Scenario-based Specification of Security Protocols and Transformation to Security Model Checkers

Thorsten Koch
thorsten.koch@iem.fraunhofer.de
Fraunhofer IEM, Germany

Stefan Dziwok
stefan.dziwok@iem.fraunhofer.de
Fraunhofer IEM, Germany

Jörg Holtmann
joerg.holtmann@iem.fraunhofer.de
Fraunhofer IEM, Germany

Eric Bodden
eric.bodden@upb.de
Paderborn University and Fraunhofer IEM, Germany

ABSTRACT

Security protocols ensure secure communication between and within systems such as internet services, factories, and smartphones. As evidenced by numerous successful attacks against popular protocols such as TLS, designing protocols securely is a tedious and error-prone task. Model checkers greatly aid protocol verification, yet any single model checker is oftentimes insufficient to check a protocol's security in full. Instead, engineers are forced to maintain multiple overlapping and hopefully non-contradicting and non-diverging specifications, one per model-checking tool—an error-prone task.

To address this problem, this paper presents VICE, a scenario-based approach to security-protocol verification. It provides a visual modeling language based for specifying security protocols *independent* of the model checker. It then automatically transforms the relevant fragments of these models into equivalent inputs to multiple model checkers. In result, VICE completely relieves the security engineer from choosing and specifying queries via a fully automatic generation of all necessary queries.

Through a case study involving real-world specifications of eight security protocols, we show that VICE is applicable in practice.

CCS CONCEPTS

• **Security and privacy** → *Cryptography*; • **Software and its engineering** → *Software notations and tools*; *Software verification and validation.*

KEYWORDS

security protocols, verification, model transformation

ACM Reference Format:
Thorsten Koch, Stefan Dziwok, Jörg Holtmann, and Eric Bodden. 2020. Scenario-based Specification of Security Protocols and Transformation to Security Model Checkers. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3365438.3410946

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MODELS '20, October 18–23, 2020, Virtual Event, Canada
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410946

1 INTRODUCTION

In 2019, the World Economic Forum [37] identified software security as the greatest technological risk for the world's population because software-intensive systems such as internet services, factories, and smartphones process critical data and provide critical services. Furthermore, Bucchiarone et al. [7] classify security as one of the grand challenges in the field of model-driven engineering. Especially the message-based communication within and between the systems mentioned above is highly vulnerable to man-in-the-middle attacks, which can cause great damage. To counter this risk, security protocols [5] can ensure the security within communication networks by executing security-related functions and applying cryptographic methods. Here, it is of absolute importance that the protocols are correctly designed and specified. However, designing and specifying such protocols is an error-prone task, due to the complex security requirements and their dependencies on the attacker model [5, 13]. As a consequence, many flaws of security protocols were only discovered after years of productive usage, e.g., it took 17 years to identify a critical flaw in the Needham-Schroeder Public-Key protocol [24].

Nowadays, to analyze whether a security protocol is correct concerning security properties such as confidentiality and authentication, security engineers typically use model checkers such as ProVerif [5], ProVerif-ATP [13], and Tamarin [26]. This, however, is not without limitations. For instance, model checking a security protocol is NP-complete for a bounded number of sessions and even undecidable if that number is unbounded [5, 10]. To deal with this problem, the various model checkers make varying approximations, which results in various analysis limitations. For this and similar reasons, a thorough verification of a security protocol typically requires multiple model checkers with different analysis capabilities.

While specifying and analyzing security protocols using (multiple) model checkers, we identified two challenges that lead to inefficient development and critical security flaws if they are insufficiently handled: (1) All model checkers have their own textual modeling and query languages, which are typically fundamentally different from one another. Moreover, each single input language requires the security engineer to have deep knowledge and experience. However, as stated above, the security engineer has to use multiple model checkers for a thorough analysis and, thus has to re-model the security protocol including its queries in the other languages repeated times. This is time-consuming and error-prone. Moreover, in our experience, the textual input languages of these

model checkers are generally rather hard to comprehend, which adds an additional burden to the engineer. (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 mistakes may remain undetected in the security protocol and 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.

Related work only marginally addresses these challenges. Fang et al. [15] provide an extension for Unified Modeling Language (UML) Interactions [30] to model security protocols and analyze them by means of ProVerif. However, their modeling and query language is very ProVerif-specific. As a consequence, the security engineer still has to learn the ProVerif modeling and query language. 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 the sufficient set of queries. Ameur-Boulifa et al. [4] present a modeling approach based on the Systems Modeling Language (SysML) [29] with the goal of enabling 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 [22]. Particularly, scenario-based notations have an intuitive representation [18] and improve the comprehension of interaction requirements for people experienced in modeling [3].

This paper presents VICE (VIsual Cryptography vErifier), a scenario-based approach to security protocol verification that is model-checker independent. VICE fully solves the two challenges mentioned above: (1) It extends the UML-compliant modeling language Modal Sequence Diagrams (MSDs) [17] to enable the visual and scenario-based specification of security primitives and protocols—independently of a specific model checker and comprehensible for software engineers knowing the UML. (2) It further provides a generic model transformation concept to transform security protocols defined as a sequence diagram into a model checker suited for these protocols. We illustrate this using our extended MSDs and the two model checkers ProVerif and ProVerif-ATP. (3) VICE encapsulates the knowledge formerly hidden within models, documents, and experts to specify and choose the necessary set of queries that the model checker shall verify to decide whether the protocol is secure w.r.t. confidentiality and authentication. VICE realizes this by an automatic generation of all necessary queries from the specified sequence diagram model.

To evaluate VICE, we conduct a case study using eight different security protocols from SPORE (the security protocols open repository)[9]—among others—using the Needham-Schroeder Public-Key protocol, which we also use as a running example. Within our case study, we show that VICE is applicable and useful in practice. In particular, it significantly reduces the effort to specify and analyze a security protocol, reduce the possibility of (critical) security flaws, and makes these tasks more accessible for non-security experts.

The remainder of the paper is structured as follows. We next introduce the fundamentals of this paper. Then, Section 3 presents our language extension to MSDs. Afterward, Section 4 introduces our transformation approach VICE. In Section 5, we conduct a case study to evaluate VICE. Section 6 covers related work. Finally, Section 7 concludes this paper with a summary and an outlook on future work.

