# Model Checking of Security-Critical Applications in a Model-Driven Approach

Marian Borek, Nina Moebius,
Kurt Stenzel, and Wolfgang Reif

Department of Software Engineering,
University of Augsburg, Germany
{borek,stenzel,moebius,reif}@informatik.uni-augsburg.de

**Abstract.** This paper illustrates the integration of model checking in SecureMDD, a model-driven approach for the development of security-critical applications. In addition to a formal model for interactive verification as well as executable code, a formal system specification for model checking is generated automatically from a UML model. Model checking is used to find attacks automatically and interactive verification is used by an expert to guarantee security properties. We use AVANTSSAR for model checking and KIV for interactive verification. The integration of AVANTSSAR in SecureMDD and the advantages and disadvantages over interactive verification with KIV are demonstrated with a smart card based electronic ticketing example.

## 1 Introduction

Security-critical system vulnerabilities are reported constantly. Such systems range from a desktop application (e.g., a browser) to security-critical systems like MasterCard and VISA [22] or the Google single sign-on password system [21]. To identify and eliminate protocol flaws during development model checking can be used. Therefore, an input model for the model checker has to be created which is then used to find attacks or automatically check security properties. There are several model checkers (e.g. NuSMV[8], SPIN[12], PRISM[11]) but only a few are tailored towards cryptographic protocols. AVANTSSAR[1] is a project for Automated Validation of Trust and Security of Service-oriented Architectures. It integrates three different model checkers (Cl-AtSe, SATMC, OFMC) by using a common input language called ASLan++[20]. It can be used to find flaws in cryptographic protocols [2] or to prove security properties under some assumptions (e.g. a fixed number of loop executions or a fixed trace length). Cryptographic protocols are based on message exchange over channels that are influenced by an attacker. Since the specification of a system using cryptographic protocols can quickly become too large and complex for a model checker due to computing resource constraints, it is necessary to abstract and simplify this specification. When security properties are to be checked for an application using cryptographic protocols (such as an electronic ticketing system) then usually the whole application has to be abstracted. This is time-consuming, error-prone and needs a lot of expertise.

SecureMDD is a model-driven approach for developing security-critical applications. From a platform independent model runnable code as well as a formal specification for interactive verification of application-specific security properties can be generated automatically. Interactive verification is time-consuming and requires expert know-how but it guarantees the properties for an arbitrary number of protocol runs and loop executions. On the other hand, classic model checking assures an automatic validation of properties, but only for fixed number of protocol runs. In our approach, model checking is meant to be an addition to interactive verification by an expert and can be used to find attacks. It is also useful to eliminate some security flaws before using interactive verification.

This paper focuses on the integration of AVANTSSAR into SecureMDD as well as on the question how far a model of an application which is used to generate runnable code can be abstracted automatically for validation of application-specific security properties with AVANTSSAR. As a result, a SecureMDD application can be model-checked automatically.

This paper is structured as follows. Section 2 gives an overview of the model-driven approach and Section 3 pictures an eTicket example. Section 4 describes the transformation from a SecureMDD model into an ASLan++ specification and section 5 shows some abstraction rules. Section 6 explains some security flaws using the eTicket example and section 7 compares model checking and interactive verification. Section 8 discusses related work and concludes this paper.

## 2 The SecureMDD Approach

SecureMDD is a model-driven approach to develop security-critical systems. From a model that represents a system, runnable code, a formal specification for interactive verification and an ASLan++ specification for model checking can be generated automatically. The formal specification is used to verify application-specific security properties for an infinite number of agents and protocol runs. The ASLan++ specification is only used to find security flaws due to the limitations like a finite number of agents and protocol runs. Examples for mentioned security-properties are that during a transfer in an online banking system no money is lost, that security-critical data remains secret or that a Dolev-Yao attacker [10] cannot harm the system.

The SecureMDD approach (see Fig. 1) uses a platform-independent UML model, a UML profile as well as a platform-independent and easy to use modeling language MEL [17] [6] to define security-critical applications. Based on the platform-independent application model a formal specification and three platform-specific models (one for each component type) are generated. The formal specification is the basis for the verification of application-specific security properties [18] using the theorem prover KIV[4]. The platform-specific models are tailored for their target platforms, e.g., Java Card for a smart card, Java for a terminal or a PC, and Java based Web services for a service. A smart card is a secure device that can be accessed only via a predefined interface and is tamper-proof: nobody has access to the operating system or the internal memory

