the root cause.

Validating Hypotheses H2

To validate H2, we simulated all 14 MSD specifications by means of the Play-out algorithm. We executed the algorithm step by step and checked in each step whether the simulation state was correct. The correct simulation includes unifying messages and arguments that have been annotated with cryptographic primitives and the handling of security assignments and conditions according to the runtime semantics as defined in Section 3.5. For the 14 modeled security protocols, we were able to find a trace through the MSD that does not cause any violation. Moreover, we modified some MSD specifications to provoke violations in the simulation, e.g., changing the key used to encrypt a message to a key that the receiver does not know. In these cases, we could not find a trace through the MSD that does not cause any violation. We repeated the simulation several times to check whether the simulation behavior was deterministic and did not find any deviations in the different simulation runs.

3.7.4 Analyzing the Results

Table 3.1 depicts the results of the case study. We are able to model all security protocols completely by means of our SECURITY MODELING PROFILE, and each MSD specification used only modeling elements introduced in Section 3.4. Moreover, we are able to simulate the MSD specifications by means of the Play-out algorithm.

To conclude the case study, we state that our SECURITY MODELING PROFILE provides means to specify security protocols in a scenario-based manner and is applicable in practice.

Table 3.1: Results of the case study

No.	Security Protocol	H1	H2
		The SECURITY MODELING PROFILE enables the modeling of realistic security protocols	The runtime extensions enable the simulative validation of realistic security protocols
1	Andrew Secure RPC [Sat89; BAN90]	●	●
2	Andrew Secure RPC (BAN) [BAN90; Low96a]	●	●
3	Bull's Authentication Protocol [BO97; RS98]	●	●
4	CH07 [vR09]	●	●
5	CCITT-X.509-Protocol [IM90]	●	●
6	Denning-Sacco Shared Key [DS81; Low00]	●	●
7	Diffie Helman [DH76]	●	●
8	Gong's Mutual Authentication Protocol [Gon89]	●	●
9	Kao Chow's Authentication Protocol [KC95]	●	●

No.	Security Protocol	H1	H2
		The Security Modeling Profile enables the modeling of realistic security protocols	The runtime extensions enable the simulative validation of realistic security protocols
10	Kerberos [BM90; NT94; Low00]	●	●
11	Needham-Schroeder Public Key [Low95]	●	●
12	Needham-Schroeder Symmetric Key [Low95]	●	●
13	Wide Mouthed Frog [BAN90; Low00]	●	●
14	Woo and Lam Mutual Authentication [WL94]	●	●

Legend: ● fulfilled, ◖ partially fulfilled, ○ not fulfilled

3.7.5 Threats to Validity

The threats to validity in our case study are as follows:

Construct Validity

The case study was designed and conducted by the same researcher who developed the approach. Since the researcher might have a bias toward the developed approach, the case study would be more significant if security experts had modeled the security protocols. To mitigate this, we conducted a literature review to collect requirements on the Security Modeling Profile and discussed our modeling approach with security experts from academia. Moreover, we discussed the case study design and its research questions with other researchers.

In addition, the case study results have yet to be evaluated by security experts from academia or industry. To mitigate this, the cases were selected from literature providing an informal description of message sequences between the different participants. Thus, we were able to compare our results with the literature.

External Validity

We only considered 14 different security protocols and, thus, cannot generalize the fulfillment of the hypothesis for all possible security protocols. Nevertheless, the selected security protocols represent typical examples; thus, we do not expect large deviations from other examples. Moreover, during the selection of the cases, we ensured that we selected security protocols that use different building blocks (e.g., encryption, hashing, etc.) in different combinations. Although we cannot guarantee that our selected cases are representative, we cover at least a broad range of security protocols.

Reliability

The case study was conducted based on the prototype implementation that might not be available in the future. To mitigate this, the implemented concepts are defined in Section 3.4 and can be newly implemented.

Moreover, the case study information and the resulting models might not be available in the future. To mitigate this, we discussed the exemplary application of our SECURITY MODELING PROFILE for two examples in detail in Section 3.3.

Finally, to analyze the results of our case study, we manually reviewed all results which is dependent on the reviewer's expertise. To mitigate this, we defined OCL rules (cf. Section 3.4) that can be automatically checked to ease the modeling of the security protocols.

3.8 Related Work

In this section, we investigate related work on approaches that enable the model-based specification of security protocols or security annotation on message-based communication.

FANG ET AL. [SLF+14; FLH+16] propose a modeling and analysis approach for security protocols. They introduce a UML profile to enable the modeling of security protocols by means of UML Interactions. Compared to our profile, their profile does not model the general concepts of security protocols but remains very close to the input language of the security model checker PROVERIF they use to verify the specified security protocols.

AMEUR-BOULIFA ET AL. [ALA19; LLA+16] present a modeling approach based on SysML to enable the specification of security aspects for embedded systems. They enhance SysML block and state machine diagrams to capture security features like secrecy and authentication. In contrast to their approach, we conceived a modeling approach based on sequence charts since they are more appropriate for the specification of requirements on message-based interactions [LT15].

LOBDDERSTED ET AL. [LBD02] present *SecureUML*, a UML-based modeling language for model-driven security. The approach enables the design and analysis of secure, distributed systems by adding mechanisms to model role-based access control. Furthermore, they provide an automatic generation of access control infrastructures based on the specified models. In contrast, we focus on modeling security protocols and not only on access controls.

UMLSec [Jür02] is a model-driven approach encompassing a UML profile for expressing security concepts, such as encryption mechanisms and attack scenarios. It provides a modeling framework to define the security properties of software components and their composition within a UML framework. Similar to UMLSec, MOEBIUS ET AL. [MSG+09] provide the model-driven approach *SecureMDD* to enable the development of security-critical applications. However, both approaches focus either on high-level security requirements or on application-specific security requirements and not, as in our approach, on basic security properties like secrecy and authentication.

3.9 Summary

This chapter presents our SECURITY MODELING PROFILE, a UML-compliant modeling language extending UML Interactions and Modal Sequence Diagrams. The SECURITY MODELING PROFILE provides a set of cryptographic primitives like (a)symmetric encryption, digital signatures, and cryptographic hash functions. These cryptographic primitives can be applied to messages and their arguments. Furthermore, the SECURITY MODELING PROFILE provides modeling elements to specify the conditional behavior of a security protocol. Moreover, we extended the runtime semantics of MSDs to enable the simulative validation of MSD specifications including modeling elements of the newly created SECURITY MODELING PROFILE.

We have developed the SECURITY MODELING PROFILE based on 19 requirements. The requirements describe typical building blocks used to specify the structure and the behavior of security protocols. We conducted a literature study investigating 54 security protocols to derive these requirements,

We implemented a prototype based on SCENARIOTOOLS MSD and evaluated the applicability in practice of our SECURITY MODELING PROFILE by means of 14 security protocols. Our evaluation results indicate that the SECURITY MODELING PROFILE provides an intuitive way to specify security-related aspects for message-based interactions. Thus, security engineers can use our SECURITY MODELING PROFILE to specify security protocols. Moreover, requirements engineers can use the profile and already compiled security protocols to specify and simulatively validate security requirements on the communication behavior of software-intensive systems.

4

Verification of Security MSDs

This chapter introduces VICE (VIsual Cryptography vErifier), a model-checking approach for automatically verifying security protocols in the symbolic model. VICE provides a model transformation concept to transform a security protocol into the input language of various symbolic model checkers. In addition, VICE automatically derives an initial set of analysis queries that the model checker shall verify to decide whether the protocol is secure concerning secrecy and authentication. Moreover, VICE translates the results of a model checker back to the security protocol. We illustrate this using the SECURITY MODELING PROFILE as described in Chapter 3 and the two symbolic model checkers PROVERIF and TAMARIN. Existing verification approaches only support one model checker [FLH$^+$16; ALA19] and either only partially support the automatic derivation of analysis queries [ALA19] or do not assist in deriving analysis queries [FLH$^+$16; MA22]. In addition, they do not process the analysis results and leave it to the security engineer to understand the analysis results [FLH$^+$16; MA22].

This chapter is structured as follows: We provide a list of our contributions in Section 4.1. Afterward, we present an overview of our model-checking approach for verifying security protocols in Section 4.2. Subsequently, we explain the main activities of VICE. First, we present the forward translation from a security-enhanced MSD specification to the VerificationModel in Section 4.3. Second, we explain the translation from the VerificationModel to the input language of PROVERIF and TAMARIN in Section 4.4 and Section 4.5. Fourth, we explain the back-translation of the analysis results in Section 4.6. Afterward, we provide information about the implementation in Section 4.7 and evaluate VICE by means of a case study in Section 4.8. We investigate related work in Section 4.9. Finally, we summarize this chapter in Section 4.10.

We published contents of this chapter in one paper ([*KDH$^+$20]). Furthermore, parts of this chapter have been contributed by the master's thesis of GOPALAKRISHNAN [+Gop21]. GOPALAKRISHNAN developed an initial concept for the forward translation from SECURITY MODELING PROFILE to the security model checker TAMARIN.

4.1 Contributions

The contributions of this chapter can be summarized as follows:

- We present our model-checking approach for the automatic verification of security protocols in the symbolic model called VICE (VIsual Cryptography vErifier). Thereby, we enable security engineers without deep knowledge of the two model checkers PROVERIF and TAMARIN to verify security protocols.

  - VICE provides a generic model transformation concept to transform security protocols specified by means of the SECURITY MODELING PROFILE into the input language of the two model checkers PROVERIF and TAMARIN.
  - For each model checker, VICE automatically derives a set of analysis queries to verify whether the security protocol is secure concerning secrecy and authentication.
  - For each model checker, VICE processes the analysis results and provides an overview to the security engineer.

- We investigate the hierarchy of authentication specification [Low97] and provide a rule set to automatically utilize the authentication specifications in the derivation of analysis queries that verify the authentication of the security protocol.

- We implement a prototype based on SCENARIOTOOLS MSD and show in an evaluation that VICE is applicable in practice.

4.2 Overview of the Model-Checking Approach for Verifying Security Protocols

This section presents an overview of our model-checking approach for the verification of security protocols in the symbolic model. As depicted in Figure 4.1, VICE encompasses eight process steps. Two of them are manually executed by security engineers and the other six are fully automated by means of model transformation techniques to help the security engineer to verify a specified security protocol without deep knowledge of the model checker. Thus, the specification of the security protocol and the inspection of the analysis results are independent of the used model checker. In the following, we introduce each step of the model-checking approach. Moreover, we provide further details about the automated steps in the subsequent sections.

In the first step, Specify Security Protocol, the security engineer manually specifies a security protocol by means of the SECURITY MODELING PROFILE as described in Chapter 3.

In the second step, Translate Security Protocol, VICE creates an intermediate model, the so-called SecurityProtocolModel. The SecurityProtocolModel provides a lightweight representation of a security protocol independent of the modeling language used in the previous step. Therefore, VICE extracts information from the specified security protocol, e.g., participants of the protocol, exchanged messages, and cryptographic primitives, and transforms them into the corresponding elements in the SecurityProtocolModel. In addition, VICE creates trace links to relate the source model's elements (e.g., the UML specification) with the target model's elements.

In the third step, Derive Analysis Queries, VICE derives a QueryModel. In particular, VICE analyzes the security protocol and creates queries to verify the secrecy and queries to verify the authentication. For the creation of secrecy queries, it investigates the properties of the participants and the security assignments occurring during the execution of the security protocol. For the creation of authentication queries, VICE utilizes a rule set based on the hierarchy of authentication

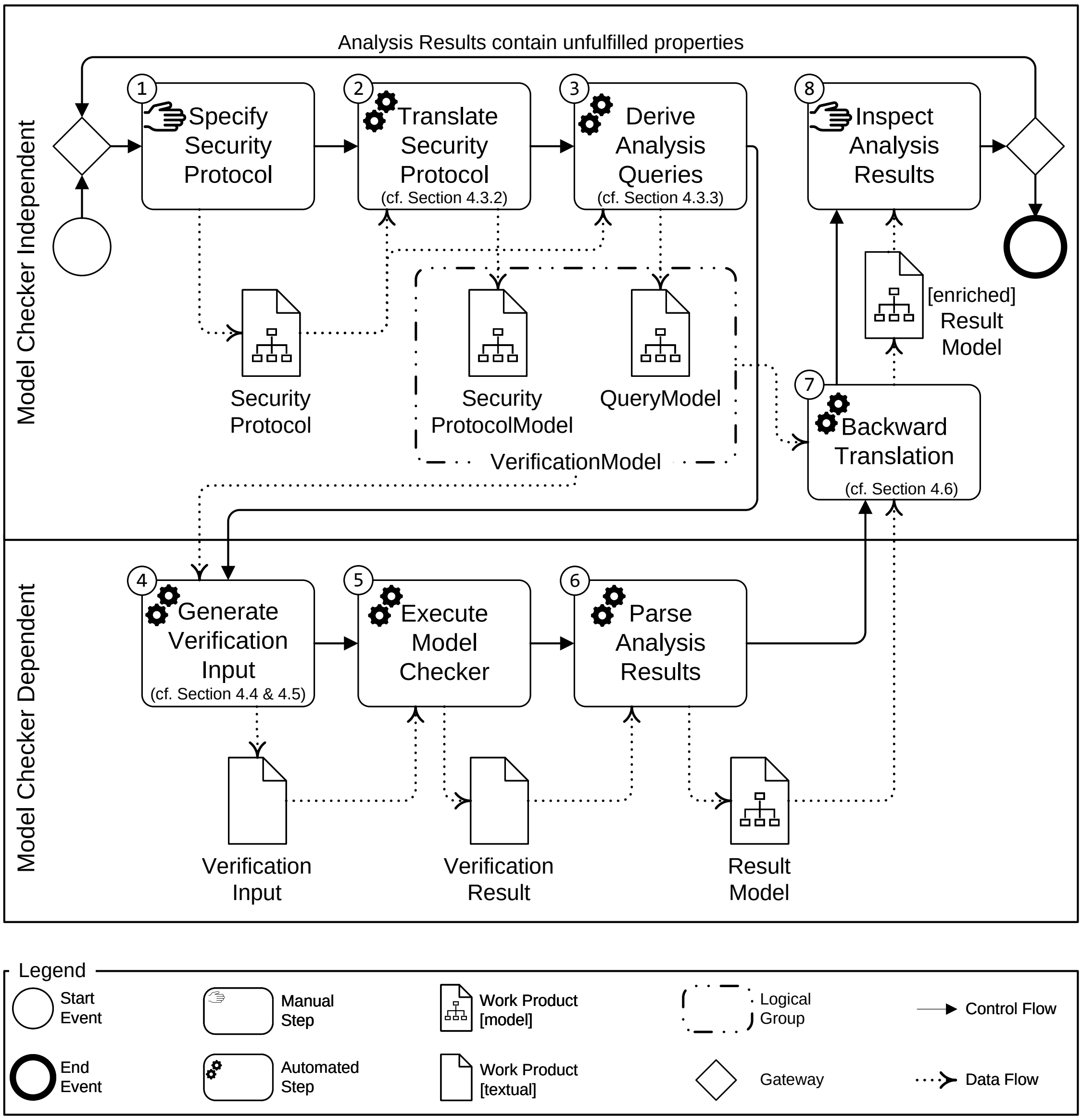


Figure 4.1: Overview of the model-checking approach for verifying security protocols

specifications specified by LOWE [Low97]. Since we analyze the security protocol in the symbolic model and assume an attacker that controls the complete channel (cf. Section 2.2), the set of queries is sufficient to comprehensively verify the protocol. The SecurityProtocolModel and the QueryModel together form the VerificationModel.

In the fourth step, Generate Verification Input, VICE uses the VerificationModel and generates the input for the used model checker. Since most existing model checkers operate on a textual input language, this step is realized by a model-to-text transformation. The generated input depends on the used model checker and encompasses the description of the security protocol and the queries the model checker shall analyze. VICE supports the input generation for PROVERIF and TAMARIN.

In the fifth step, Execute Model Checker, VICE executes the model checker with the generated input. The security engineer can choose whether both model checkers should be executed or only one of them. Although TAMARIN provides an automated and an interactive mode, VICE only uses the automated mode. After the termination of the model checker, VICE retrieves the verification results. For most model checkers, the verification results are textual and contain a statement for each query whether this query is fulfilled or not. In case it is not fulfilled, the verification results may contain a counterexample.

Thus, in the sixth step, Parse Analysis Results, VICE processes the verification results and transforms them into a ResultModel. The ResultModel stores the results for each query and contains references to the VerificationModel.

In the seventh step, Backward Translation, VICE translates the analysis results back to the level of the input language used by the security engineer. During the translation, VICE resolves all trace links created in the first step to identify the source elements in the security protocol. In particular, VICE identifies the elements to which the queries refer. Moreover, it identifies the elements of the counterexample (e.g., participants and messages) and resolves them. Based on the resolved trace elements, VICE creates a sequence diagram showing the counterexample.

Finally, in the eighth step, Inspect Analysis Results, the analysis results are shown to the security engineer such that he/she can inspect them and correct the security protocol if necessary.

4.3 Translation from MSDs to the VerificationModel

In the second step, Translate Security Protocol (cf. Figure 4.1 on page 59), VICE translates a security protocol specified by means of the SECURITY MODELING PROFILE (cf. Chapter 3) to the VerificationModel. This section introduces the VerificationModel and describes the functional principle of the model transformation from a security protocol to the VerificationModel.

4.3.1 Overview of the Metamodel Verification

Figure 4.2 depicts the package diagram of the metamodel Verification. We divide the metamodel into the packages Protocol and Query. The package Protocol provides means to specify the structure and the behavior of a security protocol. It is further subdivided into the package Primitives, Expressions, and Types. The package Query provides means to specify queries for analyzing the secrecy of protocol variables and authentication properties. Subsequently, we present the details of the packages Protocol and Queries and refer to Appendix B.1 for information about the two packages Primitives and Types. We omit the description of the package Expressions since it is identical to the metamodels described by the two grammars in Section 3.4.4 and Section 3.4.5.

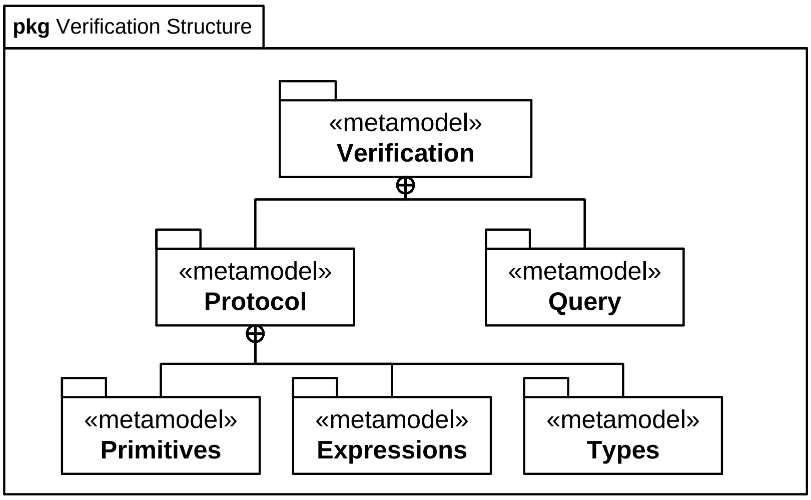


Figure 4.2: UML package diagram of the Verification metamodel

Figure 4.3 depicts the class diagram of the Verification metamodel. The class VerificationModel encompasses the two properties protocolModel of type SecurityProtocolModel and queryModel of type QueryModel. The class SecurityProtocolModel encapsulates information about the structure and behavior of the security protocol. The class QueryModel encompasses queries the model checker shall verify.

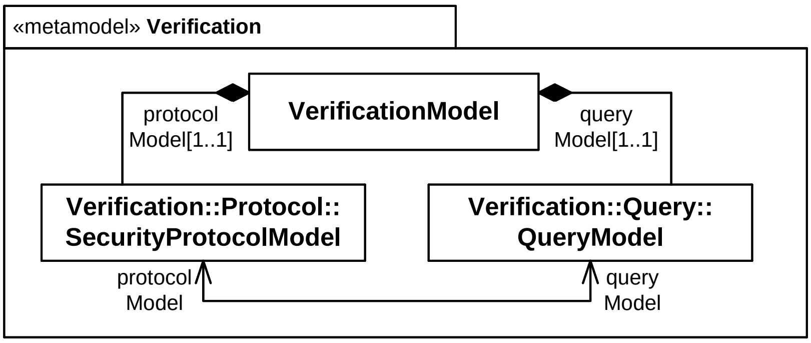


Figure 4.3: UML class diagram of the Verification metamodel

Overview of the Package Verification::Protocol

The class diagram of the package Verification::Protocol is depicted in Figure 4.4 and encompasses all classes necessary to specify the structure and behavior of a security protocol. A SecurityProtocolModel consists of exactly one SecurityProtocol. A SecurityProtocol encompasses at least two Participants but can encompass any number of Participants. The property isTrustedThirdParty of the Participant enables the distinction between participants and trusted third parties. Moreover, a Participant encompasses Properties and Events.

The class Property represents the knowledge that a Participant has before the actual execution of the security protocol.

The class Event describes an event that occurs during the execution of a security protocol. Thus, a set of events for a given Participant describes the participant's behavior, and the set of all Events describes the behavior of the security protocol. Each Event has a predecessor and a successor to specify the behavior of the complete security protocol. We distinguish three kinds of Events: MessageEvent, SecurityAssignment, and SecurityCondition.

- The class MessageEvent describes the sending or receiving of a message. Therefore, it contains a type that can be either MessageIn for receiving messages or MessageOut for sending messages. Furthermore, a MessageEvent may contain arguments. Like a Property, an argument has a type and an actual value. SecurityPrimitives may secure MessageEvents and Arguments. We distinguish five primitives: symmetric encryption, asymmetric encryption, hash-based message authentication code, digital signature, and hashing. For each primitive, the SecurityProtocolModel contains corresponding classes and each class contains references to cryptographic keys if necessary.
- The class SecurityAssignment describes the creation of a new variable or the modification of an existing variable during the execution of the security protocol. Therefore, the SecurityAssignment contains the two properties variable of type Variable and expression of type Expression.
- The class SecurityCondition describes the conditional behavior of a security protocol. Therefore, the SecurityCondition contains the property op of type ComparisionOp and the left-hand side and right-hand side expression (cf. the two associations lhs and rhs of type Expression). If the SecurityCondition evaluates to true during the execution of the security protocol, the execution progresses. Otherwise, the current run of the protocol terminates.

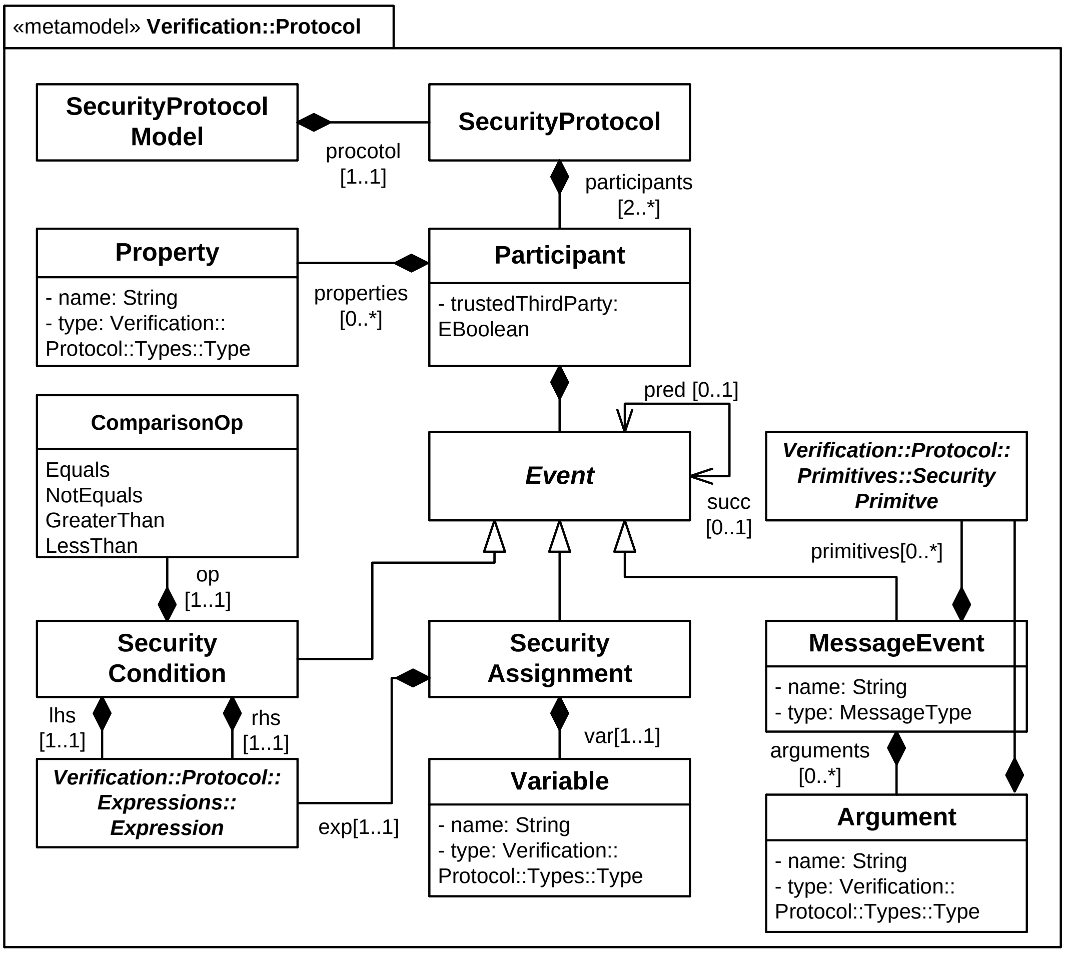


Figure 4.4: UML class diagram of the package Verification::Protocol

Overview of the Package Verification::Query

Figure 4.5 depicts the class diagram of the package Verification::Query. VICE supports the analysis of secrecy and authentication queries. Thus, the metamodel encompasses the two classes SecrecyQuery and AuthenticationQuery. Both classes extend the abstract class Query.

A SecrecyQuery either relates to a participant's property or a variable created by a security assignment. Thus, the two classes PropertyQuery and VariableQuery extend the SecrecyQuery. Each class has a property secretElement of type Property for the PropertyQuery and Variable for the VariableQuery, respectively.

As explained in Section 2.2, the analysis of whether participant A is authenticated to participant B is based on the correspondence assertion of two events. The class AuthenticationQuery is used to model the correspondence assertion and encompasses six properties. The property authenticate of type Participant describes the participant that wants to be authenticated and the property authenticationService of type Participant describes the participant that authenticates, respectively. Additionally, the two properties prevEvent and postEvent refer to an Event and form the correspondence assertion postEvent => prevEvent. Furthermore, the events in a correspondence assertion can refer to an arbitrary number of arguments. Thus, the class encompasses a list of arguments. Finally, the class encompasses the property authenticationType of type EAuthenticationSpecification. The enum EAuthenticationSpecification models the hierarchy of authentication specification as defined by LOWE [Low97].

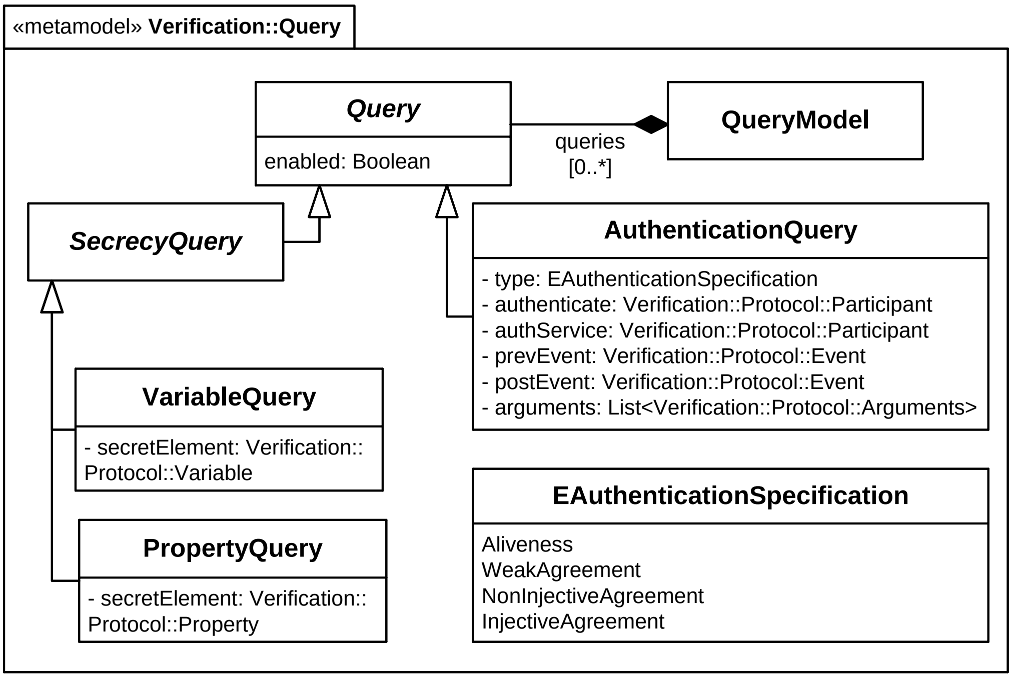

«metamodel» Verification::Query
Query
enabled: Boolean
queries
[0..*]
QueryModel
SecrecyQuery
AuthenticationQuery
- type: EAuthenticationSpecification
- authenticate: Verification::Protocol::Participant
- authService: Verification::Protocol::Participant
- prevEvent: Verification::Protocol::Event
- postEvent: Verification::Protocol::Event
- arguments: List<Verification::Protocol::Arguments>
VariableQuery
- secretElement: Verification::Protocol::Variable
PropertyQuery
- secretElement: Verification::Protocol::Property
EAuthenticationSpecification
Aliveness
WeakAgreement
NonInjectiveAgreement
InjectiveAgreement

Figure 4.5: UML class diagram of the package Verification::Query

4.3.2 Translate an MSD Specification to the SecurityProtocolModel

This section presents the functional principle of the model-to-model transformation from security protocols specified by means of our SECURITY MODELING PROFILE to the SecurityProtocolModel. Algorithm 4.1 depicts an overview of the model-to-model transformation encompassing five main steps.

At the beginning, the transformation algorithm checks if the input model is valid, i.e., the stereotype SecurityProtocol is applied to the MSD (cf. ① Derive security protocol). If the input is not valid, the algorithm terminates, otherwise it creates a new SecurityProtocol element. Then, the transformation algorithm iterates over all lifelines and derives the participants and their properties (cf. ② Derive Structure), their initial knowledge (cf. ③ Derive Initial Knowledge), and their behavior (cf. ④ Derive Behavior). Finally, the transformation algorithm iterates over all fragments contained in the interaction and preserves their ordering also in the newly created SecurityProtocol (cf. ⑤ Preserve ordering of InteractionFragments).

① Derive Security Protocol

As described in Chapter 3, an MSD specification for a security protocol encompasses exactly one MSD. Thus, we consider an MSD specification relevant for our transformation if it contains

Algorithm 4.1 Translation from an MSD Specification to the SecurityProtocolModel

```
Input: MSDSpecification
Output: SecurityProtocolModel
▷ ① Derive Security Protocol
if MSDSpecification.interaction[0].getAppliedStereotype("SecurityProtocol") $\neq$ null then
    securityProtocol = createSecurityProtocol()
    ▷ ② Derive Structure
    for each lifeline $\in$ MSDSpecification.interaction[0].lifelines do
        participant = createParticipant(lifeline.represents)
        securityProtocol.participants += participant
        ▷ ③ Derive Initial Knowledge
        deriveInitialKnowledge(participant, lifeline.represents)
        ▷ ④ Derive Behavior
        for each fragment $\in$ lifeline.fragments do
            if fragment is Message then
                participant.events += createMessageEvent(fragment)
            else if fragment is CombinedFragment then
                if fragment.getAppliedStereotype("SecurityAssignment") $\neq$ null then
                    participant.events += createSecurityAssignment(fragment)
                else if fragment.getAppliedStereotype("SecurityCondition") $\neq$ null then
                    participant.events += createSecurityCondition(fragment)
                end if
            end if
        end for
        for each fragment $\in$ MSDSpecification.interaction[0].fragments do
            ▷ ⑤ Preserve ordering of InteractionFragments
            preserveOrderingOfFragments(fragment)
        end for
    end for
end if
```

exactly one MSD and the stereotype «Security Protocol» is applied to this MSD. If this is the case, the transformation algorithm creates a new SecurityProtocol element.

② Derive Structure of the Security Protocol

In the second step, the transformation algorithm creates the structure of the security protocol. Therefore, it iterates over each lifeline in the MSD and resolves the abstract syntax link represents to identify the role representing the lifeline. The transformation algorithm creates a new Participant with the name of the role. For example, in Figure 4.6, the lifeline participantA: ParticipantA is represented by the role participantA: ParticipantA. Thus, the transformation algorithm creates a new Participant with the name participantA.

Afterward, the transformation algorithm resolves the abstract syntax link type to identify the class typing the lifeline. If the stereotype TrustedThirdParty is applied to the class, the transformation algorithm sets the property isTrustedThirdPary of the newly created Participant to true. Next, the transformation algorithm iterates over the class's properties and transforms them into corresponding properties for the newly created participant. For example, in Figure 4.6, the class ParticipantA encompasses the two properties $prop_1$ and $prop_2$, where the stereotype EncPublicKey is applied to $prop_1$ and EncPrivateKey to $prop_2$, respectively. Therefore, the transformation algorithm resolves the stereotype and creates the two properties $prop_1$ of type EncPublicKey and $prop_2$ of type EncPrivateKey. Additionally, the transformation algorithm creates the correspondence association between the two keys to express the cryptographic key pair.

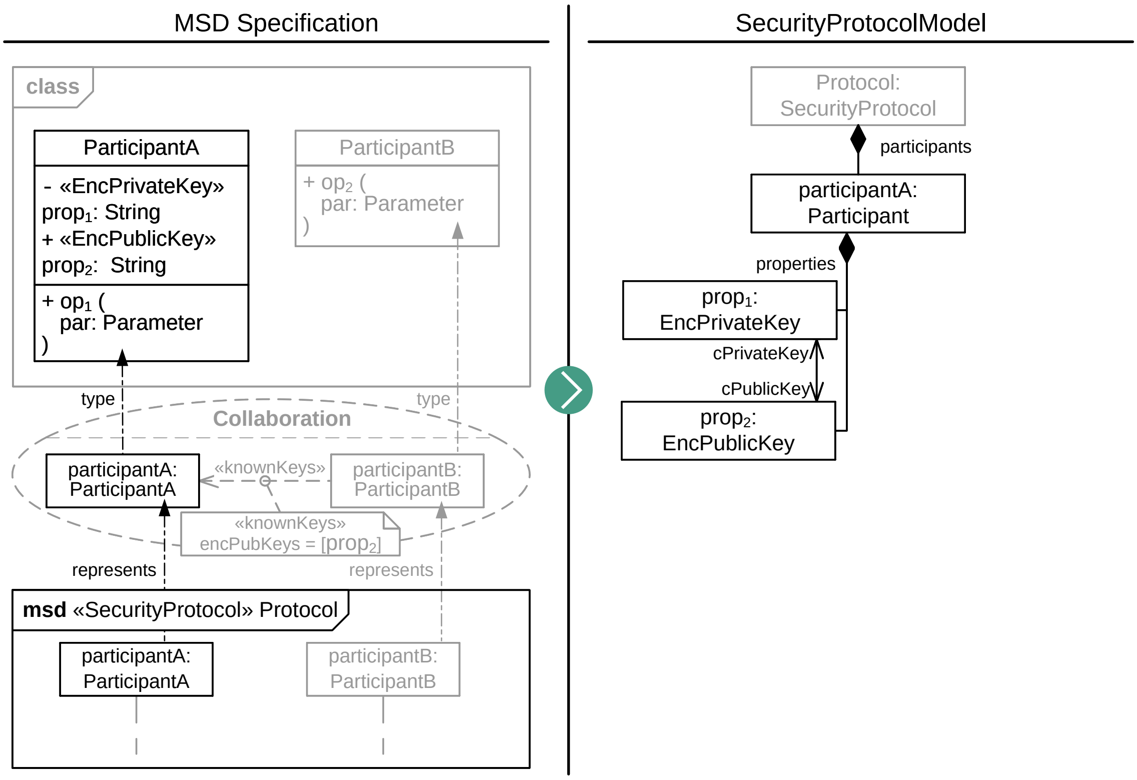

MSD Specification
SecurityProtocolModel
class
ParticipantA
- «EncPrivateKey» prop₁: String
+ «EncPublicKey» prop₂: String
+ op₁ (par: Parameter)
ParticipantB
+ op₂ (par: Parameter)
type
type
Collaboration
participantA: ParticipantA
«knownKeys»
participantB: ParticipantB
«knownKeys» encPubKeys = [prop₂]
represents
represents
msd «SecurityProtocol» Protocol
participantA: ParticipantA
participantB: ParticipantB
Protocol: SecurityProtocol
participants
participantA: Participant
properties
prop₁: EncPrivateKey
cPrivateKey
cPublicKey
prop₂: EncPublicKey

Figure 4.6: Illustration of Transformation Step 2: Derive Structure of the Security Protocol

(3) Derive Initial Knowledge

In the third step, the transformation algorithm derives the initial knowledge of each participant. Therefore, it iterates over each lifeline in the MSD and resolves the abstract syntax link represents to identify the role represented by the lifeline. Then, the transformation algorithm checks whether this role is the source of a dependency with applied stereotype «knownKeys» or «sharesKeysWith». If this is the case, the transformation algorithm identifies each cryptographic key referenced by the stereotype and its corresponding model element in the target model. Finally, the transformation algorithm adds this element to the list of known keys of the participant corresponding to the currently investigated lifeline.

For example, in Figure 4.7, the role participantB: ParticipantB knows the public key of participantA: ParticipantA (cf. dependency with applied stereotype «knownKeys»). The property $prop_2$

that is referenced by the dependency has been transformed to the element prop$_2$: EncPublicKey, and, thus, the transformation algorithm adds this element to the list of known keys of participantB: Participant (cf. association knownKeys between the two elements prop$_2$: EncPublicKey and participantB: Participant).

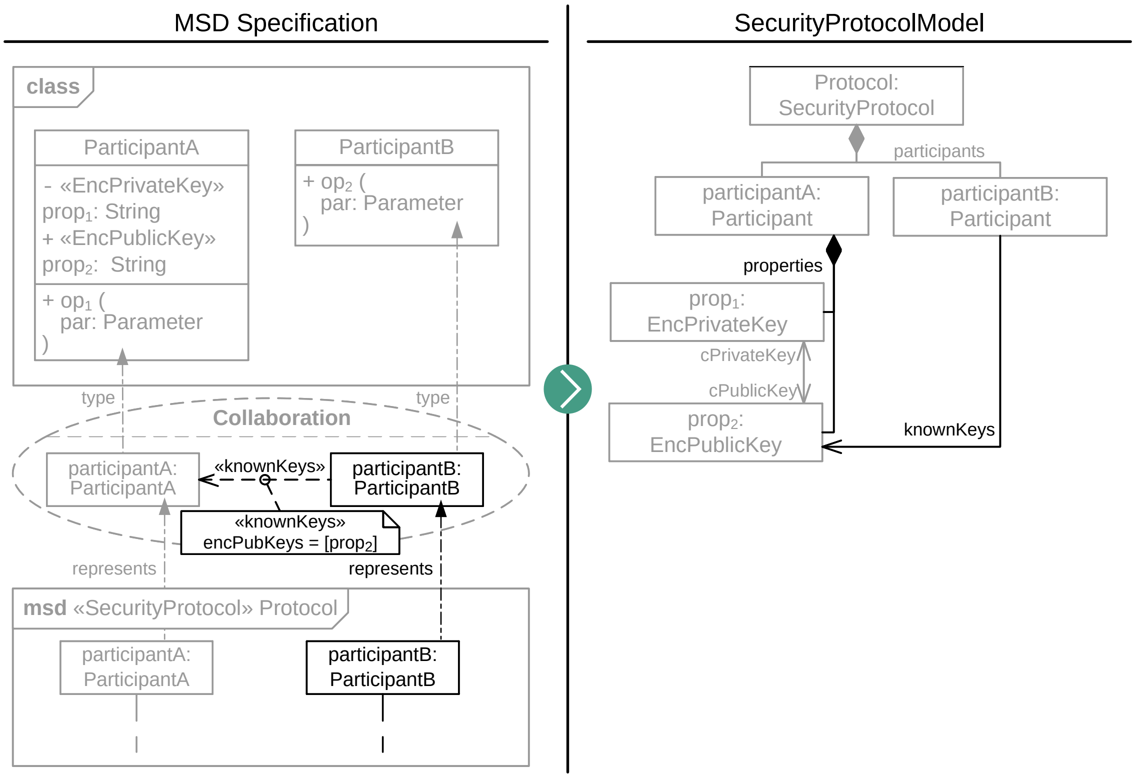

MSD Specification
SecurityProtocolModel
class
ParticipantA
- «EncPrivateKey» prop1: String
+ «EncPublicKey» prop2: String
+ op1 (par: Parameter)
ParticipantB
+ op2 (par: Parameter)
type
type
Collaboration
participantA: ParticipantA
«knownKeys»
participantB: ParticipantB
«knownKeys» encPubKeys = [prop2]
represents
represents
msd «SecurityProtocol» Protocol
participantA: ParticipantA
participantB: ParticipantB
Protocol: SecurityProtocol
participants
participantA: Participant
participantB: Participant
properties
prop1: EncPrivateKey
cPrivateKey
cPublicKey
prop2: EncPublicKey
knownKeys

Figure 4.7: Illustration of Transformation Step 3: Derive Initial Knowledge

(4) Derive Behavior

In the fourth step, the transformation algorithm creates the behavior of the security protocol. Therefore, it iterates over each lifeline in the MSD and derives a list of UML::InteractionFragments covering the lifeline. The transformation algorithm processes each fragment individually. We distinguish three kinds of fragments and describe their translation subsequently.

Translate a Message If the fragment that occurs on the lifeline is a UML::MessageOccurrenceSpecification (UML::MOS), the transformation algorithm creates a new MessageEvent with the same name and sets its type to MessageIn for a receiving UML::MOS and to MessageOut for a sending UML::MOS. Additionally, the transformation algorithm resolves the abstract syntax link signature to identify the operation and to retrieve the message parameters. Based on the message parameters, the transformation algorithm identifies the concrete message arguments and their actual values. Then, it creates the corresponding arguments for the newly created MessageEvent and relates the element in the target model that corresponds to the actual value with the newly created argument. Finally, the transformation algorithm checks whether the message or one of its arguments has applied security primitives. If this is the case, the transformation creates corresponding elements and relates the elements in the target model that correspond to the cryptographic keys with the primitives if necessary.

For example, in Figure 4.8, the MSD encompasses the messages msg$_1$() and msg$_2$(). The message msg$_1$() is sent from participantA: ParticipantA to participantB: ParticipantB and contains the argument arg. During the transformation of participantA: ParticipantA, the transformation algorithm creates two elements msg$_1$: MessageEvent, arg: Argument and adds the element msg$_1$: MessageEvent to participantA: Participant's list of events. The message msg$_2$() is sent from participantA: ParticipantA to participantB: ParticipantB and encompasses the asymmetrically encrypted argument arg. During the transformation of participantA: ParticipantA, the transformation algorithm creates two elements msg$_2$: MessageEvent, arg: Argument and adds the element msg$_2$: MessageEvent to participantA: Participant's list of events. Additionally, the transformation algorithm creates a new AsymmetricEncryption element, adds it to arg's list of primitives, and sets the properties for the cryptographic keys.