2 FOUNDATIONS

We introduce the required foundations for the understanding of this paper. First, Section 2.1 presents basic concepts about the two security model checker ProVerif and ProVerif-ATP. Finally, Section 2.2 introduces Modal Sequence Diagrams.

2.1 ProVerif

ProVerif is an automated model checker for the verification of security protocols presented by Blanchet et al. [5, 6] that is able to prove security properties such as confidentiality and authentication under the *perfect encryption assumption*. Under this assumption, encryption schemes are considered as black boxes and it is assumed that an adversary cannot learn anything from an encrypted message except if he has the corresponding key [5, 10].

ProVerif takes as input a plain text model of a security protocol specified by means of the *applied pi calculus* [1, 2]. 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.

2.1.1 Structure of a ProVerif model. We explain the three different parts of a ProVerif input model using the Needham-Schroeder Public-Key protocol [24]. The security protocol enables mutual authentication between participants. For this purpose, the security protocol relies on a trusted key server, which stores and distributes the public keys of all participants. Due to space limitations, we use only extracts from the ProVerif specification and refer to [6, Chapter 5] for the complete specification.

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

For example, in Listing 1, 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. The asymmetric decryption is specified by the function reduc aDec in Listing 1.

Variables are used to describe communication channels (e.g., free c: channel in line 1 of Listing 2) or other terms that are shared by every participant (e.g., free client : host in line 4 of Listing 2). Variables are typed either by means of predefined types (e.g., channel in line 1 of Listing 2) or by means of user-defined types (e.g., host

```
type privateKey.
type publicKey.

fun genPubKey(privateKey): publicKey.

fun aEnc(bitstring, publicKey): bitstring.
reduc forall m: bitstring, k: privateKey;
  aDec(aEnc(m, genPubKey(k)), k) = m.
```

Listing 1: ProVerif Declarations for Asymmetric Encryption

in line 3 of Listing 2). Functions and variables are by default public, and thus accessible by the attacker. If this is not intended they can be declared as private.

```
free c : channel.
type host.
free client : host.
free server : host.
```

Listing 2: ProVerif Declarations for Channels and Variables

The second part of a ProVerif input model defines the behavior of participants of the security protocol by so-called sub-processes (denoted by the keyword *let*). The behavior described within a sub-process encompasses the declaration of variables (e.g., new nonce : bitstring in line 13 of Listing 3), the sending and receiving of messages over a communication channel (e.g., out(c, (client, server)); in line 5 of Listing 3 and in(c, msg : bitstring); in line 7 of Listing 3) as well as the conditional execution of a sub-process.

```
let processClient(
  cPrivateKey : privateKey, cPublicKey : publicKey
)=
  out(c, (client, server));

  in(c, message : bitstring);
  let( sPublicKey : publicKey, =server)
    = aDec(message, cPrivateKey) in

  new nonce: bitstring;
  out(c, aEnc((client, server), sPublicKey));
.
```

Listing 3: Example for a ProVerif 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. Listing 4 depicts an excerpt of the main process of the Needham-Schroeder Public-Key protocol. In lines 9 - 11 of the main process, two sub-processes are instantiated to be run in parallel (denoted by |) in an unbounded number of sessions (denoted by !).

2.1.2 Security properties. ProVerif is able to prove reachability properties and so-called correspondence assertions, among other things. In this section, we introduce the specification of queries to enable the analysis.

Confidentiality. ProVerif is able to 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 security protocol. To analyze the confidentiality of a term M, a query of the form *query attack(M)* is included in the ProVerif input model [6].

```
process
  new cPrivateKey : private_key;
  let cPublicKey = genPubKey(cPrivateKey) in

  new sPrivateKey : private_key;
  let sPublicKey = genPubKey(sPrivateKey) in

  (
    ( !processClient(cPrivateKey, cPublicKey) )
      |
    ( !processServer(bPrivateKey, bPublicKey) )
  )
```

Listing 4: Example for a ProVerif main process

Authentication. ProVerif is able to prove authentication properties based on correspondence assertions introduced by Woo and Lam [36]. A correspondence assertion captures the relationship between events that occur in the execution of the security protocol. Informally, it can be expressed as "if an event e_1 has been executed, then the event e_2 has been previously executed" [6, p. 19]. The ability to check correspondence assertions is essential to validate authentication properties, as these typically require that certain privileged actions may only be allowed after the successful completion of prior authentication events.

In ProVerif, to mark important steps in the execution of the security protocol, sub-processes and the main process can be annotated with events. To analyze the correspondence assertions and thereby authentication properties of the security protocol, these events must be related to each other using a query of the form query: event e_1() ==> inj-event e_2() [6].

2.1.3 ProVerif-ATP. Although many flaws of security protocols have been detected under the perfect encryption assumption, it is in general too strong for most real-world protocols, since many attacks exploit properties of the cryptographic primitives. In other cases, the execution of protocols relies on some algebraic properties of the cryptographic primitives [10].

To enable the analysis of algebraic properties of the cryptographic primitives, Di Li and Tiu [13] recently introduced ProVerif-ATP as a combination of ProVerif and automated theorem proving (ATP). By these means, ProVerif-ATP is able to find flaws that would not be detected by means of ProVerif.

To specify the input model and the analysis queries, ProVerif-ATP uses the same language as ProVerif. The algebraic properties can also be accommodated in the input model using equations. Although the resulting input model is syntactically correct, it is not accepted by the ProVerif prove engine. Thus, Di Li and Tiu [13] made several changes to the ProVerif prove engine and added their own proof engine based on automated theorem proving. Due to these changes, ProVerif-ATP is only able to analyze confidentiality.

In the remainder of this paper, we use ProVerif to analyze confidentiality and authentication properties under the perfect encryption assumption and ProVerif-ATP to analyze confidentiality properties taking algebraic properties of the cryptographic primitives into account. Thereby, we address a broad range of attacks on protocols.

2.2 Modal Sequence Diagrams (MSDs)

As motivated in the introduction, scenario-based notations comprehensibly represent message-based interaction requirements, and UML Interactions [30, Clause 17] provide such a notation as a visual modeling language by means of the notion of sequence diagrams. However, UML Interactions lack precise semantics regarding universal/existential properties [17] and provide no adequate modeling constructs for the specification of temporal variables and of references to them or to parameter values [35]. The latter point leads to the problem that one cannot specify conditional behavior that depends on the values of temporal variables (typically storing computation results) or on parameter values (e.g., "if (<someComputationResult> ≥ <someValue>) then <someMessageSequence> else <anotherMessageSequence>").