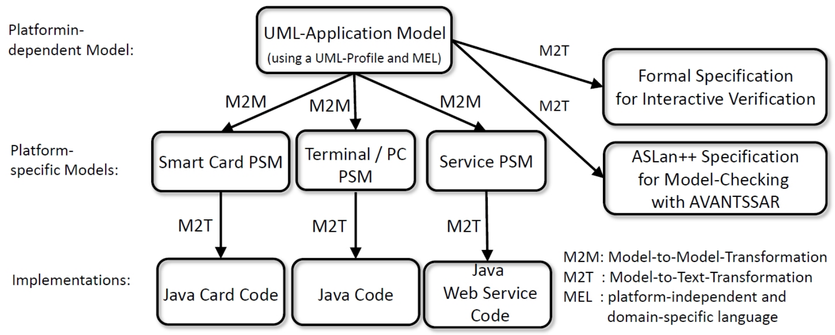
Platformin-dependent Model:
Platform-specific Models:
Implementations:
UML-Application Model (using a UML-Profile and MEL)
M2M
M2M
M2M
M2T
M2T
Formal Specification for Interactive Verification
ASLan++ Specification for Model-Checking with AVANTSSAR
Smart Card PSM
Terminal / PC PSM
Service PSM
M2T
M2T
M2T
Java Card Code
Java Code
Java Web Service Code
M2M: Model-to-Model-Transformation
M2T : Model-to-Text-Transformation
MEL : platform-independent and domain-specific language

**Fig. 1.** SecureMDD Approach

directly. A terminal is also a secure device that receives instructions from a user and can have an interface for the communication with a smart card. A PC is a personal computer where the user has access to internal storage. A service will be deployed on servers that are assumed to be secure devices. Services can be connected by terminals or other services over a network. To perform tasks, a service can orchestrate other services, or several autonomous services can collaborate together. Services can only be accessed via their specified interfaces.

The approach illustrated in Fig. 1 is fully tool-supported and all model transformations are implemented. In this paper, we focus on the integration of AVANTSSAR in SecureMDD.

## 3 Electronic Ticket Example

By using the electronic ticketing system a user is able to buy train tickets online. The tickets are stored on his smart card and can be inspected multiple times. Only a genuine inspector device is able to validate and “punch” tickets. Additionally, only the card owner is able to buy tickets and to view or delete his purchased tickets.

Fig. 2 shows the deployment diagram of the application. It defines the components, the communication structure and the abilities of the attacker. There are two kinds of users involved in the system, the card owner (*User*) and the inspector (*Inspector*). The card owner can use his home PC (*UserDevice*) to buy, show or delete tickets on the card (*ETicketCard*). To buy a new ticket, the *UserDevice* connects to a server called *ETicketServer*. An *Inspector* is able to validate and “punch” tickets on a valid card with his inspector device (*InspectorDevice*) without access to the *ETicketServer*. The attacker has full Dolev-Yao abilities on the connections. That means an attacker can read, send and suppress messages that are exchanged over any connection. This is represented by the stereotype ≪Threat≫ with the properties {*read, send, suppress*}. *InspectorDevice*, *ETicketCard* and *ETicketServer* are secure devices. This means that neither the modeled participants nor the attacker have access to their internal storage.

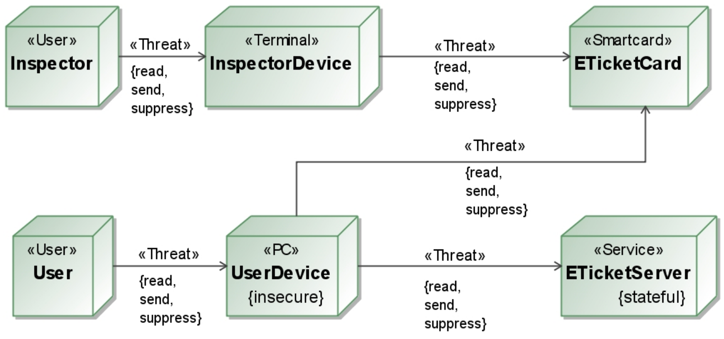

**Fig. 2.** Deployment diagram of the application

The assumptions are that a terminal is a closed and sealed device (e.g., an ATM or cash card reader) and that the attacker has no physical access to the servers running the services. *UserDevice* represents a personal computer, with its user being able to access its internal storage. Additionally, the PC is insecure, which means that an attacker also has access to the device, e.g., through malware.

Three security properties are required to hold for this system. Firstly, only tickets issued by *ETickerServer* are valid and can be "punched". Secondly, a paid ticket can not be lost (i.e., a bought ticket is stored on the server until the card has received the ticket and has sent a delete ticket confirmation to the server), even if the *UserDevice* crashes, the *ETicketCard* is removed from the card reader or because of an attacker. Thirdly, a ticket can not be "punched" twice.

## 4 Translation of a SecureMDD Model into ASLan++ Specification

This section describes the transformation from a SecureMDD model (using deployment diagrams (see Fig. 2), class diagrams and activity diagrams) into ASLan++. The focus is on the correct translation without regard to the execution time of model checking. The transformations are application-independent but are illustrated using the electronic ticketing example.

