Model-Based Testing of Network Security Protocols in Java Card Applications

Master's Thesis

Radboud University Nijmegen

Author:
Richard Ssekibuule
Student Number: 0440752

Supervisors:
Dr. Martijn Oostdijk

Dr. Jan Tretmans

Dr. Vlad Rusu

Script Number: 558

October 2006

Abstract

This research presents a combination of verification and conformance testing techniques for systems that implement network security protocols. We investigate model-based methods for detecting vulnerabilities in network security protocols and testing for correct behaviour of Java Card applications in which network security protocols are implemented.The research assumes an open environment which is insecure in which adversaries might try to gain access to restricted resources.

We present techniques for model-based testing of network security protocols in Java Card applications and analyse the use and effectiveness of verification in addition to conformance testing in enhancing the quality of a software product. We analyse the use of HLPSL in modelling Kerberos network authentication protocol in an electronic banking (E-Banking) application and also use of promela in modelling behaviour of an electronic passport(E-Passport).

We present benefits and challenges of performing verification and conformance testing of software application using formal specifications.

Acknowledgements

I take this opportunity to thank the people who have helped me in one way or another to complete my masters studies at Radboud University. Firstly, I thank my supervisors, Dr. Jan Tretmans, Dr. Martijn Oostdijk and Dr. Vlad Rusu for guiding me through this thesis work. I am very thankful to NUFFIC and Faculty of Computing and Information Technology(Makerere University), for giving an opportunity to study in The Netherlands. My teachers at Radboud university have been very helpful in guiding me through the two years of my studies. I wish to also extend my gratitude to the Security of Systems group at Radboud university, for developing the E-Passport and JMRTD API; my heartfelt thanks go to Cees-Bart Breunesse and Renè de Vries for all the help you rendered me during the testing process. I also thank the developers from IBM for development of the E-Banking application. I cannot forget to thank Zainah for the endless love that kept my heart warm in the winter days of Nijmegen. And to all my fellow students I stayed with in Muzenplaats and Nijmegen I thank you. Special thanks go to my parents and siblings for being there for me whenever I needed you.
I dedicate this thesis to my departed grand father Peter Musajja'wazza, whose wisdom and courage will always be missed.

Contents

Chapter 1

Introduction

Development of a reliable software system has to be approached in a systematic way and requires the use of appropriate tools and methods to ensure high levels of quality for the system under development.
Organisations and applications developers are faced with challenges of choosing the rightful method of testing that can be integrated in the product development life cycle to ensure these required high product qualities that can boost consumer confidence in their products. Equipped with appropriate methods of testing, organisations can be in position to enhance the quality of software they develop.
Model-based testing has been successful for many types of application tests, but little has been said about model-based testing for security properties of applications and protocols. This research investigates with practical experiments the use of model-based testing and verification techniques in testing and validation of applications that implement network security protocols in Java Card applications. In this thesis project we investigate the use of model-based testing to verify correctness of Java Card application and to ascertain whether the application conforms to the expected behaviour that is normally specificed informally.

The research work considered two systems that are used in the study of model-based testing and verification of network security protocols in Java Card applications. The applications we considered consist of an electronic banking system in which users can perform transactions from their mobiles phones and an implementation of machine readable travel document in which credentials of the document owner are kept on the smart card.

The rest of this chapter presents the motivation that led to performing this research; definitions of concepts like security protocols, formal verification and testing are discussed in detail. Java Card, smart cards and tools (AVISPA and TorX) that are used in this thesis are explained in detail for the components that we employed in the research. This chapter concludes with a look ahead into what is presented in the subsequent thesis chapters.

1.1 Motivation

Typically large software applications contain bugs and design flaws which go through project design and implementation undetected. As a result, many of the computer applications that people use on a daily basis either contain bugs or design flaws.

The world today is dependant on distributed data communications for day today businesses in which security protocols are very important to Information Technology structures like the Internet. Security protocols in these applications are used to ensure that informa-

tion is transmitted in the manner that is desirable to participating parties. Some of the desirable qualities in the communication include confidentiality and integrity, in which the former refers to the secrecy of exchanged information and the latter refers to non-tampering with information by adversaries.

Correctness in the design of security protocols is no longer considered a desirable quality, but a must have. This is due to the high stakes of finances involved in case vulnerabilities in the system are exploited by an attacker.

Formal methods are believed to be the solution to system failures that are caused by informal specifications. It was this belief among most scientists that motivated us to investigate the use of formal methods in testing of Java Card applications that are in popular use nowadays.

The research aimed to perform formal verification and conformance testing on real-life systems for security requirements of the implemented protocols.

1.2 Network Security Protocols

A security protocol is a sequence of operations that ensure protection of data. A security protocol is an abstract or concrete protocol that performs a security-related function and applies cryptographic methods[Wik06b].

Network security protocols provide abstraction of messages that are to be exchanged between parties that are performing a security related operation. Security protocols are used in network security to regulate access to restricted assets and they are considered to be difficult to implement correctly. Worse still, erroneous implementations of network security protocols can lead to undesirable losses in terms of money and life.
Security protocols are also considered by researchers to be complex objects that need to be specified, designed and verified formally. Hao et al. [CCJ03] among many other researchers have suggested and developed new techniques such combinatorial optimisation mechanisms to perform automated design of security protocols.
Researchers at the French National Institute for Research in Computer Science and Control are trying out new and hopefully better methods for automatic verification of protocols based on approximations as presented in the paper by Boichut et al. [BHK05].

1.3 Java Card and Smart Cards

This research project was performed with real-life implementations using Java Card technology. In this section, we briefly introduce the Java Card technology.
Java Card was developed and specified by SUN microsystems[Net06] to enable devices with limited memory to run applications that employ Java technology. Java Card is a standardised technology by the Joint Technical Committee One (JCT1) of the International Standards Organisation (ISO) and the International and Electronics Committee (IEC). Java Card technology adapts the Java Standard Edition platform for use on smart cards but is more restrictive than Java; for example only short is allowed as the largest data type. Smart cards are an example of devices with limited memory and they were used in the applications investigated in this research project.

Smart cards are normally plastic cards containing an integrated circuit (IC) and are considered to be difficult to forge because they are designed to be tamper resistant. Smart cards are designed in such away that if stolen, the illegal owner would not gain from a stolen smart card, or will not be able to counterfeit it (the smart card). Smart cards are a trusted computing platform that also prevent a legal owner from gaining access to certain information like cryptographic keys that are used in network security. This means that an intruder can not steal a key that is stored in a smart card without the knowledge or coercion of the rightful owner. Additionally, it is assumed that the rightful owner of the Smart card cannot alter protected information without destroying the card.

1.4 Formal Verification

As indicated in the motivation section 1.1 we investigated formal verification of security requirements for security protocols in the E-Banking and E-Passport application. In this section we present an introduction to verification.
Formal verification refers to the process of checking whether a design of a system satisfies some requirements. Formal verification is a well established mechanism for checking reactive systems that is used to compare formal specification of a system with respect to some high-level requirements. The goal of verification is to ensure that the software system fully satisfies the expected requirements.

Verification should not be confused with validation. The validation process is concerned with answering the question of whether the right product or system is being built where as the verification is about answering the question of whether the product was built right[Wik06a].

Lots of research has been done to achieve automatic verification of systems. The most popular method for automatic formal verification being based on model checking[CGL94, ACD90, Hol97].

1.5 Testing

Interestingly, there are varying definitions of testing that are provided by scientists in the field of testing. We consider two definitions that we find comprehensive; the definition provided by ISO [Gui04] and another by Tretmans [Tre04].