To address the problem of the imprecise semantics regarding universal/existential modal semantics, the Modal profile [17] syntactically extends UML Interactions with modeling constructs as known from Live Sequence Charts [11]. Thereby, this profile introduces a UML-compliant form of Live Sequence Charts, called *Modal Sequence Diagrams (MSDs)*. We present a variant of the Modal profile that additionally provides modeling constructs for temporal variables and references to them, among other things [19]. These modeling constructs and their semantics enable a precise definition of conditional behavior depending on the variables' values. In this paper, we neglect the modal semantics of MSDs but focus on the general structure of MSD specifications according to this Modal profile variant as well as on its syntax and semantics for variables and references to them.

We next introduce the overall structure of such MSD specifications and then the aspects on variables in Section 2.2.2.

2.2.1 Structure of MSD Specifications. In terms of the applied Modal profile variant, an *MSD specification* is structured by means of *MSD use cases*. These encapsulate the requirements on the message-based interaction behavior to be provided by the system under development regarding a self-contained situation. An MSD use case encompasses the participants involved in the situation as well as a set of MSDs describing the requirements on the interactions between these participants. For example, Figure 1 depicts an MSD specification excerpt. Such an MSD specification is subdivided into three parts, which we explain in the following.

UML classes provide reusable types for all MSD use cases of the specification, where the classes encompass operations that are used as message signatures as part of the actual MSDs. For example, the class diagram in the top of Figure 1 contains a class Server. Amongst others, this class encompasses an operation helloServer with the String parameters nonce and ownId.

Based on the UML classes and for each MSD use case, a UML collaboration (dashed ellipse symbol) [30, Clause 11.7] specifies the participant roles involved in the particular use case. These participants are typed by the classes mentioned above and are used as lifelines as part of the actual MSDs. For example, the collaboration in the middle of Figure 1 encompasses, among other things, the participant role server that has an abstract syntax link type to the class Server.

Based on the UML classes and collaborations, a set of MSDs is specified for each UML collaboration and thereby for each MSD

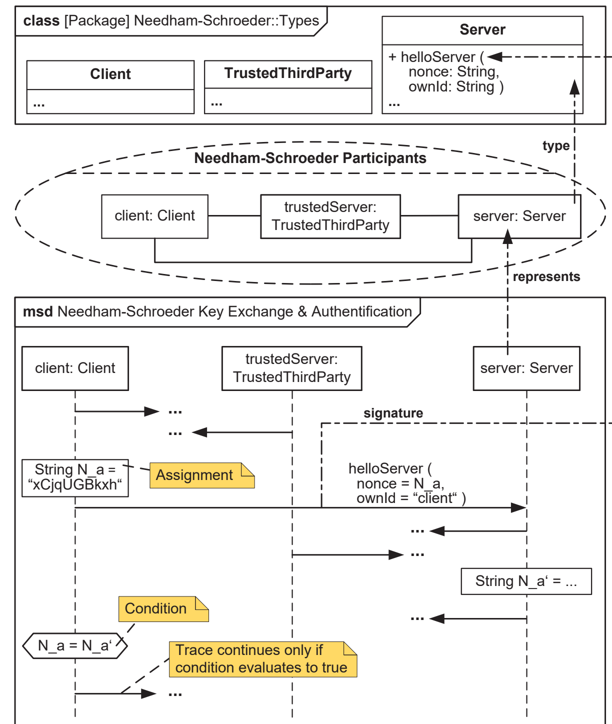

Figure 1: MSD Specification Excerpt

use case. These MSDs specify the requirements on the interactions between the participant roles involved in the use case. For example, the bottom diagram in Figure 1 depicts such an MSD. An MSD encompasses *MSD messages*, which are associated with a sending and a receiving lifeline representing the participant roles as well as an operation signature. For example, the receiving lifeline server: Server of the MSD message helloServer has an abstract syntax link represents to the equally named role in the UML collaboration. Furthermore, the MSD message helloServer is associated with the equally named operation signature of the class Server by means of the abstract syntax link signature.

2.2.2 MSD Semantics for Variables. Assignments in MSDs enable storing temporally valid, intermediate values. They are represented by rectangles that cover one or multiple lifelines and contain expressions in the form <var> = <expr>. In this context, <var> is the name of a typed *diagram variable* temporally 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) [28]. For example, the MSD in the bottom of Figure 1 contains such an assignment covering the lifeline client: Client. In this assignment, the diagram variable N_a of type String is declared, and a random string xCjqUGBkxh is assigned. The diagram variables and thereby their values can be referenced by message arguments and particularly conditions, which we explain in the remainder of this section.

The operations associated by the MSD messages can have parameters of certain types. In the case of the MSD message helloServer as part of the MSD in the bottom of Figure 1, the associated and

equally named operation in the class Server encompasses the String parameters nonce and ownId. The arguments for these parameters can reference diagram variables, for example, the first parameter nonce is specified to carry the value of the diagram variable N_a as argument. Furthermore and like in conventional UML, concrete literal values can be specified for the message parameters. For example, the second parameter ownId is specified to carry the literal values "client" as argument.

To specify conditional behavior, MSDs can contain *conditions*, which are represented as hexagons that cover one or more lifelines. Conditions contain OCL expressions that evaluate to a Boolean value, typically involving one of the diagram variables mentioned above. For example, the MSD in the bottom of Figure 1 contains a condition covering the lifeline client: Client. In this condition, the value of the diagram variable N_a is compared with the value of another diagram variable N_a'. If the expression of condition evaluates to true, the resulting execution trace is continued (the MSD "proceeds"). If the expression evaluates to false, the trace is legal but discontinues (the MSD "terminates").

3 MODELING SECURITY PROTOCOLS

In this section, we present our *Security Modeling profile* which enables to specify security primitives and apply them to the requirements on the communication behavior. In this profile, we align concepts from a study of existing security protocols from SPORE (the security protocol open repository) [9], from examples in ProVerif [5, 6], and from further security model checkers [13, 14, 26].

The Security Modeling profile extends the UML and stereotypes of the Modal profile for the specification of variables and of references to them or to parameter values. The entire profile consists of 20 stereotypes and provides OCL constraints to validate static semantics. The profile has been realized by means of Eclipse Papyrus[1]. Due to space limitations, in this section, we only illustrate an exemplary profile application.