SecureMDD uses UML that is tailored on security-critical applications. The static view is modeled by class diagrams and a deployment diagram, and the dynamic view is modeled by activity diagrams. The class diagrams define the participants, their attributes and the messages classes. The deployment diagram defines the communication structure as well as the attacker abilities. The activity diagrams contain the message exchange as well as the actions that will be executed after receiving a message (e.g., decrypting of messages, comparing of values or debiting a credit card). A platform-independent and domain-specific language called MEL [17] [6] is used. It supports assignments, object creation, local variables, comparisons and predefined operation (e.g., encrypt, sign, hash, generateNonce, etc.).

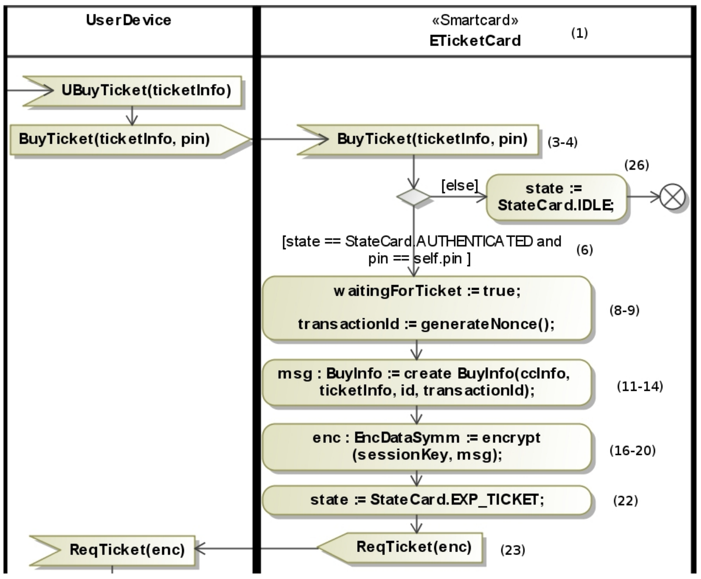
UserDevice
«Smartcard»
ETicketCard
(1)
UBuyTicket(ticketInfo)
BuyTicket(ticketInfo, pin)
BuyTicket(ticketInfo, pin)
(3-4)
(26)
[else]
state := StateCard.IDLE;
[state == StateCard.AUTHENTICATED and pin == self.pin ]
(6)
waitingForTicket := true;
transactionId := generateNonce();
(8-9)
msg : BuyInfo := create BuyInfo(ccInfo, ticketInfo, id, transactionId);
(11-14)
enc : EncDataSymm := encrypt (sessionKey, msg);
(16-20)
state := StateCard.EXP_TICKET;
(22)
ReqTicket(enc)
ReqTicket(enc)
(23)

**Fig. 3.** A part of the activity diagram used by SecureMDD to describe the behavior of ETicketCard

Fig. 3 depicts a snippet of an activity diagram that describes a part of *ETicketCard's* behavior. It describes the first step of the protocol to buy a ticket (which is initiated by the user) after a successful authentication between *ETicketCard* and *ETickerServer*. The activity diagram shows two partitions. The left one describes the participant *UserDevice* and the right one (1) represents the participant *ETicketCard*. *UserDevice* sends the message *BuyTicket* with *ticketInfo* and *pin* to an *ETicketCard*. After *ETicketCard* receives the message *BuyTicket* (3-4) it checks its state to ensure that an authentication has previously occurred and compares the received *pin* with the correct one stored in the *pin* attribute of the card(6). If the check fails, the state will be reset to *StateCard.IDLE* and the protocol step is finished (26). *StateCard* is an enumeration defined in a class diagram that can be *IDLE*, *AUTHENTICATED*, *EXP_TICKET*, etc. After a successful check *waitingForTicket* is set to true and a new *transactionId* (of type *Nonce*) is generated (8-9). *waitingForTicket*, *transactionId*, *state* and *pin* are class attributes of *ETicketCard* defined in the class diagram. After creating the local variable *msg* of type *BuyInfo* (11-14) this message is encrypted with a session key (16-20) which was exchanged in a previous authentication protocol. At the end, the state is set to *EXP_Ticket* (22) and the message *ReqTicket* with the encrypted content is sent to *UserDevice* (23).

