Incremental Development of large, secure Smart Card Applications

Nina Moebius, Kurt Stenzel, Marian Borek, Wolfgang Reif
Institute for Software and Systems Engineering
University of Augsburg
86135 Augsburg, Germany
E-Mail: {moebius, stenzel, borek, reif}@informatik.uni-augsburg.de

ABSTRACT
SecureMDD is a model-driven approach to develop security-critical applications. The focus lies on the development of smart card and service applications. Those are inherently security-critical and are based on cryptographic protocols. These protocols are difficult to design and error-prone. To guarantee the security of an application, formal verification is an inherent part of our software engineering approach. In this paper we illustrate that the SecureMDD approach is applicable for the development of large and complex applications as well. To handle the size and complexity, an incremental development method is suggested. This is illustrated with the German electronic health card application as case study.

1. INTRODUCTION
The development of cryptographic protocols is very difficult and error-prone [1, 16]. This is true even for short protocols with only few communication steps [11, 12]. The reason is the presence of an attacker who is able to read messages exchanged between two participants and interferes the communication by suppressing and sending messages. Often flaws in an application resp. in the underlying protocols are detected only after years of usage. One well-known example is EMV, a standard for payments with cash and credit cards that use a chip. In 2010 researchers found an attack on the EMV protocols that allows to make payments without entering the PIN number of the card [15].

To be able to develop secure applications based on cryptographic protocols, it is essential to integrate formal verification into the development process. Moreover, the security aspects of the application under development have to be considered in all phases of the development process. The SecureMDD approach realizes both aspects and is tailored to develop security-critical smart card and service applications. Moreover, executable code is generated automatically. The focus lies on E-Commerce applications such as ticket and payment systems as well as ‘digital identity’ applications such as electronic identity cards, passports, or electronic health cards. In the past, we evaluated and consolidated our approach using several applications that are relatively small in their size and contain only few protocols. For this kind of applications the approach works very well. Thus, the next step was to develop a large application with many complex protocols. Those applications cannot be developed within one step, i.e. an iterative process has to be defined. In our approach we use interactive verification. Proofs that have been done in an earlier iteration and even security properties may become invalid if functionality is added to an application later on. Thus, one question is how to handle the (interactive) verification in an incremental process. Another question is how to combine the model-driven approach with an incremental development method. In this paper we present this novel approach for incremental development with an electronic health card case study.

The paper is structured as follows: Section 2 gives an overview of the model-driven SecureMDD approach. Section 3 introduces our case study, the German electronic health card. Section 4 presents a method to develop large and complex applications with our approach. In Section 5 we describe the incremental verification of security properties. Section 6 relates this work to other approaches and Section 7 concludes.

2. APPROACH
The SecureMDD approach supports the development of applications that use web services, smart cards and terminals. Terminals are devices that consist of at least one smart card reader and can communicate with a smart card. Moreover, a terminal usually has a user interface to communicate with the user of the application and can also call service operations. For example, a terminal could be a Home-PC or an automat.

In this Section we shortly summarize the approach. It has been described in more detail in [13]. Fig. 1 contains an overview.

The development of an application starts with the creation of a platform-independent UML model. This is an abstract view of a system, omitting implementation details. To be able to model security-critical applications, UML was tailored to this domain by defining a UML profile. To support the modeling of the dynamic part of an application, i.e. the communication as well as the processing of messages, we defined a domain-specific language called Model Extension...
Language (MEL) which is used in UML activity diagrams. With this language it is possible to make assignments to the attributes of component classes, to create objects or to call predefined cryptographic operations. The platform-independent UML model of an application consists of all information that are needed to generate executable code as well as a formal model of the whole application automatically.

The platform-independent model is transformed into code (via platform-specific models) using model-to-model and model-to-text transformations. For terminals and web services we generate Java code. Smart cards are programmed with Java Card ([8]), which is a subset of Java and is tailored for the use on resource-constraint devices. Thus, for smart cards we generate Java Card code.

The modeled application and especially the cryptographic protocols may contain functional and security errors. We integrated model-based testing into the approach that allows to design a test case with UML and generate test cases automatically from this model (see [10]). Then, these test cases can be executed on the generated code.