In Figure 2, we apply the profile for the specification of the Needham-Schroeder Public-Key protocol. The communication between the client: Client and the server: Server is asymmetrically encrypted. Hence, our Security Modeling profile enables security engineers to annotate the messages exchanged between the client: Client and the server: Server with the stereotype «asymmetric_encryption». As for all stereotypes that are applied to messages (e.g., symmetric encryption or digital signatures), the stereotype indicates that the message is encrypted before it is sent and that it is decrypted after it is received. The cryptographic key needed to perform the encryption and decryption can be referenced within the stereotype. It can be any property that is annotated with the stereotype «PublicKey» and «PrivateKey», respectively. For example, in Figure 2, the client: Client sends the message helloServer to the server: Server. This message is asymmetrically encrypted by means of the public key pKeyS of the server: Server.

Besides messages, security protocols like the Needham-Schroeder Public-Key protocol encompass assignments and conditions. While assignments are typically used to create and manipulate protocol variables, conditions are used to describe the conditional execution of the protocol. Based on our study of existing security protocols,

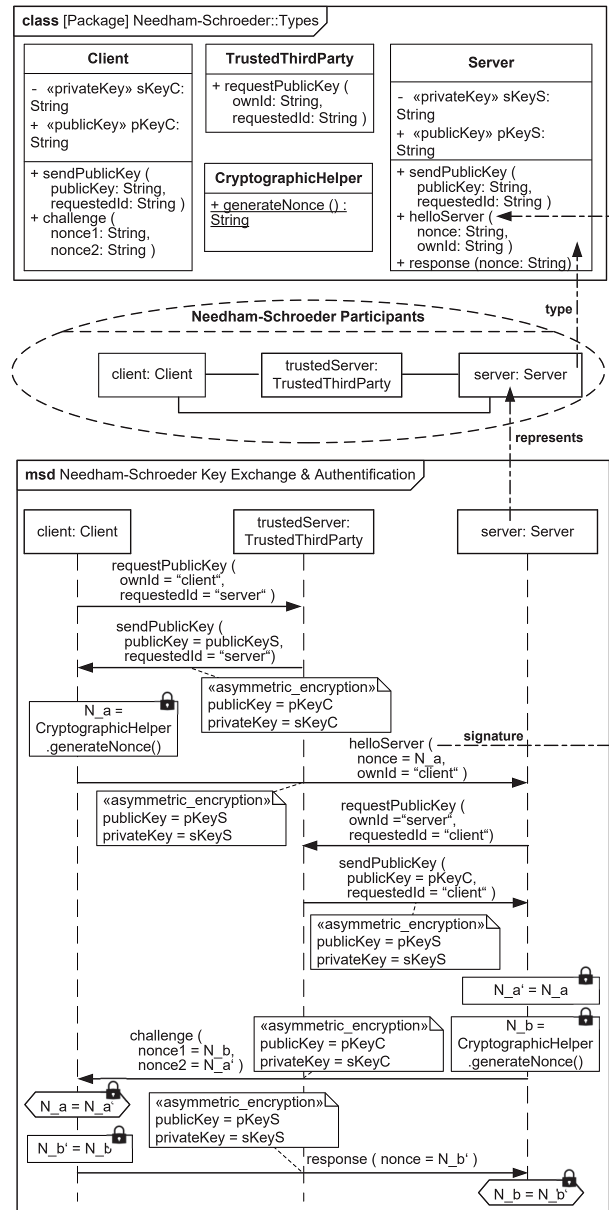

Figure 2: Extended MSD Specification for the Needham-Schroeder Public-Key protocol

we identified a commonly used set of cryptographic types (e.g., nonces, timestamps, primes, and shared keys) that are instantiated during the execution. We encapsulate the creation of corresponding diagram variables in the helper-class CryptographicHelper. In addition, there is a number of algebraic operations (e.g., addition, multiplication, and modular exponentiation), which are frequently used and summarized in the helper-class AlgebraicHelper.

To enable the specification of protocol variables, we introduce the concept of *security assignments*. Like conventional assignments (cf. Section 2.2.2), a security assignment has the form <var> = <expr>, where <var> is the name of a typed diagram variable. The term expr

[1]https://www.eclipse.org/papyrus/

can be any OCL expression evaluating to a value of the type of <var>, including operations of the two helper-classes. To visually distinguish security assignments from non-security assignments, we add a lock at the upper right corner of the rectangle. In Figure 2, we use the operation CryptographicHelper.generateNonce() to assign a random nonce to the diagram variables N_a and N_b.

In addition, to specify conditional behavior of the protocol, we introduced the concept of *security conditions*. A security condition contains OCL expressions that evaluate to a Boolean value, involving diagram variables and operations of the two helper-classes. To visually distinguish security conditions from non-security conditions, we add a lock at the upper right corner of the hexagon. In our example, we use security conditions to validate that the nonces that are received from the communication partner are the same that have been sent previously in the execution of the protocol (e.g., N_a = N_a' in Figure 2).

4 TRANSFORMATION OF SECURITY PROTOCOLS

We next describe our transformation from MSD specification to ProVerif and ProVerif-ATP input models. In Section 4.2 we then present our technique to derive verifiable constraints from the MSD specification.

4.1 Transformation from MSDs to ProVerif

VICE is designed as a model-to-text transformation and implemented by means of the Eclipse-based framework Xtend[2]. By default, VICE creates a ProVerif specification and analyzes the security protocol under the perfect encryption assumption w.r.t. confidentiality and authentication. However, if VICE detects that the MSD specification contains elements that rely on algebraic properties, it additionally generates a ProVerif-ATP specification and performs the analysis accordingly.

Since the input models of ProVerif and ProVerif-ATP are based on the same input language, the transformation rules do not differ very much. The transformation algorithm consists of four steps. In each step, it generates different parts of the input model based on the different parts of the MSD specification. In the following, we describe each step in further details:

4.1.1 Transformation Step 1 — generateProtocolPreamble. In the first step, the transformation algorithm checks if the MSD specification is specified correctly using the OCL constraints defined in our profile. If this is not the case, the transformation is aborted and the security engineer is informed accordingly. If the MSD specification is specified correctly, the transformation algorithm generates the first part of the ProVerif input model encompassing all declarations of types and functions that are supported by our profile. For example, for the stereotype «asymmetric_encryption», the transformation algorithm generates the ProVerif constructs depicted in Listing 1. Furthermore, for all operations that are provided by the CryptographicHelper and AlgebraicHelper, the transformation algorithm creates corresponding types and functions.