> According to ISO, testing is a technical operation that consists of the determination of one or more characteristics of a given product, process, or service according to a specified procedure.

> According to Tretmans[1], software testing is a technical process performed by executing/ experimenting with a product in a controlled environment, following a specified procedure with the intent of measuring one or more characteristics or quality of the software product by demonstrating the deviation of the actual status of the product from the required status or specification.

The second definition of testing is more comprehensive than the first one because it indicates the intentions and goals of performing testing. There is more to testing than only determining the characteristics of a product; testing should also involve comparison of characteristics

[1] http://www.cs.ru.nl/~tretmans

of a product to the expected or required status. We consider the definition provided by Tretmans for all references to testing.

1.5.1 Conformance Testing

Conformance testing is a type of testing used to determine whether a system meets some specified standard. Conformance testing involves testing whether an implementation conforms with respect to an implementation relation to its specification. In the definition of conformance, we are mainly interested in the implementation under test (IUT) and specifications, so that a universe of implementations IMPS, and a universe of formal specifications SPECS are assumed.
It is also assumed, that in the formalism MODS an object i_{OUT} can be used to model each implementation $OUT \in IMPS$. This hypothesis is referred to as the *test assumption*[TPHT] and it assumes that a model i_{OUT} exists. Since a test assumption exists, it allows reasoning about the implementations as if they were formal objects and conformance of implementations with respect to formal specifications expressed by means of an *implementation relation* `imp`.
The formalism presented below is defined by Terpstra et al.[TPHT] for an implementation relationship `imp`, the set of models of MODS and set of specifications SPECS.
$imp \subseteq MODS \times SPECS$: Informally, this means that an implementations relation `imp` is a relation between the set of models MODS and the set of specifications SPECS.
Implementation $OUT \in IMP$ is considered `imp`-correct with respect to the specification $s \in SPECS$, if and only if the model $i_{OUT} \in MODS$ in OUT is `imp`-related to s: i_{OUT} `imp` s.
Furthermore, a short hand notation is defined for $M_s =_{def} \{i \in MODS | \ i \texttt{ imp } s\}$ as the set of models of implementations that conform to s with respect to implementation relation `imp`.
The notion of conformance i.e. $i_{OUT} \in M_s$, and test execution, i.e., OUT `passes` T (a set of test cases) have to be linked in order to be able to indicate whether an implementation under investigation (OUT) is conformant to the specification `s` after a test execution.
Given a set of models P_T that pass the test suite T, execution of a test suite T such that the set of conforming implementations M_s, is approximated by (and preferably equal to) the set of models P_T is referred to as conformance testing[TPHT]. Additionally, the test suite $T \subseteq TESTS$ has two properties:

- Soundness: Which means that if an implementation fails a test, then it(OUT) does not conform to the specification, because all correct correct implementations and possibly some incorrect ones will pass the test: $M_s \subseteq P_T$
- Exhaustiveness: This means that if an OUT does not conform to the specification, then there exists a test in the test suite which fails: $P_T \subseteq M_s$.

These properties pose a weak requirement for conformance testing because the property $M_s = P_T$ is too strong for practical testing. The property $M_s = P_T$ requires a complete test suite which is infinite and and consequently not practical for testing.

1.5.2 Model-Based Testing

Model-based testing, refers to the type of testing in which models are used to generate test cases from requirements to be used against a system under investigation. The general popularisation of modeling in both academia and industry, has led to a growing interest in

model-based testing among scientific researchers and industrialists.
Model-based testing is being used by many researchers world wide, including the Object Management Group (OMG)[2] to define tasks that are used in developing platform independent models. Apart from allowing generation of test oracles and test cases from models, models-based testing facilitates representation of the environment in which test executions take place in order to benefit from the separation of platform independent models from platform specific models[HL03]. Model-based testing is regarded as the technology to meet the new challenges of software testing.

Model-based testing and formal verification serve complementary goals. As indicated in section 1.4 formal verification intends to show that the desired properties of the system have been implemented by proving that the model of the system has those properties. However, these properties holding for the model of the system(real implementation) does not mean that they also hold for the real implementation. Testing becomes complementary to verification, because it(testing) involves "experimenting with the real implementation in a controlled environment, following a specified procedure with the intent of measuring one or more characteristics or quality of the software product by demonstrating the deviation of the actual status of the product from the required status or specification"[3].

1.6 Tools and Languages

Now that we know what testing is all about and what model-based testing means, we now focus on the tools that we used in our investigation of model-based testing for network security protocols in real-life application.

1.6.1 AVISPA Tool

AVISPA tool is developed by the AVISPA[4] project which acronym-ed as "Automated Validation of Internet Security Protocols and Applications". AVISPA tool is developed to provide analysis of large scale Internet security-sensitive protocols and applications. The AVISPA Tool provides a suite of applications for building and analysing formal models of security protocols[arm05].
The formal models of protocols to be analysed are written in a specification language known as HLPSL; which is an acronym for High Level Protocol Specification Language[che03].

1.6.2 TorX

TorX is a conformance testing tool for reactive systems[Tor02] that is used to perform automated testing, in which test cases are automatically generated from the specification and executed against a blackbox implementation of the system.
Operations of TorX requires a real implementation and a formal specification of that implementation. The behaviour of the real implementation is described by the specification. TorX checks the behaviour of a real implementation and states whether the implementation is correct based on the observed behaviour of the system in reference to the specification.
`ioco` testing theory is used in TorX to define correctness[Tre99, Tre96] . The definition of

[2]http://www.omg.org
[3]definition of testing(1.5).
[4]http://www.avispa-project.org/

`ioco` is formally provided below:

$$i \; ioco \; s \Leftrightarrow_{def} \forall \; \sigma \; \in \; Straces(s) : \; out(i \; after \; \sigma) \subseteq \; out(s \; after \; \sigma)$$

The statement above means(informally) that, an implementation `i` is `ioco` correct with respect to the specification `s`, if and only if, for all possible behaviours of the specification, any output of the implementation is also possible with the specification. This formal notion is indicated by Tretmans ET AL. [TB03] as the starting point for a test generation algorithm which derives a test suite from the specification to test the implementation for `ioco-correctness`.

1.7 Still to come

Chapter 2 presents the research work that we did using the E-Banking application. Technical details of the E-Banking application are presented and the Kerberos network security protocol that is implemented in the application also discussed in detail. The chapter presents a detailed study of Kerberos message specifications and the mapping between the application source code and the specifications of Kerberos. The results of studying the application source code are used to obtain an HLPSL specification of the application, that is later used perform automated verification using AVISPA.

Chapter 3 presents Machine Readable Travel Documents(MRTD) and the work that was done with them(MRTDs) to perform conformance testing. We present technical details of the inspection flow process for MRTDs which includes the basic access control protocol that is tested for conformance to the International Civil Aviation Organisation(ICAO) specifications. A detailed explanation of conformance testing for the basic access control protocol implemented in the E-Passport is presented with the test results obtained using TorX automated testing tool.

Chapter 4, discusses the implications of the results that we obtained with automated verification and testing. Technical experiences from the research project are presented in addition to conclusions and a look ahead at future work.

Chapter 2

E-Banking Application

In this chapter we present the research work that was done using the E-Banking application. Technically, a functional E-Banking application consists of a mobile device or cell phone that is able to perform wireless communication with a remote application server.

A small Java program that is known as a MIDlet[Knu03] is installed on the user's mobile device to provide an interface for application access. The application allows users access their bank accounts and perform any supported transactions by using the installed graphical user interface(MIDlet). Essentially, if the transaction is not an enquiry one (For example, asking for bank balance) then, it would have to result into a debit or credit of the customer's account. The Java Card application on the Java Card would contain the authentication logic that ascertains whoever is trying to perform a business transaction.

