A Security Formal Verification Method for Protocols Using Cryptographic Contactless Smart Cards

Martin HENZL, Petr HANACEK

Dept. of Intelligent Systems, Faculty of Information Technology, Brno University of Technology, Bozetechova 1/2, 612 66 Brno, Czech Republic

ihenzl@fit.vutbr.cz, hanacek@fit.vutbr.cz

Manuscript received August 28, 2015

Abstract. *We present a method of contactless smart card protocol modeling suitable for finding vulnerabilities using model checking. Smart cards are used in applications that require high level of security, such as payment applications, therefore it should be ensured that the implementation does not contain any vulnerabilities. High level application specifications may lead to different implementations. Protocol that is proved to be secure on high level and that uses secure smart card can be implemented in more than one way; some of these implementations are secure, some of them introduce vulnerabilities to the application. The goal of this paper is to provide a method that can be used to create a model of arbitrary smart card, with focus on contactless smart cards, to create a model of the protocol, and to use model checking to find attacks in this model. AVANTSSAR Platform was used for the formal verification, the models are written in the ASLan++ language. Examples demonstrate the usability of the proposed method.*

Keywords

Security, smart card, model checking, ASLan++, formal verification, protocol

1. Introduction

Smart cards are usually used for storing some value or to provide means for user authentication, so the smart card applications usually require high level of security. Applications that use contactless smart cards are mainly payment systems, electronic ticketing, electronic vouchers, access control systems, loyalty programs, etc. In this paper we show examples of payment protocols with pre-paid credit on the card. For purposes of this paper the word protocol means the order and content of commands sent by the terminal and responses from the card and the use of cryptographic functions that will together perform the desired task in a secure way so that confidentiality, integrity, and authenticity of protected data will be ensured.

Contactless smart cards are usually simpler than smart cards with contact interface and provide limited functionality. They usually provide authentication based on symmetric keys, multiple applications and file system with access permissions. Access control is based on keys that are used for authentication, data may be encrypted using some symmetric cipher. One of the most popular and widespread contactless smart cards that uses this scheme is Mifare DESFire [1], which will be modeled in our first example to demonstrate our method. Some smart cards have more sophisticated operating system and can execute applications on their chip, such as Java Cards [2] or Basic Cards [3]. Their application logic can be modeled as well, however, in this paper we focus mainly on simple smart cards with pre-defined set of commands that cannot execute other applications on their chip. We have found out that some features of the Mifare DESFire MF3ICD40 are very dangerous and it may be very difficult to implement protocol using this card in a secure way. Although these features are not considered vulnerabilities of this smart card, they might help to introduce vulnerabilities into the implementation. The second example shows an improved Mifare DESFire. It is a fictional smart card that does not have these dangerous features.

When designing and verifying security protocols using informal techniques, some security errors may remain undetected. Formal verification methods provide a systematic way of finding protocol flaws. The protocol is specified in a formal way and the correctness of security properties is proved or disproved using formal methods and mathematics.

Many tools for verification of protocols are available. Although general purpose formal verification tools can be used to verify security protocols, such as Murphi, Spin, Isabelle, or UPPAAL, it is better and more intuitive to use one of the tools designed specifically for verification of security protocols, such as Casper [4] - a program that will take a description of a security protocol in a simple, abstract language, and produce a CSP (Communication Sequential Process) description of the same protocol, suitable for checking using FDR3 (Failure Divergence Refinement), and NRL protocol analyzer [5] - a special-purpose verification tool for analysing security protocols, written in Prolog. Other advanced tools suitable for verification of security protocols are AVISPA [6] and AVANTSSAR [7].

AVISPA (Automated Validation of Internet Security Protocols and Applications) is a tool funded by the European Union, which provides a push-button, industrial-strength technology for the analysis of large-scale Internet security-sensitive protocols and applications. Protocol models are written in the High Level Protocol Specification Language (HLPSL) [8]. AVISPA utilizes four tools for validation of security protocols: On-the-fly Model-Checker (OFMC), Constraint-Logic-based Attack Searcher (CL-AtSe), SAT-based Model-Checker (SATMC), and Tree Automata based on Automatic Approximations for the Analysis of Security Protocols (TA4SP). AVISPA is a popular tool and it was used for verification of security protocols that use smart cards multiple times.

AVANTSSAR (Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures) is a follow-up project of AVISPA, introducing new languages for describing models, the AVANTSSAR Specification Languages ASLan++ and ASLan. ASLan++ [9] is a high level formal language similar to the HLPSL, used for specifying security-sensitive service-oriented architectures, their associated security policies, and their trust and security properties. The semantics of ASLan++ is formally defined by translation to ASLan, the low-level specification language that is the input language for the back-ends of the AVANTSSAR Platform - OFMC, CL-AtSe, and SATMC.