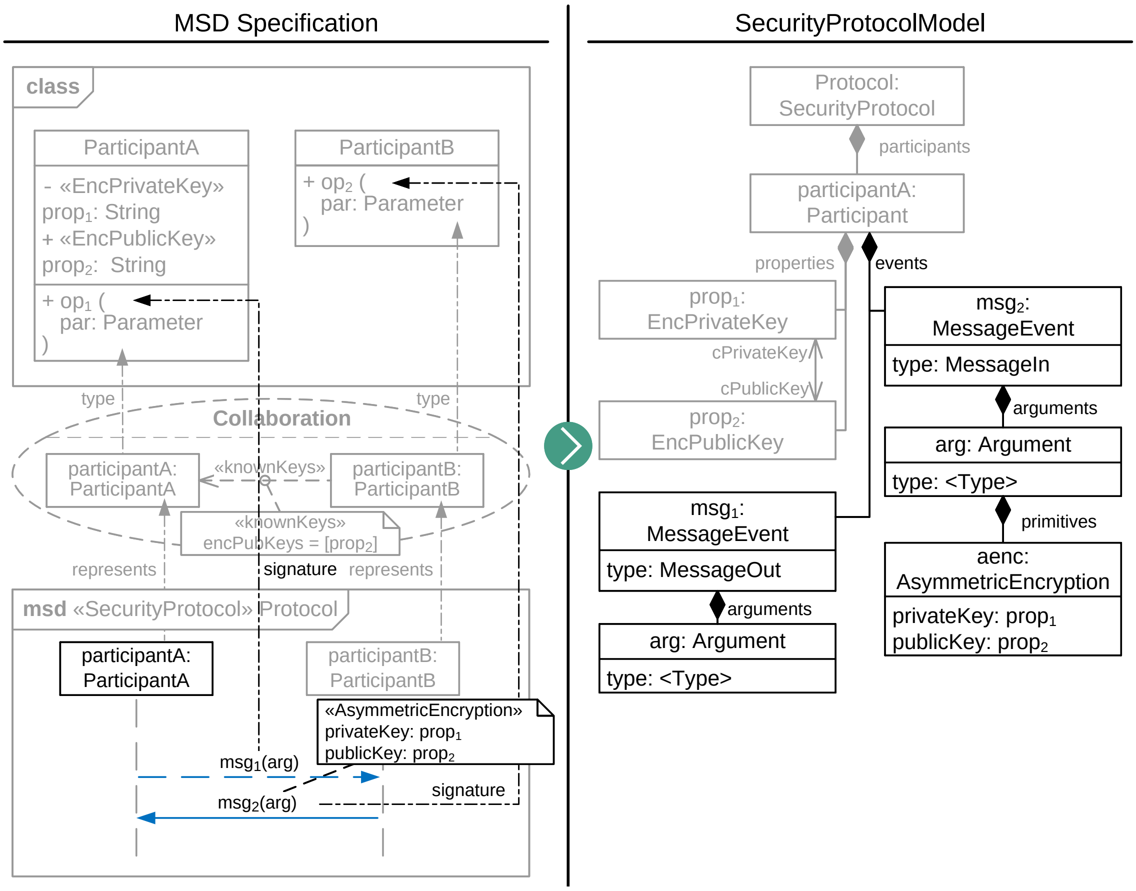


Figure 4.8: Illustration of Transformation Step 4: Derive Behavior — Translation of a Message

Translate a Security Assignment If the fragment that occurs on the lifeline is a UML::CombinedFragment with applied stereotype «SecurityAssignment», the transformation algorithm retrieves the expression describing the SecurityAssignment. As explained in Section 3.4.4, a SecurityAssignment has the form <var> = <expression>. The transformation algorithm creates a new Variable with the same name. Afterward, it transforms the expression. For the specification of expressions, we use the same metamodel as described in Section 3.4.4. Thus, transforming an expression from a security protocol to the SecurityProtocolModel is a one-to-one transformation.

For example, in Figure 4.9, the MSD encompasses the SecurityAssignment var = createNonce(). The transformation algorithm creates a new SecurityAssignment, a Variable with name var, and a new NonceGeneration expression, and relates the newly created elements to each other. Additionally, the transformation algorithm adds the element assignment: SecurityAssignment to participantA: Participant's list of events.

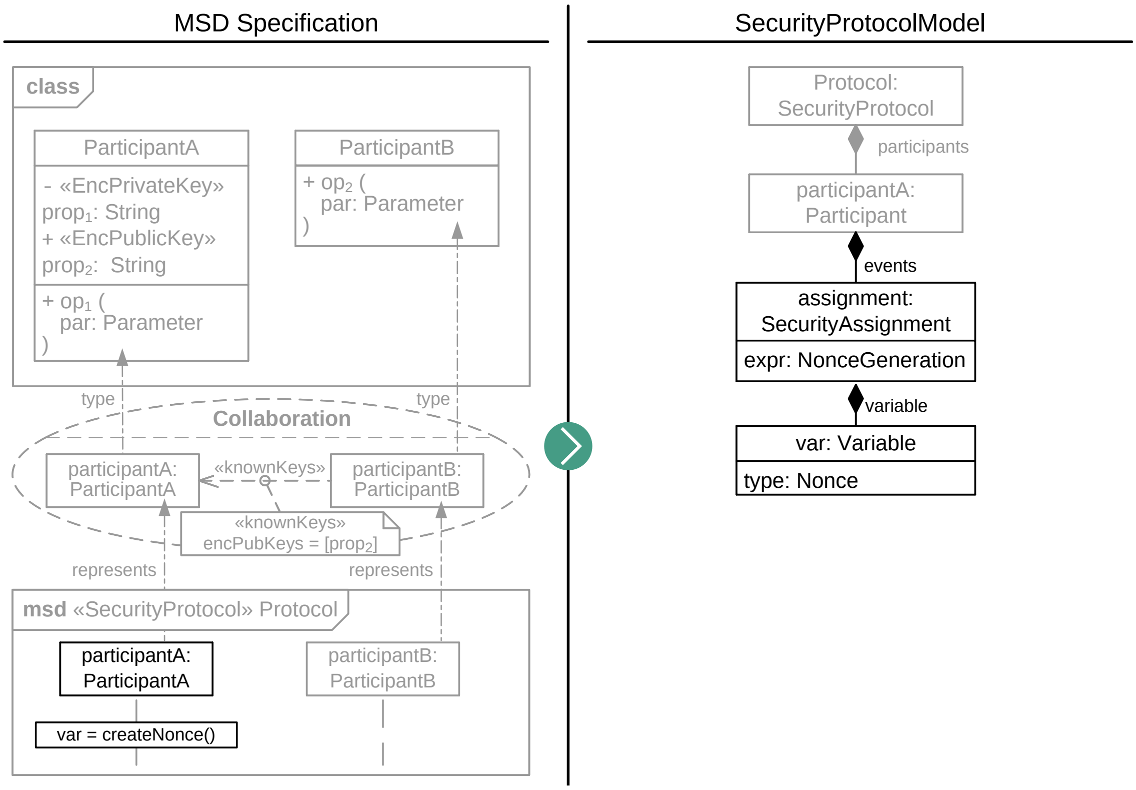

MSD Specification
SecurityProtocolModel
class
ParticipantA
- «EncPrivateKey» prop1: String
+ «EncPublicKey» prop2: String
+ op1 (par: Parameter)
ParticipantB
+ op2 (par: Parameter)
type
type
Collaboration
participantA: ParticipantA
«knownKeys»
participantB: ParticipantB
«knownKeys» encPubKeys = [prop2]
represents
represents
msd «SecurityProtocol» Protocol
participantA: ParticipantA
participantB: ParticipantB
var = createNonce()
Protocol: SecurityProtocol
participants
participantA: Participant
events
assignment: SecurityAssignment
expr: NonceGeneration
variable
var: Variable
type: Nonce

Figure 4.9: Illustration of Transformation Step 4: Derive Behavior — Translation of a Security Assignment

Translate a Security Condition If the fragment that occurs on the lifeline is a UML::CombinedFragment with applied stereotype «SecurityCondition», the transformation algorithm retrieves the expression describing the SecurityCondition. As explained in Section 3.4.5, a SecurityCondition has the form <expression> <op> <expression>. For the specification of expressions, we use the same metamodel as described in Section 3.4.5. Thus, transforming an expression from a security protocol to the SecurityProtocolModel is a one-to-one transformation.

For example, in Figure 4.10, the MSD encompasses the SecurityCondition $\mathsf{var}_1 = \mathsf{inc}(\mathsf{var}_2)$. The transformation algorithm creates a new SecurityCondition. The left-hand side of the condition is transformed to a Reference that refers to var_1. Moreover, the right-hand side is transformed to a

IncreaseExp with a Reference that refers to var_2. Additionally, the transformation algorithm adds the element condition: SecurityCondition to participantA: Participant's list of events.

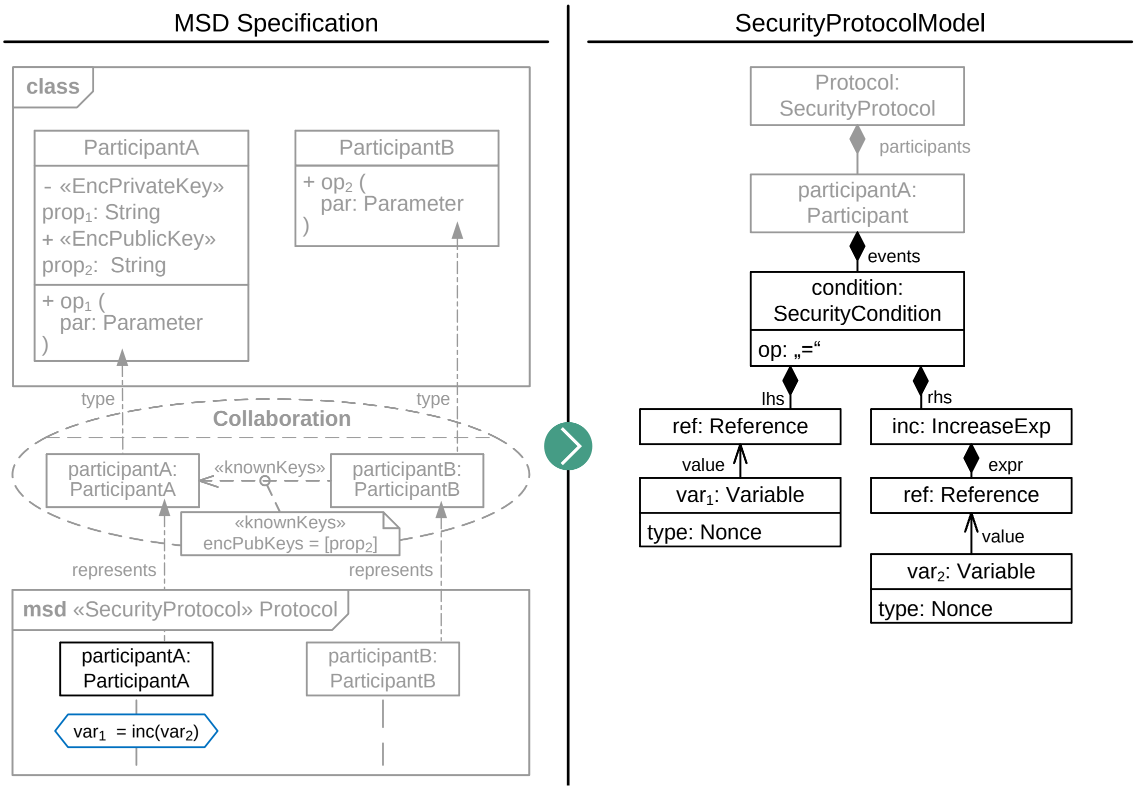


Figure 4.10: Illustration of Transformation Step 4: Derive Behavior — Translation of a Security Condition

5 Preserve ordering of InteractionFragments

The previous transformation steps have transformed the behavior of the individual participants. However, the overall behavior of the security protocol is still missing, i.e. the order in which the participants communicate with each other. Therefore, in the last step of the transformation, the transformation algorithm preserves the order of the UML::InteractionFragments contained in the MSD by transferring the order to the Events of the SecurityProtocolModel. Therefore, the transformation algorithm retrieves all UML::InteractionFragments contained in the MSD. Next, for each fragment in the list, the transformation algorithm identifies the corresponding Event in the target model and sets the preceding event prevEvent.

Figure 4.11 depicts the resulting SecurityProtocolModel after the transformation has finished all five steps for the abstract example used throughout the section.

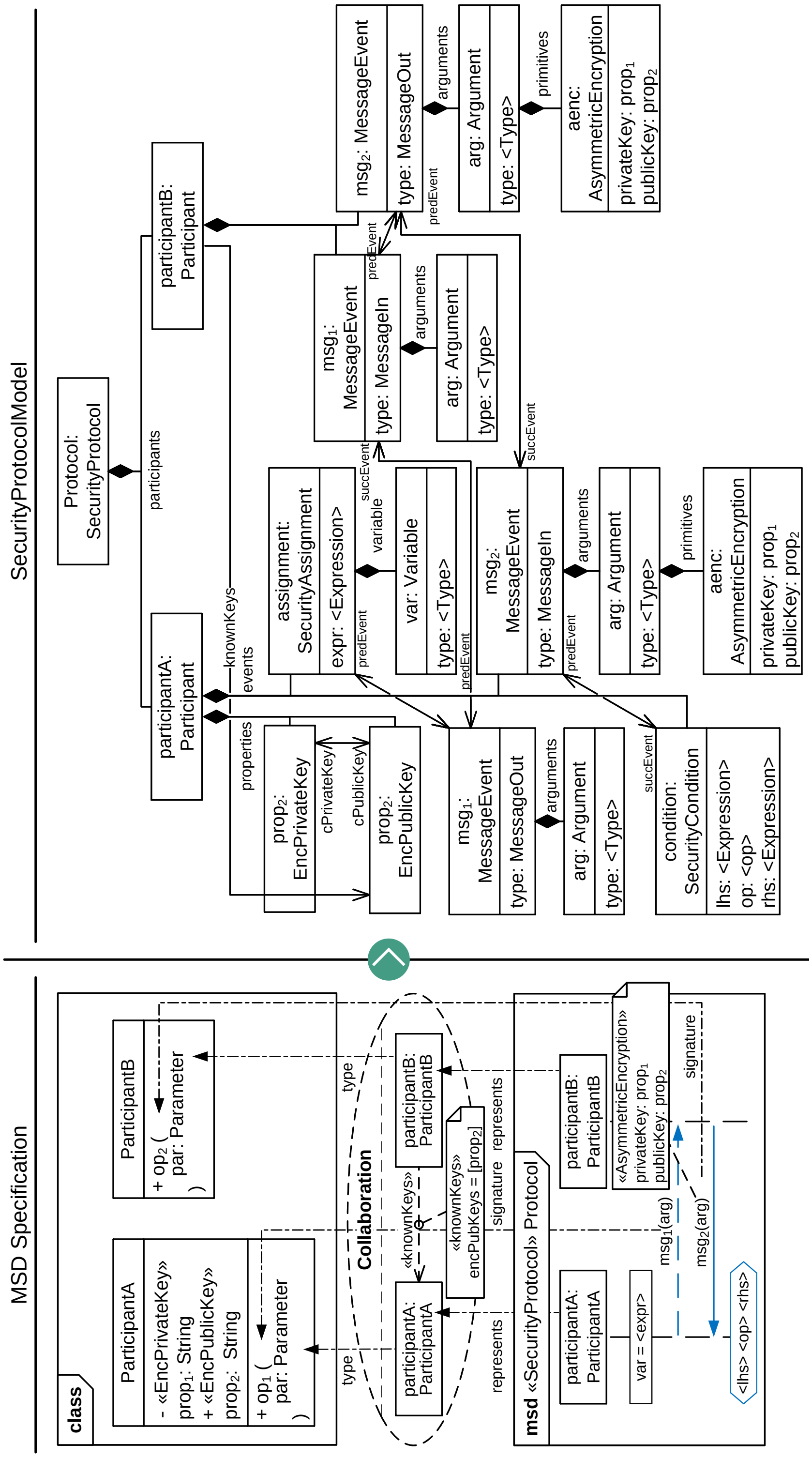


Figure 4.11: Illustration of Transformation Step 5: Preserve ordering of InteractionFragments

4.3.3 Translate an MSD Specification to the QueryModel

This section presents the automatic derivation of analysis queries based on the security protocol defined by means of the SECURITY MODELING PROFILE. Subsequently, we present the approach for deriving secrecy queries and authentication queries.

Derive Secrecy Queries

The derivation of secrecy queries encompasses two parts: In the first part, the transformation algorithm creates SecrecyQueries for all private properties contained in the UML classes of the MSD specification. Therefore, the transformation algorithm iterates over each lifeline in the MSD and resolves the abstract syntax links represents and type to identify the UML class typing the lifeline. Next, the transformation algorithm iterates over the list of properties and for each private property it uses the previously created trace link to identify the corresponding Property in the SecurityProtocolModel. Finally, the transformation algorithm creates a new PropertyQuery and sets its secretElement to the identified Property.

For example, in Figure 4.12, the lifeline participantA: ParticipantA is typed by the UML class ParticipantA. This class encompasses the private property prop_1: Property with applied stereotype «EncPrivateKey». In the transformation algorithm from an MSD specification to the SecurityProtocolModel, the transformation algorithm transforms the property to the element prop_1: EncPrivateKey. Thus, for the query derivation, the secretElement of the newly created PropertyQuery is set to the element prop_1: EncPrivateKey.

In the second part, the transformation algorithm creates SecrecyQueries for all diagram variables created by means of UML::CombinedFragments with the applied stereotype «SecurityAssignment» during the execution of the protocol. For example, in Figure 4.12, the MSD encompasses the SecurityAssignment <var> = <expr>. As in the previous part, the transformation algorithm uses the previously created trace link to identify the Variable that corresponds to the var of the SecurityAssignment. Moreover, it creates a new VariableQuery and sets its secretElement to this var.

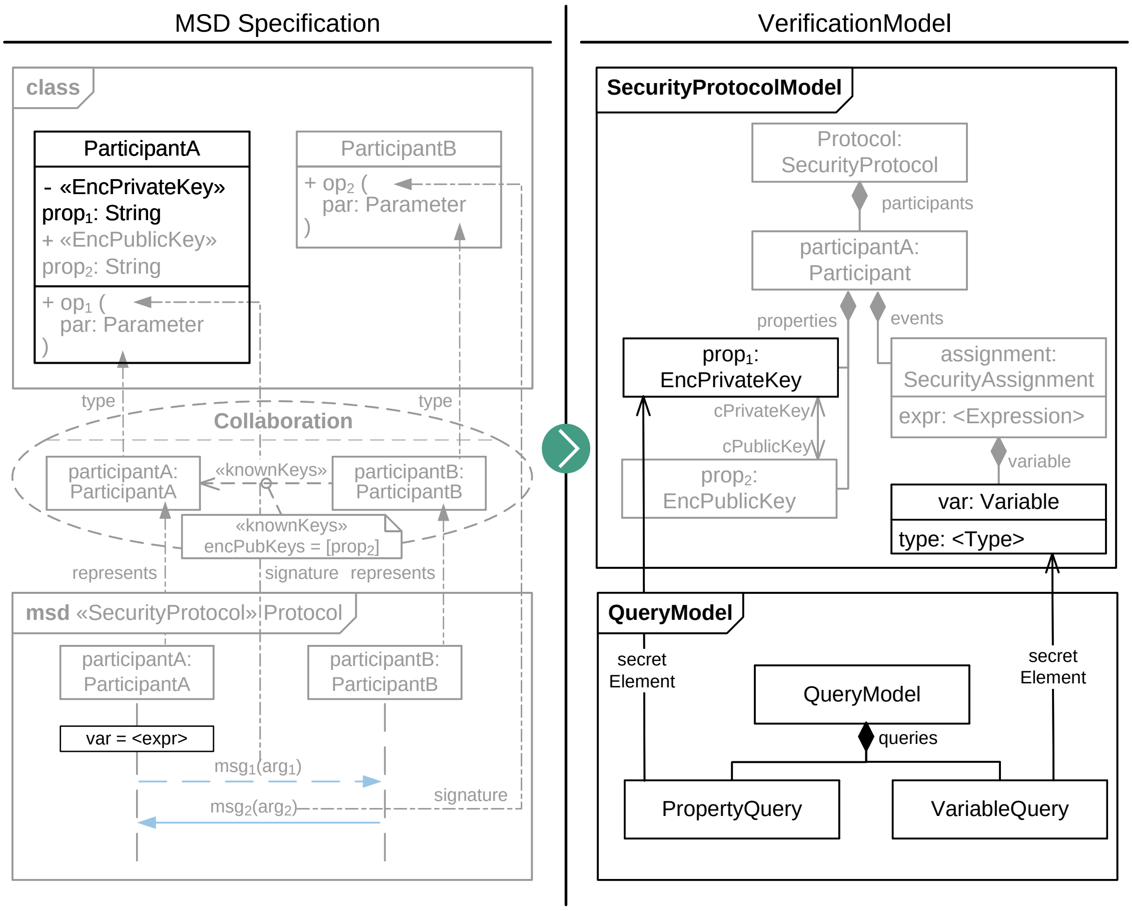


Figure 4.12: Derivation of secrecy queries

Derive Authentication Queries

As explained in Section 2.2.2, the analysis of authentication properties for a security protocol is based on correspondence assertions [Low97]. Correspondence assertions have the form e_1 ==> e_2 and capture the relationship between events that mark important steps in the execution of the security protocol.

The placement of events to annotate the important steps in the execution of the security protocol requires a deeper understanding of the security protocol. However, we apply some rules to systematically place the events in the description of the security protocol's execution. In particular, Blanchet et al. [Bla01] explain that "the event e_1 that occurs before the arrow ==> can be placed at the end of the protocol, while the event e_2 that occurs after the arrow ==> must be followed by at least one output message. Otherwise, the whole protocol can be executed without executing the latter event, so the correspondence certainly does not hold" [Bla01]. Moreover, they explain that "moving an event [e_1] that occurs before the arrow ==> toward the beginning of the protocol strengthens the correspondence property [...]. Moving an event [e_2] that occurs after the arrow ==> toward the end of the protocol also strengthens the correspondence property" [Bla01].

For example, Figure 4.13 sketches a generic protocol between the participants ParticipantA and ParticipantB. To prove the authentication of ParticipantB to ParticipantA, the security protocol has to fulfill the correspondence assertion event e_1 ==> event e_2. Applying the rule described above, the event e_1 is placed at the end of the execution of ParticipantB and the event e_2 is placed before ParticipantA sends the message msg_n to ParticipantB.

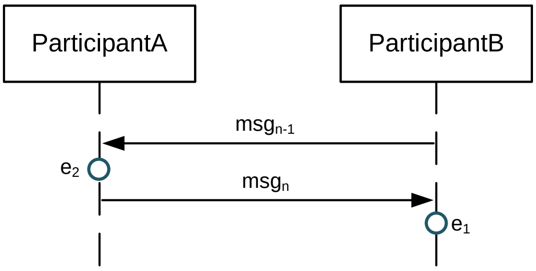


Figure 4.13: Generic security protocol showing the necessary events for authentication of ParticipantA to ParticipantB

Moreover, to prove the authentication of ParticipantA to ParticipantB, the security protocol has to fulfill the correspondence assertion event e_3 ==> event e_4. Again, we apply the rules described above and place the event e_3 at the end of the execution of ParticipantA and the event e_4 before the message msg_{n-1} is sent from ParticipantB to ParticipantA. In contrast to the previous placement, we could also strengthen the correspondence assertion by moving the event e_3 toward the beginning of the protocol, e.g., before the message msg_n (cf. Figure 4.14). However, for the sake of simplicity, in our approach, the event that occurs before the arrow is called termX for a participant X and is always placed at the end of the execution of this participant.

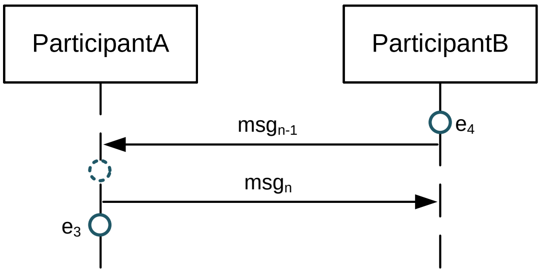


Figure 4.14: Generic security protocol showing the necessary events for authentication of ParticipantB to ParticipantA

According to BLANCHET ET AL. [Bla01], “adding arguments to the events strengthens the correspondence property.” Figure 4.15 extends the generic example shown in Figure 4.13 and adds arguments to the messages. Each message msg_k may contain arbitrary many arguments arg_k^1, …, arg_k^m. Thus, the resulting correspondence assertion to analyze the authentication of the ParticipantA to the ParticipantB is as follows:

$$\begin{aligned}&event\ e_3(arg_1^1, \ldots, arg_m^{n-1}, \ldots, arg_1^{n-1}, \ldots, arg_m^{n-1}) \quad ==> \\ &event\ e_4(arg_1^1, \ldots, arg_m^{n-1}, \ldots, arg_1^{n-1}, \ldots, arg_m^{n-1})\end{aligned}$$

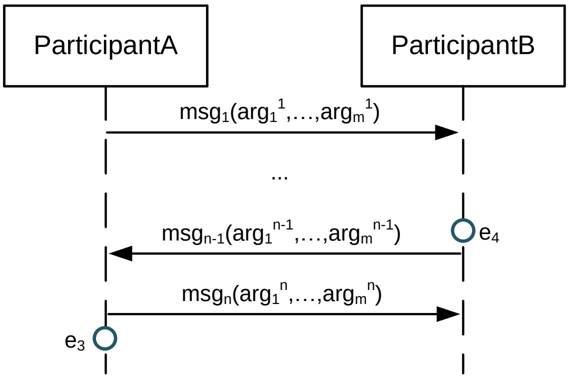


Figure 4.15: Generic security protocol showing messages with arguments and events for authentication

As described in Section 2.2.2, LOWE [Low97] defined a hierarchy of authentication specifications. We use this hierarchy and the rules to place events described above to systematically derive AuthenticationQueries based on the MSD specification.

Aliveness: "A security protocol guarantees *aliveness* to a [participant] A with another [participant] B if, whenever A completes a run of the protocol, apparently with B, then B has previously been running the protocol" [Low97].

To prove this definition of authentication, the security protocol has to fulfill the correspondence assertion *termA ==> startB*. The event *termA* is placed at the end of the execution of A. Moreover, the event *startB* is placed before the first message is sent or received by B.

Weak Agreement: "A security protocol guarantees *weak agreement* to a [participant] A with another [participant] B if, A completes a run of the protocol, apparently with B, then B has previously run the protocol, apparently with A" [Low97].

To prove this definition of authentication, the security protocol has to fulfill the correspondence assertion *termA ==> runningB*. The event *termA* is placed at the end of the execution of A and the event *runningB* is placed before the last message is sent from B to A.

Non-injective Agreement: "A security protocol guarantees *non-injective agreement* to a [participant] A with another [participant] B if, A completes a run of the protocol, apparently with B and some data values $\vec{v}$, then Bob has previously been running the protocol, apparently with A and $\vec{v}$" [Low97].

To prove this definition of authentication, the security protocol has to fulfill the correspondence assertion $termA(\vec{v}) ==> runningB(\vec{v})$. The event *termA* is placed at the end of the execution of A and the event *runningB* is placed before the last message that is sent from B to A. Moreover, the data values $\vec{v}$ are set to the arguments of the last message that is sent from B to A.

Injective Agreement: "A security protocol guarantees *injective agreement* to a [participant] A with another [participant] B if, A completes a run of the protocol, apparently with B and some data values $\vec{v}$, then Bob has previously been running the protocol, apparently with A and $\vec{v}$. In addition, each run of A has to correspond to a unique run of B" [Low97].

To prove this definition of authentication, the security protocol has to fulfill the correspondence assertion $termA(\vec{v}) ==> runningB(\vec{v})$ as before. The formalization of the uniqueness of a run is dependent on the security model checker.

Figure 4.16 illustrates the creation of an AuthenticationQuery. The protocol encompasses the two participants ParticipantA and ParticipantB. For the authentication of ParticipantA to ParticipantB, the transformation algorithm creates a new AuthenticationQuery and sets the property authenticator to participantA and the property authentication service to participantB. Depending on the selected definition of authentication, the transformation algorithm sets the properties postEvent, prevEvent, and arguments. For example, in Figure 4.16, the definition *injective agreement* is selected. Thus, the transformation algorithm sets the property postEvent to the last event that occurs in participantA's event list (cf. condition: SecurityCondition) and the property prevEvent to the last MessageEvent that is sent from participantB to participantA (cf. msg_2: MessageEvent). Finally, the transformation algorithm adds the arguments of the last message that is sent from participantB to participantA to the property arguments of the AuthenticationQuery.

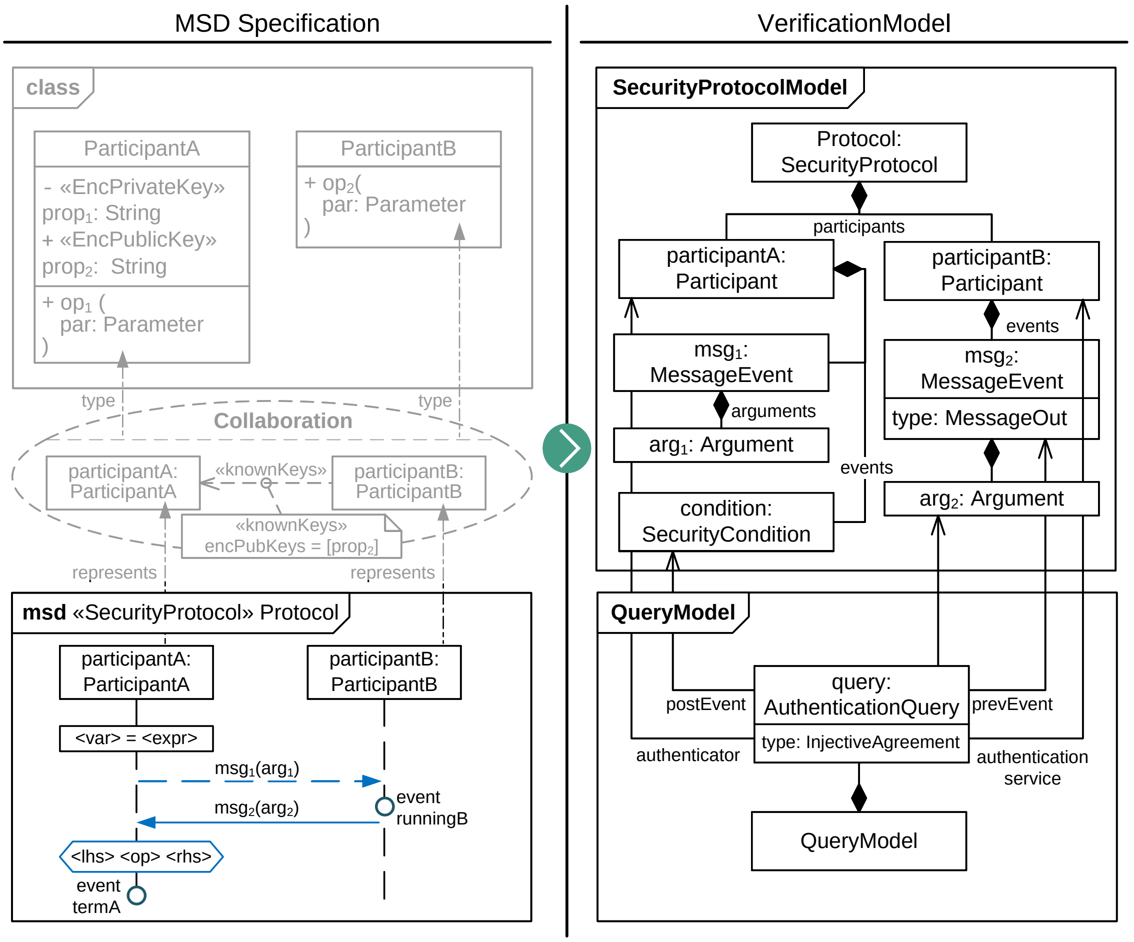


Figure 4.16: Overview of the derivation of authentication queries based on the SecurityProtocolModel

4.4 Translation from the VerificationModel to ProVerif input models

In the fourth step, Generate Verification Input (cf. Figure 4.1 on page 59), VICE uses the VerificationModel and generates the input for the used symbolic model checker. This section describes the details of the input generation for ProVerif. First, we recap the capabilities of ProVerif and discuss whether all elements of the Security Modeling Profile are supported. Second, we present the model-to-text transformation to derive the input for ProVerif.

4.4.1 Overview of the Capabilities of ProVerif in relation to the Security Modeling Profile

Table 4.1 provides an overview of the Security Modeling Profile's features and whether ProVerif is able to analyze security protocols that contain these features. As presented in Section 2.2.3, ProVerif is a tool for the automated analysis of security protocols within the symbolic model. In the following, we briefly describe how we realize the features of the SECURITY MODELING PROFILE by means of PROVERIF language constructs.

In PROVERIF, we can define sub-processes that represent participants of the security protocol. These sub-processes may encompass input parameters. We use the concept of input parameters to model properties that a participant owns or knows. Moreover, to identify and reference participants, PROVERIF models use the type host and specify a variable of type host for each participant of the security protocol. Hence, we conclude that PROVERIF supports all features of the category *Protocol Modeling*.

PROVERIF does not provide any built-in types or functions for cryptographic primitives and arithmetic expressions. Instead, PROVERIF enables the user to specify the types and functions necessary for the analysis of the security protocol. Hence, we conclude that PROVERIF supports all features of the category *Cryptographic Primitives*, *Data types*, *Expressions*. However, we have to keep in mind that PROVERIF treats these functions as black boxes and may miss attacks on the protocol that rely on the algebraic properties of the cryptographic primitives or arithmetic expressions (e.g., associativity, commutativity, and inverse). Moreover, PROVERIF does not support the consideration of time, so it is not possible to compare timestamps with each other or the current time.

PROVERIF enables the declaration of variables by means of the new operator and the modification of existing variables by means of the let operator. Moreover, PROVERIF only supports the equals comparison operator. Hence, we conclude that PROVERIF supports all features of the category *Assignments* but only partially supports the category *Conditions*. So, to overcome this restriction and avoid syntax errors in PROVERIF, we have to inform the security engineer if he/she uses other comparison operators.

Table 4.1: Overview of the Capabilities of PROVERIF in relation to the SECURITY MODELING PROFILE

Category	**Feature of the SECURITY MODELING PROFILE**	**Feature is supported by PROVERIF**
Protocol Modeling	Participants exchange messages to execute the protocol	✓
	Participants can be identified and referenced	✓
	Participants own properties like cryptographic keys	✓
	Participants know properties of other participants	✓
Cryptographic Primitives	Asymmetric Encryption	✓
	Digital Signature	✓
	HMAC	✓
	Hashing	✓
Assignments	Create a new variable	✓

Category	Feature of the SECURITY MODELING PROFILE	Feature is supported by PROVERIF
	Modify an existing variable	✓
Conditions	Conditions with the common comparison operators	✗
Data types	Cryptographic Keys	✓
	Nonce	✓
	Number	✓
	Prime	✓
	Timestamp	✓
Expressions	Arithmetic Expressions	✓
	Cryptographic Expression	✓

4.4.2 Translate the SecurityProtocolModel to PROVERIF

This section presents the functional principle of the model-to-text transformation from the SecurityProtocolModel to PROVERIF input models. Algorithm 4.2 depicts an overview of the model-to-text transformation encompassing four main steps.

At the beginning, the transformation algorithm generates a preamble encompassing all definitions necessary to specify the security protocol (cf. ① Generate Protocol Preamble). Then, the transformation algorithm generates structural information (cf. ② Generate Protocol Structure). Afterward, the transformation algorithm generates a sub-process for each participant describing the behavior of the participant (cf. ③ Generate Protocol Behavior). Finally, the transformation generates the PROVERIF main process (cf. ④ Generate Main Process).

Algorithm 4.2 Translation from a SecurityProtocolModel to PROVERIF

```
Input: SecurityProtocolModel
Output: PROVERIF Input Model
▷ ① Generate Protocol Preamble
generateProtocolPreamble()
▷ ② Generate Protocol Structure
generateChannelDescription()
for each participant ∈ SecurityProtocolModel.participants do
    generateHostDescription(participant)
    ▷ ③ Generate Protocol Behavior
    generateBehaviorForParticipant(participant)
end for
▷ ④ Generate Main Process
generateMainProcess(participant)
```

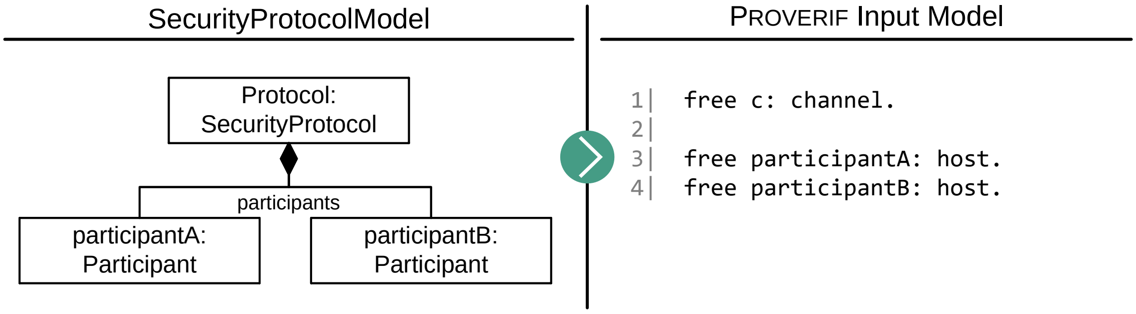
① Generate Protocol Preamble

In the first step, the transformation algorithm generates a preamble for the ProVerif input model. As explained in Section 2.2.3, ProVerif does not contain any built-in types, so security engineers create the preamble on their own. The ProVerif preamble contains all declarations of types and functions that are necessary for modeling the security protocol. Hence, we create a static preamble that encompasses all declarations of types and functions that are required to support the Security Modeling Profile, but not necessarily used by the security protocol to transform. The complete preamble is shown in Appendix B.2.

② Generate Protocol Structure

In the second step, the transformation algorithm generates structural information about the security protocol. Therefore, the transformation algorithm creates a free variable of type host for all Participants contained in the SecurityProtocolModel. Furthermore, it creates a free channel for the communication.

For example, in Figure 4.17, the SecurityProtocolModel encompasses the two participants participantA and participantB. Thus, the transformation algorithm creates the free channel (cf. line 1) and a host for participantA (cf. line 3) and participantB (cf. line 4).



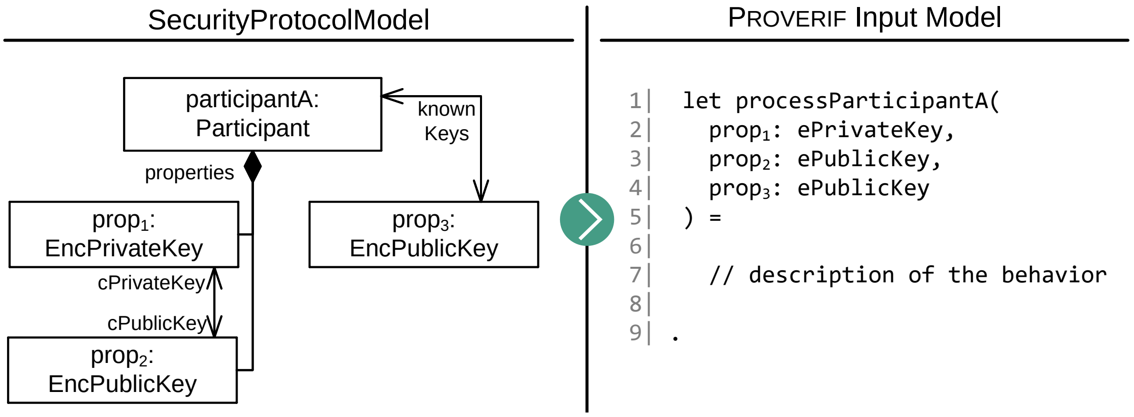
```
1| free c: channel.
2|
3| free participantA: host.
4| free participantB: host.
```

Figure 4.17: Illustration of Transformation Step 2: Generate Protocol Structure (ProVerif)

③ Generate Protocol Behavior

In the third step, the transformation algorithm generates a ProVerif process for each Participant contained in the SecurityProtocolModel. The ProVerif process for a Participant contains all properties and the set of known keys as input. Moreover, the process encompasses all events that occur in the execution of the Participant.

For example, in Figure 4.18, the SecurityProtocolModel encompasses the participant ParticipantA. The participantA has two properties prop$_1$: EncPrivateKey and prop$_2$: EncPublicKey, and knows the key prop$_3$: EncPublicKey of participantB. As output, the transformation algorithm creates a subprocess for participantA (cf. line 1) followed by the input parameter. The transformation algorithm adds the two input parameter prop$_1$: ePrivateKey, prop$_2$: ePublicKey for the properties and prop$_3$: ePublicKey for the known key (cf. lines 2–4) to the sub-process for participantA.



```
let processParticipantA(
  prop1: ePrivateKey,
  prop2: ePublicKey,
  prop3: ePublicKey
) =

  // description of the behavior

.
```

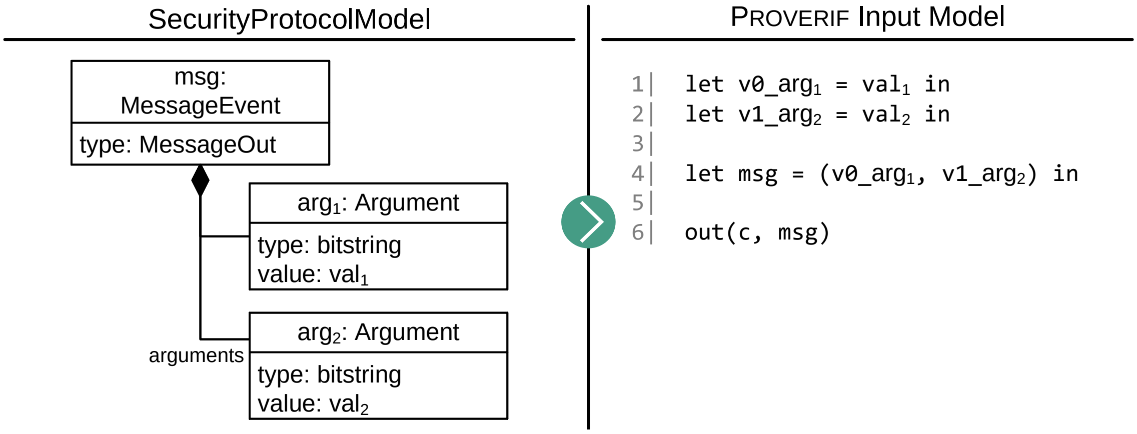
Figure 4.18: Illustration of Transformation Step 3: Generate Protocol Behavior (PROVERIF) — Creation of a sub-process for one Participant

As mentioned in Section 2.2.3, the behavior described within a sub-process encompasses the sending and receiving of messages over a communication channel, the declaration of variables and the conditional execution of a sub-process. Therefore, the transformation algorithm iterates over the ordered set of events for each participant and transforms them into the corresponding constructs in PROVERIF as explained in the following:

Translate a MessageEvent As mentioned in Section 2.2.3, in PROVERIF, a message is represented by means of a bitstring that is transmitted via a channel. The sender uses the process macro out(channel, bitstring) to send a message and the receiver uses the process macro in(channel, bitstring) to receive it. Subsequently, we describe the transformation of a MessageEvent with type MessageOut and a MessageEvent with type MessageIn.

> **Translate a MessageEvent with type MessageOut** The transformation of a MessageEvent with type MessageOut encompasses several steps. First, the transformation algorithm handles the Arguments of the MessageEvent. Therefore, the transformation algorithm retrieves the list of arguments and for each of them, it creates a new variable with the name vX_arg, where v is used as a prefix, X is a consecutive number, and arg is the name of the argument. The variables are numbered consecutively such that the same argument name or the repeated occurrence of a MessageEvent does not result in a PROVERIF warning due to a variable being rebounded. In addition, the transformation algorithm assigns the actual value of the Argument to the newly created variable (cf. let vX_arg$_1$ = val$_1$ in in Figure 4.19). Thereby, it is possible to reuse values in the resulting security protocol in PROVERIF.
>
> Second, the transformation algorithm checks if security primitives are applied to the arguments of the message or to the complete message. The transformation of security primitives applied to arguments or messages follows the same functional principle. Thus, we only present the transformation rules for security primitives applied to messages in this section and distinguish the following three cases:
>
> > **No applied security primitive** The transformation algorithm creates a new variable msg representing the bitstring of the MessageEvent and assigns the set of all argument

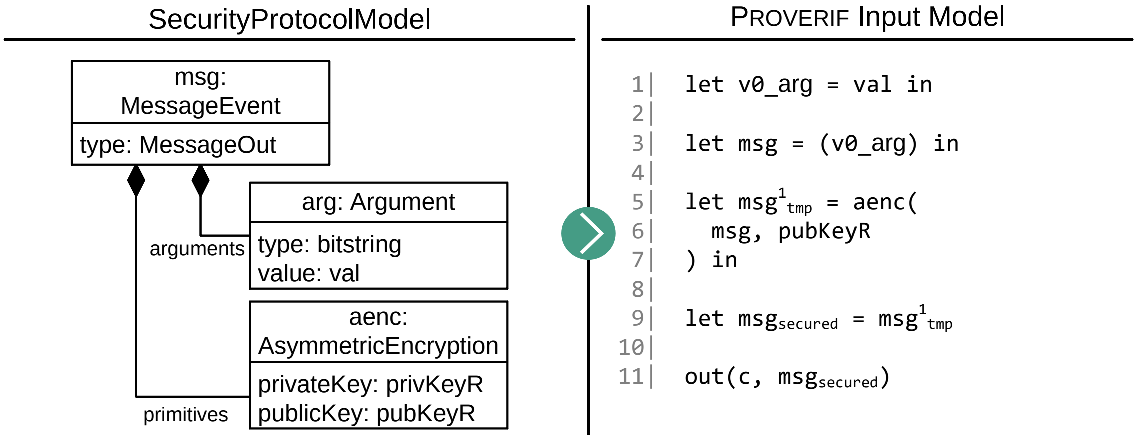
variables to this variable. If the MessageEvent contains more than one Argument, the variables are concatenated and assigned to the variable representing the message (cf. let msg = (v1_arg$_1$, v2_arg$_2$) in in Figure 4.19). Otherwise, the variable representing the argument is directly assigned to the newly created variable (cf. let msg = v1_arg in in Figure 4.20). Finally, the transformation algorithm generates an out(channel, msg) construct, where c is the channel used for the communication between the participants and msg is the bitstring to be transmitted.



```
let v0_arg1 = val1 in
let v1_arg2 = val2 in

let msg = (v0_arg1, v1_arg2) in

out(c, msg)
```

Figure 4.19: Illustration of Transformation Step 3: Generate Protocol Behavior (PROVERIF) — Translation of an outgoing MessageEvent without applied security primitive

One applied security primitive The transformation algorithm creates a new variable msg representing the bitstring of the MessageEvent and assigns the set of all argument variables to this variable. Next, it resolves the constructor that belongs to the security primitive (e.g., aenc for the security primitive AsymmetricEncryption) and applies the constructor using the variable msg and the key referenced by the primitive as input (cf. let msg$_{\text{secured}}$ = aenc(msg, pubKeyA) in in Figure 4.20). Finally, the transformation algorithm generates an out(channel, msg$_{\text{secured}}$) construct, where c is the channel used for the communication between the participants and msg$_{\text{secured}}$ is the bitstring to be transmitted.