The E-Banking application uses a SIM[1] card (which is also a Smart card) on the user's device as a trusted computing platform and also a secure storage for security credentials like cryptographic keys. We use the Java Card reference implementation (CREF) in this research to provide an emulator for a smart card reader. The architecture of this setup is shown in figure 2.1 below.

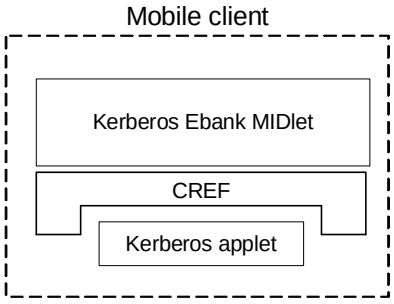

Figure 2.1: *Mobile client overview*

A detailed study of Kerberos network authentication protocol message specifications is presented in section 2.1, and the mapping between the protocol(Kerberos) specifications and the E-Banking application source code is presented in section 2.2. The mapping between Kerberos message specification and the application source code was done to obtain message sequence exchange between communicating principals and to derive the HLPSL specification that was used in performing automated verification presented in sub section 2.2.3.

[1] http://en.wikipedia.org/wiki/SIM_card

2.1 Kerberos and E-Banking Application

The Electronic Banking application, which is referred to as an E-Banking application in this research, uses Kerberos network authentication protocol[GL98, Ste88] to provide a secure environment for communication between the Java Card client and the banking Application Server hosted at the bank. The implemented security protocol is supposed to ensure that information transmitted using the wireless application will not be compromised. All information exchanged is encrypted using a secret key stored on the user's SIM(Smart) card. The secret key comprises of the user's pin code which is chosen at the time of installation and an E-Banking key which is known only to the bank. The Java Card technology has to ensure that the E-bank's key is never exposed to any client application accessing the Java Card application.

In this section we present details of Kerberos that are used in E-Banking application. A discussion of Kerberos message components that are implemented for the E-Banking application and their relevance and impact on the system is performed and presented in this section.

Figure 2.2 presents a diagrammatic overview of Kerberos, with the message exchanges presented in a Kerberos protocol.

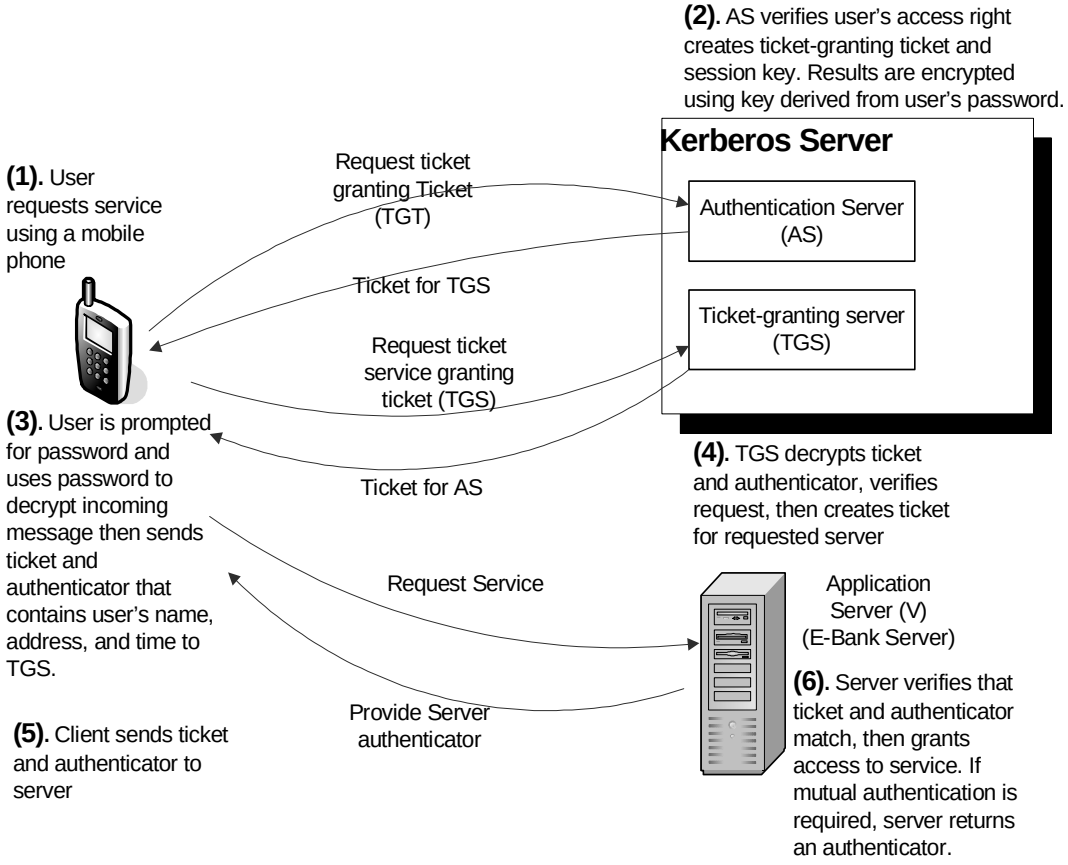

Figure 2.2: *E-Banking architecture and Kerberos overview*

The rationale of messages that are presented in figure 2.2 is explained in detail using ASN.1 message structures in sub section 2.1.

2.1.1 Kerberos Message Exchange and Specifications

This section presents the messages that are typically exchanged in a Kerberos version 5 (Kerberos 5) dialogue. We explain the essence of messages that are presented in the Kerberos RFC 1510[KN93] which provides an overview of Kerberos version 5 network authentication system. Kerberos 5 specification is fairly big and not all messages specified in RFC 1510 are compulsory in order to have a secure authentication dialogue. Some of the messages presented in this section are optional and can be optionally implemented in the application. Kerberos uses Abstract Syntax Notation One (ASN.1)[ASNa, ASNb] to define the various data structures and byte formats used in Kerberos communication. This section describes the exact contents and encoding of messages and objects that are involved in the Kerberos protocol. The ASN.1 class definitions in this paper are derived from the Kerberos RFC 1510[KN93].

Tickets and Authenticators

This section describes objects (tickets and authenticators) that are used in a Kerberos dialogue.

Ticket

A ticket is a record of items that help a client authenticate to a service. A Ticket contains the following information that is presented in ASN.1 definition:

```
Ticket ::=            [APPLICATION 1] SEQUENCE {
                      tkt-vno[0]                    INTEGER,
                      realm[1]                      Realm,
                      sname[2]                      PrincipalName,
                      enc-part[3]                   EncryptedData
}
-- Encrypted part of ticket
EncTicketPart ::=     [APPLICATION 3] SEQUENCE {
              flags[0]             TicketFlags,
              key[1]               EncryptionKey,
              crealm[2]            Realm,
              cname[3]             PrincipalName,
              transited[4]         TransitedEncoding,
              authtime[5]          KerberosTime,
              starttime[6]         KerberosTime OPTIONAL,
              endtime[7]           KerberosTime,
              renew-till[8]        KerberosTime OPTIONAL,
              caddr[9]             HostAddresses OPTIONAL,
              authorization-data[10]   AuthorizationData OPTIONAL
}
        -- encoded Transited field
TransitedEncoding ::=         SEQUENCE {
                      tr-type[0]  INTEGER, -- must be registered
                      contents[1]          OCTET STRING
}
```