Previous work on using formal methods for automated vulnerability finding in contactless smart card applications was presented by the authors of this paper in [10]. The AVISPA tool was used to create model and then SATMC model checker was used to find attack traces. The usability of the method was demonstrated on Mifare DESFire, which is a widespread contactless smart card, and possible attacks on integrity of encrypted data were found in a simple protocol. The attacker would be able to cause sending of different data to the terminal in the encrypted communication mode of Mifare DESFire by changing parameters of commands and the terminal would not have a chance to find it out. Presented attacks were found by implementing only a couple of Mifare DESFire commands.

2. Formal Verification of Protocol Implementation

This section provides description of a proposed method that can be used to create a model of contactless smart card and terminal and to define states representing attacks. This model can be then used in model checking to find attack traces in the protocol. The model takes into account the implementation details of the particular smart card which could be possibly avoided in the high level protocol verification. These details are important because wrong use of smart card commands may introduce vulnerability even if the high level definition of the protocol is secure. The ASLan++ language was chosen for protocol modeling, it can be used as input for multiple back-end model checkers of the AVANTSSAR Platform, which is suitable for purposes of security verification and is quite advanced.

A model of protocol in ASLan++ is defined by roles that can be played either by a legitimate party or by an adversary called intruder. We establish two main roles in the model description to represent the implementation - first role represents the smart card with its functionality and settings, second role represents the protocol. The protocol is executed by the terminal, the smart card only responds to commands from terminal. The protocol can be therefore identified with the terminal in our model. Contactless terminal is usually called Proximity Coupling Device (PCD), so we denote this role PCD. Contactless smart card is usually called Proximity Integrated Circuit Card (PICC). The intruder model that is used is the well-known Dolev-Yao intruder model [11]. All communication is synchronous with the intruder, the intruder intercepts messages from the legitimate user and each legitimate user receives messages only from the intruder. The intruder can be therefore identified with the network. PCD executes the protocol and communicates with PICC via the intruder, who is a man-in-the-middle. The goal of our vulnerability finding method is to find out if the intruder would be able to perform some attack in this configuration and identify the attack trace.

The state explosion problem has to be addressed. If we create precise model of the smart card and terminal functionality, the model will be too complex for the model checker, the number of states will be so high that the model checking execution time will be unacceptable. The goal of this paper is to create modeling method that will create models which can be computed using model checking in acceptable time and which describe the functionality sufficiently. We create simplified models that are weaker than the precise model would be, so more attacks can be found. Attacks that are found by the model checker can be tested and in case of false positive the model can be adjusted to be more precise and not contain the particular false vulnerability. The resulting model will be a trade-off between precision and model checker execution time.

2.1 Contactless Smart Card Model

The PICC can be seen as a state machine. The PICC reads commands from PCD, changes its internal state according to these commands, and responds back. States of the machine are determined by the internal state of the PICC logic and by the value of internal variables of the PICC, such as content of files and used cryptographic keys. Since the logic must have finite number of states and the files and keys can only have finite number of values, the number of states of the machine will be finite. The transition rules of the automaton are defined by the set of commands and parameters of these commands. Although the set of parameters will be high, it will be finite, so the number of transition rules will

be finite as well. We can therefore model the PICC behavior using a finite-state automaton, or more specifically a Mealy machine, whose output is determined by the current state and the current input. Another state machine concepts can be used instead, such as UML state machine, which is an enhanced realization of the finite-state automaton mathematical concept with characteristics of Mealy machine. However, in this paper we will describe the behavior of PICC using simple finite-state automatons and Mealy machines.

We can create the Mealy machine representing the PICC by combining an automaton describing the PICC logic and an automaton representing the state of memory. The formal definition of the PICC Mealy machine will be provided later in this paper. We can analyse the logic and memory automatons separately.

The PICC logic automaton should describe behavior of PICC as a response to the commands sent by PCD. The memory cards will result in very simple automatons, while smart cards with more complex logic like Java Cards or BASIC Cards, which allow execution of arbitrary code, will result in more complex automatons.