For the generation of ProVerif-ATP input models, further equations are generated to express the algebraic properties.

[2] https://www.eclipse.org/xtend/

4.1.2 Transformation Step 2 — generateProtocolStructure. In the second step, the transformation algorithm transfers structural information of the MSD specification into ProVerif constructs. Therefore, it generates free variables for each property of the UML classes. Furthermore, it generates a free variable of type host for all roles contained in the UML collaboration. Finally, a communication channel is modeled by means of a free channel named with the name of the UML collaboration.

4.1.3 Transformation Step 3 — generateProtocolBehavior. In the third step, the transformation algorithm generates a ProVerif subprocess for each lifeline contained in the MSD. These ProVerif sub-processes encompass all sending and receiving messages, as well as security assignments and security conditions. Therefore, the transformation algorithm iterates over the ordered set of interaction fragments for each lifeline and transforms them into corresponding constructs in ProVerif.

Due to space reasons, we illustrate the transformation approach only for an exemplary set of stereotypes contained in our profile. Nevertheless, the transformation rules for the remaining stereotypes are similar.

Transformation of a sending message occurrence specification. The transformation rules 1-3, depicted in Figure 3, illustrate the transformation of a sending message occurrence specification and differ mainly in the number of security-related stereotypes applied to the message. The transformation encompasses four steps, whereas the first two and the last one are identical for all three cases.

First, the transformation algorithm resolves the signature of the message and, thus, obtains the list of parameters. For each of them, the transformation algorithm creates a new variable of the form vX_param : bitstring, where v is used as a prefix, X is a consecutive number, and *param* is the name of the message parameter (cf. line 1 in transformation rules 1-3 in Figure 3) The variables are numbered consecutively so that the same parameter name or the repeated occurrence of a message does not result in the variable being rebound.

Second, the transformation algorithm checks whether the arguments for the message parameters have been set within the MSD. If this is the case, the transformation algorithm assigns the argument value to the message parameter (e.g., vX_param = arg; in transformation rules 1-3 in Figure 3). Hereby, it is possible to reuse values in the resulting security protocol in ProVerif.

Third, the transformation algorithm checks if security-related stereotypes are applied to the message. We distinguish the following three cases:

- **No applied security stereotype:** If no security-related stereotype is applied to the message, the transformation algorithm creates a new variable representing the bitstring of the message and assigns the set of the parameter variables to this variable (cf. line 5 of transformation rule 1 in Figure 3).
- **One applied security stereotype:** If one security-related stereotype is applied to the message, the transformation algorithm resolves the function that belongs to the stereotype (e.g., aEnc for the stereotype asymmetric_encryption). Afterward, the transformation algorithm creates a new variable representing the bitstring of the message and assigns the

result of the function to this variable. The set of message parameters is used as input of the function. If the function needs a cryptographic key, this key is referenced by the stereotype (cf. line 5 of transformation rule 2 in Figure 3).

More than one applied security stereotype: If more than one security-related stereotype is applied to the message, the transformation algorithm iteratively resolves the ProVerif function that belongs to the security stereotype and applies it as described in the previous case. The result of the former function is used as input for the next one (cf. lines 5 and 6 of transformation rule 3 in Figure 3).

Finally, the transformation algorithm generates an out(channel, msg) construct, where channel is the channel used for the communication between the participants and msg is the bitstring to be transmitted. Thereby, the out-construct represents the sending of the message (cf. line 7 of transformation rules 1-3 in Figure 3).

Transformation of a receiving message occurrence specification. The transformation rules 4-6, depicted in Figure 3, illustrate the transformation of a receiving message occurrence specification. The transformation rules are very similar to the transformation rules presented for a sending message occurrence specification.

First, the transformation algorithm creates a in(channel, msg : bitstring) construct as the counterpart of the receiving message occurrence specification (cf. in(channel, msg : bitstring) in line 1 of transformation rules 4 - 6 in Figure 3).

Second, the received bitstring msg is disassembled into its parts according to the number of applied security-related stereotypes. The idea of the transformation rules is similar to the transformation rules for a sending message occurrence specification. If no security related-stereotype is applied to the message, the transformation algorithm resolves the parameter of the message and creates a let construct of the form let(v0_param) = message (cf. transformation rule 4 in Figure 3). If one or more security-related stereotypes are applied to the message, the transformation algorithm first uses the function that corresponds to the applied security-related stereotype and proceeds with the creation of the let construct as described before (cf. transformation rule 5 and 6 in Figure 3).

Transformation of Security Assignments. As described in Section 3, a security assignment is used to either create new variables or modify existing variables. In both cases, the security assignment refers to operations of the helper-classes CryptographicHelper and AlgebraicHelper. The transformation algorithm creates a new variable v_1 for the variable of the assignment if it has not been declared previously. Furthermore, if the operation describes the modification of an existing variable v_2, the operation is applied to the variable v_2 and stored in the variable v_1.

Transformation of Security Conditions. As described in Section 2.2.2, an MSD proceeds only after a condition evaluates to true, otherwise, the MSD terminates. In ProVerif, a condition has the structure if <condition> then <P> else <Q>, where $< P >$ and $< Q >$ are sub-processes. However, if the sub-processes $< Q >$ does not contain any behavior, the else part can be omitted. Thus, the transformation algorithm only generates the if <condition> then <P> part of the condition. Therefore, the transformation algorithm resolves the variables of the condition and creates a corresponding construct.

4.1.4 Transformation Step 4 — generateMainProcess.

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.

4.2 Creation of Analysis Queries

In this section, we describe the generation of queries based on the MSD specification to enable the analysis of the security properties confidentiality and authentication.

4.2.1 Generation of Confidentiality Queries.

If security engineers want to analyze the confidentiality of protocol variables in ProVerif and ProVerif-ATP, they will have to manually add confidentiality queries of the form query attacker(secret) to the input model. In our approach, we want to reduce the effort for security engineers, and thus our transformation algorithm generates the confidentiality queries automatically based on the MSD specification.