ASLan++[20] is a textual language used by AVANTSSAR for specifying security-critical applications. The major building blocks are *entities*. They declare *types*, *symbols*, a *body* and other items. Each participant modeled as class in SecureMDD is translated into an ASLan++ entity. These entities are called agents. The local variables as well as class attributes are translated to *symbols* inside the resulting *entities*. The types of the attributes that are described by classes or primitives in SecureMDD are defined as *types* in ASLan++. A *body* section contains the dynamic part of the application as well as the communication structure and the attacker abilities on the individual communication channels.

```
1   entity ETicketCard (...){
2     ...
3     on(UserDevice -> Actor :
4     buyTicket.(ticketInfo.(...).?M_buyTicket_pin)) :
5     {
6       if ( State = statecard_authenticated & M_buyTicket_pin = Pin )
7       {
8         WaitingForTicket := t;
9         TransactionId := fresh();
10
11        L_buyTicket_msg_ccInfo ...
12        L_buyTicket_msg_ticketInfo ...
13        L_buyTicket_msg_cardID_id := Id_id;
14        L_buyTicket_msg_transactionId := TransactionId;
15
16        L_buyTicket_enc := scrypt(SessionKey, buyInfo.(
17          cCInfo.(...).ticketInfo.(...).
18          iD.(L_buyTicket_msg_cardID_id).
19          L_buyTicket_msg_transactionId
20        ));
21
22        State := statecard_exp_ticket;
23        Actor -> UserDevice : reqTicket.(L_buyTicket_enc);
24      }
25      else{
26        State := statecard_idle;
27  } } }
```

**Listing 1.** A part of the ASLan++ specification that describes the behavior of ETicketCard

List. 1 shows the ASLan++ representation of the protocol step depicted in Fig. 3. The participant *ETicketCard* is described by an *entity* (1). The ability to receive the message *BuyTicket* from *UserDevice* is specified by the *on(...)* statement (3-4). It describes a conditional branch without *else* case. If the condition *UserDevice* $\rightarrow$ *Actor : buyTicket...* (3-4) inside the *on* statement is true then the actor (*ETicketCard*) receives the message *buyTicket...* from the *UserDevice*. $\rightarrow$ describes that the attacker can read, send and suppress messages on this connection. In ASLan++ it is not possible to define complex data types (e.g., a class that contains some attributes). Hence, for sending or receiving the message *BuyTicket*, only the attributes and the type information specified by constants to avoid type confusion are used. After *ETicketCard* receives the message *BuyTicket* (3-4) it checks its *State* and the received *pin* (6). If the check fails, the state will be set to *statecard_idle* (26). If the check was successful *WaitingForTicket* is set to true (8). After that *TransactionId* is set to a "fresh" value (this value is new and unique) (9). In SecureMDD we used for this the predefined function *generateNonce*. The assignment of complex data types has to be customized. Therefore, an object assignment has to be fragmented in several assignments for existing data types. The instantiation of complex data types, in this case *BuyInfo*, needs several statements because each attribute has to be assigned separately (11-14). After the instantiation *msg* is encrypted symmetrically with *SessionKey* (16-20). Finally, the state is set to *statecard_exp_ticket* (22) and the encrypted message is sent to *UserDevice* (23). The complete SecureMDD model and ASLan++ specification of eTicket is available on our website[1].

In the following some interesting aspects of the transformation of the UML application model into ASLan++ are described. In SecureMDD a participant is able to receive any known message type while he is waiting for a message. In our ASLan++ specification this behavior is formalized with an infinite loop and a non-deterministic choice of receiving messages inside the body of each agent (see List. 2). A non-deterministic choice is defined by the *select* statement that contains conditional branches without an *else* case (*on(...)*). *A* $\rightarrow$ *Actor : M1* means that the actor receives the message *M1* from the agent *A*.

```
while(true)
{
  select
  {
    on(A->Actor : M1):{...}
    on(A->Actor : M2):{...}
    ...
    on(B->Actor : Mn):{...}
} }
```

**Listing 2.** Definition of agent behavior

If/else statements, equality checks, logical expressions (e.g., AND, OR) as well as encrypt and sign operations can be directly mapped to existing equivalent ASLan++ language constructs. SecureMDD supports lists and key-value containers, whereas ASLan++ only supports sets. For example, lists in SecureMDD contain following operations:

– *add(Element e) : void* Adds the element *e* to the end of the list.
– *remove(Element e) : void* Deletes the element *e* from the list.

Therefore, SecureMDD lists are emulated using ASLan++ sets. Each ASLan++ list element is defined as a tuple consisting of the original list element and a unique index, while the set maintains an index counter. *Add* and *remove* have been translated in a simple and efficient way into ASLan++. The add operation increases this counter and inserts a new tuple into the set consisting of the original element and the new counter value as index. This allows us to insert duplicate elements into a set. The *remove* operation for a SecureMDD list is translated to an existing remove operation on ASLan++ sets. For this operation the index can be ignored.

Arithmetic operations like addition and multiplication are supported in SecureMDD but not in ASLan++. For some examples like eTicket they are not necessary but we have also examples modeled with SecureMDD that use arithmetic operations. Currently, SecureMDD applications that use arithmetic will not be translated into ASLan++. This is a severe limitation of AVANTSSAR and the used model checkers. However, we are not aware of a model checker that supports arithmetic and is tailored on security applications.