Let M_{logic} be a deterministic finite automaton defined as a quintuple, $(Q_{\text{logic}}, \Sigma_{\text{logic}}, \sigma_{\text{logic}}, q_{\text{logic0}}, F_{\text{logic}})$, consisting of:

- a finite set of states Q_{logic}
- a finite set of input symbols Σ_{logic}
- a transition function $\sigma_{\text{logic}} : Q_{\text{logic}} \times \Sigma_{\text{logic}} \rightarrow Q_{\text{logic}}$
- a start state $q_{\text{logic0}} \in Q$
- a set of accept states $F_{\text{logic}} = Q_{\text{logic}}$ (PICC may end in all states)

The automaton describing the state of PICC memory has states determined by the content of files, values of cryptographic keys, and values of all other variables that are persistent in the PICC memory and that can be changed during the life of the card. It can be defined similarly as the M_{logic}. Let $A = a_1, a_2, ...a_n$ denote all memory blocks (files, keys, etc.), n is the number of memory blocks. Let D be a set of all possible data that can be stored in a block. Let $C_{\text{write}} = A \times D$ be a set of all write command parameters, which consist of memory address and data to be written and let $C_{\text{read}} = A$ be a set of read command parameters consisting of memory address and let c_{noop} be a command for no operation. Let M_{memory} be a deterministic finite automaton defined as a quintuple, $(Q_{\text{memory}}, \Sigma_{\text{memory}}, \sigma_{\text{memory}}, q_{memory0}, F_{\text{memory}})$, consisting of:

- a finite set of states $Q_{\text{memory}} = D_1 \times D_2 \times ... \times D_n$, where n is the number of memory blocks
- a finite set of input symbols $\Sigma_{\text{memory}} = C_{\text{write}} \bigcup C_{\text{read}} \bigcup \{ c_{\text{noop}} \}$
- a transition function $\sigma_{\text{memory}} : Q_{\text{memory}} \times \Sigma_{\text{memory}} \rightarrow Q_{\text{memory}}$ (commands for writing data C_{write} change state appropriately, C_{read} and c_{noop} do not change state)
- a start state $q_{\text{memory0}} \in Q$ (initial content of memory)
- a set of accept states $F_{\text{memory}} = Q_{\text{memory}}$ (PICC may end in all states)

The automaton describing the PICC is the combination of the automaton describing the PICC logic and the automaton representing the state of memory.

Let M be a Mealy machine defined by a 6-tuple $(S, S_0, \Sigma, \Lambda, T, G)$ consisting of the following:

- a finite set of states $S = Q_{\text{logic}} \times Q_{\text{memory}}$
- a start state $S_0 = (q_{\text{logic0}}, q_{\text{memory0}})$, which is an element of S
- a finite set of input symbols $\Sigma \subseteq \Sigma_{\text{logic}} \times \Sigma_{\text{memory}}$; input alphabet will contain only meaningful commands:
  $(write, c_i)$, where $write \in \Sigma_{\text{logic}}, c_i \in C_{\text{write}}$
  $(read, c_i)$, where $read \in \Sigma_{\text{logic}}, c_i \in C_{\text{read}}$
  (c_i, c_{noop}), where $c_i \in \Sigma_{\text{logic}} \setminus \{write, read\}$, $c_{\text{noop}} \in \Sigma_{\text{memory}}$
- a finite set called the output alphabet $\Lambda = D \bigcup R$, where R is a set of PICC status responses and D will be used for read command responses
- a transition function $T : S \times \Sigma \rightarrow S$ mapping pairs of a state and an input symbol to the corresponding next state
- an output function $G : S \times \Sigma \rightarrow \Lambda$ mapping pairs of a state and an input symbol to the corresponding output symbol

In order to reduce the number of states in the model, we can minimize the number of states in the Mealy machine M. The number of states can be reduced by reducing the number of states in M_{logic} or M_{memory} or both.

To reduce the number of states in the M_{logic} automaton, we can keep only states that perform data transfer (read or write) and join them with the supporting states that represent the chain of commands leading to data transfer. We can create complex commands that are combination of multiple real commands. Each such command results in some data transfer. This optimization reduces execution time of the model checker.

To reduce the number of states in the M_{memory}, we have to reduce the number of memory blocks that can be written to, and/or reduce the number of possible data that can be stored. The number of possible write locations on a smart card can be tremendous, good approach is to have only memory locations that the application is supposed to write to or read from and one undesired location for each file that will be used to simulate writing or reading to bad location that will corrupt the result. Using this approach the total number

of states will be reduced dramatically, which will also reduce the model checker execution time.

Now we have simplified automaton representing the PICC behavior, so we can create the PICC role in ASLan++. There are some basic concepts that can be put together to form a smart card model. These concepts are general and can be used to create a model of arbitrary smart card with pre-defined set of commands. We describe modeling of the following concepts: Authentication, Multiple Applications, File system, Permissions, Encryption, and Personalization. The following sections describe the method of creating the PICC role in ASLan++ for these concepts.