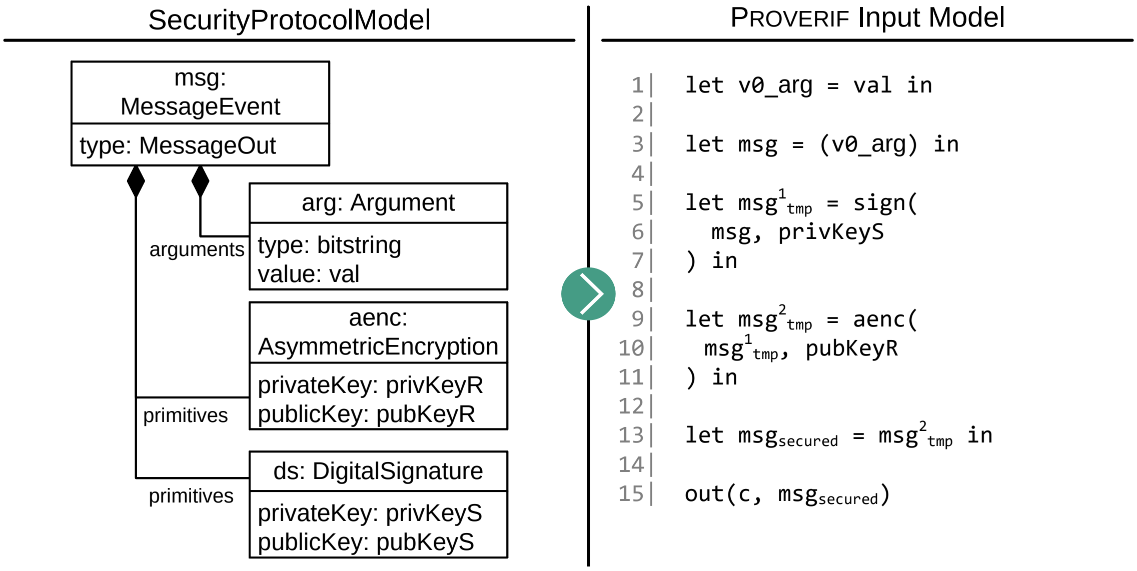
```
let v0_arg = val in

let msg = (v0_arg) in

let msg^1_tmp = aenc(
  msg, pubKeyR
) in

let msg_secured = msg^1_tmp

out(c, msg_secured)
```

Figure 4.20: Illustration of Transformation Step 3: Generate Protocol Behavior (PROVERIF) — Translation of an outgoing MessageEvent with one applied security primitive

More than one applied security primitive The transformation algorithm creates a new variable msg representing the bitstring of the MessageEvent and assigns the set of all argument variables to this variable. Next, the transformation algorithm iterates over the ordered set of security primitives. As described in the previous case, the transformation algorithm resolves the PROVERIF constructor that belongs to the security primitive. However, in case of more than one applied security primitive, the transformation algorithm uses temporal variables to store the result of the transformation of one security primitive (cf. let $\text{msg}^{1}_{\text{tmp}} = \text{sign}(\text{msg}, \text{privKeyS})$ in in Figure 4.21). This variable is then used as the input for the transformation of the next primitive (cf. let $\text{msg}^{2}_{\text{tmp}} = \text{aenc}(\text{msg}^{1}_{\text{tmp}}, \text{pubKeyR})$ in in Figure 4.21). After transforming all security primitives, the result is stored in the variable $\text{msg}_{\text{secured}}$. Finally, the transformation algorithm generates an out(channel, $\text{msg}_{\text{secured}}$) construct, where c is the channel used for the communication between the participants and $\text{msg}_{\text{secured}}$ is the bitstring to be transmitted.

Translate a MessageEvent with type MessageIn The transformation of a MessageEvent with type MessageIn encompasses several steps. First, the bitstring of the message is received by means of the PROVERIF process macro in(channel, msg: bitstring). Second, the transformation algorithm checks whether security primitives are applied to the MessageEvent. Third, the received bitstring is disassembled into its Arguments. Subsequently, we present three different cases for the transformation of security primitives.

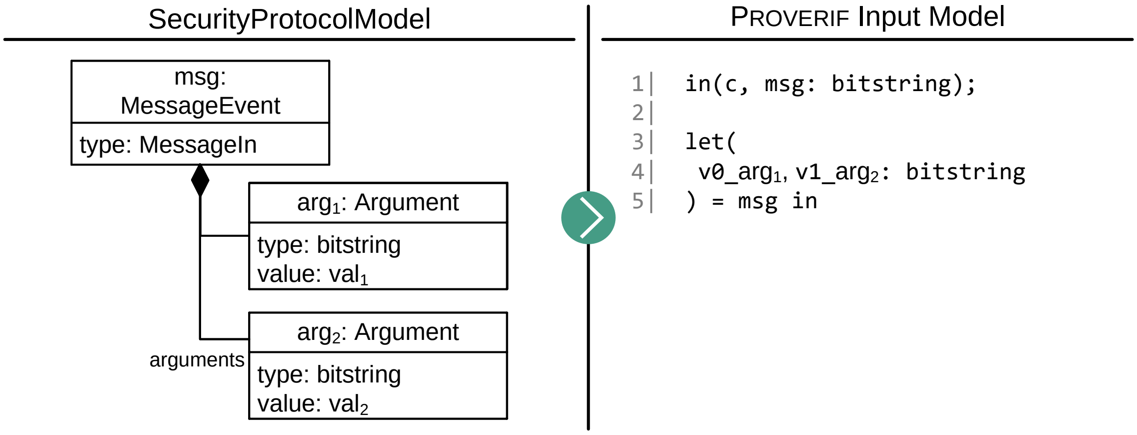
No applied security primitive The transformation algorithm stores the received bitstring in the variable msg (cf. in(c, msg: bitstring) in Figure 4.22). Then, the transformation algorithm resolves the arguments of the MessageEvent. Therefore, it checks the number of arguments the MessageEvent contains. If the MessageEvent contains only one argument, the transformation algorithm creates a new variable for the argument and assigns the received message msg to this variable. If the MessageEvent contains more than one argument, the transformation algorithm creates one variable for each argument (cf. let $(\text{v0\_arg}_1, \text{v1\_arg}_2) = \text{msg}$ in in Figure 4.22). As for the trans-



```
let v0_arg = val in

let msg = (v0_arg) in

let msg^1_tmp = sign(
  msg, privKeyS
) in

let msg^2_tmp = aenc(
  msg^1_tmp, pubKeyR
) in

let msg_secured = msg^2_tmp in

out(c, msg_secured)
```

Figure 4.21: Illustration of Transformation Step 3: Generate Protocol Behavior (PROVERIF) — Translation of an outgoing MessageEvent with more than one applied security primitives

formation of a MessageEvent with type MessageOut, we use the prefix v and a consecutive number X to enable that the same argument name or the repeated occurrence of a MessageEvent does not result in a PROVERIF warning due to the variable being rebounded.

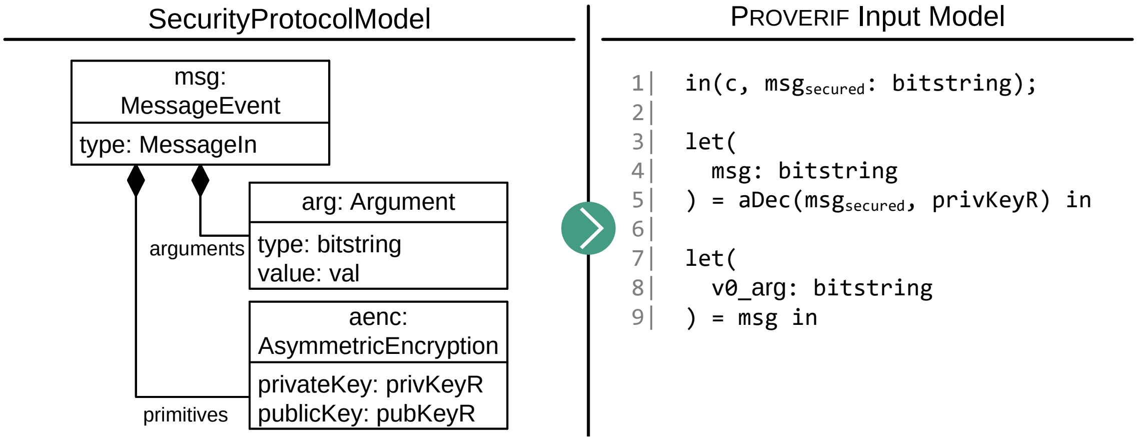


```
in(c, msg: bitstring);

let(
 v0_arg_1, v1_arg_2: bitstring
) = msg in
```

Figure 4.22: Illustration of Transformation Step 3: Generate Protocol Behavior (PROVERIF) — Translation of an incoming MessageEvent without applied security primitive

One applied security primitive The transformation algorithm stores the received bitstring in the variable msg$_{secured}$ (cf. in(c, msg$_{secured}$: bitstring) in Figure 4.23). Next, the transformation algorithm resolves the destructor that corresponds to the security primitive (e.g., adec for the security primitive AsymmetricEncryption) using the message variable msg$_{secured}$ and the key referenced by the primitive as input. The result is then assigned to the variable(s) representing the argument(s) (cf. let

$\text{msg} = \text{adec}(\text{msg}_{\text{secured}}, \text{privKeyR})$ in in Figure 4.23). Afterward, the transformation algorithm resolves the arguments of the MessageEvent as described before.



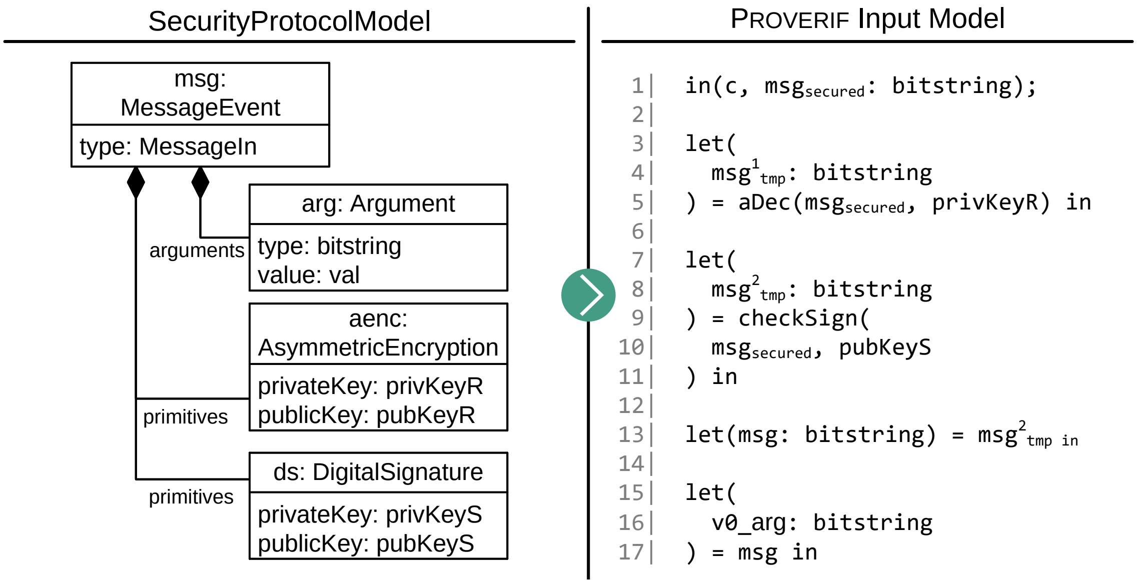
```
in(c, msg_secured: bitstring);

let(
  msg: bitstring
) = aDec(msg_secured, privKeyR) in

let(
  v0_arg: bitstring
) = msg in
```

Figure 4.23: Illustration of Transformation Step 3: Generate Protocol Behavior (PROVERIF) — Translation of an incoming MessageEvent with one applied security primitive

More than one applied security primitive The transformation algorithm stores the received bitstring in the variable $\text{msg}_{\text{secured}}$ (cf. $\text{in}(\text{c}, \text{msg}_{\text{secured}}\colon \text{bitstring})$ in Figure 4.24). Next, the transformation algorithm iterates over the ordered set of security primitives but in inverse order. For each security primitive, it resolves the destructor that corresponds to the security primitive and stores the result in a temporal variable. For example, as depicted in Figure 4.24, the result of the validation of the signature is stored in the variable $\text{msg}^{1}_{\text{tmp}}$. This variable is then used as the input for the transformation of the next primitive. After transforming all security primitives, the result is stored in the variable msg. Afterward, the transformation algorithm resolves the arguments of the MessageEvent as described before.

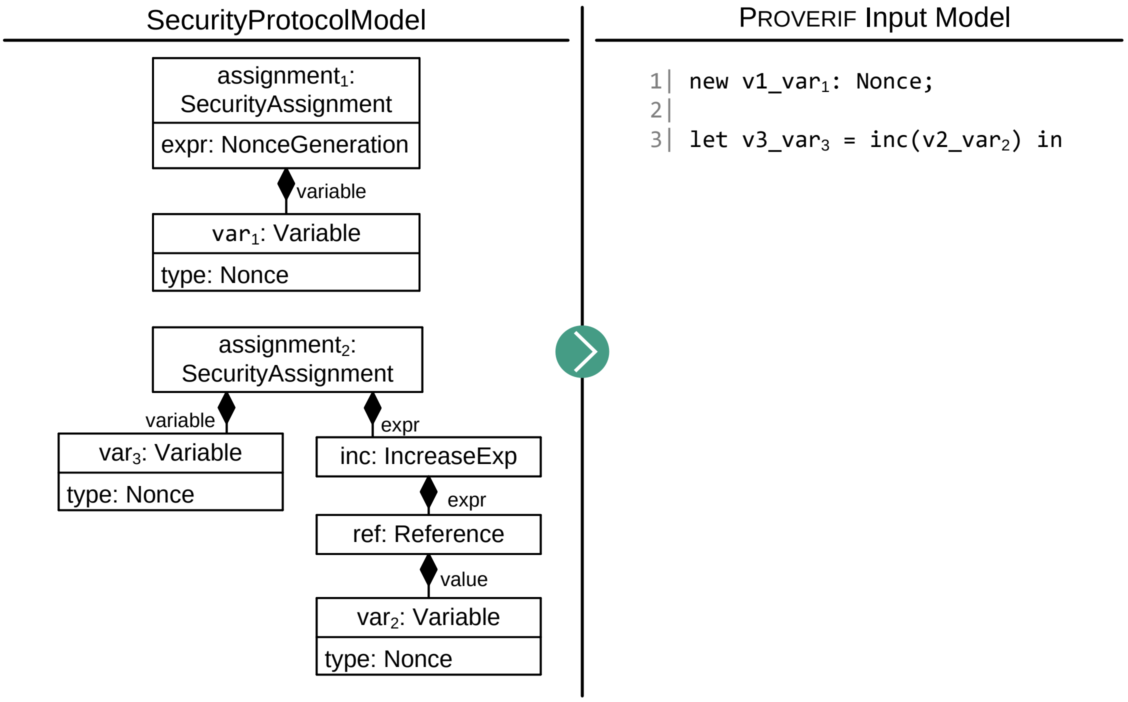
Translate a SecurityAssignment As described in Section 3.4.4, a SecurityAssignment is used to create a new variable or modify an existing variable. In both cases, the SecurityAssignment encompasses a Variable and an Expression. If the SecurityAssignment describes the creation of a variable, the transformation algorithm creates a new variable of the form new vX_var: <type>, where v is used as a prefix, X is a consecutive number, <var> is the name of the variable, and <type> is the type of the variable. Otherwise, the transformation algorithm creates a new variable of the form let vX_var = <expression> in, where <expression> is the transformed Expression.

For example, in Figure 4.25, the SecurityProtocolModel encompasses two SecurityAssignments. The first SecurityAssignment creates a new variable of type Nonce. The second SecurityAssignment increases the variable var_2 by one and assigns it to the variable var_3. As output, the transformation algorithm creates the corresponding PROVERIF constructs: new $\text{v1\_var}_1\colon$ Nonce for the first SecurityAssignment and let $\text{v3\_var}_3 = \text{inc}(\text{v2\_var}_2)$ in for the second SecurityAssignment.



```
in(c, msg_secured: bitstring);

let(
  msg^1_tmp: bitstring
) = aDec(msg_secured, privKeyR) in

let(
  msg^2_tmp: bitstring
) = checkSign(
  msg_secured, pubKeyS
) in

let(msg: bitstring) = msg^2_tmp in

let(
  v0_arg: bitstring
) = msg in
```

Figure 4.24: Illustration of Transformation Step 3: Generate Protocol Behavior (PROVERIF) — Translation of an incoming MessageEvent with more than one applied security primitive

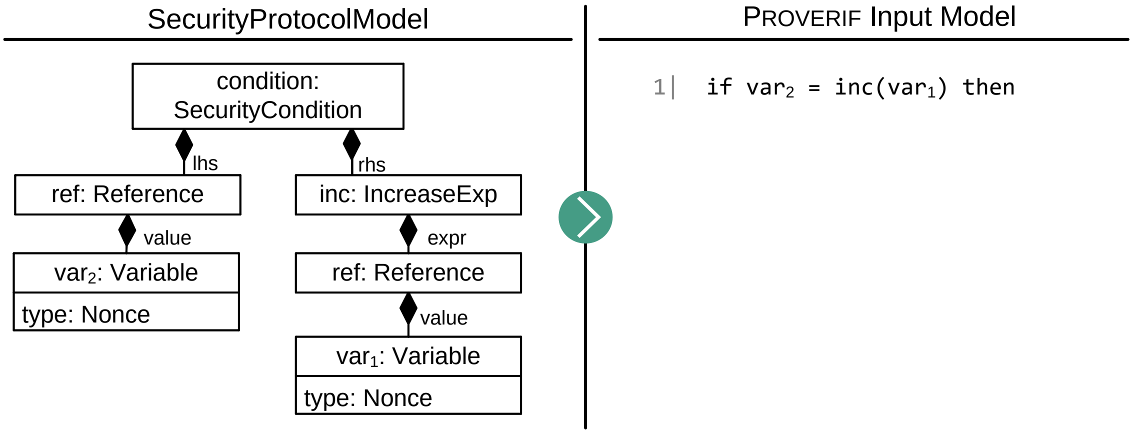


```
new v1_var_1: Nonce;

let v3_var_3 = inc(v2_var_2) in
```

Figure 4.25: Illustration of Transformation Step 3: Generate Protocol Behavior (PROVERIF) — Translation of a SecurityAssignment

Translate a SecurityCondition As described in Section 3.4.5, a SecurityCondition is used to describe the conditional behavior of a security protocol. If it evaluates to true, the security protocol proceeds, otherwise the run of the security protocol terminates. In PROVERIF, a condition has the form if <Expression> then <P> else <Q>, where P and Q are sub-processes. However, if no process is executed in the else part, the else part can be omitted. Hence, the transformation algorithm only generates the if <Expression> then <P> part of the condition.

As shown in Figure 4.26, a SecurityCondition encompasses a ComparisonOp, a left-hand side and a right-hand side of type Expression. However, PROVERIF only supports the ComparisonOp = "=", so SecurityConditions with other operators would result in an error. In the example, the left-hand side refers to a variable var_1 and the right-hand side to an IncreaseExp increasing the variable var_2. As output, the transformation algorithm creates the corresponding PROVERIF construct: if v1_var_1 = inc(v2_var_2) then.

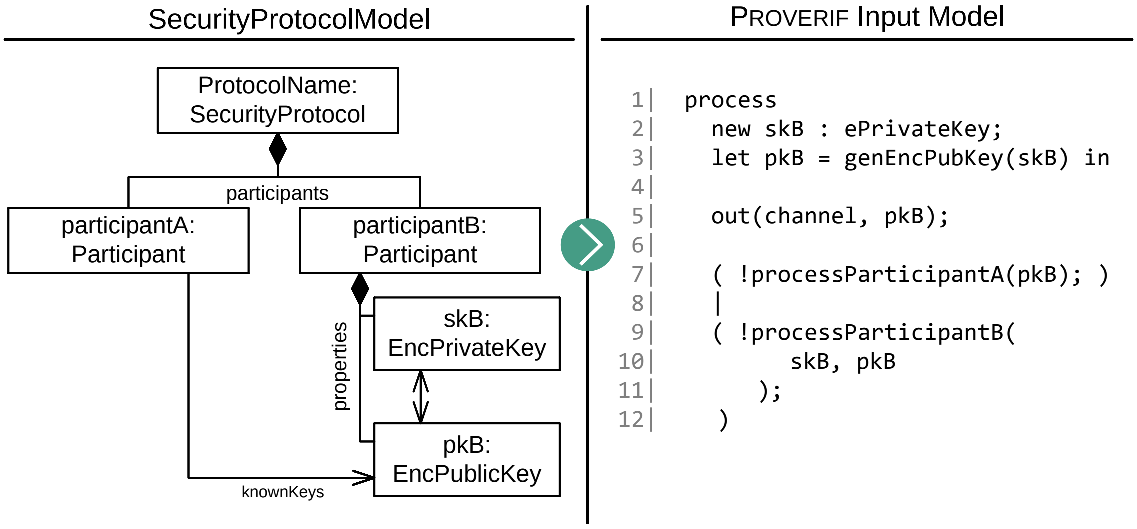


```
if var₂ = inc(var₁) then
```

Figure 4.26: Illustration of Transformation Step 3: Generate Protocol Behavior (PROVERIF) — Translation of a SecurityCondition

(4) Generate Main Process

Finally, in the last step, the transformation algorithm generates the main process. Therefore, it creates free variables for all cryptographic keys used in the security protocols. Furthermore, it creates references to the sub-processes describing the behavior of the protocol participants. For example, in Figure 4.27, the SecurityProtocolModel encompasses the two participants participantA and participantB. participantB possesses an asymmetric encryption key pair. Moreover, participantA knows the public key of participantB. As output, the transformation algorithm creates a new process with the PROVERIF macro process (cf. line 1). In addition, the transformation algorithm creates the properties for each participant. In the example, it creates the private key skB: ePrivateKey (cf. line 2). The private key is used as input for the function genPublicKey to generate the public key (cf. line 3). Afterward, the public key is sent over the channel such that all participants are aware of them (cf. line 5). Finally, the transformation algorithm calls the sub-process for each participant with the corresponding input variables (cf. lines 7-12).



```
process
  new skB : ePrivateKey;
  let pkB = genEncPubKey(skB) in

  out(channel, pkB);

  ( !processParticipantA(pkB); )
  |
  ( !processParticipantB(
        skB, pkB
     );
   )
```

Figure 4.27: Illustration of Transformation Step 4: Generate Main Process (PROVERIF)

4.4.3 Translate the QueryModel to PROVERIF

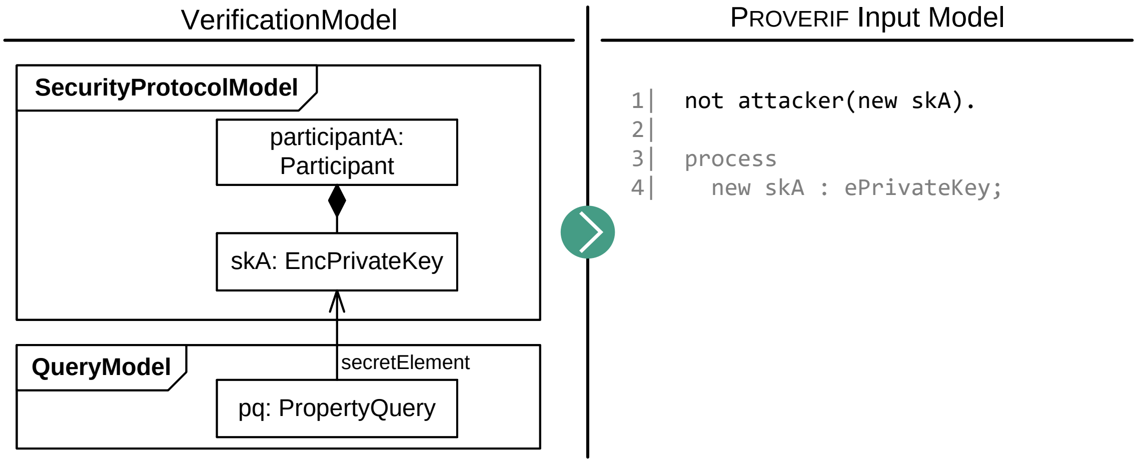
This section describes the generation of queries to enable the analysis of the security properties secrecy and authentication.

Translate a SecrecyQuery

The transformation algorithm adds secrecy queries and secrecy assumptions to the PROVERIF input model to enable the automated analysis of whether protocol variables are kept secret. As explained in Section 4.3.1, we distinguish PropertyQueries and VariableQueries.

PropertyQueries refer to a Property of a Participant. According to our transformation rules described before, the transformation algorithm creates the properties of a participant during the generation of the main process. Variables representing these properties are globally visible in the PROVERIF model. Thus, it is sufficient to add a secrecy assumption to the input model. For example, in Figure 4.28, the private asymmetric encryption key skA of the participantA should be kept secret. Thus, the transformation algorithm creates a new secrecy assumption not attacker(new skA). to the input model.

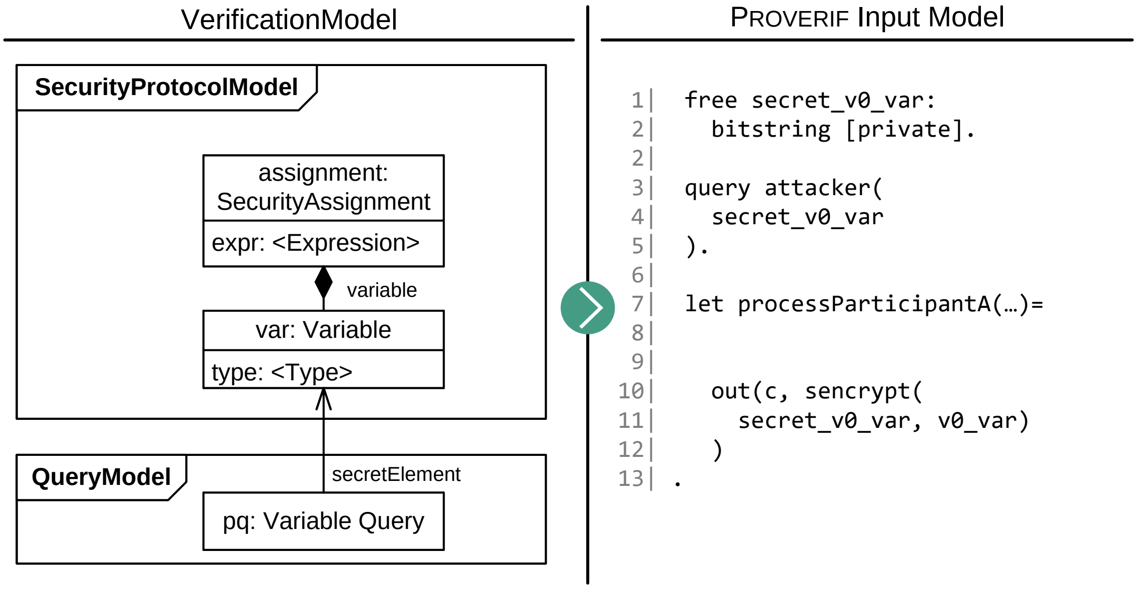
VariableQueries refer to a Variable declared in a PROVERIF sub-process representing a participant of the security protocol. The variable is only visible within this sub-process. Thus, it is not sufficient to only add the query to the input model. Instead, there are two possibilities for a security engineer to define a secrecy query. In the first possibility, the security engineer can introduce a dummy variable for each secret element. In this case, the dummy variable is symmetrically encrypted with the secret element and sent over the public channel at the end of the sub-process. If attackers can determine the content of the dummy variable, they can determine the key and compromise the secrecy of the secret element. In the second possibility, the security engineer can define the variable outside the sub-process as a global variable. In this case, the transformation algorithm could generate a secrecy assumption for the variable as described for the PropertyQueries.



```
not attacker(new skA).

process
  new skA : ePrivateKey;
```

Figure 4.28: Translating a PropertyQuery to PROVERIF

In our transformation algorithm, we use the first possibility. Thus, as shown in Figure 4.29, to create a secrecy query referencing a variable v0_var, the transformation algorithm creates a new free and private variable secret_v0_var of type bitstring (cf. free secret_v0_var: bitstring [private].). Moreover, it extends the description of the sub-process by the PROVERIF macro out(c, sencrypt(secret_v0_var, v0_var)); indicating that the publicly visible variable is symmetrically encrypted with the variable that should be kept secret. Thus, if the attacker is able to learn secret_v0_var, this means that he/she was also able to learn v0_var during the execution of the protocol.



```
free secret_v0_var:
  bitstring [private].

query attacker(
  secret_v0_var
).

let processParticipantA(…)=


  out(c, sencrypt(
    secret_v0_var, v0_var)
  )
.
```

Figure 4.29: Translating a VariableQuery to PROVERIF

Translate an AuthenticationQuery

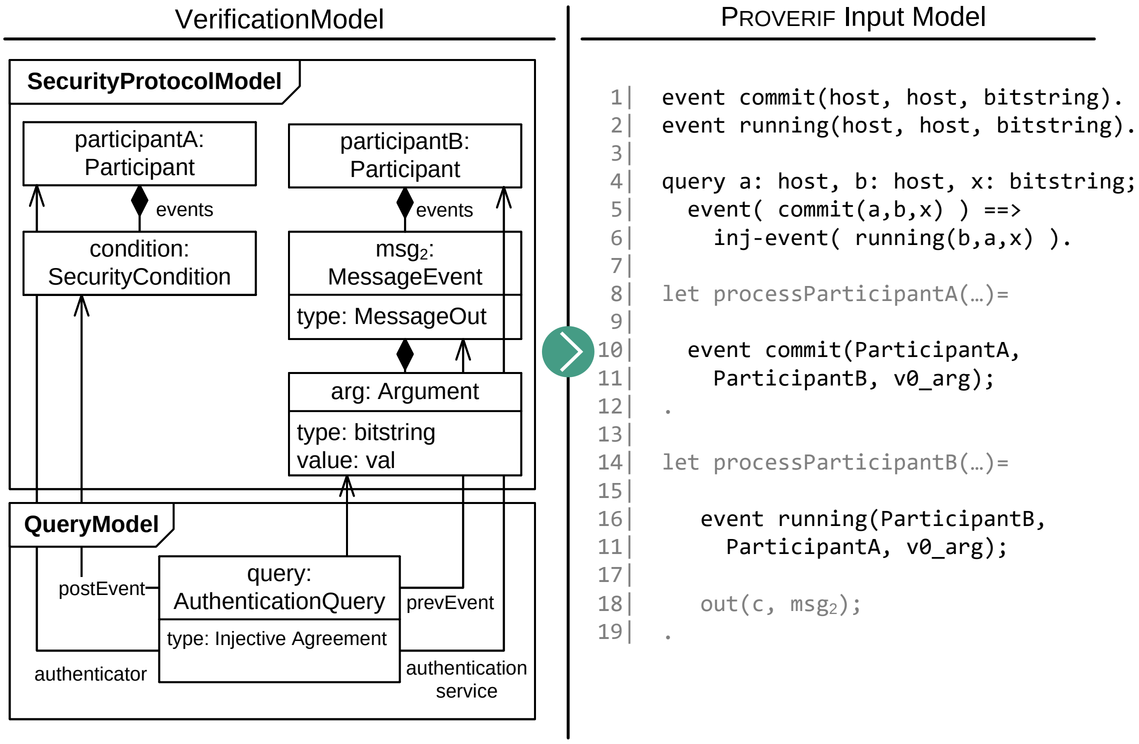
The transformation algorithm adds correspondence assertions to the PROVERIF input model to enable the automated analysis of authentication properties. Listing 4.1 depicts the formalization of the hierarchy of authentication specifications in PROVERIF.

```
(* aliveness *):
event create(host).
event commit(host, host).

query a: host, b: host;
  event(commit(a,b)) ==> event(create(b)).

(* weak agreement *)
event running(host, host).
event commit(host, host).

query a: host, b: host;
  event(commit(a,b)) ==> event(running(b,a)).

(* non-injective agreement *)
event commit(host, host, bitstring);
event running(host, host, bitstring);

query a: host, b: host, x: bitstring;
  event(commit(a,b,x)) ==> event(running(b,a,x)).

(* injective agreement *)
event commit(host, host, bitstring);
event running(host, host, bitstring);

query a: host, b: host, x: bitstring;
  event(commit(a,b,x)) ==> inj-event(running(b,a,x)).

```

Listing 4.1: Overview of Authentication Queries in PROVERIF

The transformation algorithm creates authentication queries as shown in Figure 4.30 and explained as follows: The transformation algorithm creates the events (cf. lines 1–2) including arguments for non-injective and injective authentication queries. Next, it adds the query to define the correspondence assertions (cf. lines 4–9) following the template depicted in Listing 4.1. For the three non-injective level of the authentication hierarchy, we use the PROVERIF macro event after the arrow, and for the injective agreement, we use the macro inj-event, indicating the 1:1-relationship between the two events commit and running.

Moreover, the transformation algorithm annotates the occurrence of events in the description of the behavior. Therefore, it adds the event commit at the end of the sub-process for participantA (cf. line 13) and the event running before the message described by the property prevEvent of the AuthenticationQuery is sent. In the case of non-injective and injective authentication queries, the arguments are included in the event definition.



```
event commit(host, host, bitstring).
event running(host, host, bitstring).

query a: host, b: host, x: bitstring;
  event( commit(a,b,x) ) ==>
    inj-event( running(b,a,x) ).

let processParticipantA(…)=

  event commit(ParticipantA,
    ParticipantB, v0_arg);
.

let processParticipantB(…)=

   event running(ParticipantB,
     ParticipantA, v0_arg);

   out(c, msg2);
.
```

Figure 4.30: Translating an AuthenticationQuery to PROVERIF

4.5 Translation from the VerificationModel to TAMARIN input models

In the fourth step, Generate Verification Input (cf. Figure 4.1 on page 59), VICE uses the VerificationModel and generates the input for the used symbolic model checker. This section describes the details of the input generation for TAMARIN. First, we recap the capabilities of TAMARIN and discuss whether all elements of the SECURITY MODELING PROFILE are supported. Second, we present the model-to-text transformation to derive the input for TAMARIN.

4.5.1 Overview of the Capabilities of TAMARIN in relation to the SECURITY MODELING PROFILE

Table 4.2 provides an overview of the SECURITY MODELING PROFILE's features and whether TAMARIN is able to analyze security protocols that contain these features. As presented in Section 2.2.4, TAMARIN is a tool for automated analysis of security protocols within the symbolic model. In the following, we briefly describe how we realize the features of the SECURITY MODELING PROFILE using TAMARIN language constructs.

TAMARIN uses multiset rewriting rules to model the behavior of the security protocols. We can use action facts on the left-hand side of a rewriting rule to specify properties a participant may have or know. For example, as depicted in Listing 2.8, the public key of a participant R can be specified as !Pk($R, pkR). Moreover, we can use a public variable (denoted by $) to identify and reference participants. Thus, although TAMARIN does not provide language constructs to explicitly model the behavior of a participant, we can conclude that TAMARIN supports all features of the category *Protocol Modeling*.

TAMARIN provides built-in types and functions for cryptographic primitives and arithmetic expressions. The built-ins support the features of the categories *Cryptographic Primitives* and *Data types* completely. Additionally, TAMARIN supports the declaration of custom functions. Hence, we conclude that TAMARIN also supports all features of the category *Expressions*. However, as PROVERIF, TAMARIN does not support the consideration of time, so it is not possible to compare timestamps with each other or the current time.

TAMARIN enables the declaration of variables by means of the Fr operator and the modification of existing variables by means of the let-expressions. Furthermore, TAMARIN supports several common restrictions that can be used to model the conditional behavior of the security protocol. Hence, we conclude that TAMARIN supports all features of the categories *Assignments* and *Conditions*.

Table 4.2: Overview of the Capabilities of TAMARIN in relation to the SECURITY MODELING PROFILE

Category	**Feature of the SECURITY MODELING PROFILE**	**Feature is supported by TAMARIN**
Protocol Modeling	Participants exchange messages to execute the protocol	✓
	Participants can be identified and referenced	✓
	Participants own properties like cryptographic keys	✓
	Participants know properties of other participants	✓
Cryptographic Primitives	Asymmetric Encryption	✓
	Digital Signature	✓
	HMAC	✓
	Hashing	✓
Assignments	Create a new variable	✓
	Modify an existing variable	✓
Conditions	Conditions with the common comparison operators	✓
Data types	Cryptographic Keys	✓
	Nonce	✓
	Number	✓
	Prime	✓
	Timestamp	✓

Category	Feature of the SECURITY MODELING PROFILE	Feature is supported by TAMARIN
Expressions	Arithmetic Expressions	✓
	Cryptographic Expression	✓

4.5.2 Translate the SecurityProtocolModel to TAMARIN

This section presents the functional principle of the model-to-text transformation from the SecurityProtocolModel to TAMARIN input models. Algorithm 4.3 depicts an overview of the model-to-text transformation encompassing three main steps.

Algorithm 4.3 Translation from a SecurityProtocolModel to TAMARIN

```
Input: SecurityProtocolModel
Output: TAMARIN Input Model
▷ ① Generate Protocol Preamble
generateProtocolPreamble()
▷ ② Generate Protocol Structure
for each participant ∈ SecurityProtocolModel.participants do
    createInitRules(participant)
end for
for each event ∈ SecurityProtocolModel.events do
    ▷ ③ Generate Protocol Behavior
    generateBehaviorForParticipant(participant)
end for
```

At the beginning, the transformation algorithm generates a preamble encompassing all definitions necessary to specify the security protocol (cf. ① Generate Protocol Preamble). Then, the transformation algorithm generates structural information (cf. ② Generate Protocol Structure). Afterward, the transformation algorithm iterates over all events contained in the SecurityProtocolModel and creates corresponding rewriting rules (cf. ③ Generate Protocol Behavior). "A rewrite rule [...] has a name and three parts, each of which is a sequence of facts: one for the left-hand side, one labeling the transition [...], and one for the rule's right-hand side" [Tamarin]. TAMARIN uses state facts "that indicate that a certain process is at a specific point in its execution" [Tamarin] to capture the progress of a security protocol. A state fact is used on the left-hand side of a rule to model a precondition that must hold for the rule to fire and on the right-hand side to express the postcondition after the rule has fired. A state fact has the following form in our transformation algorithm $\mathsf{State}(\$\mathsf{Sender}, \$\mathsf{Receiver}, \sim\mathsf{tid}, <\mathsf{T}_1,\ldots,\mathsf{T}_n>)$, where $\sim\mathsf{tid}$ is a unique thread identifier, $\$\mathsf{Sender}$ is the sender of the current message, $\$\mathsf{Receiver}$ is the receiver of the current message event, and $<\mathsf{T}_1,\ldots,\mathsf{T}_n>$ is a set of arguments and variables that have been exchanged or created during the previous message exchange. Listing 4.2 depicts the generic rule template that we use throughout our transformation approach.

```
rule <<event.owner>>_<<event.name>>:
let
(* Let-binding expressions are used to specify local macros, for
  instance, to assign newly created values to variable names or to
  handle cryptographic primitives *)
in
[
(* The left-hand side of the rewriting rule may encompass a state
  fact to model the precondition of a rule, the receiving of an
  incoming message, and the creation of fresh values. *)
]--[
(* The labeling part of the rewriting rule may encompass restrictions
   to model the conditional behavior of the security protocol, and
  facts that mark important assumptions of the security protocol's
  behavior. These facts can be used in lemmas to prove the security
  properties (e.g., secrecy and authentication) of the security
  protocol. *)
]->[
(* The right-hand side of the rewriting rule may encompass the
  sending of an outgoing message and a state fact to model the
  postcondition of the rule. *)
]
```

Listing 4.2: TAMARIN Template used for the Transformation Approach

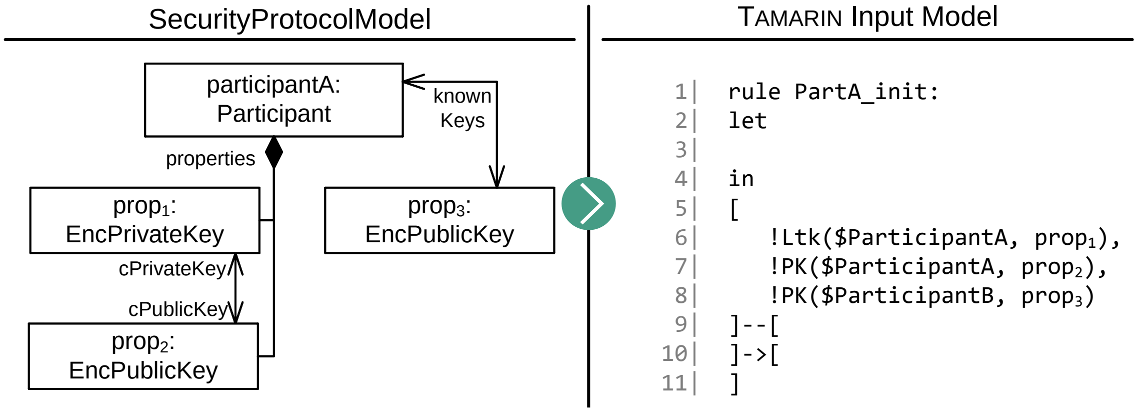
① Generate Protocol Preamble

In the first step, the transformation algorithm generates a preamble for the TAMARIN input model. As explained in Section 2.2.4, TAMARIN provides a set of built-in types that can be used to model a security protocol. Hence, the transformation algorithm creates a static preamble that imports all built-in types that are used by the security protocol. Moreover, in TAMARIN, the security engineer can use restrictions to model the conditional behavior of the security protocol. There is a set of recommended common restrictions (e.g., unique, equality). Our transformation approach adds the restrictions to the preamble that correspond to the expression of security conditions. The complete preamble is shown in Appendix B.3.

② Generate Protocol Structure

In the second step, the transformation algorithm generates structural information about the security protocol. Therefore, the transformation algorithm creates an initialization rule for all Participants contained in the SecurityProtocolModel. Within these initialization rules, the transformation algorithm creates persistent action facts for owned and known keys of the Participants.

For example, in Figure 4.31, the SecurityProtocolModel encompasses the participant participantA. The participantA has two properties prop$_1$: EncPrivateKey and prop$_2$: EncPublicKey, and knows the public key prop$_3$: EncPublicKey. Thus, the transformation algorithm creates the persistent action facts !Ltk($ParticipantA, prop$_1$), !PK($ParticipantA, prop$_2$), and !PK($ParticipantB, prop$_3$) as part of the left-hand side of the initialization rule.



```
rule PartA_init:
let

in
[
   !Ltk($ParticipantA, prop₁),
   !PK($ParticipantA, prop₂),
   !PK($ParticipantB, prop₃)
]--[
]->[
]
```

Figure 4.31: Illustration of Transformation Step 2: Generate Protocol Structure (TAMARIN)

(2) Generate Protocol Behavior

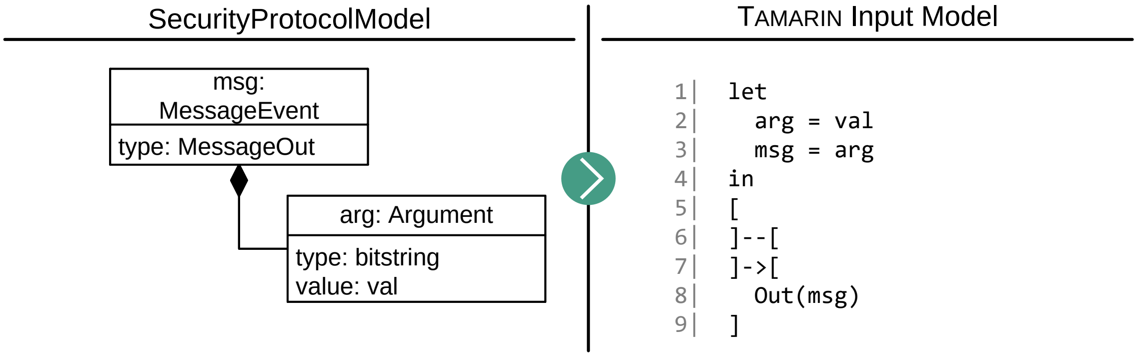
In the third step, the transformation algorithm generates the security protocol's behavior. As stated in Chapter 3, a security protocol in our SecurityProtocolModel encompasses at least two participants whereas each participant encompasses a set of events to model its behavior. Even though it would be possible to model an incoming and an outgoing message in one rewriting rule in TAMARIN, we decided for the sake of simplicity and readability to model each message event separately in one rewriting rule and relate the rules with additional state facts. Moreover, since there are no dedicated rewriting rules that handle only security conditions or security assignments, the transformation algorithm collects these events while iterating over the list of events of a participant and transforms them when transforming the next message event. Subsequently, we explain the transformation of each event type separately.