The encrypted part of the ticket is encrypted by the key shared between communicating principals (for example, a client and ticket granting server). The fields represented in the ticket are further explained below based on Kerberos RFC 1510[KN93]

a. tkt-vno: This field specifies the version number for the ticket format. We assume Kerberos version 5 for our research in regard with the E-Banking application.

b. realm: This field specifies the realm from which the ticket was issued and also identifies the realm of the server that issued a ticket. Since a Kerberos server(KDC) can only issue tickets for servers within its realm, the two should always be identical.

c. sname: This field specifies the name part of the server's identity.

d. enc-part: This field holds the encrypted encoding of the EncTicketPart sequence.

e. flags: This field indicates which of various options were used or requested when the ticket was issued. It is a bit-field, where the selected options are indicated by the bit being set (1), and the unselected options and reserved fields being reset (0). RFC 1520 provides more details about the flags that can (re)set and their meaning.

f. key: This field exists in the ticket and the KDC response and is used to pass the session key from Kerberos to the application server and the client.

g. crealm: This field contains the name of the realm in which the client is registered and in which initial authentication took place.

h. cname: This field contains the name part of the client's principal identifier.

i. transited: This field lists the names of the Kerberos realms that took part in authenticating the user to whom this ticket was issued. It does not specify the order in which the realms were transited. See section 3.3.3.1 of RFC 1510[KN93] for details on how this field encodes the traversed realms.

j. authtime: This field indicates the time of initial authentication for the named principal. It is the time of issue for the original ticket on which this ticket is based. It is included in the ticket to provide additional information to the end service in case decisions have to be made on accepting the ticket based on initial time of authentication.

k. starttime: This field in the ticket specifies the time after which the ticket is valid. Together with endtime, this field specifies the life of the ticket. If it is absent from the ticket, its value should be treated as that of the authtime field.

l. endtime: This field contains the time after which the ticket will not be honoured (its expiration time). It is also important to note that individual services may place their own limits on the life of a ticket and may reject tickets which have not yet expired. As such, this is really an upper bound on the expiration time for the ticket.

m. renew-till: This field is only present in tickets that have the RENEWABLE flag set in the flags field. It indicates the maximum endtime that may be included in a renewal. It can be thought of as the absolute expiration time for the ticket, including all renewals.

n. caddr: This field holds host addresses from which the ticket can be accepted and used. Kerberos RFC specifies that if there are no addresses, the ticket can be used from any location. The policy of requiring addresses or not is a task that is left to service administrators, who may choose to refuse issuing or accepting such tickets. Network addresses are included in the ticket to make it harder for an attacker to use stolen credentials.

o. authorization-data: The authorization-data field is used to pass authorization data from the principal on whose behalf a ticket was issued to the application service. If no authorization data is included, this field will be left out. The data in this field are specific to the end service. It is expected that the field will contain the names of service specific objects, and the rights to those objects. The authorization-data field is optional and does not have to be included in a ticket.

Authenticators

An authenticator is a record sent with a ticket to a server to certify the client's knowledge of the encryption key in the ticket, to help the server detect replays, and to help choose a "true session key" to use with the particular session. The encoding is encrypted in the ticket's session key shared by the client and the server:

```
-- Unencrypted authenticator
Authenticator ::=    [APPLICATION 2] SEQUENCE    {
       authenticator-vno[0]          INTEGER,
       crealm[1]                     Realm,
       cname[2]                      PrincipalName,
       cksum[3]                      Checksum OPTIONAL,
       cusec[4]                      INTEGER,
       ctime[5]                      KerberosTime,
       subkey[6]                     EncryptionKey OPTIONAL,
       seq-number[7]                 INTEGER OPTIONAL,
       authorization-data[8]         AuthorizationData OPTIONAL
}
```

a. authenticator-vno: This field specifies the version number for the format of the authenticator in the Kerberos message exchange.

b. crealm and cname: These fields are the same as those described for the ticket in section 2.1.1.

c. cksum: This field contains a checksum of the the application data that accompanies the application service request 2.1.1.

d. cusec: This field contains the microsecond part of the client's time-stamp. Its value (before encryption) ranges from 0 to 999999. It often appears along with ctime. The two fields are used together to specify a reasonably accurate timestamp.

e. ctime: This field contains the current time on the client's host.

f. subkey: This field contains the client's choice for an encryption key which is to be used to protect this specific application session. Unless an application specifies otherwise, if this field is left out the session key from the ticket will be used.

g. seq-number: This optional field is used to the initial(random) and sequence numbers that can be used to detect replays.

h. authorization-data: This field is the same as described for the ticket in section2.1.1. It is optional and will only appear when additional restrictions are to be placed on the use of a ticket, beyond those carried in the ticket itself.

In a Kerberos network authentication service, three main types of message exchanges are required in order for a client to gain access to server services. These messages are listed below and further discussed in detail in the subsection that follows.

a. The Authentication Service exchange,

b. The Ticket-Granting Service (TGS) exchange and

c. The Client-Server Authentication exchange.

Authentication Service Exchange

When the client wishes to access services on a given application server, the client is required to first obtain authentication credentials from the Authentication Server (AS). The authentication service exchange is used at the initiation of the login to obtain credentials for a Ticket-Granting Server(TGS), which will subsequently be used to obtain credentials for the Application Server(V) services. The client uses a secret key to decrypt the message that is received from the Authentication Server. The exchange consists of two messages: KRB_AS_REQ(AS_REQ) from the client to the Authentication Server, and KRB_AS_REP(AS_REP) or KRB_ERROR(in case of an error) in reply.

Request(KRB_AS_REQ) from Client to Authentication Server

In this protocol state, a message is sent from the client to the Authentication Server to request credentials for a service.

```
KRB-AS-REQ ::=          [APPLICATION 10] KDC-REQ

KDC-REQ ::=         SEQUENCE {
            pvno[1]                 INTEGER,
            msg-type[2]             INTEGER,
            padata[3]               SEQUENCE OF PA-DATA OPTIONAL,
            req-body[4]             KDC-REQ-BODY
}
PA-DATA ::=         SEQUENCE {
            padata-type[1]          INTEGER,
            padata-value[2]         OCTET STRING,
}

KDC-REQ-BODY ::=    SEQUENCE {
             kdc-options[0]         KDCOptions,
             cname[1]               PrincipalName OPTIONAL,
```

```
                -- Used only in AS-REQ
        realm[2]            Realm, -- Server's realm
                -- Also client's in AS-REQ
        sname[3]            PrincipalName OPTIONAL,
        from[4]             KerberosTime OPTIONAL,
        till[5]             KerberosTime,
        rtime[6]            KerberosTime OPTIONAL,
        nonce[7]            INTEGER,
        etype[8]            SEQUENCE OF INTEGER, -- EncryptionType,
                -- in preference order
        addresses[9]        HostAddresses OPTIONAL,
        enc-authorization-data[10]   EncryptedData OPTIONAL,
                -- Encrypted AuthorizationData encoding
        additional-tickets[11]       SEQUENCE OF Ticket OPTIONAL
}
```

In the structure presented above, KRB-AS-REQ uses the structure of KDC-REQ for message components representation. A string, APPLICATION 10, follows in square brackets. APPLICATION 10 means that the AS-REQ structure is identified as number 10 among the different structures of this APPLICATION. KDC-REQ that follows [APPLICATION 10] implies that the structure AS-REQ follow the definition of the structure named KDC-REQ. The fields in this message are defined below:

a. pvno: This field is included in each message, and specifies the protocol version number.

b. msg-type: This field indicates the type of a protocol message. It will almost always be the same as the application identifier associated with a message. It is included to make the identifier more readily accessible to the application. In KRB-AS-REQ this identifier is 10.