Applications

Multi-application contactless smart cards support multiple applications even from different vendors on a single card. The application on cryptographic memory card is not an executable program, it is rather a set of resources dedicated for application outside the card. The application on the card can consist of files used to store data and symmetric keys used for authentication and data encryption. The application outside the card can securely store data in the card and read them back later. This can be used for instance for payment applications or loyalty program applications, where some credit is stored on the card.

To simulate the application selection in the PICC role, we can use a variable which is set by the PCD using a $selectApplication$ command. The value of selected application is then used for file access. If we use the simplified automaton, the application selection is part of another command, such as the *read* or *write* command.

Authentication

The authentication process between smart card and terminal is usually mutual, both parties must prove possession of a common secret. The mutual three-pass authentication can be modeled as only one-pass authentication thanks to the fact that in the model we can be certain of things that we cannot be in the real environment. The way of modeling the mutual authentication process in ASLan++ using only one message pass is a fresh session key generation performed by one of the parties (PCD) and sending it encrypted using the authentication key to the other party (PICC). The other party (PICC) must check that the session key is fresh and was never used before, which would not be feasible in real environment. The fresh session key generation will prevent replay attacks.

After the one-pass authentication PCD and PICC share a common session key, which could not have been eavesdropped by the attacker, because it had been encrypted with a key not known by the attacker. The PICC knows which authentication key was used and can grant access to files accordingly. The authentication needs to be implemented in both PCD and PICC roles. The PCD always starts the communication and sends commands, so it will also generate the random session key. In case of the simplified automaton, the authentication can be part of another message.

Encryption

The high level language ASLan++ already supports modeling of communication encryption, but it does not consider various modes of encryption algorithms. In ASLan++ any data can be encrypted using symmetric or asymmetric cipher, and for purposes of modeling these ciphers are considered unbreakable, hence the intruder cannot learn the plaintext of the encrypted data unless he knows the corresponding key. But there are different modes of encryption that must be taken into account when creating a model even if the cipher algorithm itself is considered unbreakable. Symmetric ciphers are used in the following modes: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR).

In ASLan++ each block is encrypted using same key and there are no initialization vectors, so we can consider it the ECB mode. From the protocol modeling perspective, the CBC, CFB, OFB, and CTR modes do not differ. They use the initialization vector which is different for each block. We can model these modes by adding random fresh number to the data being encrypted, simulating the changing initialization vector, which will provide resistance to replay attacks.

Files and Permissions

Smart cards provide file system with permissions that can control access to each file based on the key that was used for authentication. We can model files and permissions in ASLan++ either as variables or as facts. Facts are global and more flexible, so when using facts it is possible to check content of PICC files even from the PCD role, and it is possible to add new facts and retract existing facts, which can be used to simulate flexible file system where files can be created and deleted. Let us denote the file system fact *fileSystem* and declare it with four parameters for data address, authentication keys to get read and write permission, and data itself in ASLAN++ in the following way:

```
fileSystem(text, symmetric_key, symmetric_key,
message):  fact;
```

The first parameter of the fact represents the address of the file and is of type *text*, which is the most simple type in ASLan++. The second parameter represents authentication key that must be used to obtain read permission to this file and is of type *symmetric_key*, which is an ASLan++ type for symmetric keys. Analogously, the third parameter is the authentication key for write permission. The fourth parameter represents data stored in the file and is of type *message*, which is a compound type that can store any combination of data of any other type.

Although address has a simple type, it represents a number of values that constitute the address on a real card, such as selected application number, file ID, offset, and length of data. We decided to have a separate fact for each data block that can be addressed instead of one fact per file, which results in more than one fact per file. Blocks of different lengths and offsets may overlap, so not all blocks will contain meaning-

ful data. Such blocks will contain the message *corrupted* to easily recognize unwanted data.

Long files will contain many fact definitions, but for modeling purposes we can reduce the number of possible file addresses by defining only the desired addresses and one invalid address instead of all possible invalid addresses. Reading from this invalid address will return *corrupted* and writing to this location will save *corrupted*.

Smart Card Personalization

For using in a protocol, such as payment protocol or loyalty program, the smart card must be personalized. Personalization is a process when the smart card is initially populated with data of the intended smart card user, such as name or account number. Consequently, each smart card will contain different data in files. This process should be taken into consideration when modeling the smart card protocol. The personalization process does not have to be modeled, since it usually takes place in a trusted environment. The smart card can be used in the modeled protocol only after the personalization, so we can create the model of the already personalized card. To create the model of a personalized smart card, all files must be created and populated as they would be during the personalization process.