[1] `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/`

SecureMDD and ASLan++ support a full Dolev-Yao attacker [10] who is able to read, send, and suppress messages on the fly. Secure devices in SecureMDD like an *InspectorDevice* are translated to ASLan++ as an honest agent and an insecure device like *UserDevice* becomes a dishonest agent. Dishonest means that an attacker can play the role of such an agent.

SecureMDD uses invariants to specify security properties. The invariants can be translated into ASLan++ *goals*. The validated goals for our eTicket example are described in section 6.

## 5 Automatic Abstractions

SecureMDD models like eTicket are too large for the model checkers integrated in AVANTSSAR. The eTicket case study has 52 different protocol steps (where one protocol step can lead to several transition rules in ASLan). More precisely, an eTicket ASLan++ specification that is translated one-to-one without any abstractions leads to 162 transition rules. This is a lot compared to the average 20 transition rules that are considered by the AVANTSSAR examples.

The execution time of model checking depends on several factors. One of those factors is the number of agents as well as the complexity of their behavior. The behavior of system participants can be specified in ASLan++ primarily using guards and statements (e.g., the sending or receiving of a message, a conditional branch or an assignment). ASLan++ is translated into ASLan by AVANTSSAR. ASLan uses transition rules with pre- and postconditions to define the participants' behavior. A transition rule transfers a state machine from one state to another if the precondition is true. The number of such transition rules as well as their interconnectivity is also crucial for the complexity of the system specification. The attacker capabilities are just as important. If an attacker is able to generate and send messages to a system agent, it is checked at every transition whether the attacker is capable of generating a message that could lead to a security property being violated. If loops or more than one session are specified, the complexity of the specification depends on the number of transition rule executions and on the maximum trace length. A trace contains a list of transition rules and represents one possible execution order of the specified system.

Since model checkers need a lot of computing resources which are not always available, the application models need to be abstracted. This is usually done manually [2] and only by security experts. A manually abstracted version of the full eTicket example leads to 55 transition rules. In the following some automatic abstractions are mentioned with that the generated specification has only 65 transition rules against the 162 without any abstractions. This is very close to the manually abstracted version and can not be significantly reduced further without omitting some of the applications functionality.

1. **Removing participants that are not security-critical**
   An honest agent such as a terminal in SecureMDD which only forwards messages between other agents can be omitted in the abstract specification. In order for all communication options to be preserved, new communication paths

have to be created. The attacker abilities on a new communication path is the most permissive combination of his abilities on the paths that are replaced by the new, direct communication path. If, however, such an agent that only forwards messages is dishonest (e.g., an "insecure" PC), the attacker abilities on a new communication path are "read, send, suppress". In the eTicket case study (see Fig. 2) the agent *UserDevice* can be removed. A new communication path is created between *User* and *ETicketServer*, between *User* and *ETicketCard* as well as between *ETicketCard* and *ETicketServer*. The attacker capabilities on the new created communication paths are "read, send, suppress". Using this abstraction, the resulting specification for the eTicket case study has 10% less transition rules.

2. **Deleting unused class attributes**
Some class attributes are necessary for the implemented application but are never used in the protocols. For example the attributes of *TicketInfo* (departure, destination, expiration, ...) are relevant for the real users and inspectors but not for the formal specification. Hence, if the attributes of a class are never used by the protocols (especially no constructor call of the class) and if the security properties do not refer to those class attributes, they don't need to be specified in ASLan++. Consequently, such a SecureMDD class with unused attributes is translated to an ASLan++ type. Model checking a simplified eTicket version with the model checker Cl-AtSe using this abstraction, is five times faster than a version of eTicket that does not use this abstraction.
3. **Reducing conditional branches**
In SecureMDD, the section between receiving a message and sending the next is called a protocol step and is considered to be atomic. If all steps in ASLan++ could be specified to be atomic each step could be translated into a single transition rule. However, in ASLan++ steps that contain branches are translated into several transition rules in ASLan, which increases the complexity of the specification. It is possible to merge several nested conditional branches into one by combining the branch conditions with a logical AND if only the innermost branch contains other statements. By doing so one can eliminate transition rules from the resulting ASLan specification. This abstraction can be used quite often with SecureMDD models and is executed automatically on the UML model.

   In SecureMDD, any system participant can receive any modeled message after having executed a step. For the state machine in ASLan this results in a large number of possible transition combinations. In most protocols, however, the message order is fixed by using explicit state variables that are usually checked in a branch condition immediately after receiving a message (see Fig. 3 (6)). But as already mentioned, conditional branches are to be avoided in ASLan++. Because for receiving a message a conditional branch without *else* case (on(...) see List. 1 (3-4)) is used, the mentioned abstraction would not be applicable. But because usually, in SecureMDD, all *else* cases from the state checks are the same (e.g. only set the state to idle), it is possible to get the required behavior by adding *on(true) state:= idle;* to the ASLan++ *select* statement inside the infinite loop of an entity depicted in List. 2. This is done automatically if the assumptions hold. This abstraction reduces the number of transition rules of the eTicket case study by 50% compared to a version of eTicket that does not use this abstraction.