To guarantee the security of an application, the approach integrates the formal verification of application-specific security properties (see [14]). Application-specific properties are e.g. that an electronic prescription that is stored on an electronic health card cannot be filled twice in a pharmacy. Other properties are that only genuine prescriptions can be filled, i.e. only a doctor is able to issue a prescription, and that the attacker does not get to know any prescriptions. In our opinion, for many applications application-specific properties give better guarantees to the security of an application than standard properties like secrecy, integrity or authenticity. However, standard properties are often prerequisites for proving application-specific properties and thus, have to be verified as well. Our approach generates a formal specification based on algebraic specifications and abstract state machines (ASM) [4] for the modeled application. The generated formal model is loaded into the theorem prover KIV [2] and used for interactive verification of application-specific security properties.

The security properties that are proved to hold on the formal model also have to hold on code level. To guarantee this, the generated code has to be a refinement of the generated formal model. A solution that proves the refinement for any application that is developed with the SecureMDD approach is work in progress. A first result is the definition of a calculus for QVT (the language used to implement the model-to-model transformations) in KIV. This calculus can be used to prove the correctness of QVT transformations (see [18]).

3. CASE STUDY

The German electronic health card is a large IT project. The application was planned to be introduced in Germany in 2006. The introduction was delayed because of reasons like open questions concerning the secrecy of patient data, total costs as well as missing encouragement of the doctors. Until the end of year 2012 fifty percent of the insured persons, i.e. 35 million people, must have an electronic health card. However, this first version of the card stores the same personal data that is also available on the current card. More functionality is not yet supported but will be added incrementally later.

We considered a part of the electronic health card application as case study and developed it incrementally. Since most of the protocols that were developed by gematik (the association that is responsible for the application) are obsolete at the moment, we designed our own protocols. Those protocols meet the security features defined by gematik. These are, for example, the use of certificates for the health cards and identity cards of the doctors and pharmacists or the entering of a PIN number by a patient resp. doctor or pharmacist. Another given feature is that electronic prescriptions have to be signed by the identity card of the doctor who creates the prescription. Moreover, the patient has to use a Kiosk to view the data stored on his health card instead of his own PC.

Deployment diagrams are used in SecureMDD to define the participants, the communication infrastructure as well as the attacker abilities of an application. The deployment diagram of the health card application is shown in Fig. 2.

The application is used by a Doctor who has a PC in his
office (PCDoctor). This PC is connected to a HealthInsuranceCompany that offers functionality to the doctors using (web) services. Additionally, each doctor has an identity card (a smart card called HeilberufsausweisDoctor). The doctor’s PC is connected to two smart card readers, one for the doctor’s Heilberufsausweis and one for the HealthCard. Each patient owns an electronic HealthCard. Moreover, the Pharmacists are using the application. Every Pharmacist owns an identity card called HeilberufsausweisPharmacist. Every pharmacy has a PC, called PCPharmacy, which is used for communication with electronic health cards and the identity cards of the Pharmacists. Finally, a terminal called Kiosk is part of the application. A Kiosk is a machine standing in a pharmacy that can be used by the patients to view the personal data which is stored on their health card. A user (Doctor, Patient and Pharmacist) cannot communicate with a smart card directly, the communication is done via a terminal (PCDoctor, PCPharmacy, Kiosk). We assume an attacker that is able to read messages sent over a communication channel, send messages over a channel as well as to suppress messages. In contrast to a Dolev-Yao attacker [5] who has full access to all channels, we are able to define the attacker abilities for each channel separately by annotating the stereotype ≪Threat≫ with tags read, send and suppress. For the electronic health card application we assume an attacker who has full access to all channels except the user channels. The user channels are used for interaction with the users, e.g. a graphical user interface which is used by a doctor to enter the data of a prescription.

Figure 3 shows a part of the platform-independent class diagram.

Figure 3: Part of the platform-independent class diagram

The class HealthCard represents the health card of a patient. A HealthCard stores up to twenty prescriptions that were issued for the owner of the HealthCard. A Prescription consists of three parts. First, the prescriptionData, i.e. the medicinal product, the personal data of the patient (issuedfor), the date of issue and a nonce that is unique for every prescription. Secondly, a digital signature of the PrescriptionData. This is modeled with an association sig that is annotated with stereotype ≪signed≫ (the stereotype means that in sig the signature of a PrescriptionData object is stored). The signature is created on the HeilberufsausweisDoctor during the creation of the prescription and enables to prove that a prescription is genuine. Thirdly, a prescription contains the certificate of the doctor’s Heilberufsausweis. The certificate contains a public key that can be used to verify the signature.