2.2 Modeling Application Logic

During the development, the developer can use the sequence diagram of the protocol or the flow diagram of the application as the basis for the PCD model. The PCD role should contain the logic (or simplified logic) of the application. The intruder can also play the PCD role, but he does not have to follow the logic in the role definition, he can perform arbitrary actions. The role definition applies only to the legitimate entity.

The previously described optimization of the PICC role will reduce the number of commands by making them more complex. So for example the three-pass authentication followed by the *selectApplication* command and then by the *read* command will result in only one command combining them together. This fact must be taken into account when translating the model checker results into the applicable attack paths.

2.3 Attack Definition

The attack definition must be provided for the model checker to find any attack traces. The attack is defined as a condition that should never happen in normal protocol run and that means that the intruder learned something that he should not have learned (confidentiality), or that he changed something that he should not have changed (authentication, integrity). These conditions are defined in the ASLan++ model and then translated to states that mean an attack. If the model checker finds a path to one of the attack states, a possible attack is reported. The attack trace should be evaluated and in case of false positive, refinements should be made to the model. The model checker should be run again and this process should be repeated until real attack is found or the model checker concludes that there is no attack.

Although there are means for defining security goals of confidentiality and authentication in ASLan++, these do not fit well for the purposes of our attack definitions. We will use assertions that will always hold unless an attack is under way. We can easily set goals that the protocol should achieve, covering all desired security goals, by defining assertions in PCD role that can contain information from PICC which would not be available in real environment, such as content of files (because files are modeled as global facts). The following example shows an assertion that can be used at some point in the PCD or PICC role to check content of some file on the card:

```
assert ok:
fileSystem(addressBalance,key1,key1,newBalance)
```

We can interpret this assertion as follows: if the file at address *addressBalance* contains the value *newBalance*, it is ok, otherwise the model checker will stop and an attack will be reported.

3. Sample Protocol Implementation

To demonstrate how the model creation process works and how the security attributes of a protocol can be verified, an example is provided in this section. The example introduces very simple payment system that uses Mifare DESFire like contactless smart card to store the balance. The model of communication protocol between the terminal and the card will be created and used as an input to the model checker.

Let us suppose that we need to develop a new payment protocol which uses contactless smart cards. The card will be issued to the cardholder personalized with his name and the initial balance. The cardholder will be able to pay for goods with this card. After he pays using the card, the price will be subtracted from the current balance. The balance can be increased by the authorized entity. From these basic requirements we can decide how the payment system should be implemented and create a sequence or flow diagram of the application. The developer should first create the sequence or flow diagram of the protocol and create and optimize the automaton representing the smart card, then he can create the PCD and PICC models, define conditions that represent attacks and verify using model checking. Finally, he can implement the protocol in the target programming language.

Let us create an intuitive protocol that will fulfil the stated requirements. The cardholder's name and balance will be stored on the contactless smart card in files. The card model which is used is inspired by the Mifare DESFire MF3ICD40 contactless smart card. When the cardholder puts the contactless smart card to the proximity of the contactless smart card reader

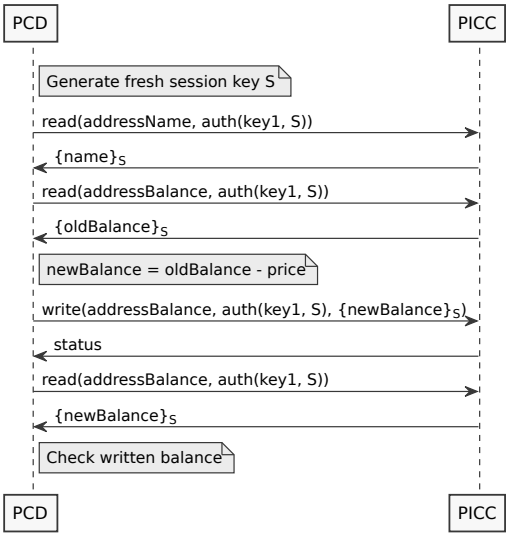

Fig. 1. Payment protocol with reduced set of commands.

at the point of sale (POS) terminal, the anti-collision procedure is performed and the payment protocol can be executed. The mutual authentication should be performed at the beginning of the transaction and the data that are then transmitted should be encrypted in both directions. The terminal first reads the cardholder's name and then the balance. If the balance is higher than the price of the goods, the price is subtracted from the balance by the POS terminal and the resulting balance is written to the smart card. To check the written value, the balance should be once more read from the card and verified. If the read balance is correct, the protocol ends and the cardholder can take the goods.