The generation of confidentiality queries encompasses two parts. In the first part, the transformation algorithm generates confidentiality queries for all private properties contained in the UML classes. For the example depicted in Figure 2, the transformation algorithm generates, the query query attacker(sKeyC) for the role client : Client and the property sKeyC : String of the UML class Client.

In the second part, the transformation algorithm generates confidentiality queries for all diagram variables created by means of security assignments during the execution of the protocol. For example, the transformation algorithm generates, amongst other things, the query query attacker(new N_a) for the diagram variable N_a = CryptographicHelper.createNonce() to check whether an attacker can learn the nonce N_a during the execution of the protocol.

This procedure generates corresponding queries for all secret variables and the security engineer can no longer forget a query. However, it may happen that the transformation algorithm is too restrictive and creates queries for diagram variables that the security engineer already knows are public. In these cases, the security engineer can define the diagram variable as public by means of a property of the security assignments, and thus no secrecy query is created.

4.2.2 Generation of Authentication Queries.

If security engineers want to analyze the authentication of protocol participants in ProVerif, they will have to manually add correspondence assertions to the ProVerif input model. As stated in Section 2.1.2, correspondence assertions have the form query: event $e_1()$ ==> inj-event $e_2()$ and are used to capture the relationship between events that mark important stages in the execution of the security protocol.

The placement of events within the ProVerif input model usually requires a deeper understanding of the security protocol. However, Blanchet et al. [6] explain that there is some flexibility in the placement of the correspondence assertions. "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 message output. Otherwise, the whole protocol can be executed without executing the latter event, so the correspondence certainly does not hold" [6].

Figure 3: Transformation rules for the transformation from MSDs to ProVerif

For the generation of correspondence assertions based on an MSD specification, we use the guidelines described above. To enable the analysis of the authentication of a participant A to a participant B, the transformation algorithm generates a correspondence assertion of the form query event terminatedB ==> inj-event lastMessageFromAtoB(). Therefore, it identifies the last message sent from A to B and annotates the generated ProVerif process of A with the event $lastMessageFromAtoB()$ accordingly. Furthermore, the transformation algorithm annotates the generated process of B with the event $terminatedB()$ at the end of the process description.

The generation of correspondence assertions is applied to each communication pair of the MSD specification. This ensures that all necessary queries are generated. However, it may happen that the transformation algorithm is too restrictive and creates correspondence assertions although the security engineer already knows that the authentication property does not hold. Hence, in future releases of VICE, we plan to enable the security engineer to exclude pairs of communication partners from analysis.

5 CASE STUDY

To evaluate the efficacy of VICE, we conduct a case study based on the guidelines by Kitchenham et al. [21] and Runeson et al. [31, 32]. Here, we investigate the applicability and usefulness of VICE in practice.

In preparation of the case study, we implemented a prototype of VICE based on ScenarioTools MSD tool suite[3] based on Eclipse Papyrus. For the transformation from MSD specifications to ProVerif and ProVerif-ATP input models, we used the Eclipse-based framework Xtend. Finally, we set up an installation of ProVerif and ProVerif-ATP within a Docker container and implemented a web interface for the communication between ScenarioTools and the two model checkers.

5.1 Case Study Context

We examine the three evaluation questions (EQ):

[3]http://scenariotools.org/projects2/

EQ1 Does VICE enable the specification of real-world security protocols?

EQ2 Does VICE generate syntactically and semantically correct ProVerif and ProVerif-ATP input models?

EQ3 Does VICE's automatic derivation of analysis queries lead to correct queries for the security analysis of ProVerif and ProVerif-ATP?

For this purpose, we select eight different real-world security protocols from SPORE (the security protocol open repository) [9]. The selected security protocols use different cryptographic primitives and, thus, present a broad range of possible security protocols. Furthermore, for each protocol, researchers have previously performed a formal analysis of potential weaknesses. Hence, we are able to validate VICE's results against theirs.

5.2 Research Hypotheses

Based on our objective and evaluation questions, we define the following four evaluation hypotheses:

H1 The security protocols of the different cases can be specified by using our extended MSDs as presented in Section 3. For evaluating H1, we model eight different security protocols. We rate the hypothesis as fulfilled if these protocols can be specified using solely the extended MSDs presented in Section 3.

H2 VICE can automatically transform the eight security protocols into syntactically correct ProVerif models. For evaluating H2, we generate the ProVerif input models for the MSD specifications created for the evaluation of H1 and analyze them in ProVerif. If the security protocols contain primitives that rely on algebraic properties, we further analyze the generated ProVerif-ATP specification. We consider H2 as fulfilled if all ProVerif models can be opened and analyzed in ProVerif and if necessary in ProVerif-ATP.

H3 The automatic derivation of analysis queries is correct and complete. For evaluating H3, we manually investigate the generated ProVerif inputs models for the MSD specifications and check whether the generated queries are complete and correctly specified. We consider H3 as fulfilled if VICE generates all queries that are necessary to prove the security of the eight security protocols.

H4 The overall security analysis of ProVerif and ProVerif-ATP leads to the same results as previously performed formal analysis of the security protocols. We consider H4 as fulfilled if the analysis results match our expectations.

5.3 Hypothesis validation

For the validation of the hypotheses, we apply VICE to the eight security protocols. We first model the MSD specifications for each security protocol and check whether our MSD extensions are expressive enough to specify the security protocol. Second, we use VICE to generate the corresponding ProVerif input models including queries. We open the generated input models in ProVerif. Finally, we perform the ProVerif analysis for each input model. We manually inspect the results of each analysis for correctness.

In addition, if the security protocol contains primitives that rely on algebraic properties, we open the generated input model in ProVerif-ATP, perform the analysis, and manually inspect the result of the analysis for correctness.

5.4 Result Analysis

The results of the case study are depicted in Table 1. For all selected security protocols, the corresponding MSD specification uses only modeling elements introduced in Section 3. Furthermore, VICE is able to transform the MSD specifications to syntactically and semantically correct input models. We were able to open and analyze each input model with ProVerif and ProVerif-ATP, respectively. Thus, we conclude that H1 and H2 are fulfilled.

Finally, we performed the security analysis based on the derived queries. The analysis results achieved are the same as stated in the literature for the different security protocols. Thus, we conclude that H3 and H4 are fulfilled.