The dynamic part of the application, i.e. the cryptographic protocols, are modeled with activity diagrams. In this case study the protocols can be divided into basic protocols and ‘functional’ protocols. Basic protocols are, for example, authentication of components or entering of PIN numbers. ‘Functional’ protocols realize the functionality of the application, e.g. creating a prescription. The basic protocols have to be executed previous to the functional protocols. In the first iteration we realized the creation (on the doctor’s PC and with the doctor’s identity card) and filling of a prescription (in the pharmacy and with the identity card of the pharmacist) as well as all basic protocols that have to be executed previously. The basic protocols were designed such that they can be executed on a doctor’s PC and with a doctor’s identity card as well as with a PC in a pharmacy and the identity card of a pharmacist. The first basic protocol authenticates an identity card (HeilberufsausweisDoctor or HeilberufsausweisPharmacist) and the HealthCard of the patient against each other and against the PC. This is done by publishing the certificates of the cards and proving the identity using challenge/response. Afterwards, the patient and the doctor resp. pharmacist have to enter the PIN numbers of their health card resp. identity card. If the PIN numbers are correct, a session key is generated and exchanged between the PC, the identity card and the health card. From now on, the protocols that contain the functionality can be executed (repeatedly). In the first iteration we realized the creation of an electronic prescription on a doctor’s PC as well as the filling of a prescription that is stored on a health card in a pharmacy. The order of protocol runs is guaranteed by using boolean flags on the cards and terminals that indicate e.g. that the pin number was entered correctly or a session key has been exchanged already.

Figure 4: Activity diagram showing the relation between some of the designed protocols

Figure 4 shows one part of the activity diagram (with expressions of the MEL language) to create a prescription. For each component that takes part in the protocol we model one partition in the diagram. The figure shows a protocol step that is executed on an electronic health card and stores a prescription on the card. Thus, the excerpt of Fig. 4 shows...
the partition of the health card. The health card receives a message called Send\textit{\text{Prescription}}. Then, the card checks if it is in the correct state (i.e. the state to process a Send\textit{\text{Prescription}} message) and the boolean flag auth\textit{HBA} is set to true (1). This flag indicates whether the health card has checked that the identity card of the doctor is genuine. This has been done in a previous protocol. If these checks fail, a protocol (with name \textit{\text{ABORT}}) that resets some values is called and the protocol execution aborts. If the checks succeed, the prescription data that was contained in the message is decrypted using the session key (2). The operation decrypt is a predefined MEL operation. Next, it is checked if the nonce of the prescription data (that was generated by the card in a previous protocol step) is the expected one and the received signature sig of the prescription data is verified (3). If the check is successful, an object of type \textit{\text{Prescription}} is created and added to the list of \textit{\text{prescriptions}} that are stored on the card. Afterwards, the state of the card is set to IDLE and a message is sent back to the doctor’s PC.

All in all, the UML model contains 15 protocols with 105 protocol steps. From the UML model we generated about 86,000 lines of code (36,000 lines of Java Card code for the smart cards, 47,000 lines of Java code for the terminals and services) get additional attributes and states. Since the class diagrams contain many classes and become confusing, we recommend to define one class diagram for all component and data classes and a separate class diagram for the message classes of each protocol. Moreover, each protocol that has been done in a previous protocol. If these checks fail, a protocol (with name \textit{\text{ABORT}}) that resets some values is called and the protocol execution aborts. If the checks succeed, the prescription data that was contained in the message is decrypted using the session key (2). The operation decrypt is a predefined MEL operation. Next, it is checked if the nonce of the prescription data (that was generated by the card in a previous protocol step) is the expected one and the received signature sig of the prescription data is verified (3). If the check is successful, an object of type \textit{\text{Prescription}} is created and added to the list of \textit{\text{prescriptions}} that are stored on the card. Afterwards, the state of the card is set to IDLE and a message is sent back to the doctor’s PC.