Figure 1 shows how the PICC role implementation of the protocol may look like when the number of PICC commands is already reduced to *read* and *write* in order to reduce model checking execution time. First two parameters of both commands are same. The first parameter is in both cases the address of data to be read or written. Mifare DESFire MF3ICD40 uses application number, file ID, offset of data in file, and length to address particular data block, so the address will represent the combination of these values. For modeling purposes, each of these combinations will be named according to the variable it will store. So for example the cardholder's name will be stored in application number 1, in file with file ID 1, with offset 0 and length 20; this particular data block address will be named *addressName* to indicate that this address is used to store the name. Other addresses will be named in the same manner. Addresses not intended to store data will also have some name.

The second parameter $auth(key1, S)$ is an authentication token. It can be seen as session key S encrypted using private key $key1$ ($key1$ is shared between legitimate entities and not known by the intruder). The PICC checks whether S is the current session key (no new authentication) or S is a fresh session key (authentication using $key1$). Every old session key (evoked by replay attack) is rejected by the PICC. The third parameter in the *write* command is the data to be written encrypted using the session key from second parameter. The response of the *read* command is the data encrypted using the session key from second parameter, the response of the *write* command is only a status message. Symmetric encryption of $oldBalance$ using key S is denoted $\{oldBalance\}_S$.

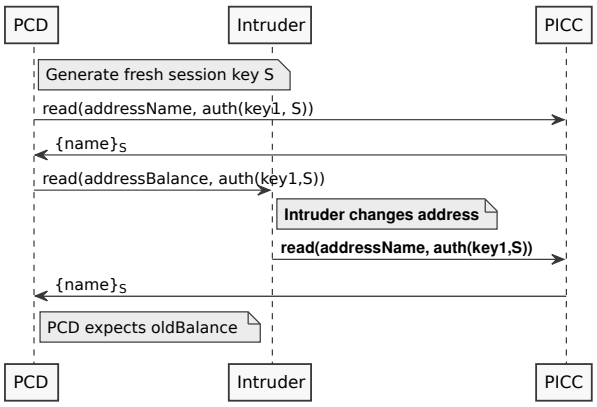

Fig. 2. Attack based on forged address.

The attack definition consists of integrity and confidentiality checks implemented using assert. There is one assert at the end of the PCD role stating that the balance on the card is equal to the $newBalance$ value. In other words, when the protocol is executed successfully and the PCD checks the written balance and comes to a point where it believes that the balance on the card is set to $newBalance$, the actual value on the card is really $newBalance$. This assert can be realized thanks to modeling of files as facts, which are visible globally. Other asserts can be used to check intermediate states of the protocol.

The ASLan++ model was translated to the ASLan format and used as an input for the Cl-Atse model checker. Several model checker runs and protocol adjustments revealed some possible attacks, which are discussed in the following sections.

3.1 Sample Verification 1

The first attack that was found was caused by the fact that the address of data blocks on the card is not cryptographically protected. The attacker changes the data address, which consists of the application number, file ID, length, and offset. This results in the PICC returning wrong data block or writing into wrong address. These attacks were described in [10]. Figure 2 shows the output of the model checker translated into the sequence diagram. Intruder's actions causing an attack are shown in bold in figures.

A countermeasure to this attack can be some integrity check on the application layer of the payment system. For purposes of integrity checking CRC or cryptographic signature can be used. CRC would be enough, because all data transmitted between PCD and PICC are encrypted, so the intruder cannot change data nor CRC, which is part of the encrypted data, without corrupting whole data block. If there is for example only one file containing the CRC protected

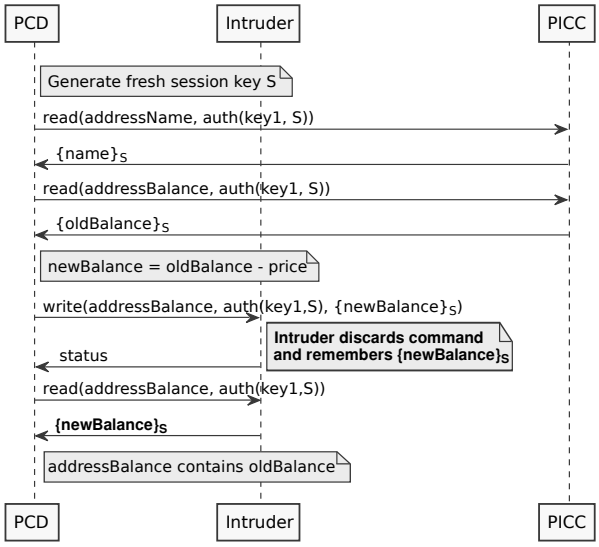

Fig. 3. Attack based on discarding write command.

data, the PCD can easily distinguish this valid data block from another data block. So let us add such integrity checking to the protocol model in the PCD role which will help PCD to distinguish valid balance from another corrupted data. Figure 3 shows an attack on this improvement that was found and that is based on discarding write command by the intruder.