Translate a MessageEvent As mentioned in Section 2.2.4, in TAMARIN, a message is transmitted via a channel. The sender uses the action fact Out(msg) to send a message and the receiver uses the action fact In(msg) to receive a message.

Translate a MessageEvent with type MessageOut The transformation of a MessageEvent with type MessageOut encompasses several steps. First, the transformation algorithm handles the Arguments of the MessageEvent. Therefore, the transformation algorithm retrieves a list of arguments and for each argument, it creates a new variable and assigns the actual value to the variable (cf. arg = val in Figure 4.32 – Figure 4.34) as part of the let-expression.

Second, the transformation algorithm checks if security primitives are applied to the arguments of the message or the complete message. The transformation of security primitives applied to arguments or messages follows the same functional principle. Thus, we only present the transformation rules for security primitives applied to messages in this section and distinguish the following three cases:

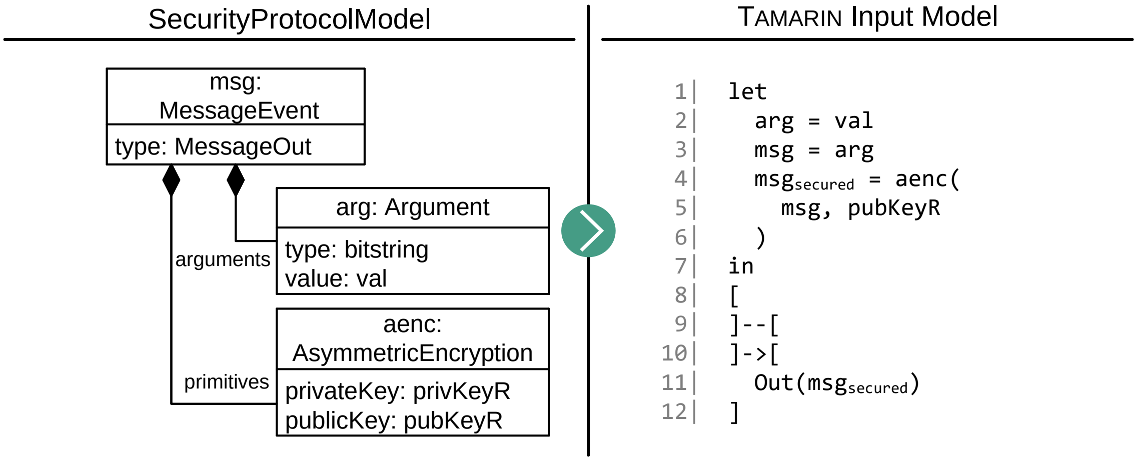
No applied security primitive The transformation algorithm creates a new variable msg representing the MessageEvent as part of the let-expression and assigns the set of all argument variables to this variable. If the MessageEvent contains more than one Argument, the variables are concatenated and assigned to the variable representing the message (e.g., $\mathsf{msg} = <\mathsf{arg}_1, \mathsf{arg}_n>$). Otherwise, the variable representing the argument is directly assigned to the newly created variable (cf. msg = arg in Figure 4.32). Finally, the transformation algorithm generates an Out(msg) construct as part of the right-hand side of the rewriting rule.



```
let
  arg = val
  msg = arg
in
[
]--[
]->[
  Out(msg)
]
```

Figure 4.32: Illustration of Transformation Step 3: Generate Protocol Behavior (Tamarin) — Translation of an outgoing MessageEvent without applied security primitive

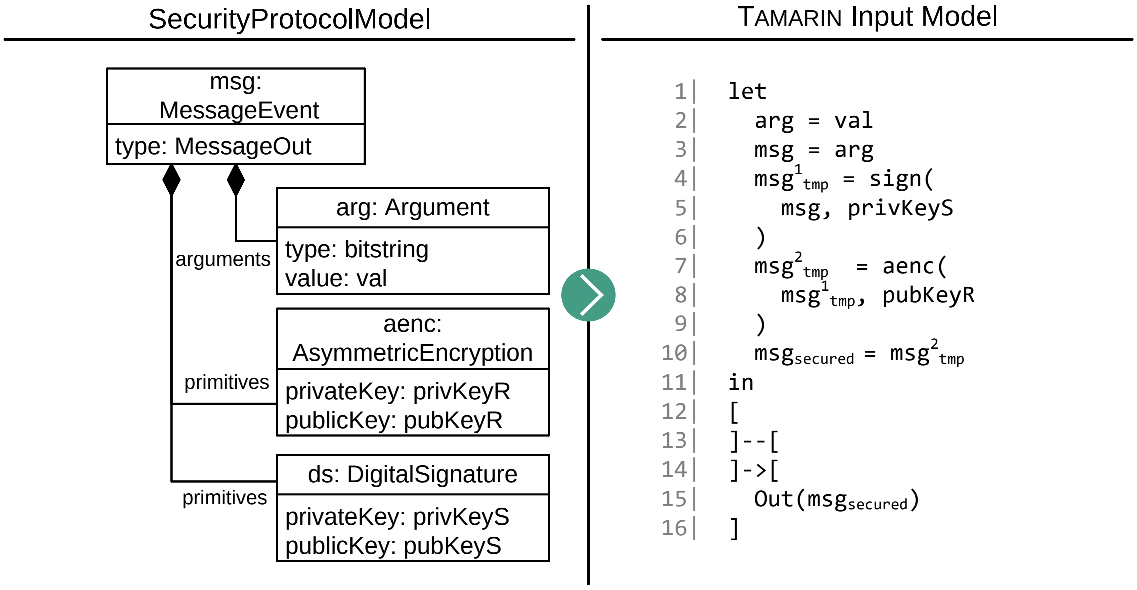
One applied security primitive The transformation algorithm creates a new variable msg representing the MessageEvent as part of the let-expression and assigns the set of all argument variables to this variable. Next, it resolves the constructor that belongs to the primitive (e.g., aenc for the security primitive AsymmetricEncryption) and applies the constructor using the variable msg and the key referenced by the primitive as input (cf., $\mathsf{msg}_{\mathsf{secured}} = \mathsf{aenc}(\mathsf{msg}, \mathsf{pubKeyR})$ in Figure 4.33). Finally, the transformation algorithm generates an $\mathsf{Out}(\mathsf{msg}_{\mathsf{secured}})$ construct as part of the right-hand side of the rewriting rule.



```
let
  arg = val
  msg = arg
  msg_secured = aenc(
    msg, pubKeyR
  )
in
[
]--[
]->[
  Out(msg_secured)
]
```

Figure 4.33: Illustration of Transformation Step 3: Generate Protocol Behavior (Tamarin) — Translation of an outgoing MessageEvent with one applied security primitive

More than one applied security primitive The transformation algorithm creates a new variable msg representing the MessageEvent as part of the let-expression and assigns the set of all argument variables to this variable. Next, the transformation algorithm iterates over the ordered set of security primitives. As described in the previous case, the transformation algorithm resolves the TAMARIN constructor that belongs to the security primitive. However, in case of more than one applied security primitive, the transformation algorithm uses temporal variables to store the result of the transformation of one security primitive (cf. $\mathsf{msg}^1_{\mathsf{tmp}} = \mathsf{sign}(\mathsf{msg}, \mathsf{privKeyS})$ in Figure 4.34). This variable is then used as the input for the transformation of the next security primitive (cf. $\mathsf{msg}^2_{\mathsf{tmp}} = \mathsf{aenc}(\mathsf{msg}^1_{\mathsf{tmp}}, \mathsf{pubKeyR})$ in Figure 4.34). After transforming all security primitives, the result is stored in the variable $\mathsf{msg}_{\mathsf{secured}}$. Finally, the transformation algorithm generates an $\mathsf{Out}(\mathsf{msg}_{\mathsf{secured}})$ construct as part of the right-hand side of the rewriting rule.



```
let
  arg = val
  msg = arg
  msg^1_tmp = sign(
    msg, privKeyS
  )
  msg^2_tmp  = aenc(
    msg^1_tmp, pubKeyR
  )
  msg_secured = msg^2_tmp
in
[
]--[
]->[
  Out(msg_secured)
]
```

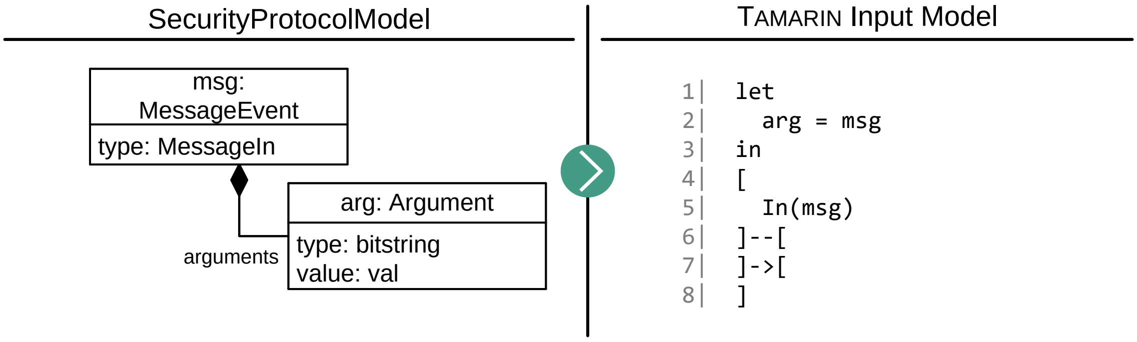
Figure 4.34: Illustration of Transformation Step 3: Generate Protocol Behavior (TAMARIN) — Translation of an outgoing MessageEvent with more than one applied security primitive

Translate a MessageEvent with type MessageIn The transformation of a MessageEvent with type MessageIn encompasses several steps. The transformation steps are similar to the transformation steps presented for a MessageEvent with type MessageOut.

First, the message is received by means of the TAMARIN action fact In(msg). Second, the transformation algorithm checks whether security primitives are applied to the MessageEvent. We distinguish three cases explained subsequently.

No applied security primitive: The transformation algorithm stores the received message in the variable msg (cf. In(msg) in Figure 4.35). Next, the transformation algorithm resolves the arguments of the MessageEvent. We distinguish three different cases based on the number of Arguments:

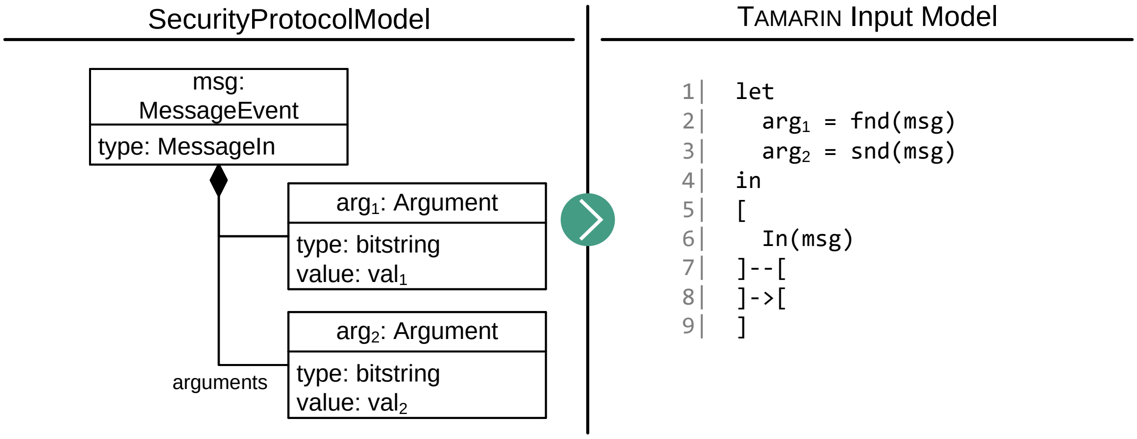
One Argument If the MessageEvent contains one argument, the transformation algorithm creates a new variable for the argument and assigns the received message msg to this variable (cf. $\mathsf{arg} = \mathsf{msg}$ in Figure 4.35).



```
let
  arg = msg
in
[
  In(msg)
]--[
]->[
]
```

Figure 4.35: Illustration of Transformation Step 3: Generate Protocol Behavior (TAMARIN) — Translation of an incoming MessageEvent with one Argument

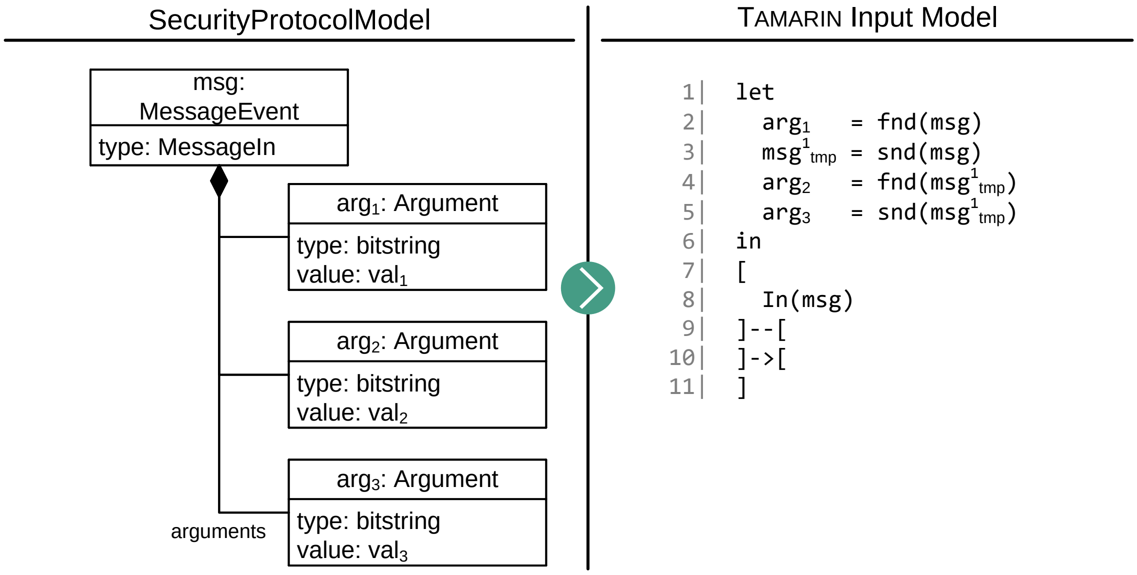
Two Arguments If the MessageEvent contains two arguments, the transformation algorithm creates a new variable for each argument and assigns the first part of the received message to the first argument variable (cf. $\mathsf{arg}_1 = \mathsf{fst}(\mathsf{msg})$ in Figure 4.36) and the second part of the received message to the second argument variable (cf. $\mathsf{arg}_2 = \mathsf{snd}(\mathsf{msg})$ in Figure 4.36).



```
let
  arg_1 = fnd(msg)
  arg_2 = snd(msg)
in
[
  In(msg)
]--[
]->[
]
```

Figure 4.36: Illustration of Transformation Step 3: Generate Protocol Behavior (TAMARIN) — Translation of an incoming MessageEvent with two Arguments

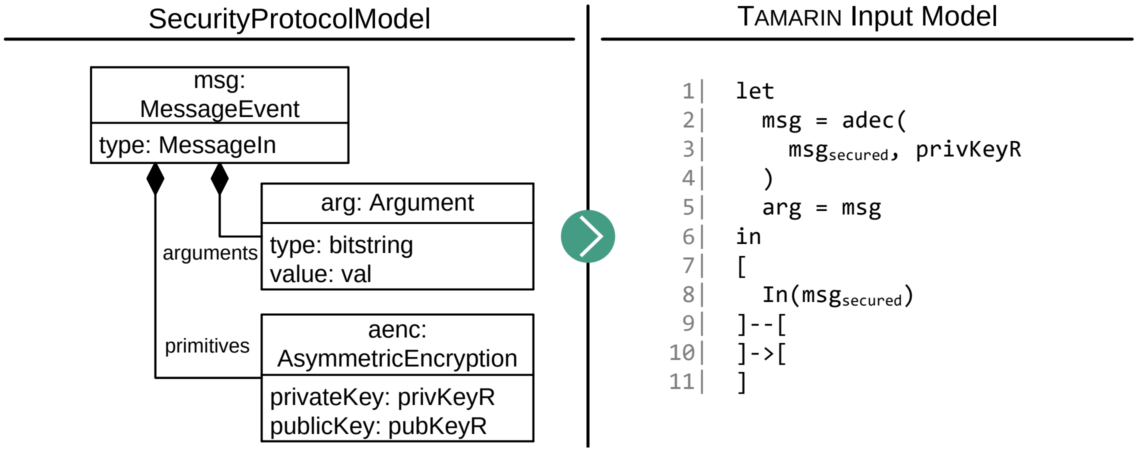
More than two Arguments If the MessageEvent contains more than two arguments, the transformation algorithm disassembles the first part of the message to the argument (cf. $\mathsf{arg}_1 = \mathsf{fst}(\mathsf{msg})$ in Figure 4.37) and the second part (= containing all other arguments) to a temporal variable (cf. $\mathsf{msg}^1_{\mathsf{arg}} = \mathsf{snd}(\mathsf{msg})$ in Figure 4.37). Next, the first part of the temporal variable is assigned to the next argument and the remainder to a new temporal variable. This procedure is repeated until all arguments of the message are assigned.



```
let
  arg1     = fnd(msg)
  msg1_tmp = snd(msg)
  arg2     = fnd(msg1_tmp)
  arg3     = snd(msg1_tmp)
in
[
  In(msg)
]--[
]->[
]
```

Figure 4.37: Illustration of Transformation Step 3: Generate Protocol Behavior (TAMARIN) — Translation of an incoming MessageEvent with more than two Argument

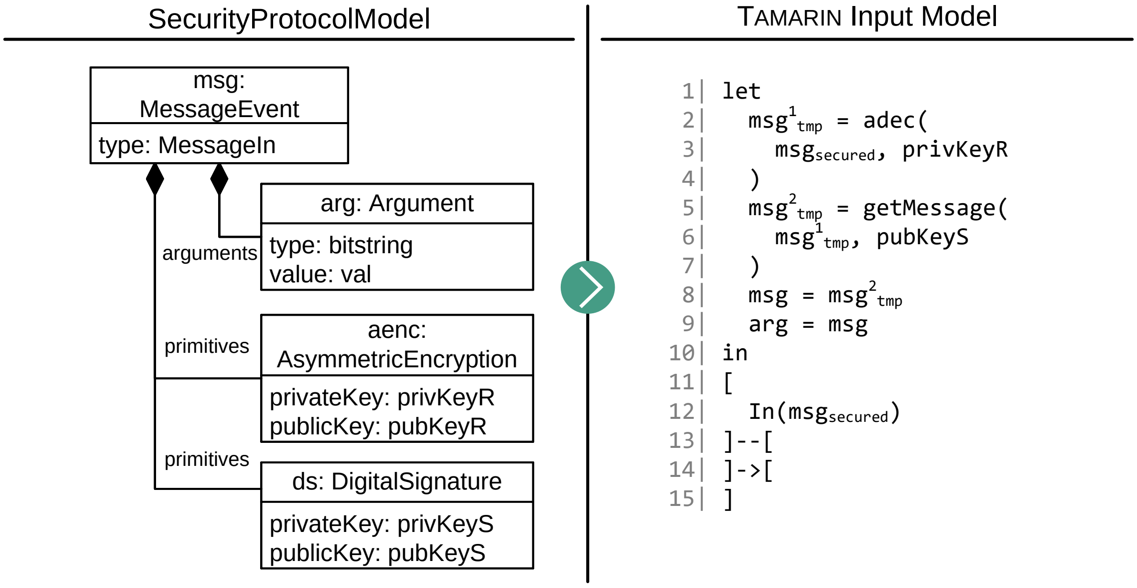
One applied security primitive The transformation algorithm stores the received bitstring in the variable $msg_{secured}$ (cf. In($msg_{secured}$) in Figure 4.38). Next, the transformation algorithm resolves the destructor that corresponds to the security primitive (e.g., adec for asymmetric encryption) using the message variable $msg_{secured}$ and the key referenced by the primitive as input. The result is then assigned to the variable(s) representing the argument(s) (cf. msg = adec($msg_{secured}$, privKeyR) in Figure 4.38). Finally, the transformation algorithm resolves the arguments of the MessageEvent as described before.



```
let
  msg = adec(
    msg_secured, privKeyR
  )
  arg = msg
in
[
  In(msg_secured)
]--[
]->[
]
```

Figure 4.38: Illustration of Transformation Step 3: Generate Protocol Behavior (TAMARIN) — Translation of an incoming MessageEvent with one applied security primitive

More than one applied security primitive The transformation algorithm stores the received bitstring in the variable $msg_{secured}$ (cf. In($msg_{secured}$: bitstring) in Figure 4.39). Next, the transformation algorithm iterates over the ordered set of security primitives but in inverse order. For each security primitive, it resolves the destructor that corresponds to the security primitive and stores the result in a temporal variable. As depicted in Figure 4.39, the result of the validation of the signature is stored in the variable msg^1_{tmp}. This variable is then used as input for the transformation of the next primitive. After transforming all security primitives, the result is stored in the variable msg. Finally, the transformation algorithm resolves the arguments of the MessageEvent as described before.



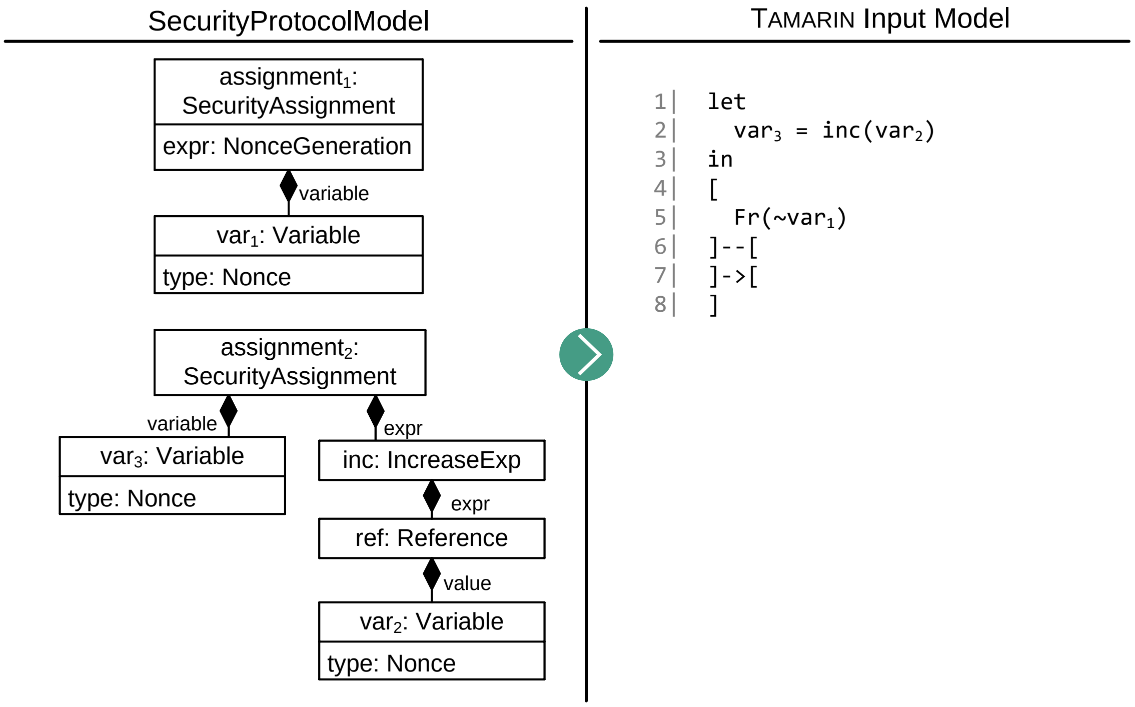
```
let
  msg^1_tmp = adec(
    msg_secured, privKeyR
  )
  msg^2_tmp = getMessage(
    msg^1_tmp, pubKeyS
  )
  msg = msg^2_tmp
  arg = msg
in
[
  In(msg_secured)
]--[
]->[
]
```

Figure 4.39: Illustration of Transformation Step 3: Generate Protocol Behavior (TAMARIN) — Translation of an incoming MessageEvent with more than one applied security primitives

Translate a SecurityAssignment As described in Section 3.4.4, a SecurityAssignment is used to create a new variable or modify an existing variable. In both cases, the SecurityAssignment encompasses a Variable and an Expression.

If the SecurityAssignment describes the creation of a new variable, the transformation algorithm creates a new statement as part of the left-hand side of the rewriting rule. The statement has the form Fr(∼var), where Fr is a built-in action in TAMARIN to create fresh values, and var is the name of the variable. Otherwise, the transformation algorithm creates a new variable of the form var = expression, where expression is the transformed Expression.

For example, in Figure 4.40, the SecurityProtocolModel encompasses two SecurityAssignments. The first SecurityAssignment creates a new variable of type Nonce. The second SecurityAssignment increases the variable var_2 by one and assigns it to the variable var_3. As output, the transformation algorithm creates the corresponding TAMARIN constructs: Fr(∼var) for the first SecurityAssignment and var_2 = inc(var_1) for the second SecurityAssignment.

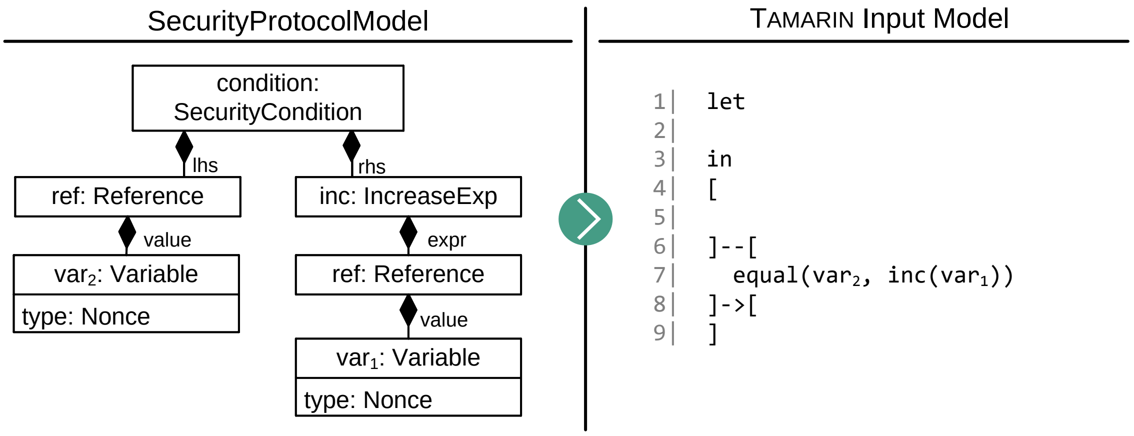


```
let
  var3 = inc(var2)
in
[
  Fr(~var1)
]--[
]->[
]
```

Figure 4.40: Illustration of Transformation Step 3: Generate Protocol Behavior (TAMARIN) — Translation of a SecurityAssignment

Translate a SecurityCondition As described in Section 3.4.5, a SecurityCondition is used to describe the conditional behavior of a security protocol. If it evaluates to true, the security protocol proceeds, otherwise the run of the security protocol is terminated. In TAMARIN, restrictions can be used to model the conditional behavior. The restrictions are placed in the labeling part of the rewriting rule.

As shown in Figure 4.41, a SecurityCondition encompasses a ComparisonOp, a left-hand side and a right-hand side Expression. In the example, the left-hand side refers to a variable var$_1$ and the right-hand side to an IncreaseExp increasing the variable var$_2$. As output, the transformation algorithm creates the corresponding TAMARIN restriction: equal(var$_1$ = inc(var$_2$)).



```
let

in
[

]--[
  equal(var2, inc(var1))
]->[
]
```

Figure 4.41: Illustration of Transformation Step 3: Generate Protocol Behavior (Tamarin) — Translation of a SecurityCondition

4.5.3 Translate the QueryModel to Tamarin

This section describes the generation of queries to enable the analysis of the security properties secrecy and authentication in Tamarin.

Translate a SecrecyQuery

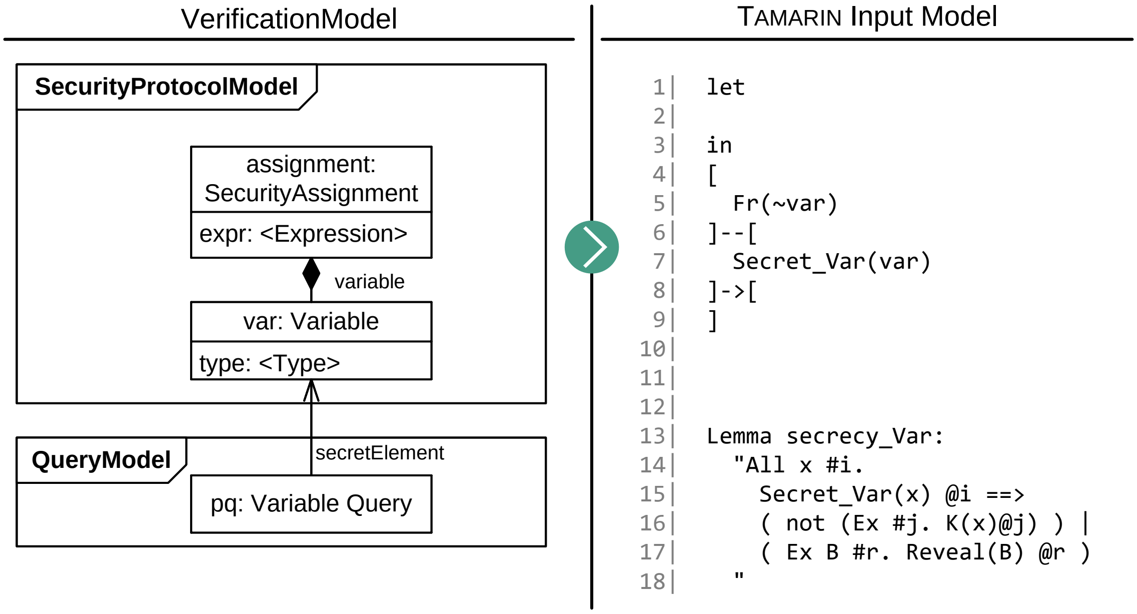
The transformation algorithm adds secrecy lemmas to the Tamarin input model to enable the automated analysis of whether protocol variables are kept secret. As explained in Section 4.3.1, we distinguish two kinds of SecrecyQueries. However, due to the modeling approach of Tamarin, both kinds are translated identically.

To model a secrecy query in Tamarin, the transformation algorithm creates a secrecy lemma as depicted in Listing 4.3 and an action fact Secrect(x) indicating that x is supposed to be secret. “The lemma states that whenever a secret action fact Secret(x) occurs at timepoint i, the adversary does not know x or an agent claimed to be honest at timepoint i has been compromised at a timepoint r [Tamarin].

```
lemma secrecy:
  "All x #i. Secret(x) @i ==>
    not (Ex #j. K(x) @j) | (Ex B #r. Reveal(B)@r & Honest(B)@i)"
```

Listing 4.3: Secrecy Lemma in Tamarin

For example, in Figure 4.42, the QueryModel contains two SecrecyQueries one referring to a property and one referring to a variable created by a security assignment. In the case of a PropertyQuery, the transformation algorithm adds the action fact to the initialization rule for the participant owing the property. In the case of a VariableQuery, the transformation algorithm adds the action fact to the rewriting rule in which the variable is created. Moreover, for each SecrecyQuery, the transformation algorithm adds a dedicated secrecy lemma to the input model. The distinction in separate secrecy lemmas makes it easier for a security engineer to understand which queries are fulfilled and which are not.



```
let

in
[
  Fr(~var)
]--[
  Secret_Var(var)
]->[
]



Lemma secrecy_Var:
  "All x #i.
    Secret_Var(x) @i ==>
    ( not (Ex #j. K(x)@j) ) |
    ( Ex B #r. Reveal(B) @r )
  "
```

Figure 4.42: Generating a Secrecy Query in TAMARIN

Translate an AuthenticationQuery

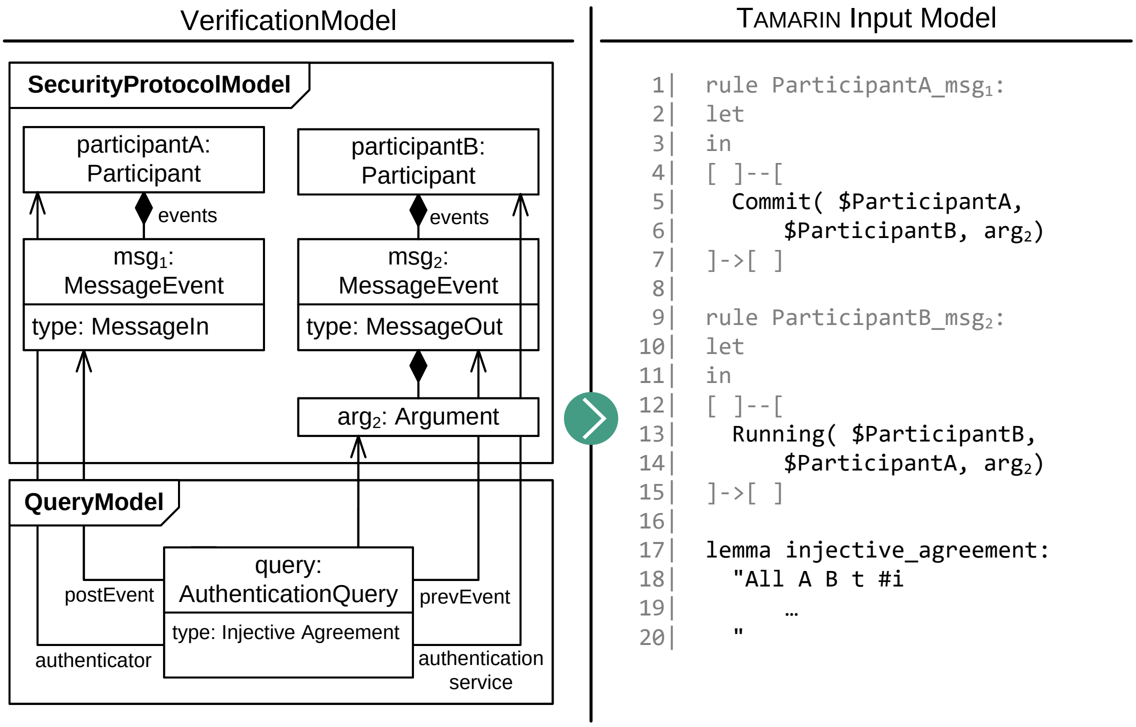
The transformation algorithm adds correspondence assertions to the TAMARIN input model to enable the automated analysis of authentication properties. Therefore, the transformation algorithm adds a lemma corresponding to the definition of authentication selected in the AuthenticationQuery. Listing 4.4 depicts the formalization of the hierarchy of authentication specifications in TAMARIN.

Afterward, the transformation algorithm adds action facts in the description of the behavior to annotate the occurrence of the two events Commit(A, B) and Create(B, id) for the definition *aliveness* and Commit(A, B) and Running(B, A) for the three other definitions. The action fact Commit(A, B) is placed at the end of A's behavior. The action fact Create(B, id) is placed in the initialization rule of B. The event Running() is placed before the last message is sent from A to B. In the case of a non-injective and injective authentication query, the transformation algorithm also adds the arguments to the action fact.

For example, in Figure 4.43, the QueryModel contains one AuthenticationQuery. The transformation algorithm adds the lemma describing the *injective agreement* to the input model. Then, the transformation algorithm adds the action fact Commit($ParticipantA, $ParticipantB, arg_2) to the rule corresponding to the postEvent of the AuthenticationQuery. Finally, the transformation algorithm adds the action fact Running($ParticipantA, $ParticipantB, arg_2) to the rule corresponding to the prevEvent of the AuthenticationQuery.

```
lemma aliveness:
"All a b t #i. Commit(a,b,t)@i ==>
  ( Ex id #j. Create(b,id) @ j ) |
  ( Ex C #r. Reveal(C) @ r & Honest(C) @ i )"

lemma weak_agreement:
" All a b t1 #i. Commit(a,b,t1) @i ==>
  ( Ex t2 #j. Running(b,a,t2) @j ) |
  ( Ex C #r. Reveal(C) @ r & Honest(C) @ i )"

lemma noninjective_agreement:
" All a b t #i. Commit(a,b,t) @i ==>
  ( Ex #j. Running(b,a,t) @j ) |
  ( Ex C #r. Reveal(C) @ r & Honest(C) @ i )"

lemma injective_agreement:
" All A B t #i.  Commit(A,B,t) @i ==>
  ( Ex #j. Running(B,A,t) @j & j < i &
    not (Ex A2 B2 #i2. Commit(A2,B2,t) @i2 & not (#i2 = #i))
  ) | ( Ex C #r. Reveal(C)@r & Honest(C) @i )"
```

Listing 4.4: Overview of Authentication Lemma in TAMARIN based on [Tamarin]



```
rule ParticipantA_msg1:
let
in
[ ]--[
  Commit( $ParticipantA,
      $ParticipantB, arg2)
]->[ ]

rule ParticipantB_msg2:
let
in
[ ]--[
  Running( $ParticipantB,
      $ParticipantA, arg2)
]->[ ]

lemma injective_agreement:
  "All A B t #i
       …
  "
```

Figure 4.43: Generating an Authentication Query in TAMARIN

4.6 Back-Translation from Security Model Checkers to MSDs

In the seventh step, Backward Translation (cf. Figure 4.1 on page 59), VICE translates the analysis results of a security model checker back to the level of the input language used to specify the security protocol. The backward translation helps security engineers to understand the results without deep knowledge of the used security model checkers. Section 4.6.1 presents the Result metamodel. Section 4.6.2 describes the functional principle of the transformation from the analysis results to the SECURITY MODELING PROFILE.

4.6.1 Overview of the Metamodel Result

Figure 4.44 depicts the class diagram of the Result metamodel. The class ResultModel refers to a VerificationModel and encompasses a set of QueryResults. A QueryResult refers to a Query contained in the QueryModel of the VerificationModel. Moreover, the QueryResult has the property result of type Boolean.

If a query cannot be fulfilled, the model checker usually provides a counterexample describing a trace that shows a possible attack. In the ResultModel, a QueryResult may encompass a Trace. This Trace consists of a set of TraceParticipants and a set of TraceMessages. A TraceParticipant refers to a Participant of the VerificationModel or to the Attacker. The TraceMessage has a sender and a receiver of type TraceParticipant. Moreover, the TraceMessage refers to a MessageEvent of the VerificationModel.

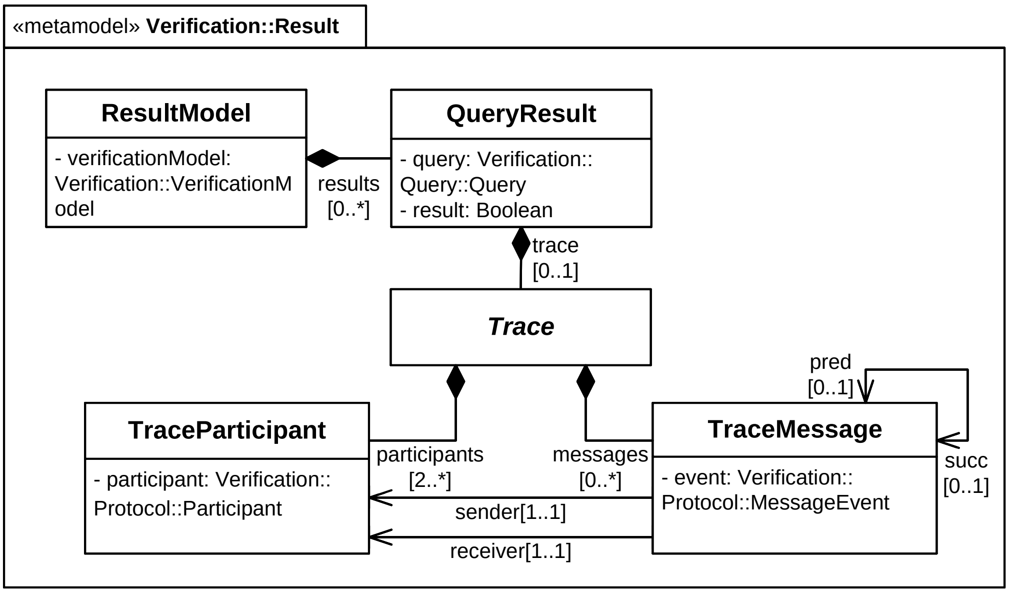

«metamodel» Verification::Result
ResultModel
- verificationModel: Verification::VerificationModel
results [0..*]
QueryResult
- query: Verification:: Query::Query
- result: Boolean
trace [0..1]
Trace
pred [0..1]
TraceParticipant
- participant: Verification:: Protocol::Participant
participants [2..*]
messages [0..*]
TraceMessage
- event: Verification:: Protocol::MessageEvent
succ [0..1]
sender[1..1]
receiver[1..1]

Figure 4.44: UML class diagram of the Result metamodel

4.6.2 Translate the Analysis Results to the SECURITY MODELING PROFILE

This section presents the functional principle of the transformation from the analysis results of a model checker to the SECURITY MODELING PROFILE. Algorithm 4.4 depicts an overview of the transformation encompassing three main steps. The step ① Parsing the Analysis Results to the ResultModel depends on the used model checker and the other two steps ② Translating the ResultModel to the Security Modeling Profile and ③ Generating a graphical representation for the ResultModel are independent of the used model checker.

Algorithm 4.4 Translation from the Analysis Results to the Security Modeling Profile

```
Input: VerificationModel, AnalysisResults
▷ ① Parsing the Analysis Results to the ResultModel
resultModel = createResultModel()
for each entry in AnalysisResults do
    queryResult = parseResult(entry)
    if queryResult.getResult = false then
        queryResult.trace = parseTrace(entry)
    end if
    resultModel.getQueryResults().add(queryResult)
end for
▷ ② Translating the ResultModel to the Security Modeling Profile
backward-translation(resultModel, VerificationModel)
▷ ③ Generating a graphical representation for the ResultModel
generate-diagram(resultModel, VerificationModel)
```

At the beginning, the transformation algorithm creates a ResultModel and parses the textual analysis results of each model checker individually. The analysis results are two-folded: First, the analysis results contain a summary of all analyzed queries and whether the queries are fulfilled or not. Second, for each analyzed query that is not fulfilled, the analysis results contain a counterexample, i.e., a successful attack on the security protocol. Hence, the transformation algorithm creates a new QueryResult for each query and sets its properties. For each counterexample, the algorithm identifies the corresponding QueryResult and creates a new Trace for this query. In the second step, the transformation algorithm translates the ResultModel back to the level of the Security Modeling Profile. The translation includes two tasks: First, the transformation algorithm translates the verification results back to the Security Modeling Profile, i.e., the properties and assignments referring to a SecrecyQuery or the participants referring to an AuthenticationQuery. Second, if a counterexample exists, the transformation algorithm translates it back to the Security Modeling Profile as well. For both tasks, the transformation algorithm exploits the traceability links that were automatically generated by QVTo during the forward translation. Thus, the transformation algorithm executes the QVTo operation *invresolve* to get the source element for a given model element in the target model. Based on this information, the transformation algorithm shows the results of the analysis to the user and in the case of a counterexample, it builds a new sequence diagram representing the counterexample on the level of the Security Modeling Profile.

In the following sections, we present the processing of the analysis results for ProVerif and Tamarin, respectively.