4. **Assuming a fixed message order**
   Another abstraction is to specify the message receive order statically in ASLan++. This means that the dynamic state check while receiving a message has to be translated into cascading receive blocks in the ASLan++ specification where the inner one can only receive a message if the outer one has received and executed a message. To guarantee that such abstraction does not lead to false positives the dynamic state checks remains. The static receive order is implemented very efficiently by the AVANTSSAR tools and leads to a major speed up that makes it possible for the first time to find security flaws in the eTicket case study.

The aforementioned abstractions are done automatically during the transformations and leads to a significant reduction of the system complexity and make model checking of medium-sized systems like eTicket with ASLan++ and a model-driven approach feasible in the first place. However, the execution time of model checking rises exponential with the number of transition rules. Therefore, larger systems like an electronic health card [16] which has 105 different protocol steps (translated and abstracted to approx. 130 transition rules) are too big for model checking application-specific properties for the whole application.

## 6 Security Flaws

AVANTSSAR is a project about "Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures". It integrates three model checkers (Cl-AtSe, OFMC and SATMC) which use the same input language called ASLan++. But not all model checkers support its full syntax. Because only Cl-AtSe[19] covers all needed syntax elements and because speed tests illustrate that all three model checkers are comparably fast [19] we decided to use Cl-AtSe for our tests. Cl-AtSe is a "Constraint Logic based Attack Searcher" for security protocols. To find attacks it uses rewriting and constraint solving techniques as well as different kinds of backward strategies. Cl-AtSe supports a *split* function to split a specification into subtasks that can be executed in parallel. The tests were performed on a 3GHz quad core computer. Without using the split function, the CPU load was constantly at 13%, with the split function we were able to use the full capacity.

For the eTicket example we have defined three application-specific security properties in ASLan++. They are used to test which kind of protocol flaws can be found, which assumptions are necessary as well as how long it takes to find those flaws.

1. **Only tickets that were issued by the eTicket server can be "punched"**
   To ensure this application-specific security property it is also necessary that only tickets that are stored on a valid card can be "punched". Hence, before an inspector "punches" a ticket, the card has to authenticate itself with the

inspector device. This is done using certificates. Then, it is ensured that the incoming messages were sent by the authenticated participant. This is done using nonces. The inspector device has to send a nonce encrypted with the certified card public key to the card and the card has to answer with the received nonce encrypted with the public key of the inspector device. If such a nonce is not used, the attacker can inject an answer message and a ticket that is not stored on a card will be "punched". That would lead to the fact that the mentioned security property does not hold. For testing the model checker we have removed the nonce. This security flaw has been detected with the abstracted version in a few seconds. The split function was not used and the assumptions were that each transition rule can be executed only once in a trace.

2. **A paid ticket cannot be lost (i.e., a bought ticket is stored on the server until it has been received by the card)**
Before a ticket can be bought, an authentication has to take place and a valid PIN has to be provided. If the user buys a ticket but does not receive it because the attacker has suppressed the message that contains the ticket, then the ticket is paid for but not stored on the buyer's card. Hence, a recovery protocol is used to be able to receive the last paid ticket until it can be stored successfully on the card. Therefore, a previously processed authentication and a boolean flag *waitingForTicket* set to true are necessary. *WaitingForTicket* is set to true before the *ReqTicket* message is sent out (see Fig. 3) and it is set to false after the ticket is received. However, after a ticket is bought but has not been received yet, it is possible that the card owner buys a new one. In this case, the first bought ticket which is stored on the server will be overwritten by the new one and the old ticket is lost. But that is against the security property. To find the flaw it is necessary that a ticket can be bought two times. Hence, for Cl-AtSe it is necessary that each transition rule can occur in a trace at least two times. This value has to be set manually and leads to a higher complexity and a higher execution time. With the abstracted version and the split function that allows a full CPU load the flaw could not be found even after a week. To ensure that this attack exists in the ASLan++ specification, we predetermined the attack trace. Then the attack was found. Another way to find the attack but without giving the full attack trace, is to omit protocol steps that are not necessary for the security property. This abstraction needs expert knowhow and can cause that some attacks can not be found. But it also reduces the complexity and raises the chance to find an attack. In this way we delete the inspector, the inspector device, all protocol steps that receive messages from the inspector device as well as the show ticket and delete ticket functionality. Then we were able to find the attack with the split function in 30 minutes.
3. **A ticket cannot be "punched" multiple times**
After a long time of analyzing the eTicket case study we have manually found a security flaw (security property is violated) that was actually hard to find and can only occur if almost all protocol steps are considered and the handshake is executed three times. Taking this knowledge into consideration we have tried to find the attack using model checking. Despite the abstraction, elimination of all not used protocol steps and the *split* function that allows a full CPU