A countermeasure to this attack can be re-authentication after data writing, which means that data that are read after re-authentication are encrypted using new session key, so the intruder cannot replay the previously eavesdropped encrypted balance. Figure 4 shows an attack on this improvement that was found and that is based on changing the address in the *write* command to another valid file. The $newBalance$ will be saved to another file and then read from this file after re-authentication for checking. The check will pass; however, the file storing the balance will contain $oldBalance$.

A countermeasure to this attack can be allowing writing to only one file and restricting writing to all other files.

The cause of all these attacks is the weakness of Mifare DESFire contactless smart card - not encrypting or signing commands, application number, file ID, length, and offset.

Note that the size of files and data being transferred between PCD and PICC is ignored in the model, so the model checker will sometimes find an attack which is not feasible in real environment due to the size limitations. These false positives can be avoided by also including size of files and data in the model.

3.2 Sample Verification 2

The second example of attack finding using formal verification methods described in this section is based on an improved protocol from the previous example using an improved contactless smart card. The contactless smart card used is a hypothetical card similar to Mifare DESFire

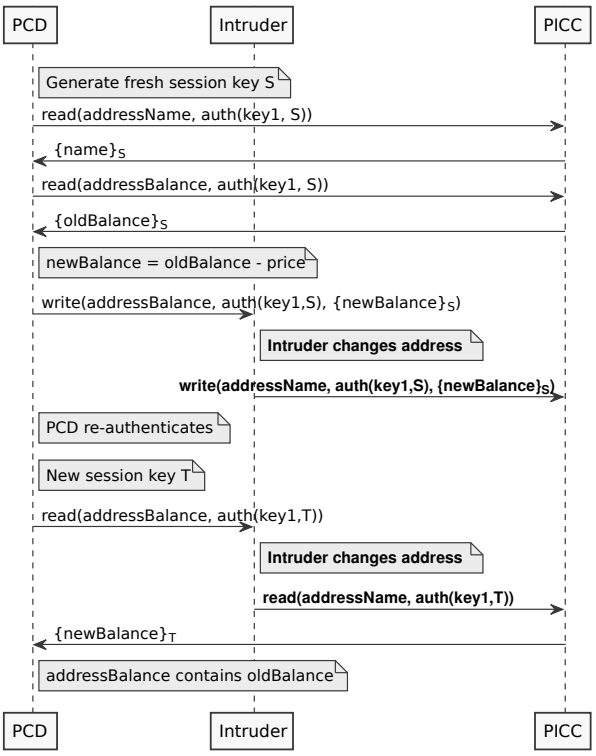

Fig. 4. Attack based on writing to another file.

with the difference that everything in the communication is encrypted (after successful authentication), not only data. This improvement will prevent attacks from the previous example. Another small change is that anyone can change the name on the smart card; this field is only informative, so there is no need to protect it. This will be modeled using $key2$, which will be known to the intruder and which will be required for granting write permission. The balance field will be protected in the same way as in the previous example, using $key1$, which is not known by the intruder.

The model checker found an attack that we call command injection attack. This attack is more sophisticated and complicated than the previous attacks. An attacker authenticates using the publicly known $key2$ and writes the forged command to the address *addressName*. Then, during the protocol run initiated by the legitimate PCD, when the PCD reads the name field from *addressName*, an attacker (man-in-the-middle) eavesdrops the forged command encrypted using current session key (not known by the attacker). He can then send the encrypted forged command to the PICC and the PICC cannot find out that it was not sent by the legitimate PCD, because it is encrypted using the current session key that was established during the authentication using $key1$, which is known only to the legitimate PCD. Despite the fact that commands and their parameters are encrypted, the intruder can execute arbitrary command, he only has to prepare it in advance. Figure 5 shows the command injection attack found by the model checker translated to human readable form with the imaginary command called $maliciousCommand$, that is here to emphasize the fact that it can be arbitrary malicious command.