(1) Translating the PROVERIF Analysis Results to the ResultModel

In PROVERIF, the summary of the analysis results starts with the keyword Verification summary and consists of multiple lines. As depicted in Listing 4.5, each line describes the results of a query and has the form 'Query' <Query> 'is' <Query Result>, where <Query> is the String describing the query and <Query Result> is a String that can have the different values true, false, or cannot be proven. For example, the summary depicted in Listing 4.5 encompasses two query results. The first query result refers to a SecrecyQuery and proves that the property prop_1 is kept secret. The second query result refers to an AuthenticationQuery and shows that A is not authenticated to B.

```
Verification summary:
Query not attacker(prop1) is true.
Query event( commit(x) ) ==> inj-event( running(x) ) is false.
```

Listing 4.5: Exemplary summary of an analysis in PROVERIF

Figure 4.45 illustrates the translation of a PROVERIF summary to the ResultModel. The transformation algorithm creates a QueryResult for each line in the summary. Moreover, it parses the <Query Result> and sets the property result of the QueryResult to true if the <Query Result> is true and false otherwise. Afterward, the transformation algorithm resolves the Query that corresponds to the line in the summary. Therefore, it iterates over the set of Queries contained in the QueryModel and checks whether the PROVERIF representation of the Query matches <Query>. If this is the case, the transformation algorithm sets the property query of the QueryResult to that Query.

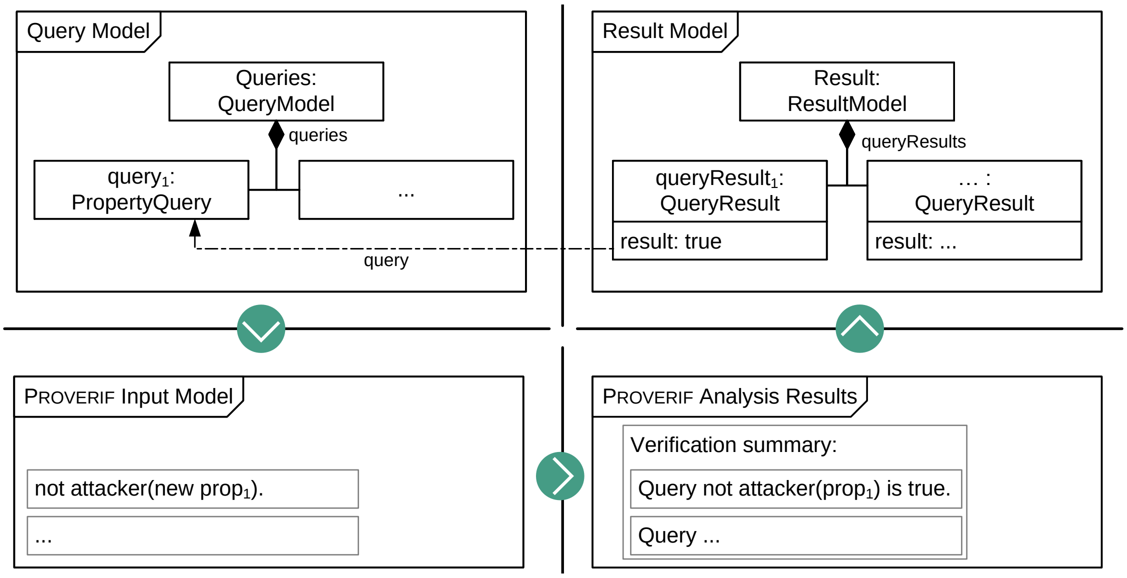


Figure 4.45: Translating a PROVERIF summary to the ResultModel

If the <Query Result> is false, the analysis results encompass a counterexample. Figure 4.46 depicts a graphical representation of a PROVERIF counterexample. Each counterexample consists of an Honest Process and an Attacker. The Honest Process corresponds to the PROVERIF main process and may create some sub-processes. For example, in Figure 4.46, the Honest Process creates the two sub-processes Alice and Bob. The creation of a sub-process is represented by a rectangle labeled with Beginning of process X, where X is the name of the process. Messages that are exchanged between the different processes are depicted by arrows. The arrows are labeled with a description of the message. For example, the first message in Figure 4.46 is labeled with (ParticipantA, ParticipantB) indicating that ParticipantA sends its identifier and ParticipantB's identifier to the public channel. Moreover, the counterexample contains information about the creation of nonces or the modification of existing variables.

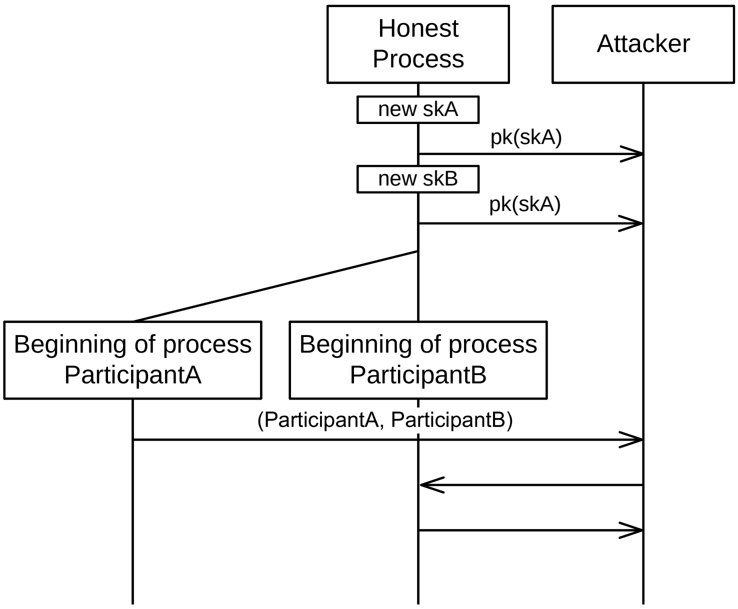

Honest Process
Attacker
new skA
pk(skA)
new skB
pk(skA)
Beginning of process ParticipantA
Beginning of process ParticipantB
(ParticipantA, ParticipantB)

Figure 4.46: Exemplary counterexample in PROVERIF

The transformation algorithm processes the counterexample as follows: At the beginning, the transformation algorithm creates a TraceParticipant for each sub-process of the counterexample. Then, it identifies the corresponding Participant contained in the VerificationModel based on the name and sets the property participant of the newly created TraceProperty accordingly. Next, the algorithm processes each process lifeline individually. The algorithm only processes message events and omits the processing of other information. For each message event that occurs on a lifeline (i.e., the start or the end of an arrow), the algorithm creates a TraceMessage and identifies the communication partner. Based on the communication partner and the message events that already occurred on the lifeline, the algorithm identifies the corresponding MessageEvent contained in the VerificationModel and sets the property event of the newly created TraceMessage accordingly.

① Translating the Tamarin Analysis Results to the ResultModel

In Tamarin, the analysis results encompass a summary of all analyzed queries and whether they are fulfilled or not. The summary starts with the keyword Section Summary: and consists of multiple lines. Each line describes the result of a particular query and has the form <Lemma> ([all-traces | exists-trace]): <Query Result>, where <Lemma> is the name of the analyzed lemma and <Query Result> is a String that can have the different values falsified or verified.

```
Section summary:
  lemma secrecy_prop1 (all-traces): verified
  lemma authentication_injective_agreement_A_B (all-traces): falsified
```

Listing 4.6: Exemplary summary of an analysis in Tamarin

The transformation algorithm processes the Tamarin summary similar to the Proverif summary. The transformation algorithm creates a QueryResult for each line in the summary. Moreover, it parses the <Query Result> and sets the property result of the QueryResult to true if the <Query Result> is verified and false otherwise. Afterward, the transformation algorithm resolves the Query that corresponds to the line in the summary. Therefore, it iterates over the set of Queries contained in the QueryModel and checks whether the Tamarin representation of the Query matches the <Query>. If this is the case, the transformation algorithm sets the property query of the QueryResult to that Query.

4.7 Implementation

This section presents an overview of our prototypical implementation to support and evaluate the concepts described throughout this chapter. The implementation is integrated into the Eclipse-based ScenarioTools MSD [ST-MSD]. In particular, we present the architecture of our implementation in Section 4.7.1 and the user interface in Section 4.7.2.

4.7.1 Security ScenarioTools (Software Architecture)

Figure 4.47 depicts the software architecture of our prototypical implementation as an extension to the tool suite Security ScenarioTools MSDs as described in Section 3.6. The entire implementation is based on the Eclipse Modeling Framework (EMF) [EMF], Eclipse Xtext [Xtext] and Eclipse QVT Operational (QVT-O) [QVTo]. In addition, our prototype encompasses the two security model checkers Proverif [Proverif] and Tamarin [Tamarin].

The model checker independent part of VICE's forwards translation (cf. Steps 1–3 in Figure 4.1 on page 59) encompasses the metamodel for the VerificationModel and the model transformation from the Security Modeling Profile to the VerificationModel. The metamodel is based on EMF and implemented by means of the component org.scenariotools.security.verification. The model transformation is implemented using QVT-O. Moreover, the package contains some Java blackbox libraries for QVT-O providing functionality to handle the evaluation of textual expressions describing security assignments and security conditions.

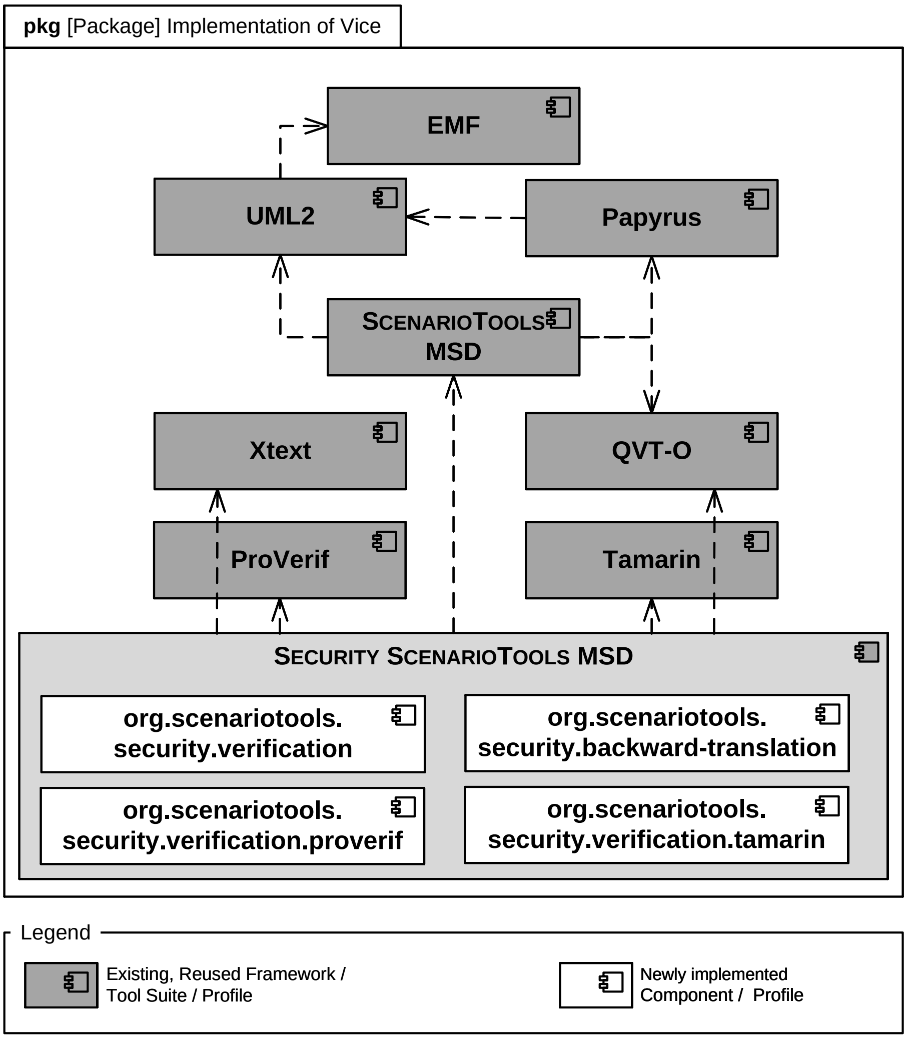

pkg [Package] Implementation of Vice
EMF
UML2
Papyrus
ScenarioTools MSD
Xtext
QVT-O
ProVerif
Tamarin
Security ScenarioTools MSD
org.scenariotools. security.verification
org.scenariotools. security.backward-translation
org.scenariotools. security.verification.proverif
org.scenariotools. security.verification.tamarin
Legend
Existing, Reused Framework / Tool Suite / Profile
Newly implemented Component / Profile

Figure 4.47: Coarse-grained architecture of the implementation and the reused components

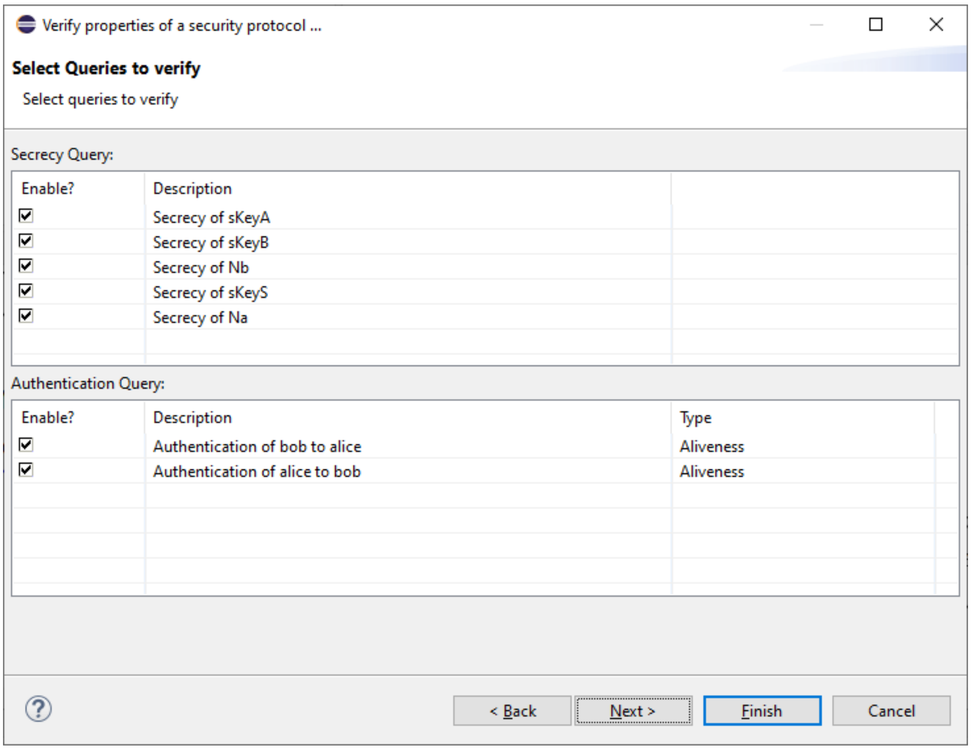
The model checker dependent part of VICE's forward translation (cf. Steps 4–5 in Figure 4.1 on page 59) encompasses the model-to-text transformation to PROVERIF and TAMARIN, and the execution of the two model checkers. The model-to-text transformations are implemented by the components org.scenariotools.security.verification.proverif and org.scenariotools.security.verification.tamarin using Xtend [Xtend].

The model checker dependent parsing of the analysis results (cf. Step 6 in Figure 4.1 on page 59) encompasses the metamodel for the ResultModel and the parsing of the PROVERIF and TAMARIN analysis results. The parsing process is implemented by the components org.scenariotools.security.verification.proverif and org.scenariotools.security.verification.tamarin. For both model checkers, the implementation encompasses the parsing of the summary. However, we have only partially implemented the parsing of the counterexample in PROVERIF and not at all in TAMARIN.

The backward translation of VICE (cf. Step 7 in Figure 4.1 on page 59) encompasses the backward translation of the ResultModel and the generation of a graphical representation of the ResultModel. The backward translation is implemented by the component org.scenariotools.security.verification.back-translation using QVT-O to resolve the elements of the VerificationModel and GraphViz to generate the sequence diagram representing the ResultModel.

4.7.2 Security ScenarioTools (User Interface)

Security ScenarioTools MSDs provides an intuitive wizard to guide the security engineer through the configuration of VICE. The wizard requires the user to specify the UML model containing the security protocol. Then, Security ScenarioTools MSDs generates the VerificationModel and provides an overview of all queries that have been generated. As shown in Figure 4.48 for the *Needham Schroeder Public Key* security protocol, the overview separates the queries into SecrecyQueries and AuthenticationQueries. For each query, the wizard provides a description providing information about the property that shall be verified and enables the security engineer to enable/disable the analysis of queries. For example, the first secrecy query relates to the variable N_a that is created during the execution of the protocol (cf. Figure 3.6 on page 37). In case of AuthenticationQueries, the wizards enables the user to select the desired level of Lowe's authentication hierarchy (e.g., Aliveness). For example, the first authentication query relates to the authentication of Alice to Bob using the level Aliveness of the Lowe's authentication hierarchy. Finally, the security engineer can configure which security model checker should be used for the analysis. By default, both model checkers are used.



Select Queries to verify

Select queries to verify

Secrecy Query:

Enable?	Description
☑	Secrecy of sKeyA
☑	Secrecy of sKeyB
☑	Secrecy of Nb
☑	Secrecy of sKeyS
☑	Secrecy of Na

Authentication Query:

Enable?	Description	Type
☑	Authentication of bob to alice	Aliveness
☑	Authentication of alice to bob	Aliveness

Figure 4.48: Overview of the Query Selection View

After these configuration steps, the wizard starts the analysis. If both model checkers are executed in the automated mode, VICE provides the analysis results to the security engineer. The result view provides an overview of whether the query is fulfilled or not fulfilled.

4.8 Evaluation

We conduct a case study based on the guidelines by KITCHENHAM ET AL. [KPP95] and RUNESON ET AL. [RH09; Run12] for evaluating VICE. Our case study investigates VICE's applicability in practice.

4.8.1 Case Study Context

We examine the three evaluation questions:

EQ1 Does VICE generate syntactically and semantically correct PROVERIF and TAMARIN input models?

EQ2 Does VICE's automatic derivation of analysis queries lead to correct queries for the security analysis of PROVERIF and TAMARIN?

EQ3 Does VICE correctly translate all analysis results (e.g., fulfillment of each query) from PROVERIF and TAMARIN back to ScenarioTools?

To answer the questions, we use the same 14 security protocols from our data collection that have been modeled successfully in the evaluation of our SECURITY MODELING PROFILE (cf. Section 3.7). The selected security protocols use different cryptographic primitives and partially rely on a trusted third party. Moreover, for some security protocols flaws have been found using symbolic model checking (e.g., the *Needham Schroeder Public Key* security protocol). For our case study, we use the flawed and the corrected version of the security protocols if existing. Thus, the selected security protocols present a broad range of possible security protocols, including correct and flawed ones.

4.8.2 Setting the Hypotheses

We define the following hypotheses for this case study. Hypotheses H1.1 and H1.2 refer to evaluation question EQ1; hypotheses H2.1 and H2.2 refer to evaluation question EQ2, and hypotheses H3.1–H5.2 refer to evaluation question EQ3.

H1.1 The different security protocols can be transformed into syntactically and semantically correct PROVERIF models. For evaluating H1.1, we generate the input models for the MSD specifications created for the evaluation of our SECURITY MODELING PROFILE in Section 3.7 and analyze them in PROVERIF. We consider H1.1 as fulfilled if all input models can be opened and analyzed in PROVERIF.

H1.2 The different security protocols can be transformed into syntactically and semantically correct TAMARIN models. For evaluating H1.2, we generate the input models for the MSD specifications created for the evaluation of our SECURITY MODELING PROFILE in Section 3.7 and analyze them in TAMARIN. We consider H1.2 as fulfilled if all input models can be opened and analyzed in TAMARIN.

H2.1 The automatic derivation of analysis queries and their translation to PROVERIF is correct and complete. For evaluating H2.1, we manually investigate the generated PROVERIF inputs models and check whether the generated queries are correctly specified. We consider H2.1 as fulfilled if the VICE generates the queries as explained in Section 4.4.3.

H2.2 The automatic derivation of analysis queries and their translation to TAMARIN is correct and complete. For evaluating H2.2, we manually investigate the generated TAMARIN inputs models and check whether the generated queries are correctly specified. We consider H2.2 as fulfilled if the VICE generates the queries as explained in Section 4.5.3.

H3.1 The security analysis of the specified security protocols is correct, i.e., PROVERIF finds the same attacks as if the security protocol were modeled manually by a security expert in the symbolic model. As mentioned above, security experts have performed a security analysis and published the results in various papers. We consider H3.1 as fulfilled if we yield the same results.

H3.2 The security analysis of the specified security protocols is correct, i.e., TAMARIN finds the same attacks as if the security protocol were modeled manually by a security expert in the symbolic model. As mentioned above, security experts have performed a security analysis and published the results in various papers. We consider H3.2 as fulfilled if we yield the same results.

H4.1 The overview of the PROVERIF analysis results is correct and complete. For evaluating H4.1, we manually investigate the results of the PROVERIF analysis and compare it with the summary shown to the user. We consider H4.1 as fulfilled if VICE's processes the analysis results correctly and shows them to the user.

H4.2 The overview of the TAMARIN analysis results is correct and complete. For evaluating H4.2, we manually investigate the results of the TAMARIN analysis and compare it with the summary shown to the user. We consider H4.2 as fulfilled if VICE's processes the analysis results correctly and shows them to the user.

H5.1 The backward translation of PROVERIF's counterexamples is correct and complete. For evaluating H5.1, we manually investigate the counterexample produced by PROVERIF and compare it with the counterexample shown to the user. We consider H5.1 as fulfilled if VICE's processes the counterexamples correctly and shows them to the user.

H5.2 The backward translation of PROVERIF's counterexamples is correct and complete. For evaluating H5.1, we manually investigate the counterexample produced by PROVERIF and compare it with the counterexample shown to the user. We consider H5.1 as fulfilled if VICE's processes the counterexamples correctly and shows them to the user.

4.8.3 Validating the Hypotheses

In the following, we validate each hypothesis separately using the prototypical implementation of VICE described in Section 4.7. Moreover, we use the MSD specifications that we created during the evaluation of our SECURITY MODELING PROFILE (cf. Section 3.7) as input for the transformation approach to generate the corresponding input models for PROVERIF and TAMARIN.

Validating Hypotheses H1.1 and H1.2

To validate H1.1 and H1.2, we executed the forward translation of VICE (cf. Steps 1–3 in Figure 4.1 on page 59) for all 14 security protocols. We disabled the derivation of analysis queries to concentrate on the generation of the security protocols. As a result, we got a VerificationModel, a PROVERIF file, and a TAMARIN file for each security protocol.

We repeated the forward translation five times for all 14 security protocols to investigate whether the forward translation produced deterministic results. The only difference in the VerificationModel and the text files is the order of elements in unordered sets, e.g., the ordering of participants in the security protocol. However, this difference does not influence the semantics. Thus, we conclude that the forward translation produces deterministic results.

Next, we checked whether the generated PROVERIF and TAMARIN files are syntactically correct. To do this, we opened each file with the corresponding model checker and used the internal functionality to report any syntax errors. The tools did not report any syntax errors. However, in the first versions of the translation to TAMARIN, some well-formedness checks failed. The reason for this was that we used a static preamble containing all possible restrictions to model the conditional behavior. After we switched to a dynamic preamble that only contained the parts that were used in the security protocol, no well-formedness check failed. Thus, we conclude that the forward translation produces syntactically correct PROVERIF and TAMARIN files.

Afterward, we opened the generated PROVERIF and TAMARIN files and manually searched for deviations between the original MSD specifications of the 14 security protocols and the resulting PROVERIF and TAMARIN input models. We did not find any deviations.

Moreover, we checked whether the generated PROVERIF and TAMARIN models are executable. For PROVERIF, we manually added a variable executedX and a secrecy query of the form query attacker(executedX) for each participant X to the PROVERIF model. The variable executedX is sent to the public channel after the last event occurred in the behavior of the participant X. If the attacker knows all variables executedX, the model is executable. Based on these queries, we conclude that all PROVERIF models are executable.

For TAMARIN, we manually added an executability lemma to the TAMARIN model. Therefore, we have annotated the first rule for each participant X with the action fact StartParticipantX() and the last message of the participant with the action fact EndParticipantX(). In addition, we have added a lemma to the TAMARIN model that holds if a trace exists that includes all annotated action facts and without an adversary that reveals one of the participants' secret keys. Based on these lemmas, we conclude that all TAMARIN models are executable.

Validating Hypotheses H2.1 and H2.2

To validate H2.1 and H2.2, we executed the forward translation of VICE (cf. Steps 1–3 in Figure 4.1 on page 59) for all 14 security protocols. For each security protocol, we generated secrecy queries and all kinds of authentication queries (aliveness, weak agreement, non-injective agreement, and injective agreement). As a result, we got for each security protocol five VerificationModels, five PROVERIF files and five TAMARIN files.

As for validating H1.1 and H1.2, we checked whether the forward translation produces deterministic and syntactically correct PROVERIF and TAMARIN files and whether the models are executable. Our checks showed that all models are syntactically correct and executable.

Validating Hypotheses H3.1 and H3.2

To validate H3.1 and H3.2, we executed the forward translation of VICE (cf. Steps 1–3 in Figure 4.1 on page 59) for all 14 security protocols and analyzed the resulting files with PROVERIF and TAMARIN. For the authentication queries, we checked whether the results are consistent, i.e., if a security protocol fulfills injective agreement, it has to fulfill the other authentication specifications as well. Moreover, we compared the results with known results from the literature. If a security flaw has been reported for the security protocol that can be found in the symbolic model, VICE finds that flaw as well. In addition, if we apply the suggested corrections for the flaw, VICE states that the security protocol is correct.

Validating Hypotheses H4.1 and H4.2

To validate H4.1 and H4.2, we executed VICE for all 14 security protocols. For each execution, we investigate the textual results provided by PROVERIF and TAMARIN and compare them with the results provided by VICE. We did not find any deviation.

Validating Hypotheses H5.1 and H5.2

As explained in Section 4.6, we only developed a concept for parsing the counterexample in PROVERIF that has only partially been implemented. Therefore, we can only validate parts of the two hypotheses.

We executed VICE for all 14 security protocols. For each counterexample produced by PROVERIF, we manually created the trace for the query following the concept described in Section 4.6. Afterward, we executed the backward translation and investigated the resulting sequence diagram. We checked if the sequence diagram only uses names that are used in the security protocol specified by means of the SECURITY MODELING PROFILE. Moreover, we checked for deviations in the message ordering of the sequence diagram and the counterexample. In both cases, we did not find any deviation.

4.8.4 Analyzing the Results

Table 4.3 summarizes the results of the case study for the PROVERIF-related hypotheses and Table 4.4 for the TAMARIN-related hypotheses. For all selected security protocols, our transformation approach is able to transform the modeled MSD specification to syntactically and semantically correct input models in PROVERIF and TAMARIN. We were able to open and analyze each input model in the corresponding model checker. Thus, we conclude that H1 and H2 are fulfilled.

Moreover, the queries generated during the transformation are correct and complete. The analysis results yield the expected results. Moreover, the result view presents a correct summary of the results at the level of the SECURITY MODELING PROFILE. Thus, we conclude that H3 and H4 are fulfilled.

However, the processing of counterexamples is only partially supported. While VICE is able to translate a given Trace correctly to the level of the SECURITY MODELING PROFILE and to create a correct graphical representation of the trace, the parsing of the counterexample for both model checkers does not work. Thus, we conclude that H5 is only partially fulfilled.

Concluding the case study, the fulfilled hypotheses indicate that our proposed model-checking approach for automatically verifying security protocols is applicable in practice. However, the correct handling of counterexamples should be improved in future work.

Table 4.3: Results of the case study for the PROVERIF-related hypotheses

No.	Security Protocol	H1.1	H2.1	H3.1	H4.1	H5.1
		VICE generates correct input models for PROVERIF	VICE correctly translates the derived analysis queries to PROVERIF	VICE yields the expected analysis results	VICE provides a correct summary of the analysis results to the user	VICE parses the counterexamples correctly and shows them to the user
1	Andrew Secure RPC [Sat89; BAN90; Low96a]	●	●	●	●	◖
2	Andrew Secure RPC (BAN) [Sat89; BAN90; Low96a]	●	●	●	●	◖
3	Bull's Authentication Protocol [BO97; RS98]	●	●	●	●	◖
4	CH07 [vR09]	●	●	●	●	◖
5	CCITT-X.509-Protocol [IM90]	●	●	●	●	◖
6	Denning-Sacco Shared Key [DS81; Low00]	●	●	●	●	◖
7	Diffie Helman [DH76]	●	●	●	●	◖
8	Gong's Mutual Authentication Protocol [Gon89]	●	●	●	●	◖
9	Kao Chow's Authentication Protocol [KC95]	●	●	●	●	◖
10	Kerberos [BM90; NT94; Low00]	●	●	●	●	◖
11	Needham-Schroeder Public Key [Low96b]	●	●	●	●	◖
12	Needham-Schroeder Symmetric Key [Low96b]	●	●	●	●	◖
13	Wide Mouthed Frog [BAN90; Low00]	●	●	●	●	◖
14	Woo and Lam Mutual Authentication [WL94]	●	●	●	●	◖

Legend: ● fulfilled, ◖ partially fulfilled, ○ not fulfilled

Table 4.4: Results of the case study for the Tamarin-related hypotheses

No.	Security Protocol	H1.1	H2.1	H3.1	H4.1	H5.1
		VICE generates correct input models for Tamarin	VICE correctly translates the derived analysis queries to Tamarin	VICE yields the expected analysis results	VICE provides a correct summary of the analysis results to the user	VICE parses the counterexamples correctly and shows them to the user
1	Andrew Secure RPC [Sat89; BAN90; Low96a]	●	●	●	●	◖
2	Andrew Secure RPC (BAN) [Sat89; BAN90; Low96a]	●	●	●	●	◖
3	Bull's Authentication Protocol [BO97; RS98]	●	●	●	●	◖
4	CH07 [vR09]	●	●	●	●	◖
5	CCITT-X.509-Protocol [IM90]	●	●	●	●	◖
6	Denning-Sacco Shared Key [DS81; Low00]	●	●	●	●	◖
7	Diffie Helman [DH76]	●	●	●	●	◖
8	Gong's Mutual Authentication Protocol [Gon89]	●	●	●	●	◖
9	Kao Chow's Authentication Protocol [KC95]	●	●	●	●	◖
10	Kerberos [BM90; NT94; Low00]	●	●	●	●	◖
11	Needham-Schroeder Public Key [Low96b]	●	●	●	●	◖
12	Needham-Schroeder Symmetric Key [Low96b]	●	●	●	●	◖
13	Wide Mouthed Frog [BAN90; Low00]	●	●	●	●	◖
14	Woo and Lam Mutual Authentication [WL94]	●	●	●	●	◖

Legend: ● fulfilled, ◖ partially fulfilled, ○ not fulfilled

4.8.5 Threats to Validity

The threats to validity in our case study are as follows:

Construct Validity

The case study was designed and conducted by the same researcher who developed the approach. Since the researcher might have a bias toward the developed approach, the case study would be more significant if security experts had modeled the security protocols, executed the transformation approach, and reviewed the results. To mitigate this, we discussed our approach for the automatic verification of security protocols with security experts from academia. Moreover, we discussed the case study design and its research questions with other researchers.

In addition, case study results have yet to be evaluated by security experts from academia or industry. To mitigate this, the cases are well known, and their description was taken from literature providing a summary of possible attacks. Thus, we were able to compare our results with the ones from the literature.

External Validity

We only considered 14 different security protocols and, thus, cannot generalize the fulfillment of the hypotheses for all possible security protocols. Nevertheless, the selected security protocols represent typical examples including different cryptographic primitives. Furthermore, some of the security protocols are correct, while others contain flaws that can be found using symbolic model checking. Thus, we do not expect large deviations from other examples. Moreover, during the selection of the cases, we took care that we select security protocols that use different building blocks (e.g., encryption, hashing, etc.) in different combinations. Although we cannot guarantee that our selected cases are representative, we cover at least a broad range of security protocols.

Reliability

The case study was conducted based on the prototype implementation that might not be available in the future. To mitigate this, the implemented concepts are defined throughout this chapter and can be newly implemented.

To analyze the results of our case study, we manually check the input models for PROVERIF and TAMARIN encompassing the description of the protocol behavior as well as the queries to verify. The researcher may have made a mistake while checking the input models. To mitigate this, we checked the input models using the internal validation functionality of the two model checkers and executed the corresponding analysis. Thereby, we can ensure that the generated input model is syntactically correct. Moreover, by comparing the results with the results from the literature, we could also judge whether the input model is semantically correct.

4.9 Related Work

Several approaches exist that focus on the specification of security protocols and their analysis using security model checkers like PROVERIF or TAMARIN. The modeling language of these approaches is either textual or graphical. However, in most approaches, only one model checker is supported and the modeling language is very close to the input language of the targeted model checker. Hence, the integration of further model checkers requires changes to the modeling language. In contrast, our SECURITY MODELING PROFILE has been designed to model the general concepts of security protocols independent of the targeted model checkers.

Furthermore, in most approaches, the security engineer has to manually specify the queries to analyze security properties. This requires a deep knowledge of the used model checker. In our approach, we automatically derive the analysis queries from our SECURITY MODELING PROFILE and thus significantly reduce the effort for the modeling and analysis of security protocols. Furthermore, we provide a basic back-translation from the model checker to the modeling tool. Thus, we additionally facilitate the analysis of security protocols.

In the following, we discuss related work in two categories: First, we present in Section 4.9.1 approaches that use a model-based specification language to specify security protocols and in Section 4.9.2 approaches that use a text-based specification language to specify security protocols. All approaches have in common that they translate the specified security protocol to a security model checker to analyze it.

4.9.1 Model-based approaches for the automated verification of security protocols

"FANG ET AL. [SLF+14; FLH+16] propose a modeling and analysis approach for security protocols. They introduce a UML profile to enable the modeling of security protocols by means of UML Interactions. In addition, they describe a translation from their UML profile to PROVERIF to verify the properties of the specified security protocol. [...] Compared to our profile, their profile does not model the general concepts of security protocols but remains very close to the input language of PROVERIF. As a consequence, the security engineer has still to learn the input language of PROVERIF. Furthermore, they do not provide any support for choosing a sufficient set of analysis queries" [*KDH+20].

"AMEUR-BOULIFA ET AL. [ALA19; LLA+16] present a modeling approach based on SysML to enable the specification of security aspects for embedded systems. They enhance SysML block and state machine diagrams to capture security features like secrecy and authentication. Furthermore, they provide a model-to-text transformation to enable the formal verification of the security concepts by means of PROVERIF. While they enable the automatic derivation of secrecy queries, the security engineer has to manually define authentication queries. In contrast to their approach, we conceived a modeling approach based on sequence charts since they are more appropriate for the specification of requirements on message-based interactions [LT15]. In addition, we support the security engineer in choosing a sufficient set of analysis queries (secrecy and authentication). Thus, the security engineer no longer has to learn the query language. Furthermore, she/he cannot make mistakes when defining queries or forget important queries" [*KDH+20].

RAIMONDO ET AL. [RBM+22] propose a modeling and analysis approach for security protocols based on UML Interactions and TAMARIN. The authors focus on the security analysis of blockchain-based protocols. They describe a translation from their UML modeling language to the Alice & Bob specification. The Alice & Bob specification is then automatically translated to TAMARIN to verify the security of the protocol. Compared to our profile, their modeling language does not model the general concepts of security protocols but focuses on blockchain-based protocols and hides basic security primitives from the security engineer. Moreover, their approach does not support the generation of analysis queries. Instead, the security engineer has to add the queries as goals to the Alice & Bob specification or to the TAMARIN input model. Hence, the security engineer has still to learn more than one specification language.

MÉRÉ ET AL. [MJP$^+$22] present their experiences on the formal verification of UML models in an industrial context. Amongst others, they informally introduce a modeling language to specify security protocols by means of UML Interaction. Furthermore, they present a translation from their UML Interaction to VerifPal, a model checker for the verification of security protocols presented by KOBEISSI ET AL. [NGM19]. Compared to our profile, their informally defined modeling language does not model the general concepts of security protocols but focuses on the modeling of blockchain-related aspects. In addition, they do not provide any support for choosing a sufficient set of analysis queries.

4.9.2 Text-based approaches for the automated verification of security protocols

KOBEISSI ET AL. [NGM19] propose a modeling analysis approach for security protocols aiming to work better for real-world practitioners. Therefore, they introduce a textual language for modeling protocols that is supposed to be easier to write and understand than the languages employed by existing tools. Based on that textual language, they have developed a formal verification approach to analyze the security protocols concerning confidentiality and authentication. Moreover, they provide a translation to PROVERIF and Coq for further analysis. As in our modeling approach, they restrict the user to a predefined set of primitives. In contrast to our approach, they use a textual input language while we use a graphical modeling language. Both approaches have in common that they model the security protocol in a notion of sequence diagrams.

BUGLIESI ET AL. [BCM$^+$16] present the textual modeling language *AnBx*, a formal protocol specification language based on the Alice & Bob notation. They define formal semantics for the language based on the AVISPA intermediate format to enable the analysis of the protocol. Moreover, the AnBx modeling language enables security engineers to specify security properties. Their approach includes a transformation from AnBx to PROVERIF [GM17]. In contrast to their approach, we automatically derive an initial set of queries for verifying the security protocol. Thereby, we enable security engineers without deep knowledge to verify security protocol. However, enabling custom queries makes the AnBx approach more flexible and points to future work for our approach.

METERE AND ARNABOLDI [MA22] present the text-based approach METACP, an automated tool to simplify the design and the analysis of security protocols. METACP enables security engineers to specify security protocols in a graphical interface using an Alice & Bob-based notation. Moreover, METACP provides different plugins to analyze the security protocol by means of PROVERIF or TAMARIN. However, in contrast to our approach, METACP does neither support the automatic derivation of analysis queries nor their manual specification. Thus, security engineers would have to change the text file manually to add analysis queries.

KELLER [Kel14] proposes an automatic translation from security protocols specified by means of the Alice & Bob notation to TAMARIN input models. Compared to [MA22], KELLER enables the automatic generation of analysis queries based on a goals section as part of the Alice & Bob notation. In contrast to our approach, they use a textual input language while we use a graphical modeling language. Both approaches have in common that they model the security protocol in a notion of sequence diagrams and that security engineers do not need to specify analysis queries directly in the security model checker's input language.

NAKABAYASHI AND OKANA [NO21] present a verification method of key-exchange protocols using TAMARIN. Their approach verifies whether a security protocol satisfies main security properties like confidentiality and authentication. To simplify the specification process, they defined a template for the security model encompassing basic lemmas used for the analysis in TAMARIN. The security protocol is specified by a specification language similar to the rewriting rules of TAMARIN. In an automated step, the security protocol specification is combined with the security model template to retrieve the TAMARIN input model. Compared to our approach, the security engineer has still to learn the TAMARIN language, which is generally hard to comprehend.

4.10 Summary

This chapter presents VICE (VIsual Cryptography vErifier), a model-checking approach for automatically verifying security protocols concerning secrecy and authentication. VICE encompasses an automatic and systematic technique for the transition from our SECURITY MODELING PROFILE to the two security model checkers PROVERIF and TAMARIN.

For this purpose, VICE encompasses two steps manually executed by a security engineer and six fully automated steps using model transformation techniques. The model transformation techniques translate the security protocols into the input language of the symbolic model checker. In particular, the model transformation generates the flow of the security protocols and all relevant queries for verifying secrecy and authentication properties. Moreover, VICE automatically executes the symbolic model checker and provides the analysis results to the user.

We implemented a prototype based on SCENARIOTOOLS MSD and evaluated the applicability in practice of VICE by means of 14 security protocols. We evaluate whether the transformation derives correct inputs for the two symbolic model checkers PROVERIF and TAMARIN. Moreover, we evaluate whether the model checkers yield the expected analysis results. Our evaluation results indicate that VICE provides an intuitive way to apply model checking for verifying security protocols. In particular, security engineers do not need deep knowledge of the used model checker since VICE completely encapsulates this knowledge.

5

Incorporation of Functional and Security MSDs

This chapter presents an extension to our scenario-based requirements engineering methodology [*HFK+16b] incorporating functional and security requirements. In particular, we introduce the MISUSE CASE MODELING PROFILE, which enables the specification of misuse cases [SO05] against the system under development. Moreover, we introduce the SECURITY PROTOCOL TEMPLATE PROFILE, which enables the systematic reuse of security protocols within scenario-based requirements specifications. Finally, by extending the Play-out algorithm, we enable requirements engineers to determine whether the specified security requirements are sufficient to mitigate the misuse cases. Existing security requirements engineering approaches (e.g., [SO05; PX05; HLM+08; FC03; MG07]) focus on the informal specification of misuse cases and the derivation of appropriate security requirements to mitigate these misuse cases. However, they do not provide an integrated analysis of functional and security-related aspects or sufficient analysis techniques to validate whether the security requirements can mitigate the specified misuse case.

This chapter is structured as follows: We provide a list of our contributions in Section 5.1. Then, we apply the two profiles MISUSE CASE MODELING PROFILE and SECURITY PROTOCOL TEMPLATE PROFILE in Section 5.2. Subsequently, we present concepts of the MISUSE CASE MODELING PROFILE and SECURITY PROTOCOL TEMPLATE PROFILE in Section 5.3 and Section 5.4. Afterward, we provide information about the implementation in Section 5.5 and evaluate both approaches by means of a case study in Section 5.6. We investigate related work in Section 5.7. Finally, we summarize this chapter in Section 5.8.

We published contents of this chapter in two papers ([*Koc18] and [*KTD+22]). In addition, parts of this chapter have been contributed by the bachelor theses of HOCHHALTER [+Hoc20] and TRIPPEL [+Tri21], as well as in the master thesis of JAZAYERI [+Jaz15]. HOCHHALTER made some initial contributions towards the specification of misuse and mitigation scenarios within the MSD specification. TRIPPEL provided an initial concept for the integration of security protocols within functional MSDs. JAZAYERI developed an initial concept for the specification of security properties by means of MSDs and their automated analysis.

5.1 Contributions

The contributions of this chapter can be summarized as follows:

- We define a UML profile called the MISUSE CASE MODELING PROFILE extending UML Interactions and Modal Sequence Diagrams for specifying misuse cases.
- We define a UML profile called the SECURITY PROTOCOL TEMPLATE PROFILE extending the SECURITY MODELING PROFILE to reuse security protocols systematically in requirement MSDs. This extension encompasses specifying parameterizable templates for security protocols and referencing these templates in scenario-based requirements specifications.
- We extend the Play-out algorithm to enable the simulation of scenario-based requirements specification specified by means of the two newly created profiles.
- We implement a prototype based on SCENARIOTOOLS MSD and show in an evaluation that our approach is applicable in practice.

5.2 Exemplary Application of the MISUSE CASE MODELING PROFILE and the SECURITY PROTOCOL TEMPLATE PROFILE

In this section, we apply the two profiles MISUSE CASE MODELING PROFILE and SECURITY PROTOCOL TEMPLATE PROFILE to the Emergency Braking & Evasion Assistance System (EBEAS). Subsequently, we specify misuse cases against the EBEAS and security measures to mitigate these misuse cases.

The MSD specification, introduced in Section 2.1.1, specifies requirements on the communication behavior of the EBEAS. The situation is as follows: If the leading: Vehicle detects an obstacle in front, it has to perform an emergency brake and inform the ego: Vehicle about the emergency brake. Then ego: Vehicle checks if it can perform an emergency brake and communicates with the vehicle rear: Vehicle to further coordinate the actions to the dangerous situation. If the situation analysis shows that a crash is unavoidable, the involved vehicles will activate their pre-crash systems.

A couple of possible attacks against the use case defined by the MSD specification exist. For example, an attacker could spoof the leading: Vehicle to send an emergency brake warning. Moreover, an attacker could initiate a man-in-the-middle attack between ego: Vehicle and rear: Vehicle.

Subsequently, we specify the man-in-the-middle attack as a misuse case using the MISUSE CASE MODELING PROFILE. Figure 5.1 depicts the UML classes for the misuse case. We reuse the classes Environment and Vehicle as defined in Section 2.1.1. Furthermore, the class diagram encompasses a class for the attacker annotated with the stereotype «Attacker». The attacker does not have any properties or operations.

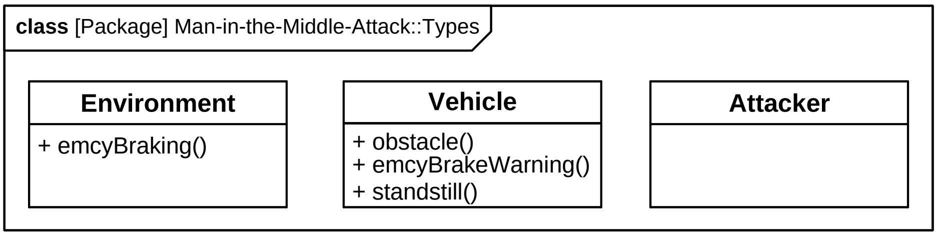


Figure 5.1: UML class diagram for the Man-in-the-Middle-Attack

Based on the classes, the UML collaboration, depicted in Figure 5.2, specifies the roles participating in the misuse case. As for the use case, the UML collaboration encompasses the four roles leading: Vehicle, ego: Vehicle, rear: Vehicle, and env: Environment. Furthermore, the UML collaboration contains the role att: Attacker. att: Attacker spoofs leading: Vehicle and ego: Vehicle to pursue the man-in-the-middle attack (cf. dependency «spoofs» in Figure 5.2).