Concluding the case study, the fulfilled hypotheses indicate that VICE for the specification and analysis of security protocols is applicable and useful in practice.

5.5 Threats to Validity

The validity of our study results is threatened by the following facts. First, we only considered eight 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 and, thus, we do not expect large deviations for other security protocols.

Second, the modeling of the security protocols has been done by the same researcher that developed the approach. Since the researcher might have a bias toward the developed model-driven approach, the case study would be more significant, if security experts would have modeled the security protocols. To mitigate this threat, we selected security protocols for which security analyses exist in literature and compared our results against these.

6 RELATED WORK

There exist many different approaches for the specification and analysis of security protocols.

Fang et al. [15, 33] 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 properties of the specified security protocol. However, they do not provide a transformation to ProVerif-ATP. 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 the sufficient set of analysis queries.

Ameur-Boulifa et al. [4, 25] 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 confidentiality and authenticity. 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

Protocol	Generated correct input models	Generated correct queries	Analysis results match expectations
Andrew Secure RPC [8]	✓	✓	✓
Andrew Secure RPC (BAN) [8]	✓	✓	✓
CH07 [34]	✓	✓	✓
Denning-Sacco Shared Key [12]	✓	✓	✓
Diffie-Hellman Key Exchange [10]	✓	✓	✓
Gong's Mutual Authentication Protocol [16]	✓	✓	✓
Needham-Schroeder Public Key [24]	✓	✓	✓
Wired Equivalent Privacy Protocol [10]	✓	✓	✓

Table 1: Results of the case study

confidentiality queries, the security engineer has to manually define authenticity 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 [22]. In addition, we support the security engineer in choosing the sufficient set of analysis queries (confidentiality and authenticity). Thus, the security engineer no longer has to learn the query language. Furthermore, he cannot make mistakes when defining queries or forget important queries.

Lobddersted et al. [23] 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 the modeling and analysis of security protocols in general and not only on access controls. In addition, we provide the verification of the security protocols to ensure that they fulfill the desired confidentiality and authenticity properties.

UMLSec [20] 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 security properties of software components and of their composition within a UML framework. Similar to UMLSec, Moebius et al. [27] 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 security properties like confidentiality and authenticity.

In summary, most approaches focus on the modeling of security protocols and their transformation into a particular model checker for analyzing security properties. 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 language. In contrast, our modeling language 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 properties from our extended MSD scenarios and thus significantly reduce the effort for the modeling and analysis of security protocols.

7 CONCLUSION AND FUTURE WORK

In this paper, we present VICE, an approach for the scenario-based specification of security protocols and their transformation into different security model checkers for confidentiality and authentication analysis purposes. To this end, we contribute two building blocks. First, we extend the UML-based scenario formalism MSDs with language constructs enabling the specification of security protocols by means of our Security Modeling UML profile. Second, we present a model transformation from such extended MSD scenarios into the two security model checker ProVerif and ProVerif-ATP, where the transformation generates all relevant queries and automatically determines if the analysis by means of ProVerif-ATP is necessary. By means of eight real-world security protocols, we evaluate whether the language constructs are adequate for the protocol specification, whether the transformation derives correct model checker inputs, and whether the model checkers yield the expected analysis results.

Our modeling language extensions enable security engineers to specify security protocols in an intuitive way by applying scenarios for message-based interactions. Particularly, the security engineers do not need any knowledge on the input and query languages of the model checkers, because the transformation completely encapsulates this knowledge. Furthermore, the engineers do not have to care about which properties they have to analyze in which model checker, because our transformation determines this decision based on the information specified in the scenarios. The evaluation indicates the practical applicability and usefulness of VICE.

Future work encompasses two aspects. First, we plan to conduct a user study to evaluate whether our modeling language is intuitive and easy to use for security engineers compared to the input languages of the security model checkers. Second, we want to integrate further model checkers (e.g., Tamarin [26]) to show that VICE is applicable to model checkers whose input and query language is completely different.

ACKNOWLEDGMENTS

This research has been partly sponsored by the project "AppSecure.nrw - Security-by-Design of Java-based Applications" funded by the European Regional Development Fund (ERDF-0801379).

REFERENCES

[1] Martín Abadi, Bruno Blanchet, and Cédric Fournet. [n.d.]. The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. http://arxiv.org/pdf/1609.03003v2

[2] Martín Abadi and Cédric Fournet. 2001. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '01*, Chris Hankin and Dave Schmidt (Eds.). ACM Press, New York, New York, USA, 104–115. https://doi.org/10.1145/360204.360213

[3] Silvia Abrahão, Carmine Gravino, Emilio Insfran, Giuseppe Scanniello, and Genoveffa Tortora. 2013. Assessing the Effectiveness of Sequence Diagrams in the Comprehension of Functional Requirements: Results from a Family of Five Experiments. *IEEE Transactions on Software Engineering* 39, 3 (2013), 327–342. https://doi.org/10.1109/TSE.2012.27

[4] Rabéa Ameur-Boulifa, Florian Lugou, and Ludovic Apvrille. 2019. SysML Model Transformation for Safety and Security Analysis. In *Security and Safety Interplay of Intelligent Software Systems*, Brahim Hamid, Barbara Gallina, Asaf Shabtai, Yuval Elovici, and Joaquin Garcia-Alfaro (Eds.). Lecture Notes in Computer Science, Vol. 11552. Springer International Publishing, Cham, 35–49. https://doi.org/10.1007/978-3-030-16874-2{_}3

[5] Bruno Blanchet. 11-13 June 2001. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001*. IEEE, 82–96. https://doi.org/10.1109/CSFW.2001.930138

[6] Bruno Blanchet. 2016. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends® in Privacy and Security* 1, 1-2 (2016), 1–135. https://doi.org/10.1561/3300000004

[7] Antonio Bucchiarone, Jordi Cabot, Richard F. Paige, and Alfonso Pierantonio. 2020. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling* 19, 1 (2020), 5–13. https://doi.org/10.1007/s10270-019-00773-6

[8] Michael Burrows, Martin Abadi, and Roger Needham. 1990. A logic of authentication. *ACM Transactions on Computer Systems (TOCS)* 8, 1 (1990), 18–36. https://doi.org/10.1145/77648.77649

[9] John Clark and Jeremy Jacob. 2002. Security Protocols Open Repository. http://www.lsv.fr/Software/spore/index.html

[10] Véronique Cortier, Stéphanie Delaune, and Pascal Lafourcade. 2006. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security* 14, 1 (2006), 1–43. https://doi.org/10.3233/JCS-2006-14101

[11] Werner Damm and David Harel. 2001. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design* 19, 1 (2001), 45–80. https://doi.org/10.1023/A:1011227529550

[12] Dorothy E. Denning and Giovanni Maria Sacco. 1981. Timestamps in key distribution protocols. *Commun. ACM* 24, 8 (1981), 533–536. https://doi.org/10.1145/358722.358740

[13] Long Di Li and Alwen Tiu. 2019. Combining ProVerif and Automated Theorem Provers for Security Protocol Verification. In *Automated Deduction – CADE 27*, Pascal Fontaine (Ed.). Springer International Publishing, Cham, 354–365.