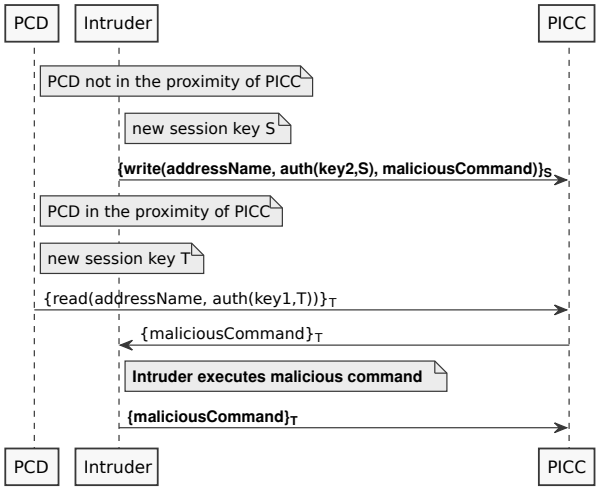

Fig. 5. Command injection attack.

The countermeasure to this attack can be any replay attack protection which will ensure the freshness of the messages. Any encryption mode different from the default ECB mode would prevent this attack.

4. Conclusion

We have presented a method that can be used for security verification of smart card protocol implementation. The focus is on contactless smart cards, because they are simpler than smart cards with contact interface, often only memory cards providing encrypted communication. This method was demonstrated on one of the most widespread contactless smart cards, the Mifare DESFire. However, the method can be used on other smart cards as well, even on more complex cards with operating system. Two sample verifications were shown to demonstrate the usability of the method. By using this method, the developer can iteratively fix the vulnerabilities found by the model checker and secure the application.

Acknowledgments

The research has been supported by the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070 and the internal BUT project FIT-S-14-2486.

References

[1] NXP SEMICONDUCTORS. *Mifare DESFire MF3ICD40 Contactless Multi-Application IC (Short Form Specification)*. [online] Available at: http://www.nxp.com/documents/short_data_sheet/075532.pdf

[2] ORACLE. *Java Card Technology*. [Online] Cited 2015-08-15. Available at: http://www.oracle.com/technetwork/java/embedded/javacard

[3] ZEIT CONTROL. *Basic Card*. [Online] Cited 2015-08-15. Available at: http://www.basiccard.com/

[4] LOWE, G. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 1998, vol. 6, p. 53–84. ISSN: 0926-227X

[5] MEADOWS, C. The NRL protocol analyzer: An overview. *The Journal of Logic Programming*, 1996, vol. 26, no. 2, p. 113–131. ISSN: 0743-1066. DOI: 10.1016/0743-1066(95)00095-X

[6] ARMANDO, A., BASIN, D., BOICHUT, Y., et al. The AVISPA tool for the automated validation of internet security protocols and applications. In *Proceedings of the 17th international conference on Computer Aided Verification (CAV'05)*, Springer-Verlag, Berlin, Heidelberg, 2005, p. 281–285. DOI: 10.1007/11513988_27

[7] AVANTSSAR. *Automated Validation of Trust and Security of Service-Oriented Architectures*. [online] Cited 2015-08-15. Available at: http://www.avantssar.eu/

[8] CHEVALIER, Y., COMPAGNA, L., CUELLAR, J., et al. A high level protocol specification language for industrial security-sensitive protocols. In *Proceedings of the Workshops on Cooperative Support for Distributed Software Engineering Processes (CSSE'04)*, Austrian Computer Society, 2004, p. 193–205.

[9] VON OHEIMB, D., MOEDERSHEIM, S. A. ASLan++ – A formal security specification language for distributed systems. In *Formal Methods for Components and Objects: 9th International Symposium*, Graz, Austria, Springer, 2010, p. 1–22. DOI: 10.1007/978-3-642-25271-6

[10] HENZL, M., HANACEK, P. Modeling of contactless smart card protocols and automated vulnerability finding. In *International Symposium on Biometrics and Security Technologies (ISBAST) 2013*. Chengdu, IEEE Computer Society, 2013, p. 141–148. DOI: 10.1109/ISBAST.2013.26

[11] DOLEV, D., YAO, A. C. *On the security of public key protocols*. Technical report, Stanford, CA, USA, 1981.

About the Authors . . .

Martin HENZL received his M.Sc. at Masaryk University and is currently a Ph.D. student at Faculty of Information Technology, Brno University of Technology. His research interests are in information technology security, especially in smart cards and applied cryptography.

Petr HANACEK is an associate professor at the Faculty of Information Technology at Brno University of Technology. He concerns with information system security, risk analysis, applied cryptography, and electronic payment systems for more than ten years.