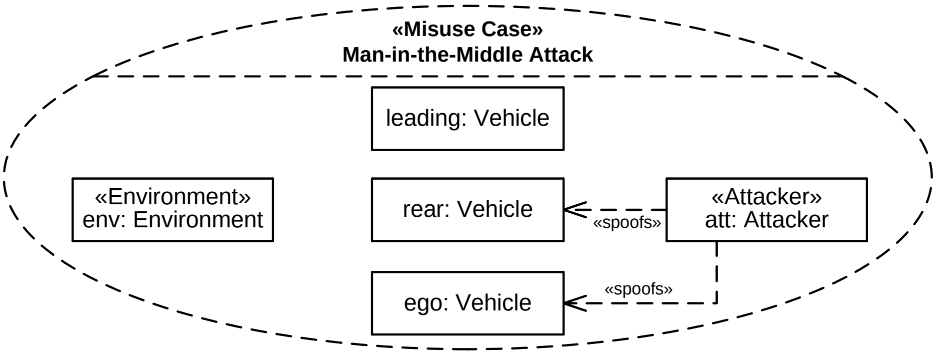


Figure 5.2: UML collaboration diagram for the Man-in-the-Middle-Attack

Figure 5.3 depicts the MSD describing the man-in-the-middle attack. The MSD encompasses the three lifelines ego: Vehicle, att: Attacker, and rear: Vehicle representing the roles defined in the UML collaboration. At the beginning of the attack, ego: Vehicle allegedly sends the message emcyBrakeWarning() to rear: Vehicle. However, this message is intercepted by att: Attacker. Then, att: Attacker forwards this message to rear: Vehicle and thus presents itself as a legitimate vehicle. ego: Vehicle checks whether it is possible to brake safely. Since this is not the case, ego: Vehicle sends the message emcyBrakeResponse(false) to att: Attacker. att: Attacker changes the parameter from “false” to “true” and forwards the message to ego: Vehicle. Due to the changed parameter, a rear-end collision occurs in the further process without an activated pre-crash system.

To mitigate the misuse case, a requirements engineer must integrate security measures into the scenario-based requirements specification. For this, they can use the cryptographic primitives of the SECURITY MODELING PROFILE or they can use existing and verified security protocols to establish secure communication.

For example, the requirements engineer could specify the requirement that all messages sent by the vehicles must be secured by a digital signature. Thus, as depicted in Figure 5.4a, the requirements engineer adds a cryptographic key pair to the properties of the class Vehicle. Moreover, as depicted in Figure 5.4b, he/she uses the dependency with the applied stereotype «knownKeys» to

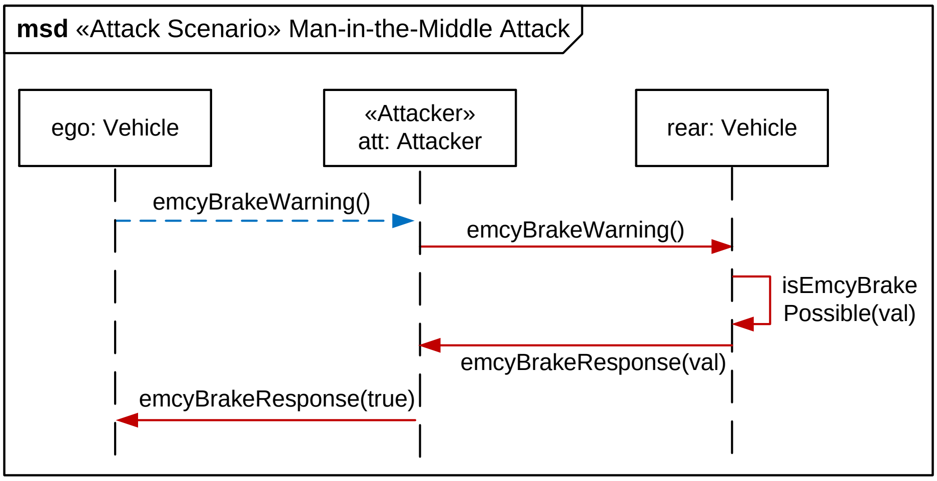


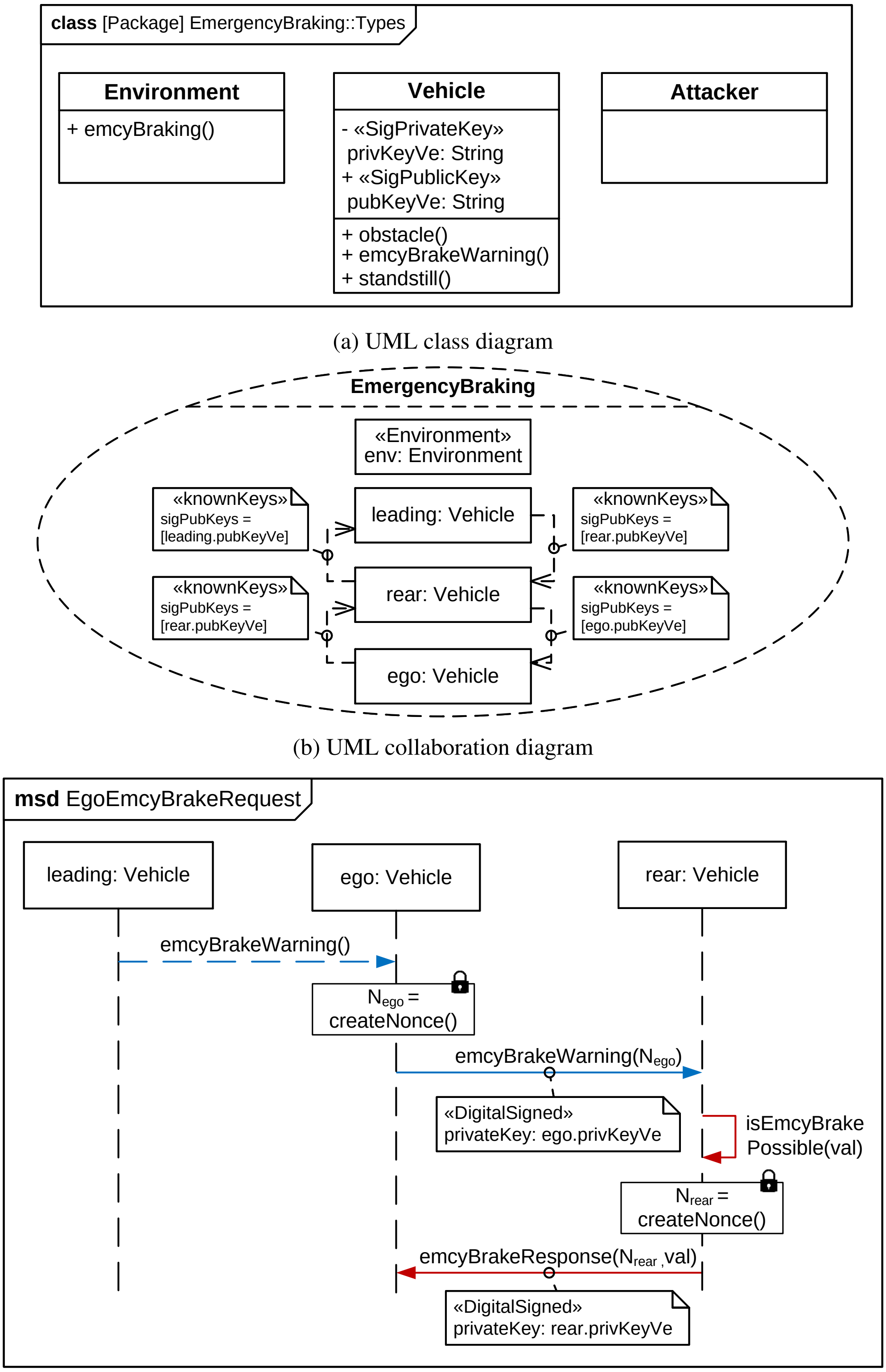
Figure 5.3: MSD for the Man-in-the-Middle-Attack

define that the vehicles know the public keys of each other. The requirements engineer adds the stereotype «DigitalSigned» to the messages and sets the properties of the stereotype accordingly (cf. Figure 5.4c). Finally, the requirements engineer creates nonces and adds these nonces as parameters to the exchanged messages to prevent replay attacks. For example, the nonce $\mathsf{N}_{\mathsf{ego}}$ is added to the message emcyBrakeWarning().

Although signing the messages would be sufficient to mitigate the man-in-the-middle attack, one problem remains. To validate the signature, the vehicles must know each other's public keys. However, storing all potential public keys on the vehicles upfront is unrealistic. Instead, the public keys could be stored at a trusted third party and sent to the vehicles on demand using a standardized protocol.

For example, the requirements engineer could decide to use a slightly modified variant of the *Needham-Schroeder Public Key* security protocol. In the modified version, the trusted third party stores both a public key for encryption and a public key for validating the signature. On request, the trusted third party sends both public keys to the requester.

To integrate the *Needham-Schroeder Public Key* security protocol into the scenario-based requirements engineering specification, the requirements engineer uses a UML::InteractionUse and applies the stereotype «SecurityProtocolReference». As depicted in Figure 5.5, the «SecurityProtocolReference» references the MSD defining the *Needham-Schroeder Public Key* security protocol and adapts the security protocol to the application context. Therefore, the ego: Vehicle has to substitute the role alice: Alice and the role rear: Vehicle has to substitute the role bob: Bob. This is realized by the roleMap, where a roleParameter of the security protocol is substituted by a roleArgument (cf. alice: Alice is substituted by ego: Vehicle in Figure 5.5). Furthermore, the propertyMap enables the substitution of properties used in the security protocols. After the execution of the protocol, the two roles know each other's public key and can use it to secure their communication (cf. emcyBrakeWarning with applied stereotype «DigitalSigned» in Figure 5.4c).


class [Package] EmergencyBraking::Types
Environment
+ emcyBraking()
Vehicle
- «SigPrivateKey» privKeyVe: String
+ «SigPublicKey» pubKeyVe: String
+ obstacle()
+ emcyBrakeWarning()
+ standstill()
Attacker
EmergencyBraking
«Environment» env: Environment
«knownKeys» sigPubKeys = [leading.pubKeyVe]
leading: Vehicle
«knownKeys» sigPubKeys = [rear.pubKeyVe]
«knownKeys» sigPubKeys = [rear.pubKeyVe]
rear: Vehicle
«knownKeys» sigPubKeys = [ego.pubKeyVe]
ego: Vehicle
msd EgoEmcyBrakeRequest
leading: Vehicle
ego: Vehicle
rear: Vehicle
emcyBrakeWarning()
N_ego = createNonce()
emcyBrakeWarning(N_ego)
«DigitalSigned» privateKey: ego.privKeyVe
isEmcyBrake Possible(val)
N_rear = createNonce()
emcyBrakeResponse(N_rear ,val)
«DigitalSigned» privateKey: rear.privKeyVe

(a) UML class diagram

(b) UML collaboration diagram

(c) MSD specification

Figure 5.4: Exemplarily application of the SECURITY MODELING PROFILE to mitigate the man-in-the-middle attack

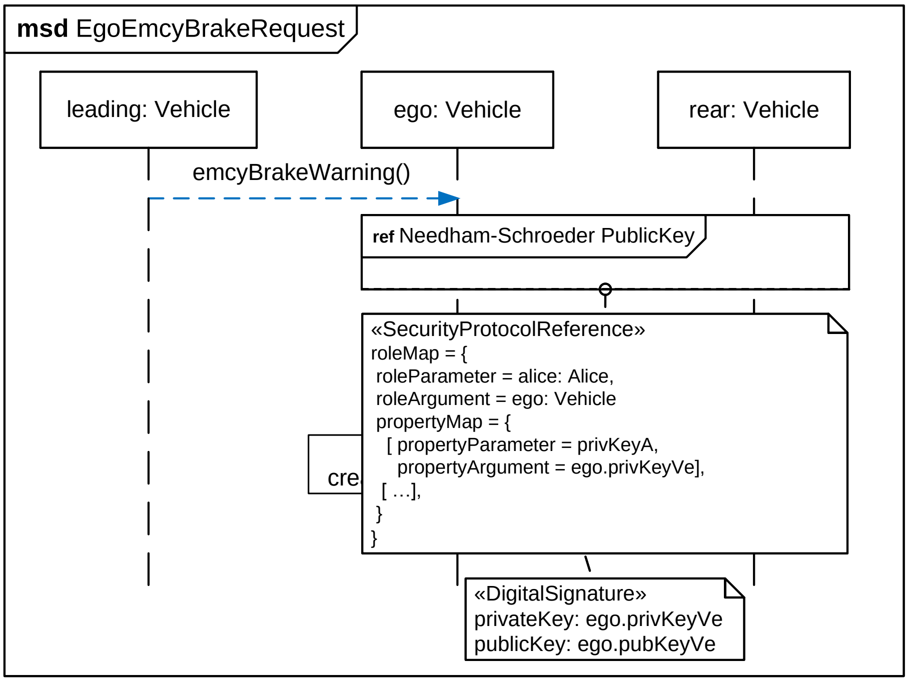

msd EgoEmcyBrakeRequest
leading: Vehicle
ego: Vehicle
rear: Vehicle
emcyBrakeWarning()
ref Needham-Schroeder PublicKey
«SecurityProtocolReference»
roleMap = {
roleParameter = alice: Alice,
roleArgument = ego: Vehicle
propertyMap = {
[propertyParameter = privKeyA,
propertyArgument = ego.privKeyVe],
[...],
}
}
cre
«DigitalSignature»
privateKey: ego.privKeyVe
publicKey: ego.pubKeyVe

Figure 5.5: Exemplarily application of the SECURITY PROTOCOL TEMPLATE PROFILE

5.3 Specification of Misuse Cases

This section introduces the MISUSE CASE MODELING PROFILE for integrating misuse cases into scenario-based requirements specifications. First, we present our modeling approach to specify misuse cases based on MSDs. Second, we describe the adaptation of the Play-out algorithm to enable the simulative validation of the resulting scenario-based requirements specification.

5.3.1 The MISUSE CASE MODELING PROFILE in Detail

SINDRE AND OPDAHL [SO05] introduce *misuse cases* as an extension of UML use cases to determine and communicate security requirements. Misusers intentionally or inadvertently initiate a misuse case and threaten at least one use case. The misuse case defines the sequence of actions that the misusers perform to interact with the system under development to exploit or harm the use case.

The MISUSE CASE MODELING PROFILE provides means to specify misuse cases against the system under development. The MISUSE CASE MODELING PROFILE is depicted in Figure 5.6 and is based on the Modal profile and the metamodel presented by SINDRE AND OPDAHL [SO05].

As mentioned in Section 2.1.1, an MSD specification is structured by means of MSD use cases where each use case encapsulates requirements on the communication behavior of the system under development. We introduce the stereotype Misuse Case extending the metaclass UML::Collaboration to distinguish between an MSD misuse case and an MSD use case. Furthermore, we introduce the stereotype threatens extending the metaclass UML::Dependency to

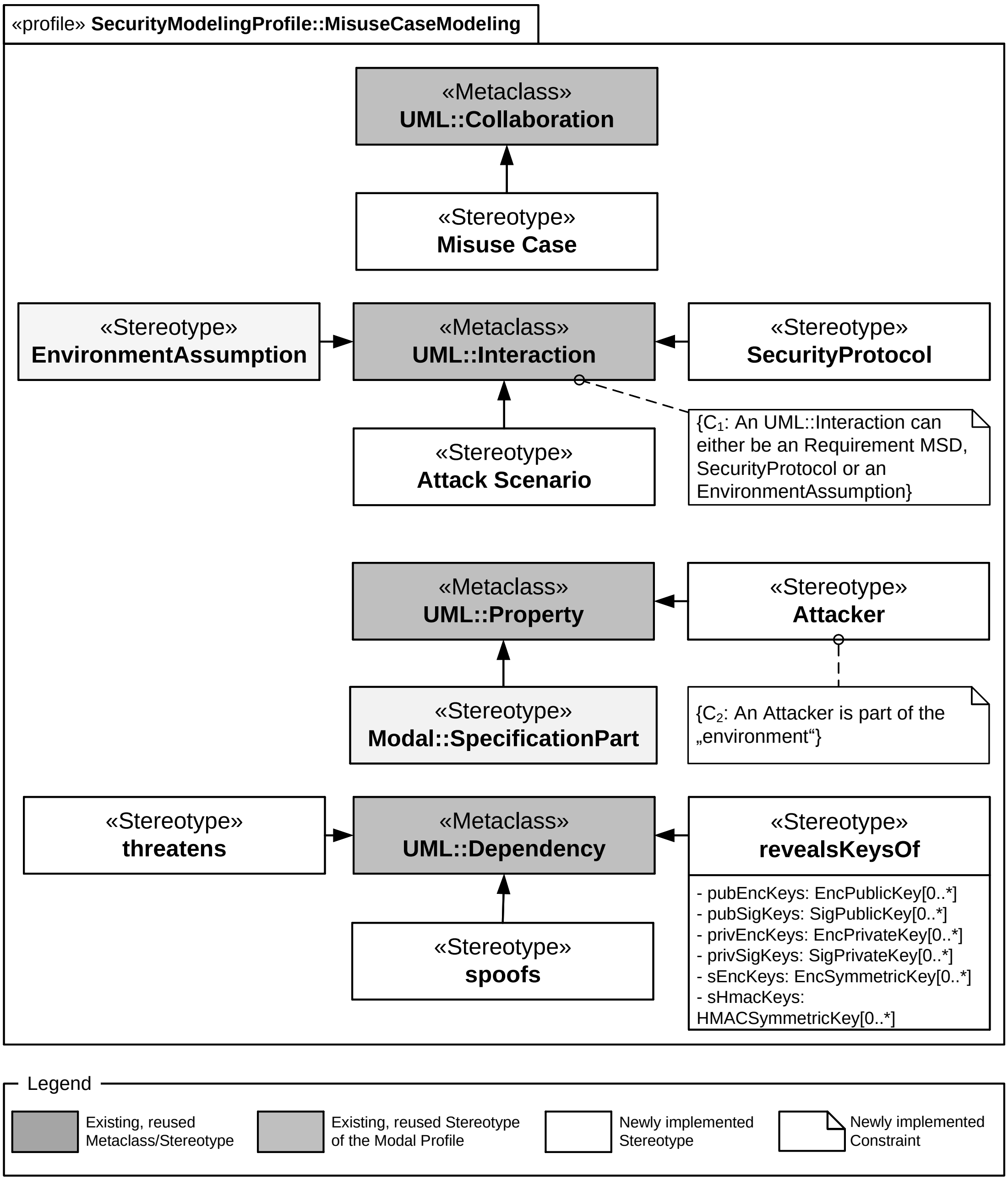

«profile» SecurityModelingProfile::MisuseCaseModeling
«Metaclass» UML::Collaboration
«Stereotype» Misuse Case
«Stereotype» EnvironmentAssumption
«Metaclass» UML::Interaction
«Stereotype» SecurityProtocol
«Stereotype» Attack Scenario
{C$_1$: An UML::Interaction can either be an Requirement MSD, SecurityProtocol or an EnvironmentAssumption}
«Metaclass» UML::Property
«Stereotype» Attacker
«Stereotype» Modal::SpecificationPart
{C$_2$: An Attacker is part of the „environment“}
«Stereotype» threatens
«Metaclass» UML::Dependency
«Stereotype» revealsKeysOf
- pubEncKeys: EncPublicKey[0..*]
- pubSigKeys: SigPublicKey[0..*]
- privEncKeys: EncPrivateKey[0..*]
- privSigKeys: SigPrivateKey[0..*]
- sEncKeys: EncSymmetricKey[0..*]
- sHmacKeys: HMACSymmetricKey[0..*]
«Stereotype» spoofs
Legend
Existing, reused Metaclass/Stereotype
Existing, reused Stereotype of the Modal Profile
Newly implemented Stereotype
Newly implemented Constraint

Figure 5.6: The MISUSE CASE MODELING PROFILE in Detail

specify that an MSD misuse case threatens an MSD use case. Moreover, an MSD misuse case only contains Attack Scenarios. An Attack Scenario describes the interaction between an attacker and the system under development based on MSDs. The stereotype Attack Scenario extending the metaclass UML::Interaction enables to distinguish between an Attack Scenario and a security protocol or a requirements/assumption MSD. We define the constraint C_1 to ensure that only one stereotype is applied to the UML::Interaction.

The participants of a misuse case do not differ from participants in requirements/assumption MSDs. The participants may possess properties and send and receive messages with no, one or more parameters. A misuse case also includes an attacker, specified by the stereotype Attacker. An attacker is part of the environment and aims to harm the system. As participants, an attacker may have properties and send and receive messages. Moreover, an attacker may spoof one or more participants within an attack scenario. Thus, we introduce the stereotype spoofs extending the metaclass UML::Dependency to specify that an attacker spoofs a participant.

An attacker may have initial knowledge, e.g., knowing other participants' public and/or private keys. To realize this, the MISUSE CASE MODELING PROFILE provides the stereotype revealedKeysOf extending the metaclass UML::Dependency. The stereotype shall be used only in a UML::Collaboration with annotated stereotype Misuse Case. The source is always an attacker (e.g., role with annotated stereotype Attacker), while the target specifies the role that owns the knowledge/property that the attacker reveals. Thus, the stereotype revealedKeysOf encompasses lists for cryptographic keys defined in the SECURITY MODELING PROFILE (cf. Section 3.4.2).

5.3.2 Extension of the Runtime Semantics to Support the MISUSE CASE MODELING PROFILE

In this section, we introduce extensions to the semantics of MSDs (cf. Section 2.1.2 and Section 3.5) necessary to define the behavior of misuse cases specified by means of the MISUSE CASE MODELING PROFILE. Thereby, we enable the simulative validation of misuse cases using the Play-out algorithm.

As defined in Section 5.3.1, an attacker is part of the environment, i.e., not controlled by the system under development. Thus, as environment objects, an attacker can send messages at any time if the message does not cause any violation in an active MSD. Moreover, an attacker may spoof other roles in the attacked MSD use case. The handling of spoofed roles requires an extension to the definition of unification.

Originally, a message event is unifiable with a message in an MSD if the event name equals the message name and if the sending and receiving lifelines of the message are bound to the sending and receiving objects. If an attacker spoofs a role, a message is *attack unifiable* if the event name equals the message name and if either the sending or receiving lifelines of the message are bound to the sending or receiving objects, respectively. Moreover, the spoofed role is marked as *attacked*.

As explained in Section 3.4, a message event may be secured by means of cryptographic primitives. A message event with applied cryptographic primitives is *attack unifiable* regardless of whether the attacker knows the correct cryptographic keys. However, if the attacker knows the correct cryptographic key (i.e., the dependency revealedKeysOf contains the corresponding key), the attacker can change the message during the execution of the attack scenario.

If the cut is in front of the first message of an attack scenario, the requirements engineer can decide whether he/she wants to take the execution path of the requirements MSDs or the attack scenario. If he/she decides to take the execution path of the attack scenario, the attack scenario becomes active. As in the case of requirements MSDs, an attack scenario progresses if message events occur that are unifiable with messages in the attack scenario.

Intuitively, an attack scenario is successful if it progresses without causing any violation in the remaining MSD specification. Moreover, the successful execution of an attack scenario must not cause a liveness violation in an affected MSD. A requirement MSD is affected by an attack scenario if it contains roles marked as attacked. A successful attack scenario causes a *security violation*. As a hot violation, a security violation violates the requirements and causes the Play-out algorithm to terminate.

5.4 Integrating Security Protocols into Scenario-based Requirements Specifications

Requirements engineers can use existing and verified security protocols to establish a secure communication channel. Although security protocols remain the same regardless of the application in which they are used, they cannot be modeled just once using MSDs and then used in arbitrary applications. The reason for this is that the MSD specification of a security protocol and the MSD specification describing the functional requirements on the communication behavior do not rely on the same model elements (cf. Figure 5.7). Both are independent models with different types used to create the specification. Thus, it is not easily possible for requirements engineers to use a security protocol within the scenario-based requirements specification. Hence, requirements engineers need a systematic approach for integrating security protocols into scenario-based requirements specifications.

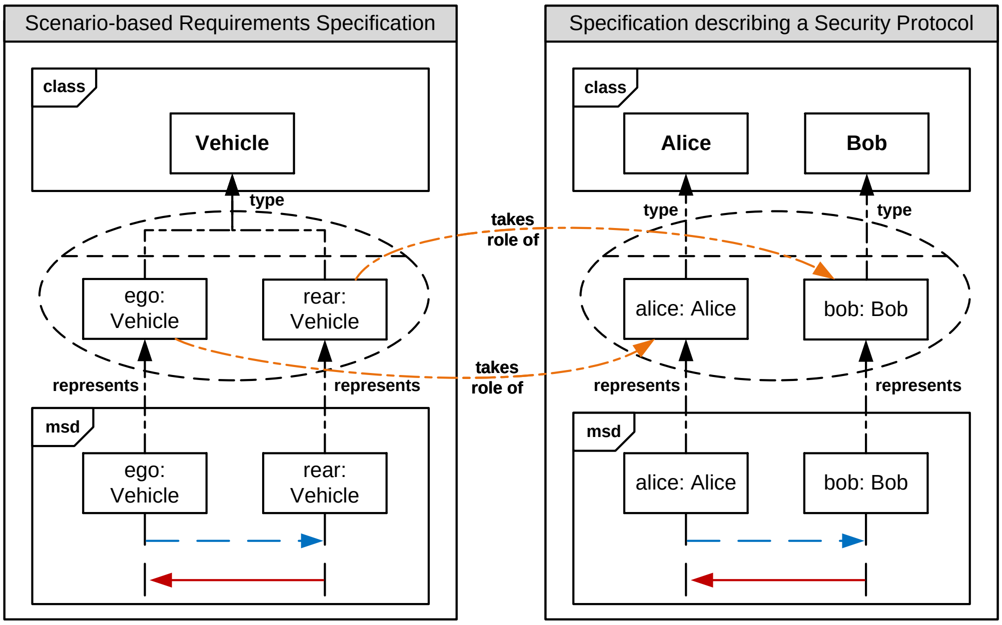


Figure 5.7: Illustration of the problem of integrating a security protocol into the scenario-based requirements specification of an application

Intuitively, one possible idea to relate the classes of the two MSD specifications would be to use the concept of UML inheritance. For that, the UML classes used as types for the scenario-based requirements specification would extend the UML classes used as types for the security protocol specification. Thus, the scenario-based requirements specification types inherit the operations and properties, which enable their representative roles to participate in the security protocol [+Jaz15].

However, this solution has several drawbacks. "First, [...] multiple roles in a scenario-based requirements specification can be typed by the same class. Thus, it would be unclear which role of the scenario-based requirements specification takes which role in the security protocol. Second, if a role of the scenario-based requirements specification executes the same security protocol with different participants, the same properties of a type must be used in different contexts and, thus, would be overwritten" [*KTD$^+$22]. Next, we present a template-based modeling approach to adapt a security protocol to the application context of a scenario-based requirements specification to address these drawbacks.

5.4.1 The SECURITY PROTOCOL TEMPLATE PROFILE in Detail

In this section, we introduce our SECURITY PROTOCOL TEMPLATE PROFILE, which implements a template-based modeling approach to integrate security protocols specified by the SECURITY MODELING PROFILE in a scenario-based requirements specification based on MSDs. The SECURITY PROTOCOL TEMPLATE PROFILE is depicted in Figure 5.8 and provides two main concepts.

First, security engineers need modeling elements to annotate those parts of the security protocols that must be adapted to the application context. Typically, this includes the participants of the security protocols and their properties (e.g., cryptographic keys). Our SECURITY PROTOCOL TEMPLATE PROFILE introduces the concept of *template parameters* annotating elements that the scenario-based requirements specification may substitute. The stereotype RoleTemplate can be applied to the lifelines of the MSD, and the stereotype PropertyTemplate can be applied to the properties of the UML classes.

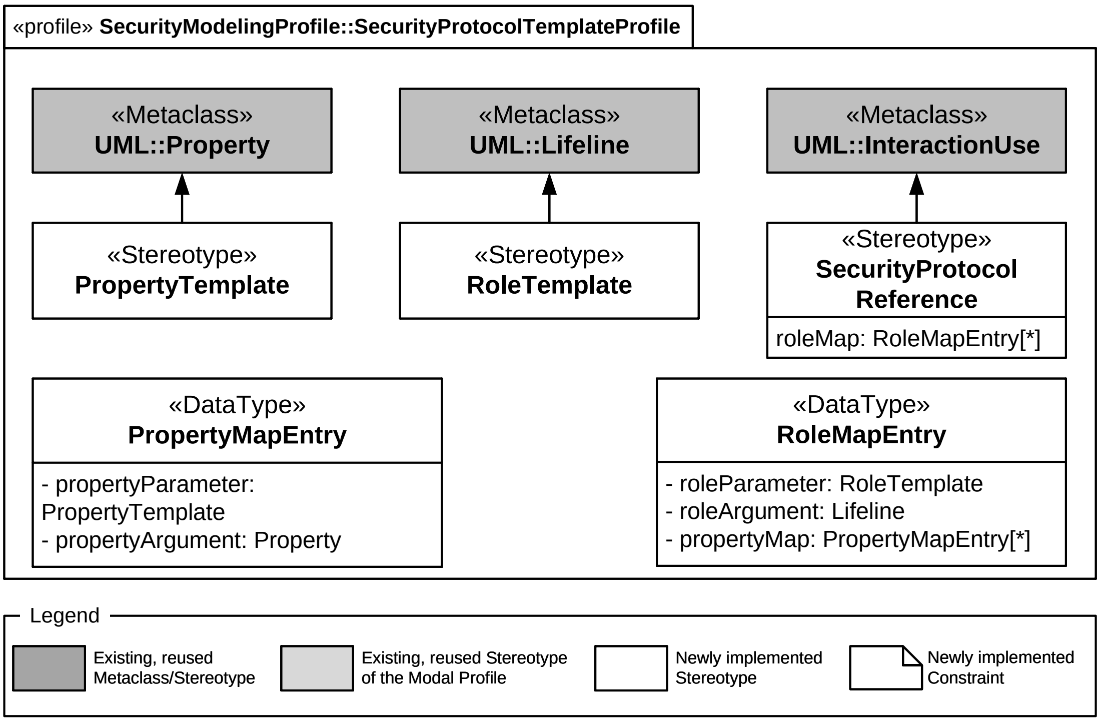

«profile» SecurityModelingProfile::SecurityProtocolTemplateProfile
«Metaclass» UML::Property
«Metaclass» UML::Lifeline
«Metaclass» UML::InteractionUse
«Stereotype» PropertyTemplate
«Stereotype» RoleTemplate
«Stereotype» SecurityProtocol Reference
roleMap: RoleMapEntry[*]
«DataType» PropertyMapEntry
- propertyParameter: PropertyTemplate
- propertyArgument: Property
«DataType» RoleMapEntry
- roleParameter: RoleTemplate
- roleArgument: Lifeline
- propertyMap: PropertyMapEntry[*]
Legend
Existing, reused Metaclass/Stereotype
Existing, reused Stereotype of the Modal Profile
Newly implemented Stereotype
Newly implemented Constraint

Figure 5.8: The SECURITY PROTOCOL TEMPLATE PROFILE in Detail

Second, requirements engineers need modeling elements to reference a security protocol and substitute the elements annotated as template parameters. A UML::InteractionUse enables the reuse of existing interactions. However, a UML::InteractionUse does not allow to specify structural substitutions. Thus, we introduce the stereotype SecurityProtocolReference, which extends the UML::InteractionUse. The stereotype SecurityProtocolReference inherits the property refersTo of type UML::Interaction. This property is used to specify the interaction that defines the behavior of the security protocol.

"In addition to the inherited properties, the stereotype SecurityProtocolReference encompasses a RoleMap. The RoleMap is a list of type RoleMapEntry mapping the roles from the security protocol template to roles in the referencing MSD. The list has at least two elements, the initiator and the responder of the security protocol. The concrete mapping is specified in the data type RoleMapEntry. For this, the data type has two properties: roleParameter and roleArgument. While the roleParameter corresponds to a role in the security protocol, the roleArgument captures the role of the application context. Apart from the two properties used to define the role mapping, the RoleMapEntry encompasses a property called PropertyMap. It maps properties from the security protocol template to properties in the referencing security protocol" [*KTD$^+$22].

5.4.2 Extension of the Runtime Semantics to Support the SECURITY PROTOCOL TEMPLATE PROFILE

We adapt the Play-out algorithm to support the modeling approach for integrating security protocols in scenario-based requirements specifications presented in the previous section. The adaptation is restricted to evaluating the SecurityProtocolReference and resolving its defined properties. Apart from that, the algorithm behaves as described in Section 3.5.

Figure 5.9 depicts the resulting MSD after the Play-out algorithm resolved the SecurityProtocolReference defined in the MSD depicted in Figure 5.7.

"If the cut of an active MSD is immediately before the SecurityProtocolReference, it is directly evaluated. Therefore, the Play-out algorithm resolves the substitutions specified by the SecurityProtocolReference and adds the messages defined by the referenced security protocol to the referencing MSD. The substitution process encompasses two steps" [*KTD$^+$22].

First, the Play-out algorithm evaluates the role mapping and substitutes the roles accordingly. For example, in Figure 5.7, the role ego: Vehicle substitutes the alice: Alice and the role rear: Vehicle substitutes the bob: Bob. Furthermore, the role trustedServer: TrustedThirdParty is added to the resulting specification since it is not part of any mapping. "Note that it is not relevant whether the lifelines of the security protocol template are part of the environment or the system. Instead, the specification kind is taken from the referencing lifelines. As a result of the substitution, the roles of the requirements specification can now send and receive the messages of the security protocol" [*KTD$^+$22].

Second, the Play-out algorithm evaluates the property mapping specified for each role mapping. The Play-out algorithm searches for all references to a property in the security protocol specification and replaces them with the new reference. This applies to message parameters and to the references in the stereotypes of the SECURITY MODELING PROFILE. After completing the two steps, the Play-out algorithm continues as described in Section 3.5.

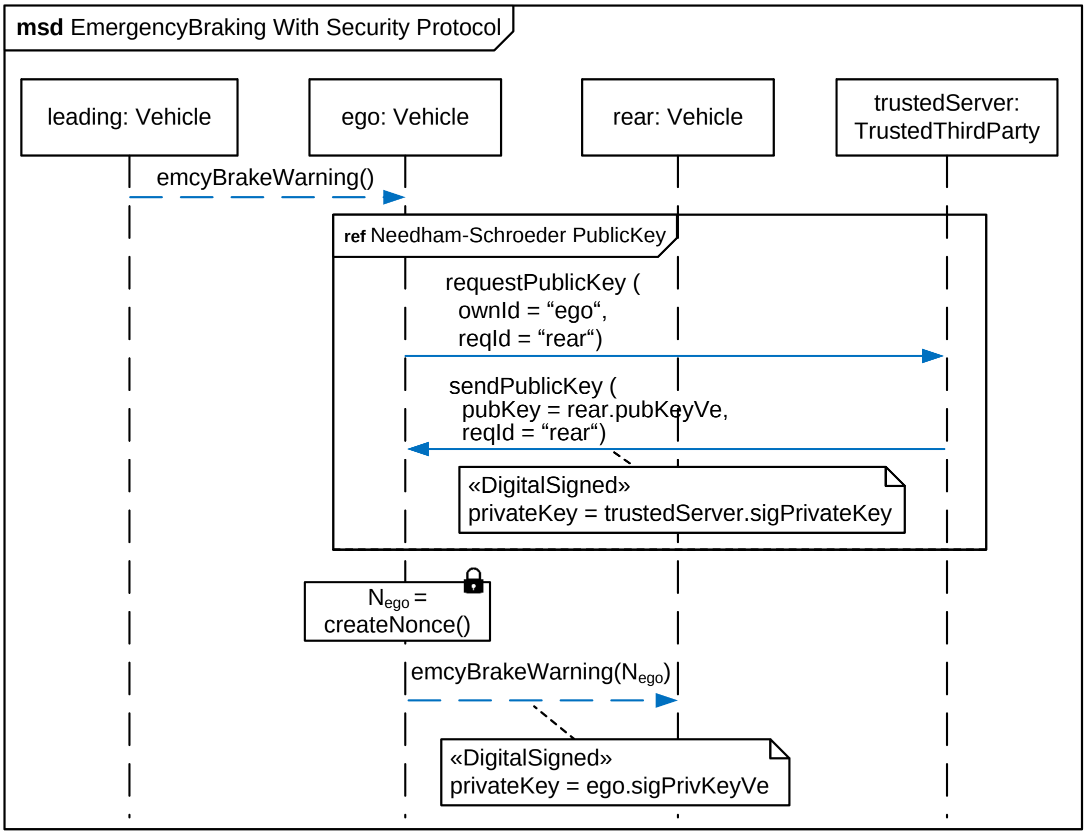

msd EmergencyBraking With Security Protocol
leading: Vehicle
ego: Vehicle
rear: Vehicle
trustedServer: TrustedThirdParty
emcyBrakeWarning()
ref Needham-Schroeder PublicKey
requestPublicKey (ownId = "ego", reqId = "rear")
sendPublicKey (pubKey = rear.pubKeyVe, reqId = "rear")
«DigitalSigned» privateKey = trustedServer.sigPrivateKey
N_ego = createNonce()
emcyBrakeWarning(N_ego)
«DigitalSigned» privateKey = ego.sigPrivKeyVe

Figure 5.9: Resulting MSD after the handling of the SecurityProtocolReference

5.5 Implementation

This section presents an overview of our prototypical implementation to support and evaluate the concepts described throughout this chapter. The implementation is integrated into the Eclipse-based SCENARIOTOOLS MSD tool suite [ST-MSD]. In particular, we present the architecture of our implementation in Section 5.5.1 and the user interface in Section 5.5.2.

5.5.1 Security ScenarioTools (Software Architecture)

Figure 5.10 depicts the software architecture of our prototypical implementation. The entire implementation is based on the Eclipse Modeling Framework (EMF) [EMF] and Eclipse Papyrus [Papyrus].

The extended tool suite SECURITY SCENARIOTOOLS MSD provides the MISUSE CASE MODELING PROFILE and SECURITY PROTOCOL TEMPLATE PROFILE as well as the corresponding runtimes. Both profiles extend the UML metamodel as part of the component UML2 and the Modal profile as part of the SCENARIOTOOLS MSD tool suite. The components Misuse Case Runtime and Security Template Runtime implement the runtime extensions necessary to simulate the misuse cases and the template-based MSD specifications, respectively.

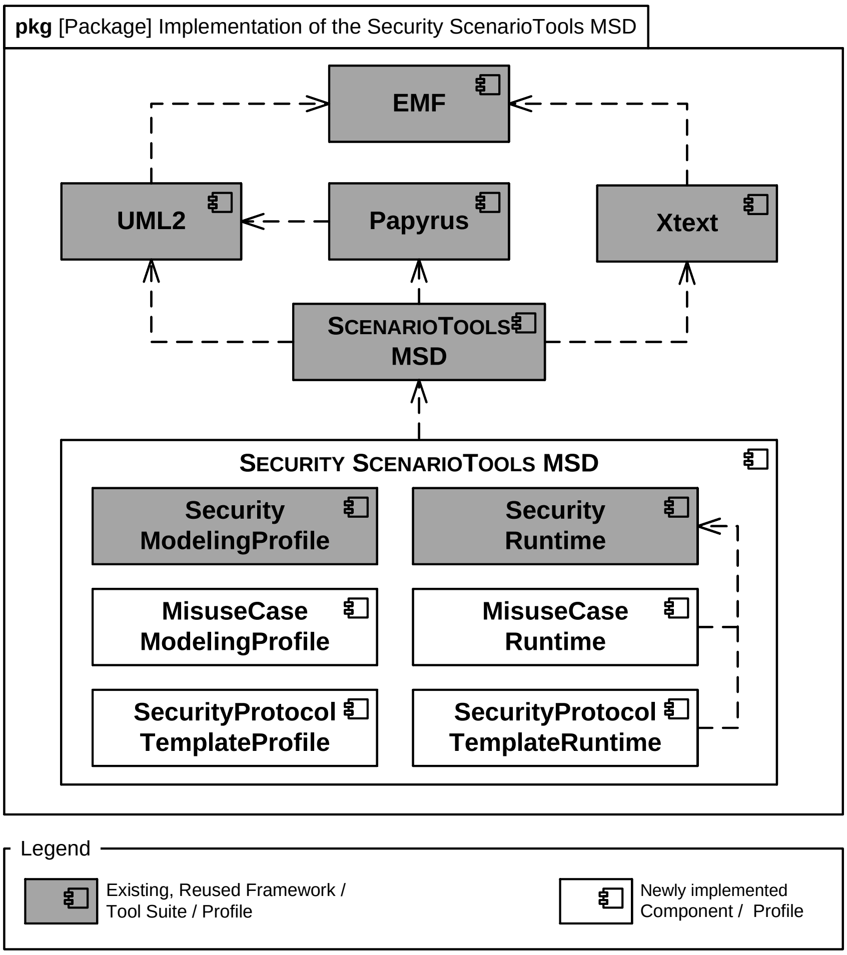

pkg [Package] Implementation of the Security ScenarioTools MSD
EMF
UML2
Papyrus
Xtext
SCENARIOTOOLS MSD
SECURITY SCENARIOTOOLS MSD
Security ModelingProfile
Security Runtime
MisuseCase ModelingProfile
MisuseCase Runtime
SecurityProtocol TemplateProfile
SecurityProtocol TemplateRuntime
Legend
Existing, Reused Framework / Tool Suite / Profile
Newly implemented Component / Profile

Figure 5.10: Detailed overview of the Software Architecture

5.5.2 Security ScenarioTools (User-Interface)

The user interface of the SECURITY SCENARIOTOOLS MSD tool suite provides a modeling and a simulation perspective. The modeling and simulation perspectives are similar to the ones described in Section 3.6.2. The modeling perspective is based on the Papyrus modeling editors and is used to specify the different parts of the MSD specification. The simulation perspective provides information necessary for the simulation of an MSD specification. In particular, the simulation perspective provides a set of enabled messages and the user can select the message that should be sent next.

5.6 Evaluation

In this section, we conduct a case study based on the guidelines by KITCHENHAM ET AL. [KPP95] and RUNESON ET AL. [RH09; Run12] for evaluating our approach. In our case study, we investigate the applicability in practice of our MISUSE CASE MODELING PROFILE and SECURITY PROTOCOL TEMPLATE PROFILE.

5.6.1 Case Study Context

We examine five evaluation questions (EQ):

> **EQ1** Does the MISUSE CASE MODELING PROFILE enable the specification of misuse cases for real-world scenario-based requirements specifications?
>
> **EQ2** Does the extension to the runtime semantics enable the simulation of misuse cases against the system under consideration?
>
> **EQ3** Does the SECURITY PROTOCOL TEMPLATE PROFILE enable the specification of security protocol templates for real-world security protocols?
>
> **EQ4** Does the SECURITY PROTOCOL TEMPLATE PROFILE enable the use of security protocol templates in a scenario-based requirements specification?
>
> **EQ5** Does the extension to the runtime semantics enable the simulation of scenario-based requirements specifications that include security protocol templates?

For this purpose, we specify two use cases from the automotive domain and reuse the 14 security protocols that have been successfully modeled in the evaluation of the SECURITY MODELING PROFILE (cf. Section 3.7). The first case is the Emergency Braking & Evasion Assistance System (EBEAS). The EBEAS is an advanced driver assistance system that is supposed to reduce the risk of rear-end collisions in case of obstacles in front of a vehicle. Therefore, the EBEAS combines autonomous braking systems and autonomous emergency steering systems with vehicle-to-vehicle communication technology.

The second case describes an over-the-air update scenario. By means of an over-the-air update, OEMs can install a software update without bringing a vehicle into the repair shop. For this, the OEM or a supplier initiates the software update. Then, this update is forwarded to the vehicle using mobile networks like GSM, UMTS, or LTE. Once a vehicle receives the update, it stores the update until there is an opportunity for the safe execution of the installation.

To answer the questions, we use the scenario-based requirements specification of the two use cases as a basis. We specify different misuse cases against the two use cases. Moreover, we integrate 14 different security protocols into each use case. The selected security protocols use different cryptographic primitives and partially rely on a trusted third party. Thus, they present a broad range of possible security protocols and, thereby, cover all elements of our template-based approach.

5.6.2 Setting the Hypotheses

We define the following hypotheses for this case study.

> **H1** The MISUSE CASE MODELING PROFILE enables the specification of misuse cases against scenario-based requirements specifications. We rate H1 as fulfilled if each misuse case can be specified using solely the approach presented in Section 5.3.