c. padata: The padata (pre-authentication data) field consist of authentication information which may be needed before credentials can be issued or decrypted. This field may also contain information needed by certain extensions to the Kerberos protocol. For example, it might be used to initially verify the identity of a client before any response is returned. This is accomplished with a padata field with padata-type equal to PA-ENC-TIMESTAMP and padata-value defined as follows:

```
padata-type     ::= PA-ENC-TIMESTAMP
padata-value    ::= EncryptedData -- PA-ENC

PA-ENC   ::= SEQUENCE {
        patimestamp[0]              KerberosTime, -- client's time
        pausec[1]                   INTEGER OPTIONAL
}
```

d. padata-type: The padata-type element of the padata field indicates the way that the padata-value element is to be interpreted. Negative values of padata-type are reserved for unregistered use; non-negative values are used for a registered interpretation of the element type.

e. padata-value After obtaining the client's time with patimestamp, and pausec containing the microseconds which may be omitted if a client will not generate more than one request per second. The ciphertext (padata-value) consists of the PA-ENC sequence, encrypted using the client's secret key. PA-ENC can also be encoded AP-REQ(2.1.1 to hold ticket(TGT) and authenticator details.

f. req-body: This field is a placeholder delimiting the extent of the remaining fields. If a checksum is to be calculated over the request, it is calculated over an encoding of the KDC-REQ-BODY sequence which is enclosed within the req-body field.

g. kdc-options: This field appears in the KRB_AS_REQ 2.1.1 and KRB_TGS_REQ 2.1.1 requests to the KDC and indicates the flags that the client wants set on the tickets as well as other information that is to modify the behaviour of the KDC. For example, a client might need a forwardable ticket (a ticket that can be forwarded to a different KDC server). Similarly, a client might also request a renewable ticket (one that can be renewed after expiry).

h. cname and sname: These fields are the same as those described for the ticket in section.2.1.1 If sname is absent, the name of the server is taken from the name of the client in the ticket passed as additional-tickets.

i. enc-authorization-data: The enc-authorization-data, if present is an encoding of the desired authorization-data encrypted under the sub-session key if present in the Authenticator, or alternatively from the session key in the ticket-granting ticket, both from the padata field in the KRB_AP_REQ.

j. realm: This field specifies the realm part of the server's principal identifier. In the AS exchange, this is also the realm part of the client's principal identifier.

k. from: This field is included in the KRB_AS_REQ and KRB_TGS_REQ ticket requests when the requested ticket is to be postdated. It specifies the desired start time for the requested ticket.

l. till: This field contains the expiration date requested by the client in a ticket request.

m. rtime: This field is the requested renew-till time sent from a client to the KDC in a ticket request. It is optional.

n. nonce: This field is part of the KDC request and response. It it intended to hold a random number generated by the client. If the same number is included in the encrypted response from the KDC, it provides evidence that the response is fresh and has not been replayed by an attacker.

o. etype: This field specifies the desired encryption algorithm to be used in the response.

p. addresses: This field is included in the initial request for tickets, and optionally included in requests for additional tickets from the ticket-granting server. It specifies the addresses from which the requested ticket is to be valid.

q. additional-tickets: Additional tickets may be optionally included in a request to the ticket-granting server.

Reply(KRB_AS_REP) from Authentication Server to Client

The KRB_AS_REP message format is used for the reply from the KDC for reply from the authentication server. The key used to encrypt the ciphertext part of the reply depends on the message type. For KRB_AS_REP, the ciphertext is encrypted in the client's secret key, and the client's key version number is included in the key version number for the encrypted data.

```
KRB-AS-REP ::=    [APPLICATION 11] KDC-REP

KDC-REP ::=   SEQUENCE {
              pvno[0]                     INTEGER,
              msg-type[1]                 INTEGER,
              padata[2]                   SEQUENCE OF PA-DATA OPTIONAL,
              crealm[3]                   Realm,
              cname[4]                    PrincipalName,
              ticket[5]                   Ticket,
              enc-part[6]                 EncryptedData
}

EncASRepPart ::=    [APPLICATION 25] EncKDCRepPart

EncKDCRepPart ::=   SEQUENCE {
            key[0]                        EncryptionKey,
            last-req[1]                   LastReq,
            nonce[2]                      INTEGER,
            key-expiration[3]             KerberosTime OPTIONAL,
            flags[4]                      TicketFlags,
            authtime[5]                   KerberosTime,
            starttime[6]                  KerberosTime OPTIONAL,
            endtime[7]                    KerberosTime,
            renew-till[8]                 KerberosTime OPTIONAL,
            srealm[9]                     Realm,
            sname[10]                     PrincipalName,
            caddr[11]                     HostAddresses OPTIONAL
}
```

NOTE: In EncASRepPart, the application code in the encrypted part of a message provides an additional check that the message was decrypted properly.

a. pvno and msg-type: These fields are described above in section 2.1.1 and msg-type is KRB_AS_REP.

b. padata: This field is described in detail in section 2.1.1.

c. crealm, cname, srealm and sname: These fields are the same as those described for the ticket in section 2.1.1.

d. ticket: The newly-issued ticket, from section 2.1.1.

e. enc-part: This field is a place holder for the ciphertext and related information that forms the encrypted part of a message. The description of the encrypted part of the message follows each appearance of this field.

f. key: This field is the same as described for the ticket in 2.1.1

g. last-req: This field is returned by the KDC and specifies the time(s) of the last request by a principal.

h. nonce: This field is described above in section 2.1.1.

i. key-expiration: The key-expiration field is part of the response from the KDC and specifies the time that the client's secret key is due to expire. The expiration might be the result of password aging or an account expiration.

j. flags, authtime, starttime, endtime, renew-till and caddr: These fields are duplicates of those found in the encrypted portion of the attached ticket (see section 2.1.1).

The Ticket Granting Server(TGS) exchange

This section explains format and contents of messages that are exchanged in Kerberos between a client and TGS.

Request(KRB_TGS_REQ) sent to Ticket-Granting server (TGS)

```
KRB-TGS-REQ ::=         [APPLICATION 12] KDC-REQ

KDC-REQ ::=         SEQUENCE {
            pvno[1]                 INTEGER,
            msg-type[2]             INTEGER,
            padata[3]               SEQUENCE OF PA-DATA,
            req-body[4]             KDC-REQ-BODY
}
PA-DATA ::=         SEQUENCE {
            padata-type[1]          INTEGER,
            padata-value[2]         OCTET STRING,
                                    -- encoded AP-REQ
}
KDC-REQ-BODY ::=    SEQUENCE {
             kdc-options[0]         KDCOptions,
             cname[1]               PrincipalName OPTIONAL,
                                    -- Used only in AS-REQ
             realm[2]               Realm, -- Server's realm
                                    -- Also client's in AS-REQ
             sname[3]               PrincipalName OPTIONAL,
             from[4]                KerberosTime OPTIONAL,
             till[5]                KerberosTime,
             rtime[6]               KerberosTime OPTIONAL,
             nonce[7]               INTEGER,
             etype[8]               SEQUENCE OF INTEGER, -- EncryptionType,
```

```
                                    -- in preference order
            addresses[9]          HostAddresses OPTIONAL,
            enc-authorization-data[10]   EncryptedData OPTIONAL,
                                     -- Encrypted AuthorizationData encoding
            additional-tickets[11]       SEQUENCE OF Ticket OPTIONAL
}
```

Details of fields in the messages for this session are described in section2.1.1. Exceptional information to note is the non optional padata field, which was previously optional in KRB_AS_REQ(see sub section 2.1.1). The padata field encoded as AP-REQ(2.1.1 is used to hold authentication information which includes a ticket(TGT) and authenticator.