load the flaw could not be found even after a week. For the attack a handshake between card and server has to be processed and a ticket has to be bought, stored on the card and the delete ticket confirmation that should be sent to the server has to be suppressed. Then the ticket has to be stamped by an inspector device. For that the public keys have to be exchanged between card and inspector device and their certificates has to be verified. Additionally, a ticket has to be chosen and then stamped. After that a new Ticket has to be bought to set the *WaitingForTicket* flag to true. This means that a new handshake has to be processed but this time the message to buy the ticket has to be suppressed. After that the first bought ticket that is still stored on the server can be recovered by processing a handshake and the recovery. That replaces the stamped ticket with the same ticket but it is not stamped. After that the ticket can be stamped a second time. This attack requires 73 steps, which is a lot.

## 7 Comparison: Model Checking vs Interactive Verification

Existing ASLan++ specifications that consider security applications can often be checked within a few minutes. But to achieve this in the first place, the real applications are abstracted manually. In case of a real application not the whole system is considered but only a manually chosen part. For example, for our eTicket case study that has 52 different protocol steps (whereby one protocol step can lead to several transition rules in ASLan) the ASLan++ specification also considers that a user is able to view his purchased tickets. Because the exchanged messages to view purchased tickets are not relevant for the considered properties, a manual abstracted specification would omit those messages. But this has to be done by an expert because it is non-trivial which parts of the application can be omitted. Additionally, the assumptions (e.g., only one or maybe two protocol runs are considered) are too restrictive.

In contrast to classic model checking, with interactive verification it is possible to verify security properties of an application that uses cryptographic protocols for an arbitrary number of agents and protocol runs. The formal model for interactive verification with KIV is based on algebraic specifications and Abstract State Machines (ASMs) [7]. It specifies a world in which agents exchange messages according to the protocols, and an attacker tries to break the security. The interactive verification of the mentioned security properties for eTicket (see 6) by an expert requires approx. three weeks to verify the properties for all possible protocol runs and for an arbitrary but finite number of agents. For model checking of medium-sized applications the application model has to be abstracted to reduce the search space. If these abstractions are done manually they are time-consuming, error-prone and require expert know-how. Interactive verification is also time-consuming and requires expert know-how but with "simplifier rules" that are generated automatically by KIV some verification steps can be automated. Arithmetic is also a difficult task for most model checkers. For example, ASLan++ only provides a *successor* as well as an *equal* function. Other model checkers like NuSMV[8] provide basic arithmetic like addition, subtraction, multiplication, comparison etc. But in those model checkers, a fixed range of values has to be specified. Model checkers that are able to verify properties for an arbitrary number of agents and protocol runs are currently work in progress. OFMC[15] is a model checker that implements a fixpoint module which uses an over-approximation of the search space to allow a verification for an unbounded number of transitions. To cope with the infinite set of traces, OFMC uses some abstractions that are not safe, which can lead to false attacks. It is also possible that the abstractions run into non-termination and nothing can be proved with the fixpoint module. Although OFMC is integrated in AVANTSSAR, it only supports a subset of full ASLan, which is not enough to specify our eTicket example. Hence, we were not able to test the fixpoint module of OFMC.

## 8 Related Work

Model checking is still used with a manual abstraction of an existing application [2]. Such abstractions can be done systematically and fault-preserving [13] but they are still time-consuming and need expert know-how. In our model-driven approach, runnable code as well as formal specifications are generated automatically from a model. To ensure that the generated code and the formal specification fit together, the resulting formal specification must not be adjusted manually. Hence, abstractions have to be done automatically. There are model-driven approaches that already generate formal specifications automatically for model checking.

The approach developed by Deubler et al. [9] considers the model-driven development of secure service-based systems and uses SMV to automatically check role-based access control policies. SecureUML [5] is also a model-driven approach that defines application behavior with UML. It uses Maude and Spin for model checking and Isabelle for interactive verification. However, it is tailored to role-based access control applications. Arsac et al. [3] use BPMN for modeling security-critical business processes and AVANTSSAR for model checking of security properties like role-based access control.

But all these approaches are not able to check or verify application-specific security properties. That is because they focus on the interaction between agents and do not model the full application behavior. Hence, by using those approaches one is only able to validate high level security properties like secrecy, integrity, authentication, authorization and role-based access control. But many system requirements are application-specific like "only tickets that were issued by a valid ticket server can be punched". Additionally, a few of them also generate code from the application model automatically but this code has to be extended by logic that is usually also security-critical. By combining our approach called SecureMDD with AVANTSSAR it is possible to model check the full system behavior for application-specific properties automatically.

UMLsec [14] describes another model-driven approach that uses sequence diagrams or state charts to model the system behavior and integrates model checkers (e.g., Spin) and automated theorem provers (e.g., SPASS) to check security properties like secrecy and integrity but are not restricted to those. UMLsec does not integrate AVANTSSAR and has not demonstrated the limitations of model checking in a model-driven approach.