H2 The specified misuse cases can be evaluated correctly according to the runtime semantics defined in Section 5.3. This includes the validation of two different kinds of MSD specifications. First, the use case does not include any security measures. Second, the use case includes security measures to secure the communication between the involved rules. We rate H2 as fulfilled if the misuse cases are correctly analyzed during the simulative validation by means of the extended Play-out algorithm.

H3 All security protocols can be specified as security protocol templates using our SECURITY PROTOCOL TEMPLATE PROFILE as presented in Section 5.4.1. We rate H3 as fulfilled if each security protocol can be specified as a security protocol template using solely the approach presented in Section 5.4.1.

H4 All security protocol templates can be referenced in a scenario-based requirements specification. Therefore, all parameters specified by the template must be substituted by model elements of the scenario-based requirements specification using our approach presented in Section 5.4.1. We consider H4 as fulfilled if all security protocol templates can be correctly used in the two scenario-based requirements specifications using our approach presented in Section 5.4.1.

H5 The references to all security protocol templates can be evaluated correctly according to the runtime semantics defined in Section 5.4.2. We rate H5 as fulfilled if the contents of the security protocol templates are correctly inserted into the referencing scenario-based requirements specifications during the simulative validation by means of the extended Play-out algorithm.

5.6.3 Validating the Hypotheses

In the following, we validate each hypothesis separately using the prototypical implementation of our approach described in Section 5.5.

Validating Hypothesis H1

To validate H1, we created an MSD specification for the EBEAS and the over-the-air update scenario. Moreover, we specified five misuse cases against each use case. All misuse cases have in common that the attacker is actively interacting with the system under consideration. During the modeling process, we checked whether the MISUSE CASE MODELING PROFILE is expressive enough to specify misuse cases or whether any language constructs are missing. Afterward, we executed the Papyrus validate function on each MSD specification to check for syntactical problems and unfulfilled OCL constraints. Papyrus returns that no problem exists in any MSD specification.

Validating Hypothesis H2

To validate H2, we simulated the MSD specifications for the EBEAS and the over-the-air update scenario. We executed the algorithm step by step and checked in each step whether the simulation state was correct. The correct simulation includes the creation of attack scenarios if we decide to take the execution path of an attacker and the correct unification of messages in the attack scenarios according to the runtime semantics defined in Section 5.3.2. We repeated the simulation several times to check whether the simulation behavior was deterministic and did not find any deviations in the different simulation runs.

Validating Hypothesis H3

To validate H3, we modified the MSD specification for the 14 security protocols used to evaluate the SECURITY MODELING PROFILE. For each MSD specification, we added the stereotype «RoleTemplate» to each lifeline; except for lifelines that represent a trusted third party. Additionally, we added the stereotype «PropertyTemplate» to each property of the classes typing the annotated lifelines. As before, we used the Papyrus validate function on each MSD specification to check for syntactical problems and unfulfilled OCL constraints and Papyrus did not report any.

Validating Hypothesis H4

To validate H4, we used the MSD specification for the EBEAS and the over-the-air update scenario. We added a UML::InteractionUse to the MSD specifications and imported all 14 security protocols separately. If necessary, we added additional properties for the cryptographic keys to the classes of the EBEAS and the over-the-air update scenario. Afterward, we applied the stereotype «SecurityProtocolReference» to the UML::InteractionUse and specified the Role2RoleMap accordingly. As before, we used the Papyrus validate function on each MSD specification to check for syntactical problems and unfulfilled OCL constraints and Papyrus did not report any.

Validating Hypothesis H5

To validate H5, we simulated the MSD specifications for the EBEAS and the over-the-air update scenario, including the security protocols and the misuse cases created before. We executed the algorithm step by step and checked in each step whether the simulation state was correct. The correct simulation includes inserting the security protocol template in the active MSD. Moreover, all messages must be in the correct order and the properties must be substituted and referenced correctly where necessary. Finally, we also used the execution path of attack scenarios to check whether the integrated security protocols correctly mitigate these. As before, we repeated the simulation several times to check whether the simulation behavior was deterministic and did not find any deviations in the different simulation runs.

5.6.4 Analyzing the Results

By using the MISUSE CASE MODELING PROFILE, we can specify and validate misuse cases against the EBEAS and the over-the-air update scenario. However, the specified misuse cases only consider active attackers. The MISUSE CASE MODELING PROFILE does not support the specification of passive attackers, e.g., attackers that only eavesdrop on the communication between two participants. Future work may investigate the possibility of creating a domain-specific language to model eavesdropping attacks. Additionally, future work may investigate the possibility of specifying security properties for the MSD specification and generate misuse cases based on these properties. This would reduce the manual effort for a requirements engineer. Due to the mentioned restrictions, we consider H1 only as partially fulfilled and H2 as fulfilled.

Moreover, by using the SECURITY PROTOCOL TEMPLATE PROFILE, we can model all relevant information for specifying the 14 security protocols as security protocol templates and reference them in the MSD specifications of the two cases. Thus, we consider H3 and H4 as fulfilled. Finally, we executed the MSD specifications that reference the security protocol template using our extended Play-out algorithm. During the execution, the SecurityProtocolReference was evaluated. Subsequently, we observed that for each security protocol and simulation run the contents of the security protocol template are inserted correctly into the active MSD. Additionally, all messages were in the correct order and were sent between the lifelines specified as arguments to their original sending and receiving lifelines. Furthermore, the properties were correctly substituted and referenced where necessary. Thus, we consider H5 as fulfilled.

To conclude the case study, the (partially) fulfilled hypotheses indicate that our approach is applicable in practice.

5.6.5 Threats to Validity

The threats to validity in our case study are as follows:

Construct Validity

The case study was designed and conducted by the same researcher who developed the approach. Since the researcher might have a bias toward the developed approach, the case study would be more significant if security experts had modeled the security protocols. To mitigate this, we discussed the case study design and its research questions with other researchers.

External Validity

For the evaluation of H1–H2, we only considered two use cases and a small set of misuse cases against these use cases. Thus, we cannot generalize the fulfillment of the hypotheses for all possible combinations of use and misuse cases. Nevertheless, the specified misuse cases present typical examples; at least for active attackers; thus, we do not expect large deviations from other misuse cases.

For the evaluation of H3–H5, we only considered two application scenarios and 14 different security protocols. Thus, we cannot generalize the fulfillment of the hypotheses for all possible combinations of application scenarios. Nevertheless, the selected application scenarios and security protocols represent typical examples; thus, we do not expect large deviations from other examples.

Reliability

The case study was conducted based on the prototype implementation that might not be available in the future. To mitigate this, the implemented concepts are defined throughout this chapter and can be newly implemented.

Moreover, the case study information and the resulting models might not be available in the future. To mitigate this threat, we discussed the exemplary application of our MISUSE CASE MODELING PROFILE and our SECURITY PROTOCOL TEMPLATE PROFILE in detail in Section 5.2.

5.7 Related Work

This section discusses related work in two categories: First, in Section 5.7.1, we present approaches to identify and analyze misuse cases against the system under development. Second, in Section 5.7.2, we present approaches that consider the specification of security mechanisms and their reuse in developing the system under development.

5.7.1 Approaches for the Identification and Analysis of Misuse Cases

"Many approaches exist that specify misuse cases [SO05] or abuse cases [PX05] as negative scenarios to specify what is not allowed to happen during the execution of the system. Furthermore, several other approaches worked on the elicitation and specification of security requirements, for example, security use cases [Fir03; Fir07]; UMLsec [Jür02; Jür05], a framework for security requirements engineering [HLM$^+$08]; SQUARE [MS05]; security requirements methods based on i* framework [LYM03]; Secure Tropos [GMZ06]" [*Koc18].

However, all mentioned approaches mainly work on eliciting and documenting threats and security requirements. "They do not provide any integration into existing functional requirements engineering methodologies or development processes. In addition, they do not provide sufficient analysis techniques to validate whether the specified security requirements are fulfilled and that a potential attacker is not able to execute the specified attack" [*Koc18].

WHITTLE ET AL. [WWH08] develop an approach to formalize misuse cases by means of extended interaction overview diagrams (EIODs) and execute these misuse cases against scenario specifications of the system under development to validate whether it is sufficiently secure to mitigate them. They want to overcome the issues with informal and non-analyzable misuse cases. However, their approach specifying functional aspects does not contain any notation of modality or time. Thus, we adapted their approach in the context of scenario-based requirements specifications based on MSDs.

5.7.2 Approaches for the Specification of Security Mechanisms

"Several approaches consider the specification of security mechanisms and their reuse in an application's requirements and design phase. For example, MOUHEB ET AL. [MTL$^+$09] propose an aspect-oriented modeling approach to integrate security mechanisms (e.g., security protocols like TLS) into software design models (e.g., UML Interactions). Therefore, the authors present a

UML profile to specify security mechanisms. This profile introduces a transparent and automatic approach that weaves the security mechanism into the design models. The approach is similar to ours since the authors present an approach to reuse security mechanisms that people with limited security knowledge can use. However, while the authors only consider the behavior of a security mechanism, we also consider structural properties. Therefore, we can better adapt the security mechanisms to the application context. Furthermore, we provide a simulative validation of the resulting specification" [*Koc18].

"Ray et al. [RLF^{+}04] present an approach to incorporate role-based access control (RBAC) policies into the design of applications. The RBAC policies are defined independently of the application that has to implement the policies. To bridge the gap between the policy definition and the application design, Ray et al. specify policies by means of UML diagram templates. These UML diagram templates can be used to integrate application-specific policies into the design of the application. The approach is similar to our approach since the authors present an approach to reuse security mechanisms. Furthermore, both approaches enable the validation of the resulting specifications. However, in contrast to our approach, Ray et al. consider role-based access control and only consider the specification and analysis of structural models like UML class and object diagrams" [*Koc18].

"Moreover, many approaches consider the modeling of system and software security using UML and SysML. For example, Jürjens [Jür02] propose UMLSec as a model-driven approach for integrating security-related information in UML specifications. UMLSec encompasses a UML profile for expressing security mechanisms, including secure information flow, confidentiality, and access control" [*Koc18].

"Lobberstedt et al. [LBD02] present SecureUML, a UML-based modeling language for model-driven security. The approach enables the design and analysis of secure, distributed systems by adding mechanisms to model role-based access control. Furthermore, they provide an automatic generation of access control infrastructures based on the specified models" [*Koc18].

"Roudier and Apvrille [RA15] present SysML-Sec, a modeling approach based on SysML that enables the specification of security aspects for embedded systems. They enhance SysML block and state machine diagrams to capture security-related features. In contrast to the other two approaches, SysML-Sec also covers safety-related features" [*Koc18].

"However, all these approaches mainly focus on extending the UML/SysML notations to reflect security concerns better. In contrast, our approach addresses the systematic reuse of existing security mechanisms. Therefore, our approach separates the specification and application of security mechanisms and allows engineers with limited knowledge to rely on these mechanisms to secure their applications" [*Koc18].

5.8 Summary

This chapter presents an extension to our scenario-based requirements engineering methodology [*HFK^{+}16b] incorporating functional and security requirements. In particular, we introduce the Misuse Case Modeling Profile to specify misuse cases against a scenario-based requirements specification and the Security Protocol Template Profile to reuse security protocols specified by the Security Modeling Profile systematically. Moreover, we extend the runtime semantics of MSD to enable the correct interpretation of the new modeling elements in the Play-out algorithm. Thereby, requirements engineers can validate whether the specified misuse cases are successful or whether the specified security measures (e.g., usage of primitives or integration of security protocols) are sufficient to mitigate the misuse cases. Our evaluation shows that the MISUSE CASE MODELING PROFILE and the SECURITY PROTOCOL TEMPLATE PROFILE are applicable in practice.

6

Conclusion

This chapter summarizes the challenges and contributions of this thesis in Section 6.1 and points to directions for future work in Section 6.2.

6.1 Summary

The widespread usage of software-intensive systems significantly increases the risk of cyber-attacks. Thus, security has become one of the most crucial technology risks for the world's population [Wor23]. However, specifying and applying state-of-the-art security measures like security protocols requires deep knowledge of the used concepts and tools. In particular, security engineers use several symbolic model checkers like PROVERIF and TAMARIN to verify whether a security protocol is secure concerning secrecy and authentication. However, since both model checkers have their own textual modeling and query languages, they must learn both languages and repeatedly model a security protocol, including its queries in several languages. Moreover, in developing a software-intensive system, requirements engineers model the message-based communication between and within software-intensive systems. However, current requirements engineering approaches concentrate on functional or security requirements. Thereby, requirements engineers do not identify conflicts introduced by further security measures, e.g., unintended timing behavior of the system due to an exhausting message exchange caused by the security protocol.

The contributions of this thesis enable security and requirements engineers to apply security techniques without deep knowledge of the underlying concepts and tools. We successfully implemented all concepts as extensions to the tool suite SCENARIOTOOLS MSD and showed in case studies based on realistic examples that all concepts are applicable in practice.

As our first contribution, we provide the UML-compliant SECURITY MODELING PROFILE for the scenario-based specification of security protocols. On the one hand, the SECURITY MODELING PROFILE enables security engineers to model security protocols in a scenario-based way. On the other hand, the SECURITY MODELING PROFILE enables requirements engineers to specify requirements on the message-based communication for their application by using cryptographic primitives. Moreover, the SECURITY MODELING PROFILE provides several constraints to avoid specification errors by the security engineers or the requirements engineers.

As our second contribution, we present VICE (VIsual Cryptography vErifier), a model-checking approach for automatically verifying security protocols in the symbolic model. VICE reduces the knowledge necessary to apply model checking for security protocols. In particular, VICE provides a model transformation concept to transform a security protocol into the input language of the two symbolic model checkers PROVERIF and TAMARIN. Moreover, VICE automatically derives an initial set of analysis queries that the model checker shall verify to decide whether the protocol is secure concerning secrecy and authentication. Furthermore, VICE provides the results of the analysis to the security engineer.

As our third contribution, we present a scenario-based approach for specifying and validating functional and security requirements. In particular, we extend our scenario-based requirements engineering methodology [*HFK$^+$16b] with techniques from the misuse-case specification. We enable requirements engineers to model conceivable attacks and validate whether the system under development resists the attack. Moreover, to integrate mitigations in the specification of the system under development, we provide a template-based approach to reuse security protocols and configure them to the context of the system under development. By extending existing analysis techniques for MSDs, we enable the requirements engineer to determine in the early phase of the development whether the modeled security requirements are sufficient to avoid threats and whether the security requirements negatively influence the system's functional behavior.

In combination, our three contributions address the increasing risk of cyber-attacks against software-intensive systems. In particular, our contributions help security engineers with little knowledge of security modeling checking to apply state-of-the-art security model checkers like PROVERIF and TAMARIN to verify security protocols. Furthermore, our contributions help requirements engineers with little knowledge of security to use security measures (e.g., cryptographic primitives and/or security protocols) to secure their system under development.

6.2 Future Work

The contributions of this thesis provide several aspects that may be the topic of future research. All contributions require evaluations using security engineers and requirements engineers from various domains, e.g., automotive, avionics, and automation. In the following, we specify six aspects for future research:

Extend VICE to support computational model checkers
Within our thesis, we only considered model checkers in the symbolic but not in the computational model. However, when applying state-of-the-art computational model checkers like CryptoVerif [Bla23] and EasyCrypt [BGH$^+$11], the same challenges occur as for the symbolic model checkers (cf. Section 1.1.1). Thus, future work may extend VICE to support the input and query generation for computational model checkers.

Enable security engineers to specify security protocols using an Alice & Bob notation
VICE provides a generic model transformation concept to transform security protocols into the modeling language of various symbolic model checkers. Within our thesis, we use the SECURITY MODELING PROFILE to specify security protocols in a scenario-based manner. Future work may provide a textual language to specify security protocols based on the Alice & Bob notation. Thereby, our model-checking approach would become applicable to security engineers unfamiliar with UML. While the transformation concepts into the various model checkers are independent of the used specification language, future work

needs to develop a transformation from the AnB specification language to our VerificationModel. Moreover, future work may provide a transformation from the AnB specification language to the SECURITY MODELING PROFILE to enable requirements engineers to reuse the security protocol to secure their system under development.

Enable security engineers to specify custom analysis queries
VICE automatically derives a set of queries to verify whether the security protocol is secure regarding secrecy and authentication. However, in the current approach, security engineers can only adjust the derived analysis queries to their needs or add additional queries to the input model if they change the input file. Future work may provide a domain-specific language for the specification of custom queries and their transformation into security model checkers. In addition, future work may evaluate the applicability of providing suggestions to strengthen the generated queries.

Enable security engineers to adjust the attack model used for the security analysis
VICE automatically generates input for PROVERIF and TAMARIN to enable the analysis of a security protocol in the symbolic model based on the DOLEV-YAO model [DY83]. However, for analyzing real-world security protocols like TLS, it is often necessary to relax the strong assumption of the Dolev-Yao model. Therefore, future work may provide a domain-specific language for specifying assumptions for the attacker's capabilities and their transformation into security model checkers.

Enable security engineers to generate source code for a verified security protocol
Several approaches enable security engineers to generate source code for a specified security protocol. For example, AnBx is a tool for generating a Java-based implementation for security protocols specified in an AnB-based language [BCM+16]. Moreover, MetaCP automatically generates a C-based implementation for a security protocol [MA22]. Contrary to the two mentioned approaches, ProScript provides an approach for implementing secure messaging protocols based on JavaScript and permits the automated extraction of protocol models directly from the implementation to be analyzed amongst others in PROVERIF [KBB17]. Future work may evaluate the applicability of such approaches in the context of VICE and add the most fitting approach.

Enable requirements engineers to select a security protocol from a library
We provide a template-based approach to reuse security protocols in the scenario-based requirements specification of software-intensive systems. Future work may provide a catalog containing a set of verified security protocols. The catalog could provide several filtering opportunities, for example, by name or by the properties the security protocols provide. Moreover, the catalog-based approach could also improve the application of the security protocol by guiding the requirements engineer through the process of adjusting the template to the context of the system under development.

Enable requirements engineers to generate misuse cases
We provide a specification and analysis approach for misuse cases. However, applying this approach can take time depending on the system under development. Future work may provide a domain-specific language to specify properties the system under development has to provide. Based on these properties, various misuse cases could be automatically generated.

Bibliography

The bibliography is structured into four parts: my own publications, the theses I supervised, foreign literature, and tool suites and tool frameworks.

Own Publications

In this section, I do not only list publications that contribute to my PhD thesis but all publications that I wrote during my time of a Research Associate. The publication key of all my publications have the prefix * to identify them easily within this thesis. Details about my contributions to the papers listed under by own publications are given in Appendix C.

[*ABD$^+$19] K. Altemeier, M. Becker, S. Dziwok, T. Koch and S. Merschjohann. “Was fehlt (bisher) um Apps sicher zu entwickeln? — Prozesse, Werkzeuge und Schulungen für sichere Apps by Design”. In: *Projektmanagement und Vorgehensmodelle 2019 (PVM 2019)*. Ed. by M. Mikusz. Gesellschaft für Informatik. Lecture Notes in Informatics (LNI), Oct. 2019.

[*FHK$^+$18a] M. Fockel, J. Holtmann, T. Koch and D. Schmelter. “Formal, Model- and Scenario-based Requirement Patterns”. In: *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*. Jan. 2018.

[*FHK$^+$18b] M. Fockel, J. Holtmann, T. Koch and D. Schmelter. *Model-based Requirement Pattern Catalog*. Tech. rep. tr-ri-17-354. Paderborn, Germany: Software Engineering Department, Fraunhofer IEM, Oct. 2018.

[*HFK$^+$16a] J. Holtmann, M. Fockel, T. Koch and D. Schmelter. “Requirements Engineering - Zusatzaufgabe oder Kernkompetenz?” In: *OBJEKTspektrum* RE/2016 (2016).

[*HFK$^+$16b] J. Holtmann, M. Fockel, T. Koch, D. Schmelter, C. Brenner, R. Bernijazov and M. Sander. *The MechatronicUML Requirements Engineering Method: Process and Language*. Tech. rep. tr-ri-16-351. Software Engineering Department, Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute, 2016.

[*KDH+20] T. Koch, S. Dziwok, J. Holtmann and E. Bodden. "Scenario-Based Specification of Security Protocols and Transformation to Security Model Checkers". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, 2020, pp. 343–353. isbn: 9781450370196. doi: 10.1145/3365438.3410946. url: https://doi.org/10.1145/3365438.3410946.

[*KHD14] T. Koch, J. Holtmann and J. DeAntoni. "Generating EAST-ADL Event Chains from Scenario-Based Requirements Specifications". In: *Proceedings of the 8th European Conference on Software Architecture (ECSA 2014)*. Ed. by P. Avgeriou and U. Zdun. Vol. 8627. Lecture Notes in Computer Science (LNCS). Springer, Aug. 2014, pp. 146–153.

[*KHL17] T. Koch, J. Holtmann and T. Lindemann. "Flexible Specification of STEP Application Protocol Extensions and Automatic Derivation of Tool Capabilities". In: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*. Feb. 2017.

[*KHL18] T. Koch, J. Holtmann and T. Lindemann. "Model-Driven STEP Application Protocol Extensions Combined with Feature Modeling Considering Geometrical Information". In: *Model-Driven Engineering and Software Development*. Ed. by S. Pires Luís Ferreiraand Hammoudi and B. Selic. Springer International Publishing, 2018, pp. 173–197. isbn: 978-3-319-94764-8.

[*KHS+16] T. Koch, J. Holtmann, D. Schubert and T. Lindemann. "Towards Feature-based Product Line Engineering of Technical Systems". In: *3rd International Conference on System-Integrated Intelligence: New Challenges for Product and Production Engineering*. Ed. by B. Denkena, K.-D. Thoben and A. Trächtler. Elsevier, Aug. 2016, p. 00.

[*KMM+20] T. Koch, M. Meyer, F.-B. Masud and H. Runschke. "Softwareentwicklung wie am Fließband". In: *Proceedings of the Software Engineering 2020*. Lecture Notes in Informatics (LNI). Gesellschaft fuer Informatik, Feb. 2020, pp. 209–214.

[*Koc18] T. Koch. "Towards Scenario-based Security Requirements Engineering for Cyber-Physical Systems". In: *International Workshop on Security for and by Model-Driven Engineering (SecureMDE 2018)*. June 2018.

[*KTD+22] T. Koch, S. Trippel, S. Dziwok and E. Bodden. "Integrating Security Protocols in Scenario-based Requirements Specifications". In: *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development*. Jan. 2022.

[*MHK+15] J. Meyer, J. Holtmann, T. Koch and M. Meyer. "Generierung von AUTOSAR-Modellen aus UML-Spezifikationen". In: *10. Paderborner Workshop Entwurf mechatronischer Systeme*. Ed. by J. Gausemeier, R. Dumitrescu, F.-J. Rammig, W. Schäfer and A. Trächtler. Vol. 343. Verlagsschriftenreihe des Heinz Nixdorf Instituts, Paderborn. Heinz Nixdorf Institut, Apr. 2015, pp. 159–172.

Supervised Thesis

[+Gop21] S. GOPALAKRISHNAN. “Security Protocol Verification by means of Tamarin”. Masters’s Thesis. Universität Paderborn, June 2021.

[+Hal16] J. HALTERMANN. “Steuerbarkeit und Verfolgbarkeit von Änderungen an Produkten im Umfeld von ALM- und PLM-Systemen”. Bachelor’s Thesis. Universität Paderborn, Aug. 2016.

[+Hei15] M. HEINZMANN. “Integriertes Anforderungsmanagement im Product-Lifecycle-Management”. Bachelor’s Thesis. Universität Paderborn, June 2015.

[+Hoc20] R. HOCHHALTER. “Szenariobasierte Modellierung von Angriffsmodellen auf Basis von Modal Sequence Diagrammen”. Bachelor’s Thesis. Universität Paderborn, June 2020.

[+Jaz15] B. M. JAZAYERI. “Early Prediction of Security Properties for Mechatronic Systems”. Master’s Thesis. Universität Paderborn, Jan. 2015.

[+Kai20] A. KAISER. “Spezifikation von Anforderungen an sichere Kommunikationsprotokolle”. Bachelor’s Thesis. Universität Paderborn, Sept. 2020.

[+Tri21] S. TRIPPEL. “Incorporating Security Protocols in Scenario-based Functional Requirements Specifications”. Bachelor’s Thesis. Universität Paderborn, June 2021.

Foreign Publications

[ABD+15] D. ADRIAN, K. BHARGAVAN, Z. DURUMERIC, P. GAUDRY, M. GREEN, J. A. HALDERMAN, N. HENINGER, D. SPRINGALL, E. THOMÉ, L. VALENTA, B. VANDERSLOOT, E. WUSTROW, S. ZANELLA-BÉGUELIN AND P. ZIMMERMANN. “Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS ’15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 5–17. ISBN: 9781450338325. DOI: 10.1145/2810103.2813707. URL: https://doi.org/10.1145/2810103.2813707.

[ABF16] M. ABADI, B. BLANCHET AND C. FOURNET. *The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication*. 2016. URL: http://arxiv.org/pdf/1609.03003v2.

[AF01] M. ABADI AND C. FOURNET. “Mobile values, new names, and secure communication”. In: *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’01*. Ed. by C. HANKIN AND D. SCHMIDT. New York, New York, USA: ACM Press, 2001, pp. 104–115. ISBN: 1581133367. DOI: 10.1145/360204.360213.

[AGI+13] S. ABRAHAO, C. GRAVINO, E. INSFRAN, G. SCANNIELLO AND G. TORTORA. “Assessing the Effectiveness of Sequence Diagrams in the Comprehension of Functional Requirements: Results from a Family of Five Experiments”. In: *IEEE Transactions on Software Engineering* 39.3 (2013), pp. 327–342. ISSN: 0098-5589. DOI: 10.1109/TSE.2012.27.

[ALA19] R. Ameur-Boulifa, F. Lugou and L. Apvrille. "SysML Model Transformation for Safety and Security Analysis". In: Lecture Notes in Computer Science (2019). Ed. by B. Hamid, B. Gallina, A. Shabtai, Y. Elovici and J. Garcia-Alfaro, pp. 35–49. doi: 10.1007/978-3-030-16874-2_3.

[AN96] M. Abadi and R. Needham. "Prudent engineering practice for cryptographic protocols". In: *IEEE Transactions on Software Engineering* 22.1 (1996), pp. 6–15. issn: 0098-5589. doi: 10.1109/32.481513.

[Aus96] D. M. Auslander. "What is Mechatronics?" In: *IEEE/ASME Transactions on Mechatronics* 1.1 (1996), pp. 5–9. issn: 1083-4435. doi: 10.1109/3516.491404.

[BAN90] M. Burrows, M. Abadi and R. Needham. "A logic of authentication". In: *ACM Transactions on Computer Systems (TOCS)* 8.1 (1990), pp. 18–36. issn: 0734-2071. doi: 10.1145/77648.77649.

[BBB+19] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao and B. Parno. *SoK: Computer-Aided Cryptography*. Cryptology ePrint Archive, Paper 2019/1393. 2019. url: https://eprint.iacr.org/2019/1393.

[BBD+17] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub and J. K. Zinzindohoue. "A messy state of the union: taming the composite state machines of TLS". In: *Commun. ACM* 60.2 (2017), pp. 99–107. issn: 0001-0782. doi: 10.1145/3023357. url: https://doi.org/10.1145/3023357.

[BBK17] K. Bhargavan, B. Blanchet and N. Kobeissi. "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate". In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 483–502. doi: 10.1109/SP.2017.26.

[BCD+17] D. Basin, C. Cremers, J. Dreier and R. Sasse. "Symbolically Analyzing Security Protocols Using Tamarin". In: *ACM SIGLOG News* 4.4 (2017), pp. 19–30. doi: 10.1145/3157831.3157835. url: https://doi.org/10.1145/3157831.3157835.

[BCM+16] M. Bugliesi, S. Calzavara, S. Mödersheim and P. Modesti. "Security Protocol Specification and Verification with AnBx". In: *Journal of Information Security and Applications* (2016).

[BGH+11] G. Barthe, B. Grégoire, S. Heraud and S. Z. Béguelin. "Computer-Aided Security Proofs for the Working Cryptographer". In: *Advances in Cryptology – CRYPTO 2011*. Ed. by P. Rogaway. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 71–90. isbn: 978-3-642-22792-9.

[Bla01] B. Blanchet. "An efficient cryptographic protocol verifier based on prolog rules". In: *Computer Security Foundations Workshop, 2001. Proceedings. 14th IEEE*. 2001, pp. 82–96. isbn: 0-7695-1147-3. doi: 10.1109/CSFW.2001.930138.

[Bla12] B. Blanchet. “Security Protocol Verification: Symbolic and Computational Models”. In: *Principles of Security and Trust*. Ed. by P. Degano and J. D. Guttman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–29. isbn: 978-3-642-28641-4.

[Bla16] B. Blanchet. “Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif”. In: *Foundations and Trends® in Privacy and Security* 1.1-2 (2016), pp. 1–135. issn: 2474-1558. doi: 10.1561/3300000004.

[Bla23] B. Blanchet. *CryptoVerif: a Computationally-Sound Security Protocol Verifier (Initial Version with Communications on Channels)*. 2023. arXiv: 2310.14658 [cs.CR]. url: https://arxiv.org/abs/2310.14658.

[BLF$^+$14] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti and P. Y. Strub. “Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS”. In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 98–113. doi: 10.1109/SP.2014.14.

[BM90] S. M. Bellovin and M. Merritt. “Limitations of the Kerberos authentication system”. In: *ACM SIGCOMM Computer Communication Review* 20.5 (1990), pp. 119–132. issn: 0146-4833. doi: 10.1145/381906.381946.

[BO97] J. Bull and D. J. Otway. *The authentication protocol*. Ed. by Defence Research Agency. Technical Report DRA/CIS3/PROJ/CORBA/SC/1/CSM/436-04/03. 1997.

[BST21] D. Basin, R. Sasse and J. Toro-Pozo. “Card Brand Mixup Attack: Bypassing the PIN in non-Visa Cards by Using Them for Visa Transactions”. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 179–194. isbn: 978-1-939133-24-3. url: https://www.usenix.org/conference/usenixsecurity21/presentation/basin.

[Bun21] Bundesamt für Sicherheit in der Informationstechnik. “Die Lage der IT-Sicherheit in Deutschland 2020”. In: (2021).

[CDL06] V. Cortier, S. Delaune and P. Lafourcade. “A Survey of Algebraic Properties Used in Cryptographic Protocols”. In: (2006). doi: 10.5555/1239376.1239377.

[CH98] D. Carrel and D. Harkins. *The Internet Key Exchange (IKE)*. RFC 2409. Nov. 1998. doi: 10.17487/RFC2409. url: https://www.rfc-editor.org/info/rfc2409.

[CJ02] J. Clark and J. Jacob. *Security Protocols Open Repository*. 2002. url: http://www.lsv.fr/Software/spore/index.html.

[CJ95] J. Clark and J. Jacob. “On the security of recent protocols”. In: *Information Processing Letters* 56.3 (1995), pp. 151–155. issn: 0020-0190. doi: https://doi.org/10.1016/0020-0190(95)00136-Z. url: https://www.sciencedirect.com/science/article/pii/002001909500136Z.

[CJ97] J. A. Clark and J. L. Jacob. *A survey of authentication protocol literature: Version 1.0*. Report. 1997. url: https://eprints.whiterose.ac.uk/72494/.

[CKM20] C. Cremers, B. Kiesl and N. Medinger. "A Formal Analysis of IEEE 802.11's WPA2: Countering the Kracks Caused by Cracking the Counters". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1–17. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/cremers.

[Com19] Competence Center Industrial Security. "Industrial Security im Maschinen- und Anlagenbau: Ergebnisse der VDMA-Studie und Handlungsempfehlungen". In: (2019).

[CVB06] C. Caleiro, L. Viganò and D. Basin. "On the semantics of Alice & Bob specifications of security protocols". In: *Theoretical Computer Science* 367.1 (2006), pp. 88–122. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2006.08.041. URL: https://www.sciencedirect.com/science/article/pii/S0304397506005755.

[DH76] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638.

[DHR+18] J. Dreier, L. Hirschi, S. Radomirovic and R. Sasse. "Automated Unbounded Verification of Stateful Cryptographic Protocols with Exclusive OR". In: *IEEE 31th Computer Security Foundations Symposium*. Piscataway, NJ: IEEE, 2018, pp. 359–373. ISBN: 978-1-5386-6680-7. DOI: 10.1109/CSF.2018.00033.

[DS81] D. E. Denning and G. M. Sacco. "Timestamps in key distribution protocols". In: *Communications of the ACM* 24.8 (1981), pp. 533–536. ISSN: 00010782. DOI: 10.1145/358722.358740.

[DT19] L. Di Li and A. Tiu. "Combining ProVerif and Automated Theorem Provers for Security Protocol Verification". In: *Automated Deduction – CADE 27*. Ed. by P. Fontaine. Lecture Notes in Artificial Intelligence. Cham: Springer International Publishing, 2019, pp. 354–365. ISBN: 978-3-030-29436-6.

[DY83] D. Dolev and A. Yao. "On the security of public key protocols". In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. ISSN: 0018-9448. DOI: 10.1109/TIT.1983.1056650.

[FC03] D. G. Firesmith and F. Consulting. "Engineering Security Requirements". In: *Journal of Object Technology* 2 (2003), pp. 53–68.

[Fir03] D. Firesmith. "Security Use Cases". In: *The Journal of Object Technology* 2.3 (2003), p. 53. DOI: 10.5381/jot.2003.2.3.c6. URL: http://www.jot.fm/issues/issue_2003_05/column6.pdf.

[Fir07] D. G. Firesmith. "Engineering Safety and Security Related Requirements for Software Intensive Systems". In: *29th International Conference on Software Engineering*. Los Alamitos, Calif. [u.a.]: IEEE Computer Society, 2007, p. 169. ISBN: 0-7695-2892-9. DOI: 10.1109/ICSECOMPANION.2007.35.

[FLH+16] K. Fang, X. Li, J. Hao and Z. Feng. "Formal Modeling and Verification of Security Protocols on Cloud Computing Systems Based on UML 2.3". In: *IEEE TrustCom/BigDataSE/ISPA 2016*. Piscataway, NJ: IEEE, 2016, pp. 852–859. ISBN: 978-1-5090-3205-1. DOI: 10.1109/TrustCom.2016.0148.

[GJM99] J. A. GARAY, M. JAKOBSSON AND P. MACKENZIE. "Abuse-Free Optimistic Contract Signing". In: *Advances in Cryptology — CRYPTO' 99*. Ed. by G. GOOS, J. HARTMANIS, J. VAN LEEUWEN AND M. WIENER. Vol. 1666. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 449–466. ISBN: 978-3-540-66347-8. DOI: 10.1007/3-540-48405-1_29.

[GM17] R. GARCIA AND P. MODESTI. "An IDE for the Design, Verification and Implementation of Security Protocols". In: *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2017, pp. 157–163. DOI: 10.1109/ISSREW.2017.69.

[GMZ06] P. GIORGINI, H. MOURATIDIS AND N. ZANNONE. "Modelling security and trust with secure tropos". In: *Integrating Security and Software Engineering: Advances and Future Vision* (2006), pp. 160–189.

[Gon89] L. GONG. "Using one-way functions for authentication". In: *ACM SIGCOMM Computer Communication Review* 19.5 (1989), pp. 8–11. ISSN: 0146-4833. DOI: 10.1145/74681.74682.

[Gre11] J. GREENYER. "Scenario-based Design of Mechatronic Systems". PhD thesis. University of Paderborn, 2011. URL: http://dups.ub.uni-paderborn.de/hs/urn/urn:nbn:de:hbz:466:2-7690.

[GV03] T. GENET AND V. VIET TRIEM TONG. "Verification of Copy-Protection Cryptographic Protocol using Approximations of Term Rewriting Systems". In: (Apr. 2003).

[Har00] D. HAREL. "From Play-In Scenarios to Code: An Achievable Dream". In: *Fundamental Approaches to Software Engineering*. Ed. by G. GOOS, J. HARTMANIS, J. VAN LEEUWEN AND T. MAIBAUM. Vol. 1783. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 22–34. ISBN: 978-3-540-67261-6. DOI: 10.1007/3-540-46428-X_3.

[Har01] D. HAREL. "From play-in scenarios to code: an achievable dream". In: *Computer* 34.5 (2001), pp. 53–60. ISSN: 00189162. DOI: 10.1109/2.895118.

[HC95] T. HWANG AND Y.-H. CHEN. "On the security of SPLICE/AS — The authentication system in WIDE Internet". In: *Information Processing Letters* 53.2 (1995), pp. 97–101. ISSN: 00200190. DOI: 10.1016/0020-0190(94)00175-X.

[HLL$^+$95] T. HWANG, N.-Y. LEE, C.-M. LI, M.-Y. KO AND Y.-H. CHEN. "Two Attacks on Neuman-Stubblebine Authentication Protocols". In: *Information Processing Letters* 53.2 (1995), pp. 103–107. ISSN: 00200190.

[HLM$^+$08] C. B. HALEY, R. LANEY, J. D. MOFFETT AND B. NUSEIBEH. "Security Requirements Engineering: A Framework for Representation and Analysis". In: *IEEE Transactions on Software Engineering* 34.1 (2008), pp. 133–153. ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70754.

[HM03] D. HAREL AND R. MARELLY. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Secaucus, NJ, USA: Springer-Verlag New York, Inc, 2003. ISBN: 3540007873.

[HM08] D. Harel and S. Maoz. “Assert and negate revisited: Modal semantics for UML sequence diagrams”. In: *Software & Systems Modeling* 7.2 (2008), pp. 237–252. issn: 1619-1366. doi: 10.1007/s10270-007-0054-z. url: http://dx.doi.org/10.1007/s10270-007-0054-z.

[Hol19] J. Holtmann. “Improvement of software requirements quality based on systems engineering”. In: (2019). doi: 10.17619/UNIPB/1-730.

[HRD10] J. Hassine, J. Rilling and R. Dssouli. “An Evaluation of Timed Scenario Notations”. In: *J. Syst. Softw.* 83.2 (2010), pp. 326–350. issn: 0164-1212. doi: 10.1016/j.jss.2009.09.014. url: http://dx.doi.org/10.1016/j.jss.2009.09.014.

[HS07] T. A. Henzinger and J. Sifakis. “The Discipline of Embedded Systems Design”. In: *Computer* 40.10 (2007), pp. 32–40. issn: 00189162. doi: 10.1109/MC.2007.364.

[IEE99] IEEE. “IEEE Standard for Information Technology- Telecommunications and Information Exchange Between Systems-Local and Metropolitan Area Networks-Specific Requirements-Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications”. In: *IEEE Std 802.11-1999* (1999).

[IM90] C. I’Anson and C. Mitchell. “Security defects in CCITT recommendation X.509”. In: *ACM SIGCOMM Computer Communication Review* 20.2 (1990), pp. 30–34. issn: 0146-4833. doi: 10.1145/378570.378623.

[Jür02] J. Jürjens. “UMLsec: Extending UML for Secure Systems Development”. In: *The unified modeling language: Model engineering, concepts, and tools ; proceedings*. Ed. by J.-M. Jézéquel. Vol. 2460. Lecture Notes in Computer Science. Berlin [u.a.]: Springer, 2002, pp. 412–425. isbn: 978-3-540-44254-7. doi: 10.1007/3-540-45800-X_32.

[Jür05] J. Jürjens. *Secure Systems Development with UML*. Berlin/Heidelberg: Springer-Verlag, 2005. isbn: 3-540-00701-6. doi: 10.1007/b137706.

[JV96] M. Just and S. Vaudenay. “Authenticated multi-party key agreement”. In: *Advances in Cryptology — ASIACRYPT ’96*. Ed. by K. Kim and T. Matsumoto. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 36–49. isbn: 978-3-540-70707-3.

[KBB17] N. Kobeissi, K. Bhargavan and B. Blanchet. “Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach”. In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017, pp. 435–450. doi: 10.1109/EuroSP.2017.38.

[KC95] I.-L. Kao and R. Chow. “An efficient and secure authentication protocol using uncertified keys”. In: *ACM SIGOPS Operating Systems Review* 29.3 (1995), pp. 14–21. issn: 0163-5980. doi: 10.1145/206826.206832.

[Kel14] M. Keller. “Converting Alice&Bob Protocol Specifications to Tamarin”. Bachelor’s Thesis. ETH Zürich, Aug. 2014.

[KHN$^+$14] C. Kaufman, P. E. Hoffman, Y. Nir, P. Eronen and T. Kivinen. *Internet Key Exchange Protocol Version 2 (IKEv2)*. RFC 7296. Oct. 2014. doi: 10.17487/RFC7296. url: https://www.rfc-editor.org/info/rfc7296.

[KNT19] N. Kobeissi, G. Nicolas and M. Tiwari. *Verifpal: Cryptographic Protocol Analysis for the Real World*. 2019.

[KPP95] B. Kitchenham, L. Pickard and S. L. Pfleeger. “Case studies for method and tool evaluation”. In: *IEEE Software* 12.4 (1995), pp. 52–62. issn: 07407459. doi: 10.1109/52.391832.

[KSL92] A. Kehne, J. Schönwälder and H. Langendörfer. “A nonce-based protocol for multiple authentications”. In: *SIGOPS Oper. Syst. Rev.* 26.4 (1992), pp. 84–89. issn: 0163-5980. doi: 10.1145/142854.142872. url: https://doi.org/10.1145/142854.142872.

[LBD02] T. Lodderstedt, D. Basin and J. Doser. “SecureUML: A UML-Based Modeling Language for Model-Driven Security”. In: *UML 2002 The Unified Modeling Language*. Ed. by J.-M. Jézéquel, S. Cook and H. Hussmann. SpringerLink Bücher. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2002, pp. 426–441. isbn: 978-3-540-45800-5.

[LLA$^+$16] F. Lugou, L. W. Li, L. Apvrille and R. Ameur-Boulifa. “SysML Models and Model Transformation for Security”. In: (2016), pp. 331–338. doi: 10.5220/0005748703310338.

[Low00] G. Lowe. “A Family of Attacks upon Authentication Protocols”. In: 5 (2000).

[Low95] G. Lowe. “An attack on the Needham-Schroeder public-key authentication protocol”. In: *Information Processing Letters* 56.3 (1995), pp. 131–133. issn: 00200190. doi: 10.1016/0020-0190(95)00144-2.

[Low96a] G. Lowe. “Some new attacks upon security protocols”. In: *Proceedings 9th IEEE Computer Security Foundations Workshop*. IEEE Comput. Soc. Press, 1996, pp. 162–169. isbn: 0-8186-7522-5. doi: 10.1109/CSFW.1996.503701.

[Low96b] G. Lowe. “Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by G. Goos, J. Hartmanis, J. Leeuwen, T. Margaria and B. Steffen. Vol. 1055. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 147–166. isbn: 978-3-540-61042-7. doi: 10.1007/3-540-61042-1_43.

[Low97] G. Lowe. “A hierarchy of authentication specifications”. In: *Computer Security Foundations Workshop X*. Los Alamitos: IEEE Computer Society Press, 1997, pp. 31–43. isbn: 0-8186-7990-5. doi: 10.1109/CSFW.1997.596782.

[Low98] G. Lowe. “Towards a completeness result for model checking of security protocols”. In: *Proceedings. 11th IEEE Computer Security Foundations Workshop (Cat. No.98TB100238)*. IEEE Comput. Soc, 1998, pp. 96–105. isbn: 0-8186-8488-7. doi: 10.1109/CSFW.1998.683159.

[LT15] G. Liebel and M. Tichy. “Comparing Comprehensibility of Modelling Languages for Specifying Behavioural Requirements”. In: *HuFaMo@MoDELS*. 2015.

[LYM03] L. LIU, E. YU AND J. MYLOPOULOS. "Security and privacy requirements analysis within a social setting". In: *Journal of Lightwave Technology*. IEEE Comput. Soc, 2003, pp. 151–161. ISBN: 0-7695-1980-6. DOI: 10.1109/ICRE.2003.1232746.

[LZK20] T. LAUSER, D. ZELLE AND C. KRAUSS. "Security Analysis of Automotive Protocols". In: *Proceedings of the 4th ACM Computer Science in Cars Symposium*. CSCS '20. Feldkirchen, Germany: Association for Computing Machinery, 2020. ISBN: 9781450376211. DOI: 10.1145/3385958.3430482. URL: https://doi.org/10.1145/3385958.3430482.

[MA22] R. METERE AND L. ARNABOLDI. "Automating cryptographic protocol language generation from structured specifications". In: *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*. Ed. by S. GNESI, N. PLAT, A. HARTMANNS AND I. SCHAEFER. New York, NY, USA: ACM, 2022, pp. 91–101. ISBN: 9781450392877. DOI: 10.1145/3524482.3527654.

[Mei13] S. MEIER. "Advancing automated security protocol verification". PhD thesis. ETH Zurich, 2013. DOI: 10.3929/ETHZ-A-009790675.

[MG07] H. MOURATIDIS AND P. GIORGINI. "SECURE TROPOS: A SECURITY-ORIENTED EXTENSION OF THE TROPOS METHODOLOGY". In: *International Journal of Software Engineering and Knowledge Engineering* 17.02 (2007), pp. 285–309. ISSN: 0218-1940. DOI: 10.1142/S0218194007003240.