Reply(KRB_TGS_REP)from Ticket-Granting Server (TGS) to Client

In the KRB_TGS_REP, the ciphertext is encrypted in the sub-session key from the Authenticator, or if absent, the session key from the ticket-granting ticket used in the request. In that case, no version number will be present in the EncryptedData sequence.

```
TGS-REP ::=   [APPLICATION 13] KDC-REP

KDC-REP ::=   SEQUENCE {
              pvno[0]                    INTEGER,
              msg-type[1]                INTEGER,
              padata[2]                  SEQUENCE OF PA-DATA OPTIONAL,
              crealm[3]                  Realm,
              cname[4]                   PrincipalName,
              ticket[5]                  Ticket,
              enc-part[6]                EncryptedData
}
EncTGSRepPart ::=   [APPLICATION 26] EncKDCRepPart

EncKDCRepPart ::=   SEQUENCE {
            key[0]                       EncryptionKey,
            last-req[1]                  LastReq,
            nonce[2]                     INTEGER,
            key-expiration[3]            KerberosTime OPTIONAL,
            flags[4]                     TicketFlags,
            authtime[5]                  KerberosTime,
            starttime[6]                 KerberosTime OPTIONAL,
            endtime[7]                   KerberosTime,
            renew-till[8]                KerberosTime OPTIONAL,
            srealm[9]                    Realm,
            sname[10]                    PrincipalName,
            caddr[11]                    HostAddresses OPTIONAL
}
```

Description of fields in this message structure is similar to that of the exchange explained in section2.1.1.

The Client-Server Authentication Exchange

This section is used to explain messages that are exchanged between the client and server for acquiring services. application server.

Request(KRB_AP_REQ) sent to the application server from the client

The KRB_AP_REQ(AP_REQ) message contains the Kerberos protocol version number, the message type KRB_AP_REQ, an options field to indicate any options in use, and the ticket and authenticator themselves. The KRB_AP_REQ message is often referred to as the "authentication header".

```
AP-REQ ::=      [APPLICATION 14] SEQUENCE {
                pvno[0]                         INTEGER,
                msg-type[1]                     INTEGER,
                ap-options[2]                   APOptions,
                ticket[3]                       Ticket,
                authenticator[4]                EncryptedData
}

APOptions ::=   BIT STRING {
                reserved(0),
                use-session-key(1),
                mutual-required(2)
}
```

Kerberos RFC 1510 provides the following definitions for the fields in the client-to-server request:

a. pvno and msg-type: These fields are described above in section 2.1.1. msg-type is KRB_AP_REQ.

b. ap-options: This is a bit-field that appears in the application request (KRB_AP_REQ) and affects the way the request is processed by indicating the options that have been selected.

c. ticket: This field is a ticket authenticating the client to the server.

d. authenticator: This field contains the authenticator, which includes the client's choice of a subkey. Its encoding is described in section 2.1.1

Reply(KRB_AP_REP) from the application server to client request

The KRB_AP_REP message contains the Kerberos protocol version number, the message type, and an encrypted timestamp. The message is sent in response to an application request (KRB_AP_REQ) where the mutual authentication option has been selected in the ap-options field.

```
AP-REP ::=          [APPLICATION 15] SEQUENCE {
            pvno[0]                     INTEGER,
            msg-type[1]                 INTEGER,
            enc-part[2]                 EncryptedData
}

EncAPRepPart ::=    [APPLICATION 27]     SEQUENCE {
            ctime[0]                    KerberosTime,
            cusec[1]                    INTEGER,
            subkey[2]                   EncryptionKey OPTIONAL,
            seq-number[3]               INTEGER OPTIONAL
}
```

NOTE: in EncAPRepPart, the application code in the encrypted part of a message provides an additional check that the message was decrypted properly. The encoded EncAPRepPart is encrypted in the shared session key of the ticket. The optional subkey field can be used in an application-arranged negotiation to choose a per association session key.

a. pvno and msg-type: These fields are described above in section 5.4.1. msg-type is KRB_AP_REP.

b. ctime: This field contains the current time on the client's host.

c. cusec: This field contains the microsecond part of the client's timestamp.

d. subkey: This field contains an encryption key which is to be used to protect this specific application session.

e. seq-number: This optional field is used to hold the initial(that could be randomly chosen) and sequence numbers that are useful in detecting replay attacks.

2.2 Application Verification

In this section we present a mapping between the Kerberos message specifications and the E-Banking application source code. The work and discussion that is presented in this section is used as a foundation for obtaining a formal specification for the protocol that was implemented for the E-Banking application.

2.2.1 Kerberos Message Exchanges in E-Banking Application

In this section, we present classes and methods that implement standard Kerberos 5 message exchanges in the E-Banking application. In this section we relate Kerberos message specifications presented in section 2.1.1 to the E-Banking application and present classes and method that implement objects in Kerberos Version 5 specification and the message exchanges.
The methods that are used in the E-Banking systems to ensure that messages are in the ASN.1 format are defined by the ASN1DataTypes class. The ASN1DataTypes class handles all the low-level functionality required to author and process ASN.1 data structures. The class contains methods that author ASN.1 data structures and methods that are responsible for processing messages that have already been authored in ASN.1 received from remote principals. This class provides several methods for low level processing of message encoding and decoding.
Since a Kerberos dialogue consists of several messages that are exchanged between the client and server environment with an critical purpose of facilitating a secure communication, we need to use an example related to the E-Banking application to relate message structures and objects described in section 2.1.1 to the implemented classes and methods for the system. For the discussion in this section, we use an example of a client sending money to another account holder. Actually most of the operations for an E-Banking application are not completely implemented. For example there is no module for client to ask for bank balances; but the sending money module is sufficient for our discussion.
We present Kerberos dialogue in the E-Banking application using three phases, namely:

- Message exchange between client and Authentication server,
- Message exchange between client and Ticket-Granting server and
- Message exchange between client and Application server

Our discussion of message exchanges and the corresponding classes and methods that implement Kerberos version 5 details in the E-Banking application are presented in sub sections 2.2.1, 2.2.1 and 2.2.1. The descriptions in this section do not cover details on instantiating a Java Card applet onto a client's device. It is assumed that the E-Banking MIDlet and Java Card applet are already installed and that the user's key and bank key are also securely kept in the Java Card on the mobile device.
The messages have also been enumerated in order to have a diagrammatic representations of the exchange at the end of the Kerberos based dialogue. The diagram also acts as an affirmative check to the correct high-level implementation of Kerberos.

Message exchange between Client and Authentication server

1. The user launches a MIDlet application that has been installed on the mobile device of the user. The MIDlet is implemented by `KerberosEBank` class, which provides the E-Banking application interfaces for sending money and navigation menus. In case a user wishes to send money to another person, the mobile device user provides a username and password to the MIDlet application. The password is a shared secret that is known to the user and the Kerberos Server, technically known as the Key Distribution Center (KDC).

2. The MIDlet passes the username and password to the Kerberos client. Kerberos client is implemented as Java Card application in `JavaCardKerberosClient` class. It is the Kerberos client's duty to establish a context for secure communication with the E-Bank server. So the client has to author a Ticket Granting Ticket(TGT) request[1].