## 9 Conclusion

Model checking can be used to find protocol flaws in security-critical systems. In this paper we successfully integrate model checking using AVANTSSAR into SecureMDD, a model-driven approach for security-critical applications. We have written model-transformations to automatically transform a SecureMDD model into ASLan++. The transformation automatically does abstractions on the UML input model to reduce the complexity of the resulting specification. The generated specification is very close to a manually written one that specifies the full application functionality. We have defined application-specific security properties for an eTicket application and have shown that within one week and a 3GHz quad core computer some attacks could be found but not all. The properties are checked for the whole system that has 52 protocol steps and represents a real and medium-sized application. Because the system complexity rises exponential with each protocol step, even larger applications than our eTicket case study are too big for model checking of application-specific properties for the whole system without omitting functionality. Finally, we have shown the difference between interactive verification and model checking. Future work could be to annotate protocol parts in the SecureMDD model that should be omitted in the resulting ASLan++ specification to reduce the system complexity.

We come to the conclusion that model checking enriches a model-driven approach for security-critical applications greatly. Such an approach with automatic generation and abstraction of formal specifications avoids expert know-how about formal methods as needed for interactive verification. But for large systems the complexity of model checking the whole system is too big. Furthermore, classic model checking is suitable to find application-specific security flaws but for verification (arbitrary number of agents and protocol runs) of large systems, interactive theorem proving is needed.

## References

1. Armando, A., et al.: The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 267–282. Springer, Heidelberg (2012)
2. Armando, A., Carbone, R., Compagna, L., Cuellar, J., Tobarra, L.: Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In: Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, pp. 1–10. ACM (2008)
3. Arsac, W., Compagna, L., Pellegrino, G., Ponta, S.E.: Security validation of business processes via model-checking. In: Erlingsson, Ú., Wieringa, R., Zannone, N. (eds.) ESSoS 2011. LNCS, vol. 6542, pp. 29–42. Springer, Heidelberg (2011)
4. Balser, M., Reif, W., Schellhorn, G., Stenzel, K., Thums, A.: Formal system development with KIV. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783, pp. 363–366. Springer, Heidelberg (2000)

5. Basin, D., Doser, J., Lodderstedt, T.: Model Driven Security: From UML Models to Access Control Infrastructures. ACM Transactions on Software Engineering and Methodology, 39–91 (2006)
6. Borek, M., Moebius, N., Stenzel, K., Reif, W.: Model-driven development of secure service applications. In: 2012 35th Annual IEEE Software Engineering Workshop (SEW), pp. 62–71. IEEE (2012)
7. Börger, E., Stärk, R.F.: Abstract State Machines—A Method for High-Level System Design and Analysis. Springer (2003)
8. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
9. Deubler, M., Grünbauer, J., Jürjens, J., Wimmel, G.: Sound development of secure service-based systems. In: Proceedings of the 2nd International Conference on Service Oriented Computing, pp. 115–124. ACM (2004)
10. Dolev, D., Yao, A.C.: On the Security of Public Key Protocols. In: Proc. 22nd IEEE Symposium on Foundations of Computer Science. IEEE (1981)
11. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
12. Holzmann, G.: The model checker spin. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
13. Hui, M.L., Lowe, G.: Fault-preserving simplifying transformations for security protocols. Journal of Computer Security 9(1), 3–46 (2001)
14. Jürjens, J.: Model-based security engineering with UML. In: Aldini, A., Gorrieri, R., Martinelli, F. (eds.) FOSAD 2004/2005. LNCS, vol. 3655, pp. 42–77. Springer, Heidelberg (2005)
15. Mödersheim, S., Viganò, L.: The open-source fixed-point model checker for symbolic analysis of security protocols. In: Foundations of Security Analysis and Design V. LNCS, vol. 5705, pp. 166–194. Springer, Heidelberg (2009)
16. Moebius, N., Stenzel, K., Borek, M., Reif, W.: Incremental development of large, secure smart card applications. In: Proceedings of the Workshop on Model-Driven Security. ACM (2012)
17. Moebius, N., Stenzel, K., Reif, W.: Modeling Security-Critical Applications with UML in the SecureMDD Approach. International Journal on Advances in Software 1(1) (2008)
18. Moebius, N., Stenzel, K., Reif, W.: Formal verification of application-specific security properties in a model-driven approach. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 166–181. Springer, Heidelberg (2010)
19. Turuani, M.: The CL-atse protocol analyser. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 277–286. Springer, Heidelberg (2006)
20. von Oheimb, D., Mödersheim, S.: ASLan++ — A formal security specification language for distributed systems. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 1–22. Springer, Heidelberg (2011)
21. ZDNet. Attackers hit google single sign-on password system (2010)
22. ZDNet. Chip and pin is broken, say researchers (2010)