[14] Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, and Ralf Sasse. 09.07.2018 - 12.07.2018. Automated Unbounded Verification of Stateful Cryptographic Protocols with Exclusive OR. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 359–373. https://doi.org/10.1109/CSF.2018.00033

[15] Kunding Fang, Xiaohong Li, Jianye Hao, and Zhiyong Feng. 23.08.2016 - 26.08.2016. Formal Modeling and Verification of Security Protocols on Cloud Computing Systems Based on UML 2.3. In *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 852–859. https://doi.org/10.1109/TrustCom.2016.0148

[16] Li Gong. 1989. Using one-way functions for authentication. *ACM SIGCOMM Computer Communication Review* 19, 5 (1989), 8–11. https://doi.org/10.1145/74681.74682

[17] David Harel and Shahar Maoz. 2008. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software & Systems Modeling* 7, 2 (2008), 237–252. https://doi.org/10.1007/s10270-007-0054-z

[18] Jameleddine Hassine, Juergen Rilling, and Rachida Dssouli. 2010. An Evaluation of Timed Scenario Notations. *Journal of Systems and Software* 83, 2 (2010), 326–350. https://doi.org/10.1016/j.jss.2009.09.014

[19] Jörg Holtmann, Markus Fockel, Thorsten Koch, David Schmelter, Christian Brenner, Ruslan Bernijazov, and Marcel Sander. 2016. *The MechatronicUML Requirements Engineering Method: Process and Language*. Technical Report tr-ri-16-351. Software Engineering Department, Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute.

[20] Jan Jürjens. 2002. UMLsec: Extending UML for Secure Systems Development. In *The unified modeling language*, Jean-Marc Jézéquel (Ed.). Lecture Notes in Computer Science, Vol. 2460. Springer, Berlin [u.a.], 412–425. https://doi.org/10.1007/3-540-45800-X{_}32

[21] Barbara Kitchenham, Lesley M. Pickard, and Shari Lawrence Pfleeger. 1995. Case studies for method and tool evaluation. *IEEE Software* 12, 4 (1995), 52–62. https://doi.org/10.1109/52.391832

[22] Grischa Liebel and Matthias Tichy. 2015. Comparing Comprehensibility of Modelling Languages for Specifying Behavioural Requirements. In *Proceedings of the First International Workshop on Human Factors in Modeling (HuFaMo)*. 17–24.

[23] Torsten Lodderstedt, David Basin, and Jürgen Doser. 2002. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *≪UML≫ 2002 — The Unified Modeling Language*, Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 426–441.

[24] Gavin Lowe. 1996. Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Tiziana Margaria, and Bernhard Steffen (Eds.). Lecture Notes in Computer Science, Vol. 1055. Springer Berlin Heidelberg, Berlin, Heidelberg, 147–166. https://doi.org/10.1007/3-540-61042-1{_}43

[25] Florian Lugou, Letitia W. Li, Ludovic Apvrille, and Rabéa Ameur-Boulifa. 2016. SysML Models and Model Transformation for Security. In *MODELSWARD 2016*, Slimane Hammoudi (Ed.). SCITEPRESS - Science and Technology Publications Lda, Setúbal, 331–338. https://doi.org/10.5220/0005748703310338

[26] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 696–701. https://doi.org/10.1007/978-3-642-39799-8$\backslash${_}48

[27] Nina Moebius, Kurt Stenzel, Holger Grandy, and Wolfgang Reif. 16.03.2009 - 19.03.2009. SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications. In *2009 International Conference on Availability, Reliability and Security*. IEEE, 841–846. https://doi.org/10.1109/ARES.2009.22

[28] Object Management Group (OMG). 2014. *OMG Object Constraint Language (OCL) – Version 2.4*. OMG Document Number: formal/14-02-03.

[29] Object Management Group (OMG). 2017. *OMG Systems Modeling Language (OMG SysML) – Version 1.5*. OMG Document Number: formal/2017-05-01.

[30] Object Management Group (OMG). 2017. *OMG Unified Modeling Language (OMG UML) – Version 2.5.1*. OMG Document Number: formal/2017-12-05.

[31] Per Runeson (Ed.). 2012. *Case study research in software engineering: Guidelines and examples* (1st ed. ed.). Wiley, Hoboken, N.J. https://doi.org/10.1002/9781118181034

[32] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (2009), 131–164. https://doi.org/10.1007/s10664-008-9102-8

[33] Gang Shen, Xiaohong Li, Ruitao Feng, Guangquan Xu, Jing Hu, and Zhiyong Feng. 2014. An Extended UML Method for the Verification of Security Protocols. In *2014 19th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, Piscataway, NJ, 19–28. https://doi.org/10.1109/ICECCS.2014.12

[34] Ton van Deursen and Saša Radomirović. 2009. Attacks on RFID Protocols. *Cryptology ePrint Archive* 2008, 310 (6 Aug. 2009), 1–56.

[35] Marc-Florian Wendland, Martin Schneider, and Øystein Haugen. 2013. Evolution of the UML Interactions Metamodel. In *Proceedings MODELS 2013 (LNCS)*. Springer, 405–421. https://doi.org/10.1007/978-3-642-41533-3_25

[36] Thomas Woo and Simon S. Lam. 24-26 May 1993. A semantic model for authentication protocols. In *Proceedings 1993 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE Comput. Soc. Press, 178–194. https://doi.org/10.1109/RISP.1993.287633

[37] World Economic Forum. 2019. *Global risks 2019: Insight report* (14th edition ed.). World Economic Forum, Geneva.