   Authoring a TGT request

3. The Kerberos client authors a request for the AS to issue a TGT. The Java Card Kerberos client authors a TGT request using the `getTicketResponse()` method, which is inherited(in the Object-Oriented sense) from the `KerberosClient` class.

```
response = kc.getTicketResponse (userName,
                                 kdcServerName,
                                 realmName,
                                 null,
                                 null);
```

   The name of the variable `response` is not to be confused with the request action that is being performed. The variable is used to hold the response from the Authentication Server (AS) which is part of the Key Distribution Center.
   We immediately realise in this request that the developers have included the username of the client(`userName`), the identity of the Authentication Server (`kdcServerName`) and the realm (`realmName`) of the client. The kerberosTicket and key fields are set to null which is understandable since the user's key is not supposed to travel across the network. We also note that the time settings and the nonce request are not included in the arguments that are provided to `getTicketResponse()` method, however these (time setting and nonce) arguments are catered for by local variables in `getTicketResponse()`as indicated in the code listing below. We also need to note repeated use of ASN1DataTypes class that is used to encode messages into Kerberos structures that we discussed in section 2.1.1.

```
byte pvno[] = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC, 1,
        getIntegerBytes(5));

msg_type = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC, 2,
           getIntegerBytes(10));
```

[1]Refer to subsection 2.1.1 for ASN.1 structural details of TGT request

```
byte kdc_options[] = getTagAndLengthBytes(
            ASN1DataTypes.CONTEXT_SPECIFIC, 0,
               getBitStringBytes(new byte[5]));

byte generalStringSequence[] = getSequenceBytes(
             getGeneralStringBytes(userName));
byte name_string[] = getTagAndLengthBytes(
             ASN1DataTypes.CONTEXT_SPECIFIC, 1, generalStringSequence);

byte name_type[] = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC,
             0, getIntegerBytes(ASN1DataTypes.NT_PRINCIPAL));

byte principalNameSequence[] = getSequenceBytes(concatenateBytes(
              name_type, name_string));

byte cname[] = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC, 1,
               principalNameSequence);

byte realm[] = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC, 2,
               getGeneralStringBytes(realmName));

byte sgeneralStringSequence[] = concatenateBytes(
               getGeneralStringBytes(serverName),
               getGeneralStringBytes(realmName));

byte sname_string[] = getTagAndLengthBytes(
               ASN1DataTypes.CONTEXT_SPECIFIC, 1,
               getSequenceBytes(sgeneralStringSequence));

byte sname_type[] = getTagAndLengthBytes(
               ASN1DataTypes.CONTEXT_SPECIFIC, 0,
               getIntegerBytes(ASN1DataTypes.NT_UNKNOWN));

byte sprincipalNameSequence[] = getSequenceBytes(concatenateBytes(
               sname_type, sname_string));

byte sname[] = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC, 3,
               sprincipalNameSequence);

byte till[] = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC, 5,
               getGeneralizedTimeBytes(new String("19700101000000Z")
                           .getBytes()));

byte nonce[] = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC, 7,
               getIntegerBytes(getRandomNumber()));

byte etype[] = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC, 8,
               getSequenceBytes(getIntegerBytes(3)));
```

```
byte req_body[] = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC,
                  4, getSequenceBytes(concatenateBytes(kdc_options,
                    concatenateBytes(cname, concatenateBytes(realm,
                     concatenateBytes(sname, concatenateBytes(till,
                      concatenateBytes(nonce, etype)))))));
```

4. The Kerberos client sends the request to the Authentication Server. Using datagram object in `getTicketResonse()` method of `KerberosClient`class the client prepares to send the message to the Authentication Server. After creating a Datagram object, `getTicketResponse()` method calls send() method to send the ticket request to AS.

   ```
   Datagram dg = dc.newDatagram(ticketRequest, ticketRequest.length);
   dc.send(dg);
   ```

   Datagram Connection variable `dc` is used to establish a connection with the KDC server once the application is launched on the J2ME device by a user.

   ```
   dc = (DatagramConnection)Connector.open("datagram://"
                          +kdcAddress+":"+kdcPort);
   ```

   The parameters passed to the DatagramConnection include the address of the KDC server (`kdcAddress`) and the port(`kdcPort`) on which the server is listening for connection.

Processing TGT request

5. The Authentication server is expected to extract the username of the client and the retrieve the user's password from it's internal database to be used to derive a session key. This derived key is used to encypt a response message to the client's request. The clients can also derive their uniques keys using password-username combination to decrypt the message that has been sent by the AS. Only the client the with the right password can decrypt the encrypted text portion of the TGT.

6. The AS sends the reply message to the requesting Kerberos client. We assume that the KDC will send back a correct message according to the structure represented by KRB_AS_REP in sub section 2.1.1. This message is expected to contain a two blocks of encrypted data. The first one being encrypted by a key based on the user's password, which should include the session key to be used between the client and TGS, time options and the identity of the TGS. The second encrypted block is a ticket itself that is expected to include the session key identifying for the client to the TGS.

Processing TGT reply

7. After receiving a response from the AS, the Java Card client now needs to extract the Ticket-Granting Ticket(TGT) and the session key shared between the client and the Ticket-Granting Server(TGS) from the reply. The MIDlet passes the encrypted portion of the TGT to the Java Card applet as shown in the code snippet below so that decryption of the AS response is processed.

```
tk = new TicketAndKey();
jcKClient = new JavaCardKerberosClient();
byte[] ticketCipher = jcKClient.getTicketCipher(response, tk);
```

Message exchange between client and Ticket-Granting Server(TGS)

Authoring a Service Ticket request

8. The `getTicketCipher()` method which takes the `response` from the Authentication Server(see item 2 above.) and TicketAndKey object as parameters performs the following actions in this dialogue:

   - It extracts the encrypted portion from the message and returns it to the KerberosEBank MIDlet.
   - The method sets the ticket (`ticketAndKey.setTicket(ticket);`) in the TicketAndKey object that was passed as `tk` to `getTicketCipher()` method.

   The session key that is sent by the Authentication also needs to be retrieved. This is obtained by the MIDlet client after obtaining reference to the key manager on the Java Card applet and then using the `getKey()` method of the `KerberosKeyManager` class on the applet. Now we have three items, the encrypted portion of the TGT, the ticket and the session key needed to communicate with the Ticket-Granting Server. The Kerberos client now needs to author a request for a service ticket. The structure of this message is defined in section 2.1.1, which includes a TGT and an encrypted structure known as an authenticator. Using the ticket and session key obtained from the message exchange between the client and the Authentication server, a new response is requested from the KDC server using the code snippet below. `tk` is an object of type `ticketAndKey` that is used to obtain the ticket and key from the Authentication Server response.

```
response = kc.getTicketResponse (
                              userName,
                              e_bankName,
                              realmName,
                              tk.getTicket(),
                              tk.getKey()
                            );
```