[MJP$^+$22] M. MÉRÉ, F. JOUAULT, L. PALLARDY AND R. PERDRIAU. "Feedback on the formal verification of UML models in an industrial context". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. Ed. by E. SYRIANI AND H. SAHRAOUI. New York, NY, USA: ACM, 2022, pp. 121–131. ISBN: 9781450394666. DOI: 10.1145/3550355.3552454.

[Möd09] S. MÖDERSHEIM. "Algebraic Properties in Alice and Bob Notation". In: *2009 International Conference on Availability, Reliability and Security*. 2009, pp. 433–440. DOI: 10.1109/ARES.2009.95.

[MS05] N. R. MEAD AND T. STEHNEY. "Security quality requirements engineering (SQUARE) methodology". In: *ACM SIGSOFT Software Engineering Notes* 30.4 (2005), pp. 1–7. ISSN: 0163-5948. DOI: 10.1145/1082983.1083214.

[MS13] C. MEYER AND J. SCHWENK. *Lessons Learned From Previous SSL/TLS Attacks - A Brief Chronology Of Attacks And Weaknesses*. 2013.

[MSC$^+$13] S. MEIER, B. SCHMIDT, C. CREMERS AND D. BASIN. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols". In: *Computer Aided Verification*. Ed. by D. HUTCHISON, T. KANADE, J. KITTLER, J. M. KLEINBERG, F. MATTERN, J. C. MITCHELL, M. NAOR, O. NIERSTRASZ, C. PANDU RANGAN, B. STEFFEN, M. SUDAN, D. TERZOPOULOS, D. TYGAR, M. Y. VARDI, G. WEIKUM, N. SHARYGINA AND H. VEITH. Vol. 8044. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 696–701. ISBN: 978-3-642-39798-1. DOI: 10.1007/978-3-642-39799-8_48.

[MSG+09] N. Moebius, K. Stenzel, H. Grandy and W. Reif. "SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications". In: *International Conference on Availability, Reliability and Security, 2009*. Piscataway, NJ: IEEE, 2009, pp. 841–846. isbn: 978-1-4244-3572-2. doi: 10.1109/ARES.2009.22.

[MTL+09] D. Mouheb, C. Talhi, V. Lima, M. Debbabi, L. Wang and M. Pourzandi. "Weaving security aspects into UML 2.0 design models". In: *Proceedings of the 13th workshop on Aspect-oriented modeling - AOM '09*. Ed. by O. Aldawud, W. Cazzola, T. Cottenier, J. Gray, J. Kienzle and D. Stein. New York, New York, USA: ACM Press, 2009, p. 7. isbn: 9781605584515.

[MV15] C. Miller and C. Valasek. "Remote Exploitation of an Unaltered Passenger Vehicle". In: *Black Hat USA* (2015).

[NGM19] K. Nadim, N. Georgio and T. Mukesh. *Verifpal: Cryptographic Protocol Analysis for the Real World*. 2019. url: https://eprint.iacr.org/2019/971.

[NO21] M. Nakabayashi and Y. Okano. "Verification Method of Key-Exchange Protocols With a Small Amount of Input Using Tamarin Prover". In: *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems*. ASSS '21. Virtual Event, Hong Kong: Association for Computing Machinery, 2021, pp. 43–50. isbn: 9781450384032. doi: 10.1145/3457340.3458301. url: https://doi.org/10.1145/3457340.3458301.

[NS78] R. M. Needham and M. D. Schroeder. "Using encryption for authentication in large networks of computers". In: *Communications of the ACM* 21.12 (1978), pp. 993–999. issn: 00010782. doi: 10.1145/359657.359659.

[NS87] R. M. Needham and M. D. Schroeder. "Authentication revisited". In: *SIGOPS Oper. Syst. Rev.* 21.1 (1987), p. 7. issn: 0163-5980. doi: 10.1145/24592.24593. url: https://doi.org/10.1145/24592.24593.

[NS93] B. C. Neuman and S. G. Stubblebine. "A note on the use of timestamps as nonces". In: *ACM SIGOPS Operating Systems Review* 27.2 (1993), pp. 10–14. issn: 0163-5980. doi: 10.1145/155848.155852.

[NT09] J. Nicolás and A. Toval. "On the generation of requirements specifications from software engineering models: A systematic literature review". In: *Information and Software Technology* 51.9 (2009), pp. 1291–1307. issn: 09505849. doi: 10.1016/j.infsof.2009.04.001.

[NT94] B. C. Neuman and T. Ts'o. "Kerberos: an authentication service for computer networks". In: *IEEE Communications Magazine* 32.9 (1994), pp. 33–38. issn: 0163-6804. doi: 10.1109/35.312841.

[Obj17a] Object Management Group. *OMG Systems Modeling Language (OMG SysML)*. 2017. url: http://www.omg.org/spec/SysML/1.5/.

[Obj17b] Object Management Group. *OMG Unified Modeling Language (OMG UML) – Version 2.5.1*. 2017. url: http://www.omg.org/spec/SysML/1.4/.

[OR01] G. O'Shea and M. Roe. "Child-proof authentication for MIPv6 (CAM)". In: *ACM SIGCOMM Computer Communication Review* 31.2 (2001), pp. 4–8. ISSN: 0146-4833. DOI: 10.1145/505666.505668.

[OR87] D. Otway and O. Rees. "Efficient and timely mutual authentication". In: *ACM SIGOPS Operating Systems Review* 21.1 (1987), pp. 8–10. ISSN: 0163-5980. DOI: 10.1145/24592.24594.

[Orm98] H. Orman. *The OAKLEY Key Determination Protocol*. RFC 2412. Nov. 1998. DOI: 10.17487/RFC2412. URL: https://www.rfc-editor.org/info/rfc2412.

[Pau00] L. C. Paulson. "Relations Between Secrets: The Yahalom Protocol". In: *Security Protocols*. Ed. by B. Christianson, B. Crispo, J. A. Malcolm and M. Roe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 73–77. ISBN: 978-3-540-45570-7.

[Poh10] K. Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3642125778.

[Poo10] R. Poovendran. "Cyber-Physical Systems: Close Encounters Between Two Parallel Worlds". In: *Proceedings of the IEEE* 98.8 (2010), pp. 1363–1366. ISSN: 0018-9219. DOI: 10.1109/JPROC.2010.2050377.

[PX05] J. J. Pauli and D. Xu. "Misuse case-based design and analysis of secure software architecture". In: *International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II*. IEEE, 2005, 398–403 Vol. 2. ISBN: 0-7695-2315-3. DOI: 10.1109/ITCC.2005.199.

[RA15] Y. Roudier and L. Apvrille. "SysML-Sec: A model driven approach for designing safe and secure systems". In: *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2015, pp. 655–664.

[RBM+22] M. Raimondo, S. Bernardi, S. Marrone and J. Merseguer. "An approach for the automatic verification of blockchain protocols: the Tweetchain case study". In: *Journal of Computer Virology and Hacking Techniques* (2022). DOI: 10.1007/s11416-022-00444-z.

[Res18] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[RH09] P. Runeson and M. Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2 (2009), pp. 131–164. ISSN: 1382-3256. DOI: 10.1007/s10664-008-9102-8.

[RLF+04] I. Ray, N. Li, R. France and D.-K. Kim. "Using uml to visualize role-based access control constraints". In: *Proceedings of the ninth ACM symposium on Access control models and technologies - SACMAT '04*. Ed. by T. Jaeger and E. Ferrari. New York, New York, USA: ACM Press, 2004, p. 115. ISBN: 1581138725.

[RS98] P. RYAN AND S. A. SCHNEIDER. "An attack on a recursive authentication protocol A cautionary tale". In: *Information Processing Letters* 65.1 (1998), pp. 7–10. ISSN: 00200190. DOI: 10.1016/S0020-0190(97)00180-4.

[Run12] P. RUNESON, ed. *Case study research in software engineering: Guidelines and examples*. 1st ed. Hoboken, N.J: Wiley, 2012. ISBN: 9781118104354. DOI: 10.1002/9781118181034.

[Sat89] M. SATYANARAYANAN. "Integrating security in a large distributed system". In: *ACM Transactions on Computer Systems (TOCS)* 7.3 (1989), pp. 247–280. ISSN: 0734-2071. DOI: 10.1145/65000.65002.

[Sch90] C. P. SCHNORR. "Efficient Identification and Signatures for Smart Cards". In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by G. BRASSARD. New York, NY: Springer New York, 1990, pp. 239–252. ISBN: 978-0-387-34805-6.

[Sch96] B. SCHNEIER. *Applied cryptography: Protocols, algorithms, and source code in C*. 2nd ed. New York: J. Wiley & Sons, 1996. ISBN: 9780471117094. URL: http://proquest.tech.safaribooksonline.de/9780471117094.

[SLF^{+}14] G. SHEN, X. LI, R. FENG, G. XU, J. HU AND Z. FENG. "An Extended UML Method for the Verification of Security Protocols". In: *2014 19th International Conference on Engineering of Complex Computer Systems*. IEEE, 2014, pp. 19–28. ISBN: 978-1-4799-5482-7. DOI: 10.1109/ICECCS.2014.12.

[SO05] G. SINDRE AND A. L. OPDAHL. "Eliciting security requirements with misuse cases". In: *Requirements Engineering* 10.1 (2005), pp. 34–44. ISSN: 0947-3602. DOI: 10.1007/s00766-004-0194-4.

[SR96] V. SHOUP AND A. RUBIN. "Session Key Distribution Using Smart Cards". In: *Advances in Cryptology — EUROCRYPT '96*. Ed. by G. GOOS, J. HARTMANIS, J. VAN LEEUWEN AND U. MAURER. Vol. 1070. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 321–331. ISBN: 978-3-540-61186-8. DOI: 10.1007/3-540-68339-9_28.

[SS11] SHEILA FRANKEL AND SURESH KRISHNAN. *IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap*. 2011. DOI: 10.17487/RFC6071. URL: https://www.rfc-editor.org/info/rfc6071.

[STP12] E. SIKORA, B. TENBERGEN AND K. POHL. "Industry needs and research directions in requirements engineering for embedded systems". In: *Requirements Engineering* 17.1 (2012), pp. 57–78. ISSN: 0947-3602. DOI: 10.1007/s00766-011-0144-x.

[SW07] W. SCHÄFER AND H. WEHRHEIM. "The Challenges of Building Advanced Mechatronic Systems". In: *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*. Ed. by LIONEL C. BRIAND AND ALEXANDER L. WOLF. 2007, pp. 72–84.

[Tel96] TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU. *ITU-T recommendation Z.120 (10/96): Message Sequence Chart (MSC)*. 1996.

[TMN90] M. TATEBAYASHI, N. MATSUZAKI AND D. B. NEWMAN. "Key Distribution Protocol for Digital Mobile Communication Systems". In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by G. GOOS, J. HARTMANIS, D. BARSTOW, W. BRAUER, P. BRINCH HANSEN, D. GRIES, D. LUCKHAM, C. MOLER, A. PNUELI, G. SEEGMÜLLER, J. STOER, N. WIRTH AND G. BRASSARD. Vol. 435. Lecture Notes in Computer Science. New York, NY: Springer New York, 1990, pp. 324–334. ISBN: 978-0-387-97317-3. DOI: 10.1007/0-387-34805-0{\textunderscore}30.

[VDI04] VDI. *Design methodology for mechatronic systems (VDI 2206)*. 2004.

[vR09] T. VAN DEURSEN AND S. RADOMIROVI. "Attacks on RFID Protocols". In: *Cryptology ePrint Archive* 2008.310 (2009), pp. 1–56.

[WL93] T. WOO AND S. S. LAM. "A semantic model for authentication protocols". In: *Proceedings*. Los Alamitos, Calif: IEEE Computer Society Press, 1993, pp. 178–194. ISBN: 0-8186-3370-0. DOI: 10.1109/RISP.1993.287633.

[WL94] T. Y. C. WOO AND S. S. LAM. "A lesson on authentication protocol design". In: *ACM SIGOPS Operating Systems Review* 28.3 (1994), pp. 24–37. ISSN: 0163-5980. DOI: 10.1145/182110.182113.

[Wor23] WORLD ECONOMIC FORUM. *Global risks 2021: Insight report*. 18th Edition. Geneva: World Economic Forum, 2023.

[WWH08] J. WHITTLE, D. WIJESEKERA AND M. HARTONG. "Executable misuse cases for modeling security concerns". In: *Proceedings of the 30th International Conference on Software Engineering: May 10-18, 2008, Leipzig, Germany*. Ed. by W. SCHÄFER. New York, N.Y.: ACM Press, 2008, p. 121. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368106.

[YOM90] S. YAMAGUCHI, K. OKAYAMA AND H. MIYAHARA. "Design and implementation of an authentication system in WIDE Internet environment". In: *IEEE TENCON'90: 1990 IEEE Region 10 Conference on Computer and Communication Systems. Conference Proceedings* (1990), 653–657 vol.2.

Tool Suites and Tool Frameworks

[EMF] ECLIPSE MODELING FRAMEWORK (EMF). URL: http://www.eclipse.org/modeling/emf.

[Obj14] OBJECT MANAGEMENT GROUP (OMG). *OMG Object Constraint Language (OCL) – Version 2.4*. OMG Document Number: formal/14-02-03. 2014.

[Papyrus] PAPYRUS MODELING ENVIRONMENT. URL: http://www.eclipse.org/papyrus.

[Proverif] PROVERIF: CRYPTOGRAPHIC PROTOCOL VERIFIER IN THE FORMAL MODEL. URL: https://bblanche.gitlabpages.inria.fr/proverif/.

[QVTo] ECLIPSE QVT OPERATIONAL. URL: http://projects.eclipse.org/projects/modeling.mmt.qvt-oml.

[ST-MSD] SCENARIOTOOLS MSD. URL: http://scenariotools.org/projects2/msd.

[Tamarin] TAMARIN PROVER. URL: https://tamarin-prover.github.io/.

[UML] OBJECT MANAGEMENT GROUP. *OMG Unified Modeling Language (OMG UML)*. 2015. URL: http://www.omg.org/spec/UML/2.5/.

[Xtend] ECLIPSE XTEND. URL: https://www.eclipse.org/xtend/.

[Xtext] ECLIPSE XTEXT. URL: https://www.eclipse.org/Xtext/.

List of Figures

List of Tables

List of Listings

List of Algorithms

A

Supplementing Materials for the Specification of Security MSDs

This appendix presents supplementing materials for the analysis of existing security protocols as presented in Chapter 3. We analyzed 54 security protocols to identify typical building blocks. In particular, we analyzed the usage of cryptographic primitives (cf. Section A.1), the usage of algebraic operations (cf. Section A.2), and the usage of data types (cf. Section A.3).

A.1 Analyzing the Usage of Cryptographic Primitives in Security Protocols

Table A.1: Usage of security primitives in the security protocols of our literature study

No.	Security Protocol	Asymmetric Encryption	Symmetric Encryption	Digital Signatures	MAC	Hashing
1	Andrew Secure RPC [Sat89; BAN90]	○	●	○	○	○
2	Andrew Secure RPC (BAN concrete) [BAN90; Low96a]	○	●	○	○	○
3	Andrew Secure RPC (BAN modified) [BAN90]	○	●	○	○	○
4	Andrew Secure RPC (Lowe modified) [Low96a]	○	●	○	○	○
5	Bull's Authentication Protocols [BO97; RS98]	○	●	○	●	○
6	CCITT X.509 (v1) [BAN90; AN96]	●	○	●	○	○
7	CCITT X.509 (v2) [BAN90; IM90]	●	○	●	○	●
8	CCITT X.509 (v3) [BAN90; AN96; IM90]	●	○	●	○	○
9	CCITT X.509 (BAN modified) [BAN90]	●	○	●	○	○
10	CH07 [vR09]	○	○	○	○	●
11	Child-proof Authentication for MIPv6 (CAM) [OR01]	○	○	●	○	●

No.	Security Protocol	Asymmetric Encryption	Symmetric Encryption	Digital Signatures	MAC	Hashing
12	Denning-Sacco-Shared-Key [DS81; Low00]	○	●	○	○	○
13	Denning-Sacco-Shared-Key (Lowe modified) [Low00]	○	●	○	○	○
14	Diffie-Helman [DH76]	○	○	○	○	○
15	GJM [GJM99]	○	○	●	○	○
16	Gong's Mutual Authentication Protocol [Gon89]	●	○	○	○	●
17	Internet Key Exchange (IKEv1) [CH98]	○	●	●	○	●
18	Internet Key Exchange (IKEv2)[KHN+14]	○	●	●	●	○
19	Kao Chow Authentication (v1) [KC95; CJ97]	○	●	○	○	○
20	Kao Chow Authentication (v2) [KC95]	○	●	○	○	○
21	Kao Chow Authentication (v3) [KC95]	○	●	○	○	○
22	Kerberos [BM90; NT94; Low96a]	○	●	○	○	○
23	Kerberos (Nonce-based Improvement of Kerberos (KSL)) [BM90; NT94; Low96a; KSL92]	○	●	○	○	○
24	Kerberos (Nonce-based Improvement of Kerberos (KSL, Lowe modified))[Low96a]	○	●	○	○	○
25	Needham-Schroeder Public Key [NS78]	●	○	●	○	○
26	Needham-Schroeder Public Key (Lowe modified) [Low95]	●	○	●	○	○
27	Needham-Schroeder Symmetric Key [NS78]	○	●	○	○	○
28	Needham-Schroeder Symmetric Key (Amended-version) [NS87]	○	●	○	○	○
29	Neumann Stubblebine [NS93]	○	●	○	○	○
30	Neumann Stubblebine (Hwang modified) [NS93; HLL+95]	○	●	○	○	○
31	Oakley Key Determination Protocol [Orm98]	●	●	●	○	●
32	Otway Rees [OR87]	○	●	○	○	○
33	Schnorr's Identification Protocol [Sch90]	○	○	○	○	○
34	Shamir-Rivest-Adleman Three Pass Protocol [CJ97]	○	●	○	○	○

No.	Security Protocol	Asymmetric Encryption	Symmetric Encryption	Digital Signatures	MAC	Hashing
35	SK3[SR96]	○	●	○	○	○
36	Smart-Right (view-only) [GV03]	○	●	○	○	●
37	SPLICE/AS [YOM90]	●	○	●	○	○
38	SPLICE/AS (Clark and Jacob modified) [CJ95]	●	○	●	○	○
39	SPLICE/AS (Hwang and Chen modified) [HC95]	●	○	●	○	○
40	TMN [TMN90]	●	●	○	○	○
41	Transport Layer Security (TLS 1.3) [Res18]	●	●	●	●	●
42	Wide Mouthed Frog [BAN90]	○	●	○	○	○
43	Wide Mouthed Frog (Lowe modified) [Low00]	○	●	○	○	○
44	Wired Equivalent Privacy (WEP) [IEE99]	○	○	○	○	○
45	Woo and Lam Mutual Authentication [WL94]	○	●	○	○	○
46	Woo and Lam Pi [WL94]	○	●	○	○	○
47	Woo and Lam Pi 1 [WL94]	○	●	○	○	○
48	Woo and Lam Pi 2 [WL94]	○	●	○	○	○
49	Woo and Lam Pi 3 [WL94]	○	●	○	○	○
50	Woo and Lam Pi f [WL94]	○	●	○	○	○
51	Yahalom [BAN90; CJ97]	○	●	○	○	○
52	Yahalom (modified version by Lowe) [Low98]	○	●	○	○	○
53	Yahalom (simplified version by BAN) [BAN90]	○	●	○	○	○
54	Yahalom (strengthened version by Paulson) [Pau00]	○	●	○	○	○

Legend: ● used in the security protocol, ○ not used in the security protocol

A.2 Analyzing the Usage of Algebraic Operations in Security Protocols

Table A.2: Usage of algebraic operations in the security protocols of our literature study

No.	Security Protocol	XOR	Addition	Subtraction	Multiplication	Division	Modular exponentiation
1	Andrew Secure RPC [Sat89; BAN90]	○	●	○	○	○	○
2	Andrew Secure RPC (BAN concrete) [BAN90; Low96a]	○	●	○	○	○	○
3	Andrew Secure RPC (BAN modified) [BAN90]	○	●	○	○	○	○
4	Andrew Secure RPC (Lowe modified) [Low96a]	○	●	○	○	○	○
5	Bull's Authentication Protocols [BO97; RS98]	●	○	○	○	○	○
6	CCITT X.509 (v1) [BAN90; AN96]	○	○	○	○	○	○
7	CCITT X.509 (v2) [BAN90; IM90]	○	○	○	○	○	○
8	CCITT X.509 (v3) [BAN90; AN96; IM90]	○	○	○	○	○	○
9	CCITT X.509 (BAN modified) [BAN90]	○	○	○	○	○	○
10	CH07 [vR09]	●	○	○	○	○	○
11	Child-proof Authentication for MIPv6 (CAM) [OR01]	○	○	○	○	○	○
12	Denning-Sacco-Shared-Key [DS81; Low00]	○	○	○	○	○	○
13	Denning-Sacco-Shared-Key (Lowe modified) [Low00]	○	○	●	○	○	○
14	Diffie-Helman [DH76]	○	○	○	○	○	●
15	GJM [GJM99]	○	○	○	○	○	○
16	Gong's Mutual Authentication Protocol [Gon89]	○	○	○	○	○	○
17	Internet Key Exchange (IKEv1) [CH98]	○	○	○	○	○	●
18	Internet Key Exchange (IKEv2) [KHN$^+$14]	○	○	○	○	○	●
19	Kao Chow Authentication (v1) [KC95]	○	○	○	○	○	○
20	Kao Chow Authentication (v2) [KC95]	○	○	○	○	○	○
21	Kao Chow Authentication (v3) [KC95]	○	○	○	○	○	○
22	Kerberos [BM90; NT94; Low96a]	○	○	○	○	○	○
23	Kerberos (Nonce-based Improvement of Kerberos (KSL)) [BM90; NT94; Low96a; KSL92]	○	○	○	○	○	○

No.	Security Protocol	XOR	Addition	Subtraction	Multiplication	Division	Modular exponentiation
24	Kerberos (Nonce-based Improvement of Kerberos (KSL, Lowe modified)) [Low96a]	○	○	○	○	○	○
25	Needham-Schroeder Public Key [NS78]	○	○	○	○	○	○
26	Needham-Schroeder Public Key (Lowe modified) [Low95]	○	○	○	○	○	○
27	Needham-Schroeder Symmetric Key [NS78]	○	○	●	○	○	○
28	Needham-Schroeder Symmetric Key (Amended-version) [NS87]	○	○	●	○	○	○
29	Neumann Stubblebine [NS93]	○	○	○	○	○	○
30	Neumann Stubblebine (Hwang modified) [NS93; HLL^{+}95]	○	○	○	○	○	○
31	Oakley Key Determination Protocol [Orm98]	○	○	○	○	○	●
32	Otway Rees [OR87]	○	○	○	○	○	○
33	Schnorr’s Identification Protocol [Sch90]	○	●	○	●	○	●
34	Shamir-Rivest-Adleman Three Pass Protocol [CJ97]	○	○	○	○	○	○
35	SK3 [SR96]	●	○	○	○	○	○
36	Smart-Right (view-only) [GV03]	○	○	○	○	○	○
37	SPLICE/AS [YOM90]	○	●	○	○	○	○
38	SPLICE/AS (Clark and Jacob modified) [CJ95]	○	●	○	○	○	○
39	SPLICE/AS (Hwang and Chen modified) [HC95]	○	●	○	○	○	○
40	TMN [TMN90]	○	○	○	○	○	○
41	Transport Layer Security (TLS 1.3) [Res18]	○	○	○	○	○	●
42	Wide Mouthed Frog [BAN90]	○	○	○	○	○	○
43	Wide Mouthed Frog (Lowe modified) [Low00]	○	○	○	○	○	○
44	Wired Equivalent Privacy (WEP) [IEE99]	○	○	○	○	○	○
45	Woo and Lam Mutual Authentication [WL94]	○	○	○	○	○	○
46	Woo and Lam Pi [WL94]	○	○	○	○	○	○
47	Woo and Lam Pi 1 [WL94]	○	○	○	○	○	○
48	Woo and Lam Pi 2 [WL94]	○	○	○	○	○	○

No.	Security Protocol	XOR	Addition	Subtraction	Multiplication	Division	Modular exponentiation
49	Woo and Lam Pi 3 [WL94]	○	○	○	○	○	○
50	Woo and Lam Pi f [WL94]	○	○	○	○	○	○
51	Yahalom [BAN90; CJ97]	○	○	○	○	○	○
52	Yahalom (modified version by Lowe) [Low98]	○	○	○	○	○	○
53	Yahalom (simplified version by BAN) [BAN90]	○	○	○	○	○	○
54	Yahalom (strengthened version by Paulson) [Pau00]	○	○	○	○	○	○

Legend: ● used in the security protocol, ○ not used in the security protocol

A.3 Analyzing the Usage of Data Types in Security Protocols

Table A.3: Usage of data types in the security protocols of our literature study

No.	Security Protocol	Nonce	Number	Timestamp	Identifier	Cryptographic Key
1	Andrew Secure RPC [Sat89; BAN90]	●	○	○	●	●
2	Andrew Secure RPC (BAN concrete) [BAN90; Low96a]	●	○	○	●	●
3	Andrew Secure RPC (BAN modified) [BAN90]	●	○	○	●	●
4	Andrew Secure RPC (Lowe modified) [Low96a]	●	○	○	●	●
5	Bull's Authentication Protocols [BO97; RS98]	○	●	○	●	●
6	CCITT X.509 (v1) [BAN90; AN96]	●	○	●	●	●
7	CCITT X.509 (v2) [BAN90; IM90]	●	○	●	●	●
8	CCITT X.509 (v3) [BAN90; IM90]	●	○	●	●	●
9	CCITT X.509 (BAN modified) [BAN90]	●	○	○	●	●
10	CH07 [vR09]	●	○	○	●	●
11	Child-proof Authentication for MIPv6 (CAM) [OR01]	●	○	●	●	●
12	Denning-Sacco-Shared-Key [DS81; Low00]	○	○	●	●	●
13	Denning-Sacco-Shared-Key (Lowe modified) [Low00]	●	○	●	●	●
14	Diffie-Helman [DH76]	○	●	○	●	○
15	GJM [GJM99]	○	○	○	●	●
16	Gong's Mutual Authentication Protocol [Gon89]	○	●	○	●	○
17	Internet Key Exchange (IKEv1) [CH98]	●	○	○	●	●
18	Internet Key Exchange (IKEv2) [KHN^{+}14]	●	○	○	●	●
19	Kao Chow Authentication (v1) [KC95]	●	○	○	●	●
20	Kao Chow Authentication (v2) [KC95]	●	○	○	●	●
21	Kao Chow Authentication (v3) [KC95]	●	○	○	●	●
22	Kerberos [BM90; NT94; Low96a]	●	○	●	●	●
23	Kerberos (Nonce-based Improvement of Kerberos (KSL)) [BM90; NT94; Low96a; KSL92]	●	○	●	●	●

No.	Security Protocol	Nonce	Number	Timestamp	Identifier	Cryptographic Key
24	Kerberos (Nonce-based Improvement of Kerberos (KSL, Lowe modified)) [Low96a]	●	○	●	●	●
25	Needham-Schroeder Public Key [NS78]	●	○	○	●	●
26	Needham-Schroeder Public Key (Lowe modified) [Low95]	●	○	○	●	●
27	Needham-Schroeder Symmetric Key [NS78]	●	○	○	●	●
28	Needham-Schroeder Symmetric Key (Amended-version) [NS87]	○	○	○	●	●
29	Neumann Stubblebine [NS93]	○	○	●	●	●
30	Neumann Stubblebine (Hwang modified) [NS93; HLL^{+}95]	○	○	●	●	●
31	Oakley Key Determination Protocol [Orm98]	●	○	○	●	●
32	Otway Rees [OR87]	●	○	○	●	●
33	Schnorr's Identification Protocol [Sch90]	○	●	○	●	●
34	Shamir-Rivest-Adleman Three Pass Protocol [CJ97]	○	●	○	●	●
35	SK3 [SR96]	○	○	○	●	●
36	Smart-Right (view-only)[GV03]	○	○	○	●	●
37	SPLICE/AS [YOM90]	●	○	●	●	●
38	SPLICE/AS (Clark and Jacob modified) [CJ95]	●	○	●	●	●
39	SPLICE/AS (Hwang and Chen modified) [HC95]	●	○	●	●	●
40	TMN [TMN90]	○	○	○	●	●
41	Transport Layer Security (TLS 1.3) [Res18]	○	●	○	●	●
42	Wide Mouthed Frog [BAN90]	○	○	●	●	●
43	Wide Mouthed Frog (Lowe modified) [Low00]	●	○	●	●	●
44	Wired Equivalent Privacy (WEP) [IEE99]	○	○	○	●	●
45	Woo and Lam Mutual Authentication [WL94]	●	○	○	●	●
46	Woo and Lam Pi [WL94]	●	○	○	●	●
47	Woo and Lam Pi 1 [WL94]	●	○	○	●	●
48	Woo and Lam Pi 2 [WL94]	●	○	○	●	●
49	Woo and Lam Pi 3 [WL94]	●	○	○	●	●

No.	Security Protocol	Nonce	Number	Timestamp	Identifier	Cryptographic Key
50	Woo and Lam Pi f [WL94]	●	○	○	●	●
51	Yahalom [BAN90; CJ97]	○	○	○	●	●
52	Yahalom (modified version by Lowe) [Low98]	○	○	○	●	●
53	Yahalom (simplified version by BAN) [BAN90]	○	○	○	●	●
54	Yahalom (strengthened version by Paulson) [Pau00]	○	○	○	●	●

Legend: ● used in the security protocol, ○ not used in the security protocol

B

Supplementing Materials for the Verification of Security MSDs

This appendix presents supplementing materials for the concepts presented in Chapter 4. Section B.1 describes three metamodels defining parts of the VerificationModel. Afterward, Section B.2 and Section B.3 present further transformation rules from the VerificationModel to PROVERIF and TAMARIN, respectively.

B.1 Supplementing Materials for the Translation from MSDs to the VerificationModel

As described in Section 4.3.1, the metamodel Verification is subdivided into several packages. This section presents an overview of the packages Verification::Protocol::Primitives and Verification::Protocol::Types.

B.1.1 Overview of the Package Verification::Protocol::Primitives

The class diagram of the package Verification::Protocol::Primitives is depicted in Figure B.1 and encompasses all classes necessary to specify security primitves. The package provides the abstract class SecurityPrimitve and five classes extending the abstract class.

Next, we provide an overview of the five classes extending the abstract class SecurityPrimitve:

- The class AsymmetricEncryption describes that a message or an argument is asymmetrically encrypted. The class encompasses the properties publicKey: EncPublicKey used for the encryption and privateKey: EncPrivateKey used for the decryption.
- The class DigitalSignature describes that a message or an argument is digitally signed. This signature is added to the message and sent to the receiver of the message. The class encompasses the properties privateKey: SigPrivateKey used for the creation of the digital signatur and publicKey: SigPublicKey used for the validation of the signature.
- The class SymmetricEncryption describes that a message or an argument is symmetrically encrypted. Therefore, the class encompasses the property symmetricKey: EncSymmetricKey specifying the key used for the encryption and decryption.

- The class HMAC describes that a hash-based message authentication code is added to a message and covers all or only part of the message's arguments. Therefore, the class encompasses the property symmetricKey: HMACSymmetricKey specifying the key used for creating and validating the hash-based message authentication code.

- The class Hashed describes that a message or an argument of a message is cryptographically hashed.

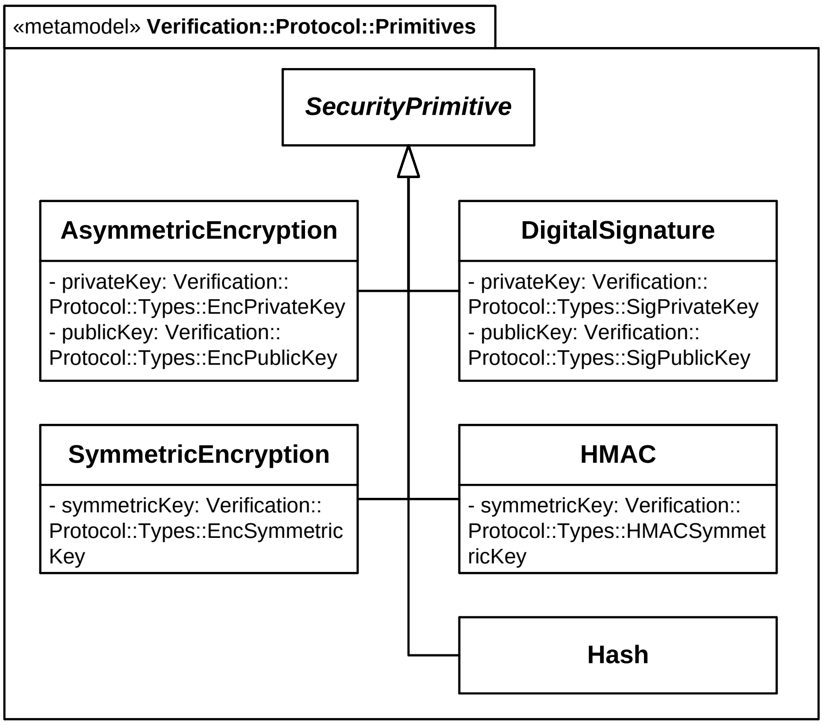

«metamodel» Verification::Protocol::Primitives
SecurityPrimitive
AsymmetricEncryption
- privateKey: Verification::Protocol::Types::EncPrivateKey
- publicKey: Verification::Protocol::Types::EncPublicKey
DigitalSignature
- privateKey: Verification::Protocol::Types::SigPrivateKey
- publicKey: Verification::Protocol::Types::SigPublicKey
SymmetricEncryption
- symmetricKey: Verification::Protocol::Types::EncSymmetricKey
HMAC
- symmetricKey: Verification::Protocol::Types::HMACSymmetricKey
Hash

Figure B.1: Class diagram of the package Verification::Protocol::Primitives

B.1.2 Overview of the Package Verification::Protocol::Types

The class diagram of the package Verification::Protocol::Types is depicted in Figure B.2. The package provides primitive types and types for cryptographic keys. The class PrimitiveType has a property type of type PrimiveTypes. The abstract class CryptographicKey is extended by six classes.

Next, we provide an overview of the classes extending the abstract class CryptographicKey.

- The class EncSymmetricKey provides means to specify a cryptographic key used for symmetric encryption. In symmetric encryption, several communication partners use the same key to encrypt and decrypt the communication. The association identicalKeys relates symmetric keys of participants with each other.

- The class HMACSymmetricKey provides means to specify a cryptographic key used to create a hashed message-authentication code (HMAC). As for the stereotype EncSymmetricKey, the stereotype identicalKeys relates symmetric keys of participants with each other.

- The two classes EncPrivateKey and EncPublicKey define a key pair used for asymmetric encryption. The EncPublicKey specifies the public key used for the encryption, while the EncPrivateKey specifies the private key used for the decryption. The association keyPair is used to relate the two parts of the key pair.
- The two classes SigPrivateKey and SigPublicKey define a key pair used to create and validate a digital signature. The SigPrivateKey specifies the private key used for creation, while the SigPublicKey specifies the private key used validating the signature. The association keyPair is used to relate the two parts of the key pair.

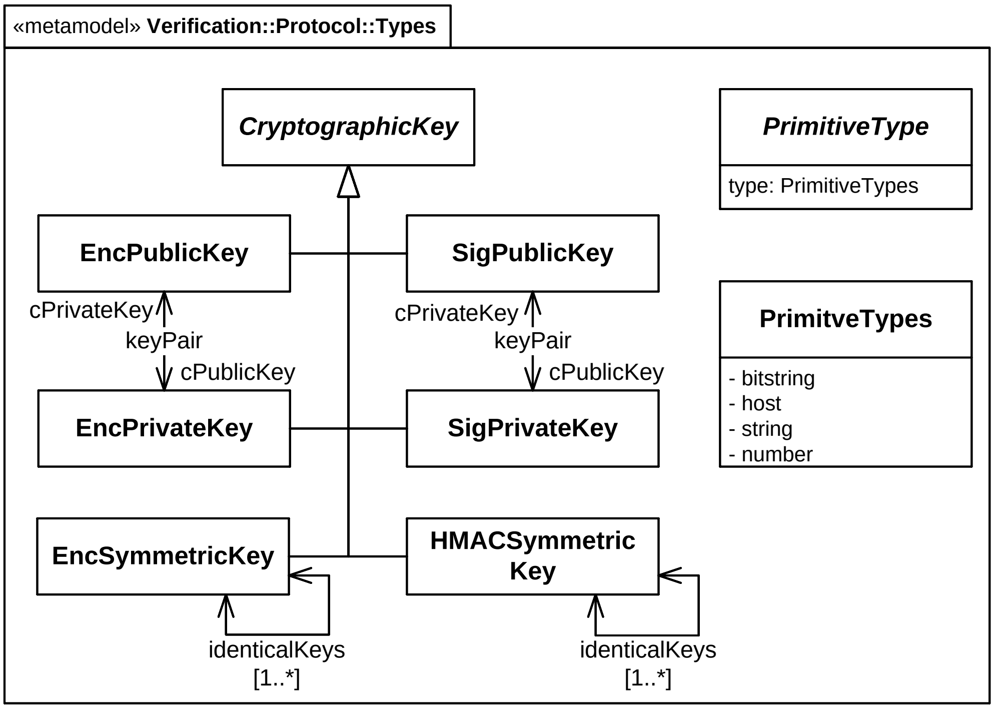


Figure B.2: Class diagram of the package Verification::Protocol::Types

B.2 Supplementing Materials for the Translation from the SecurityProtocolModel to PROVERIF

This section provides supplementing materials for the translation from the SecurityProtocolModel to PROVERIF.

B.2.1 Definition of the Protocol Preamble in PROVERIF

```
(* Definition of the Protocol Preamble *)

type host.

(* XOR Operation *)

fun xor(bitstring, bitstring) : bitstring.
const XOR_ZERO : bitstring.

(* Modular Exponentiation *)
fun exp(bitstring, bitstring) : bitstring.
fun mod(bitstring, bitstring) : bitstring.

equation forall P:bitstring, G:bitstring, x:bitstring, y:bitstring;
  mod(exp(exp(G, x), y), P) = mod(exp(exp(G, y), x), P).

(* Increase / Decrease *)
fun inc(bitstring) : bitstring.
reduc forall x: bitstring; dec(inc(x)) = x.

(* Concat *)
fun concat(bitstring, bitstring) : bitstring.

(* Symmetric Encryption *)
type symmetricKey.

fun senc(bitstring, symmetricKey): bitstring.

reduc forall m: bitstring, k: symmetricKey; sdec(senc(m,k),k) = m.

(* Asymmetric Encyption *)

type ePrivateKey.
type ePublicKey.

fun generatePublicKey4Encryption(ePrivateKey): ePublicKey.
fun aenc(bitstring, ePublicKey): bitstring.

reduc forall m: bitstring, k: ePrivateKey; adec(aenc(m,
    generatePublicKey4Encryption(k)), k) = m.

(* Digital Signature *)

type sPrivateKey.
type sPublicKey.

```

```
fun generatePublicKey4Signature(sPrivateKey): sPublicKey.
fun sign(bitstring, sPrivateKey): bitstring.

reduc forall m: bitstring, k: sPrivateKey; getMessage(sign(m,k)) = m.
reduc forall m: bitstring, k: sPrivateKey; verify(sign(m,k),
    generatePublicKey4Signature(k)) = m.
```

Listing B.1: PROVERIF Protocol Preamble

B.3 Supplementing Materials for the Translation from the Security Protocol Model to TAMARIN

This section provides supplementing materials for the translation from the SecurityProtocolModel to TAMARIN.

B.3.1 Definition of the Protocol Preamble in TAMARIN

```
theory verificationModel.securityProtocolModel.protocol.name

begin

  // ====================================
  // == Import Builtin Functions       ==
  // ====================================
  builtins:
    asymmetric-encryption,
    diffie-hellman,
    hashing,
    multiset,
    signing,
    symmetric-encryption,
    xor

  // ====================================
  // == Custom Functions and Equations ==
  // ====================================

  functions: dec/1, inc/1
  equations: inc(dec(x))=x

  // ====================================
  // == PKI Infrastructure (Setup)     ==
  // ====================================

  rule RegisterPublicEncryptionKey:
    [ Fr(∼skA) ] --[Secret(∼skA), Gen($A)]->[ !Ltk($A, ~skA), !PK($A,
   pk(∼skA)), Out(pk(∼skA)) ]

  rule RevealPrivateEncryptionKey:
    [ !Ltk(A, ltkA) ] --[ Reveal(A)  ]-> [ Out(ltkA) ]


  // ====================================
  // == Symmetric Key (Setup)          ==
  // ====================================

  rule RegisterSymmetricEncryptionKey:
    [ Fr(∼k) ] --[ KeyGen($A), Secret(∼k) ]-> [ !
   SymmetricEncryptionKey($A, ∼k) ]


  rule RevealSymmetricEncryptionKey:
    [ !SymmetricEncryptionKey(A, ∼k) ] --[ Reveal(A) ]-> [ Out(∼k) ]

  // ====================================
  // == Common Restrictions            ==
  // ====================================

  restriction unique:
    "All x #i #j. UniqueFact(x) @#i & UniqueFact(x) @#j ==> #i = #j"

  restriction Equality:
    "All x y #i. Eq(x,y) @#i ==> x = y"

  restriction Inequality:
    "All x #i. Neq(x,x) @ #i ==> F"

  restriction OnlyOnce:
    "All #i #j. OnlyOnce()@#i & OnlyOnce()@#j ==> #i = #j"

  restriction LessThan:
    "All x y #i. LessThan(x,y)@#i ==> Ex z. x + z = y"

  restriction GreaterThan:
    "All x y #i. GreaterThan(x,y)@#i ==> Ex z. x = y + z"
end
```

Listing B.2: TAMARIN Protocol Preamble

C

Own Publication Contributions

[*ABD+19] This publication presents the results of a survey on the state of secure software development. The study was conducted within the research project AppSecure.nrw. I contributed to the design and evaluation of the study and to all sections in joint work with the other authors. Furthermore, I presented the paper.

[*FHK+18a] This publication presents requirement patterns for MSDs. I contributed to parts of the requirement patterns. Furthermore, I contributed to parts of the paper and reviewed the overall paper.

[*FHK+18b] This publication is a supplementing technical report for [*FHK+18a] and presents requirements pattern for MSDs. I contributed to parts of the requirement patterns. Furthermore, I contributed to parts of the paper and reviewed the overall paper.

[*HFK+16a] This publication introduces maturity levels for requirements engineering based on the author's industrial experiences. I contributed to all sections in joint work with the other authors.

[*HFK+16b] This publications consolidates the MSD syntax and semantics that this thesis works with, and introduces the EBEAS. I was one of the main authors of the EBEAS and a section that describes the usage of MSDs by example requirements on the EBEAS. In addition, I reviewed other parts of the paper.

[*KDH+20] This publication introduces the Security Modeling Profile and the model transformations from MSD specifications to ProVerif input models. Thereby, it presents preliminary work for Chapter 3 and Chapter 4. I am the main author of the publication, coordinated its creation, contributed to all sections, and presented the paper.

[*KHD14] This paper introduces model transformations from MSD specifications to CCSL models. I contributed to all sections and reviewed the overall publication.

[*KHL17] This publications present results from a research project on the data exchange by means of STEP models for mechatronic production systems. I developed the concepts of the paper in joint work with the other authors. Furthermore, I coordinated the creation of the publication, contributed to all sections, and presented the paper.

[*KHL18] This publications is an extended version of [*KHL17] and presents results from a research project on the data exchange by means of STEP models for mechatronic production systems. I developed the concepts of the paper in joint work with the other authors. Furthermore, I coordinated the creation of the publication, contributed to all sections, and presented the paper.

[*KHS$^+$16] This publications present results from a research project on variant modeling and on the data exchange by means of STEP models for mechatronic production systems. I developed the concepts of the paper in joint work with the other authors. I coordinated the creation of the publication, contributed to all sections, and presented the paper.

[*KMM$^+$20] This publication presents a experience report on the introduction of software development improvements in a machinery and plant engineering company. I coordinated the creation of the publication, contributed to all sections, and presented the paper.

[*Koc18] This publications presents first ideas on the integration of security and functional requirements to enable their early analysis. Thereby, it presents preliminary work for Chapter 4. I wrote and presented the paper.

[*KTD$^+$22] This publication introduces the approach for security protocol templates and their refercene in a MSD specification. Thereby, it presents preliminary work for Chapter 5. The initial concepts have been developed in [+Tri21]. I am the main author of the publication, coordinated its creation, contributed to all sections, and presented the paper.

[*MHK$^+$15] This publication describes an automatic derivation of initial AUTOSAR models from UML software design specifications in the automotive sector. I contributed to parts of the paper and reviewed the overall paper.