1. The part of the application that is developed in the iteration is modeled with UML. To do this, the existing UML model has to be extended. Especially, the existing class diagrams have to be expanded with additional data and message classes. Furthermore, the existing component classes (i.e. classes for smart cards, terminal and services) get additional attributes and states. Since the class diagrams contain many classes and become confusing, we recommend to define one class diagram for all component and data classes and a separate class diagram for the message classes of each protocol. Moreover, each protocol that has been done in a previous protocol. If these checks fail, a protocol (with name \textit{\text{ABORT}}) that resets some values is called and the protocol execution aborts. If the checks succeed, the prescription data that was contained in the message is decrypted using the session key (2). The operation decrypt is a predefined MEL operation. Next, it is checked if the nonce of the prescription data (that was generated by the card in a previous protocol step) is the expected one and the received signature sig of the prescription data is verified (3). If the check is successful, an object of type \textit{\text{Prescription}} is created and added to the list of \textit{\text{prescriptions}} that are stored on the card. Afterwards, the state of the card is set to IDLE and a message is sent back to the doctor’s PC.

2. The UML model is validated. The validation includes consistency as well as syntax checks.

3. Executable code is generated from the UML model using the implemented model transformations.

4. The generated code is tested. For the new functionality added in this iteration, tests have to be modeled with UML. For all tests (old and new ones), test cases are generated and executed. If a test fails, the incorrect protocols are corrected in the UML model and steps 1 - 4 are repeated.

5. The last step of the iteration is the verification of security properties.

Using (interactive) verification within an incremental development process has two potential problems. The first is, that an application may become insecure if additional functionality is added in a later iteration. This is due to the fact that the attacker is able to mix up protocol runs. One example is that a hash value is computed in two different protocol runs (on different inputs) but the input has the same type, e.g. contains a pin number, a nonce and a number. In this case, an attacker could memorize the hash value of the first protocol run and replay this hash value in a run of the second protocol. If the second protocol is added, the application may become insecure.

The second problem is that some properties that hold and were already proven for a formal model may become invalid if additional protocols are added to the UML model. To prove security properties usually auxiliary properties are required. In the electronic health card application, one of these properties was that only doctors and pharmacists are able to see an electronic prescription. This implies that patients who own a health card never see a prescription (they only get the medicine in the pharmacy). In a later iteration the functionality for the Kiosk component was added. With a Kiosk it is possible for a patient to view his personal data. The answer to this problem is to avoid properties that may become invalid in a later iteration and draw the reasoning on properties that will never be restricted after adding further protocols.

The incremental development of the electronic health card case study worked very well. We performed three iterations.

\footnote{http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/}
In each iteration functionality was added but the existing protocols were not changed. In the second iteration we added protocols to create and read out data that is needed in case of emergency (e.g., intolerances or diabetes). Moreover, we realized the basic and functional protocols for the Kiosk. Especially, we support the viewing of emergency data and the prescriptions as well as the deletion of emergency data. In the third iteration the health insurance companies were connected to the application by comparing the personal data of a patient (e.g., name and address) with the data stored at the insurance company and update the data on the card if necessary.

5. INCREMENTAL VERIFICATION

The formal model uses algebraic specifications for data types and the attacker knowledge (similar to [17]) and an Abstract State Machine (ASM) [4] for the protocol steps and attacker actions. A security property is formulated as a Dynamic Logic (DL) [6] formula and has the following structure:

\[
\text{init(...)} \rightarrow \text{[ASM(...)] property(...)}
\]

If the ASM is started in an initial state then after termination the property holds.

The ASM nondeterministically chooses a step to execute or to terminate:

\[
\text{ASM(...)} \{ \text{while (not stop) \{choose action in STEP(action); stop := ?; \}} \}
\]

In this manner the ASM contains all possible traces of things that can happen in the “world”. A proof of a security property always works by using invariants of one STEP as lemmas. However, a security property is usually not an invariant by itself, but only together with additional properties:

\[
\text{property(...) \land inv_1(...) \land inv_2(...) \land \ldots \rightarrow [STEP(action)] property(...)}
\]

The additional properties inv_1, inv_2 and so on must also be proved to be invariants. For large applications like the health card it is essential to prove invariants incrementally.

Fig. 5 shows the specification structure of the invariants on top of the ASM specification (many more specifications below the ASM are omitted). Each Inv-... specification contains one or more invariant(s). They depend on the invariants defined in lower specifications. The proof that a prescription is genuine (in Inv-Prescriptions-genuine) requires a proof that the attacker never gains access to a symmetric session key (in Inv-no-SymmKeys) which in turn requires the proof that the attacker cannot introduce his own key pair into other components (Inv-no-PrivateKeys). A session key is exchanged by encrypting it together with a nonce with the public key of the other component. Therefor, it is important that this key is not an attacker key, and that the session key is protected by the nonce that is a response to a challenge (otherwise the session key could come from the attacker). This additionally means that the attacker must never learn the challenge, and so on.

If new protocols are added to an application the ASM changes because it includes rules for the new protocol steps now. This means that all invariant proofs become invalid and have to be re-proved. In interactive proofs this usually requires significant additional effort. However, the effort can be reduced to almost zero in our incremental approach if the following rules are observed:

1. Do not modify the behavior of existing protocols because this requires substantial changes to proofs, definitions, and properties.

2. Make sure that the invariant proofs run automatically by defining suitable simplification rules. Then the re-proof will run automatically for the old protocol steps.

3. Split a large invariant into many mini parts such that each part states a property for only one or two components. E.g., define a property that the attacker does not learn the session key of the health card, and another that he does not learn the session key of the doctor’s PC, and so on, instead of one big property. Then, if a new component like the Kiosk is added we only need one new mini property, and all proofs for the old properties remain valid. Otherwise we would have to change the big property, and all proofs for this property would be invalidated.

A re-proof runs automatically for the old protocol steps, and sometimes stops for the new steps. Then simplification rules must be added and proved so that the re-proof completes. If the invariant no longer holds then add new mini properties and re-prove automatically. In summary, the effort is almost the same as in the non-incremental case. The overhead of the re-proofs is negligible since they are automatic. Indeed, the incremental approach has the additional benefit that it is easier for the persons doing the proofs to comprehend the smaller set of protocols.

We have verified several security properties for the developed application. One property states that all prescriptions that are filled in any pharmacy were created on a doctor’s PC. Since we assume that the terminals are free of malicious
software and the attacker has no direct access to the terminals, this implies that only genuine prescriptions are filled in a pharmacy. A second property is that a prescriptions can be filled only once. Every prescription contains a nonce, i.e. a random number that is used only once. The property states that the list of all prescriptions that are filled does not contain prescriptions with duplicate nonces. This implies that it is not possible to make a copy of a prescription or transfer a prescription to a second health card and fill it.

The proofs consist of about 60,000 proof steps (e.g. execution of an assignment) and less than 4,000 interactions (e.g. instantiation of a quantifier). In addition to the security properties about 800 auxiliary theorems have been verified and used in the main proofs. During the development we have found two flaws. The first one was a missing nonce in a message to prevent replay attacks. The second one concerned the identity cards of the doctors and pharmacists. To differ between these two cards, a flag is contained in the certificates that are stored onto the cards. In the protocols it has to be checked if the identity card belongs to a doctor. In some protocols this check was missing which allows a pharmacist to create a prescription.

6. RELATED WORK

There are some approaches related to ours but all differ in certain aspects. Most related are UMLSec [9] and SecureUML [3] that both aim to develop security-critical systems with extended UML. SecureUML is tailored to role-based access control applications and supports specific authorization constraints with OCL. UMLSec allows to express security properties like secrecy, integrity and role-based access control with stereotypes. Both do not consider an incremental development method. The Security Development Lifecycle (SDL) [7] is a software development process introduced by Microsoft. Its aim is to extend existing software processes with security practices that are grouped in seven phases. SDL defines security procedures that supplements existing development processes but is not tailored to develop applications based on cryptographic protocols.

7. CONCLUSION

In this paper we introduced our incremental development method for security-critical smart card applications. The model-driven approach generates executable code as well as a formal model which can be used directly as input for the theorem prover KIV. Automation and replay mechanisms allow for an incremental development of the application as most of the effort spent in the verification carries over from one iteration to another. Our approach makes use of tailored UML. We illustrated the usability of our approach by modeling the German electronic health card, which is a real-world application of considerable size.

8. REFERENCES