   We also take notice of the `if` block implemented in `getTicketResponse()` method that caters for parametes with a non null `KerberosTicket` field. The listing below provides details of that if block of code which this time is providing preauthentication data and an authenticationHeader that were not part of the message (2.1.1) previous sent to the Authentication Server since the `KerberosTicket` parameter was null.

```
if (kerberosTicket != null) {
   msg_type = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC, 2,
        getIntegerBytes(12));
   sname_string = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC,
        1, getSequenceBytes(getGeneralStringBytes(serverName)));
```

```
   sname_type = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC,
        0, getIntegerBytes(ASN1DataTypes.NT_UNKNOWN));

   sprincipalNameSequence = getSequenceBytes(concatenateBytes(
        sname_type, sname_string));

   sname = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC, 3,
        sprincipalNameSequence);

   byte[] req_body_sequence = getSequenceBytes(concatenateBytes(
        kdc_options, concatenateBytes(realm, concatenateBytes(
           sname, concatenateBytes(till, concatenateBytes(
          nonce, etype))))));

   req_body = getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC, 4,
         req_body_sequence);

   byte[] cksum = getChecksumBytes(
          getMD5DigestValue(req_body_sequence), getIntegerBytes(7));

   byte[] authenticationHeader = getAuthenticationHeader(
           kerberosTicket, realmName, userName, cksum, key, 0);

   byte[] padata_sequence = getSequenceBytes(concatenateBytes(
           getTagAndLengthBytes(ASN1DataTypes.CONTEXT_SPECIFIC, 1,
            getIntegerBytes(1)), getTagAndLengthBytes(
              ASN1DataTypes.CONTEXT_SPECIFIC, 2,
              getOctetStringBytes(authenticationHeader))));

   byte[] padata_sequences = getSequenceBytes(padata_sequence);

   byte[] padata = getTagAndLengthBytes(
          ASN1DataTypes.CONTEXT_SPECIFIC, 3, padata_sequences);

   ticketRequest = getTagAndLengthBytes(
          ASN1DataTypes.APPLICATION_TYPE, 12,
           getSequenceBytes(concatenateBytes(pvno, concatenateBytes(
            msg_type, concatenateBytes(padata, req_body)))));
} else {
   ticketRequest = getTagAndLengthBytes(
          ASN1DataTypes.APPLICATION_TYPE, 10,
          getSequenceBytes(concatenateBytes(pvno, concatenateBytes(
          msg_type, req_body))));
}//end of if block
```

The `getAuthenticationHeader()` method is used to build the `padata_value` structure explained in sub section 2.1.1. The `getAuthenticationHeader()` is used to

provide the TGT and build an authenticator encrypted with the session key extracted from the Authentication server's response to the Ticket-Granting Server. The authenticator certifies the client's knowledge of the session key. The service ticket request also specifies the name of the server (E-Bank's business logic server) which the client wishes to connect to.

9. The client sends the service ticket request to the TGS. The `getTicketResponse()` method is still responsible for sending the client message requesting the TGS for a service ticket. The explanation of objects used to send a service ticket request is identical to the one given in sub section 2 for requesting TGT.

```
Datagram dg = dc.newDatagram(ticketRequest, ticketRequest.length);
dc.send(dg);
```

Processing Service Ticket request

10. The Ticket-Granting Server is expected to process the client request and send back a response that is similar to that of the Authentication Server. According to the Kerberos standard message specification described in section 2.1.1, the Ticket-Granting Server (TGS) is expected to send a message containing a plain text block and two extra blocks encrypted with different keys. The first one is encrypted with session key shared between the client and the TGS so that the client can descrypt the message and read the details of that block. The contents of this block which notably include the session key to be used between the client and the application server are described in section 2.1.1. The second one is the service ticket that is encrypted with a server's secret key so that only the server can decrypt the ciphered text. A service ticket would typically contain a session key to be used for communication between the application server and the client, time options and the identity of the client that is expected to be the recipient of the service ticket.

11. The TGS then sends a reply for the service ticket request to the client. We also assume that this reply from the Ticket-Granting Server conforms to the standard Kerberos message specification.

Processing Service Ticket reply

12. The client now needs to process the message that has been received from the TGS. The code snippet below from `KerberosEBank` MIDlet is used obtain the Ticket and Key from the TGS response (`response`).

```
tk = kc.getTicketAndKey( response, tk.getKey());
```

The method `getTicketAndKey()` is define in `KerberosClient` class and takes two arguments, a response(`response`) from TGS and a decryption key obtained from the `tk` object using the getKey() method. The key that is used was obtained by the client from the Authentication Server and kept by TicketAndKey object `tk`.
The client also extracts the service ticket using `getTicketAndKey()` method.

Message exchange between Client and Application server

Authoring Service Request

13. Now the client can establish a secure communication context since it has obtained a session key to establish a connection with the Application Server(SS).
The `createKerberosSession()` method implemented in the `KerberosEBank` class handles aspect of creating a secure communication context. The `createKerberosSession()` method uses the parameters described below:

    - The `ticketContent` which is assigned the service ticket.
    - `clientRealm` that is assigned the realm of the requesting client.
    - `clientName` holds the name of the requesting client.
    - `sequenceNumber`that holds the sequence number of the message.
    - `encryptionKey` represents the session key that is to be used for secure message exchange between the client and application server.
    - `inStream` a DataInputStream for receiving message from the application server and
    - `outStream` a DataOutputStream for sending messages to the application server.

    The code below shows details of how the `createKerberosSession()` method is called by the MIDlet application.

```
boolean isEstablished = kc.createKerberosSession (
          tk.getTicket(),
          realmName,
          userName,
          i, tk.getKey(), is, os );
```

    On examining `createKerberosSession()` method we realise that the application uses Generic Security Services Application Programmers Interface (GSS-API) to establish a secure communication context with the application server.
    GSS-API enables enterprise applications that implement various dissimilar technologies with varying security requirements to authenticate clients across differing technologies. The GSS-API was designed by the Internet Engineering Task Force(IETF) to provide an application programming interface that provides authentication to communicating parties while insulating details of the underlying technology[KU00].
    RFC 1964[Lin] defines protocols, procedures, and conventions to be employed by peers implementing the GSS-API when using Kerberos Version 5 technology. RFC 1964 defines additional ASN.1 message structures that are to be used for message exchange between the server and the client.
    The context establishment token described in RFC 1964 is define as shown below.

```
InitialContextToken ::=
 [APPLICATION 0] IMPLICIT SEQUENCE {
         thisMech        MechType
                       -- MechType is OBJECT IDENTIFIER
```

```
                    -- representing "Kerberos V5"
          innerContextToken ANY DEFINED BY thisMech
                    -- contents mechanism-specific;
                    -- ASN.1 usage within innerContextToken
                     -- is not required
}
```

The first field(`thisMech`) inside the `InitialContextToken`, defines the security technology that is to be used, which is Kerberos Version 5 in the E-Banking application. The `innerContextToken` field defined in `InitialContextToken` contains either of the Kerberos messages KRB_AP_REQ or KRB_AP_REP) that we described in sub sections 2.1.1 and 2.1.1 respectively.

Processing Service Reply and Sending Service Request

If the `createKerberosSession()` method returns true, then we know that that the client has successfully established a secure session with the E-Bank server.

14. The client sends the message to the Application Server(E-Bank server) using the `sendSecureMessage()` method implemented in the `KerberosClient` class.
`sendSecureMessage()` method implements five parameters that are briefly discussed below:

    - `message` parameter was used in the implementation of `sendSecureMessage()` method to store value from the MIDlet input form.
    - `sub_sessionKey` this parameter is assigned the subsession key that is used to create an authenticator for the client. Kerberos messages that form the authenticator structure are encrypted with this key that was previous obtained from the Ticket-Granting Server as explained in sub section 9
    - `seqNumber` this parameter keeps track of the sequence of messages that have been sent.
    - `inStream` a DataInputStream for receiving message from the application server and
    - `outStream` a DataOutputStream for sending messages to the application server.

    The code below show details of how the `sendSecureMessage()` method is called by the MIDlet application.

```
byte[] rspMessage = kc.sendSecureMessage(
     "Transaction of Amount:"+txt_amount.getString()
        + " From: "+userName
        +" To: "+txt_sendTo.getString(),
        tk.getKey(), i, is, os );
```

    It is important to note that a ticket was already sent to the E-Banking server when a secure communication context was being established. This was achieved by the message sent to the E-Bank server, using `ticketContent` parameter that was passed to `createKerberosSession()` method. The service ticket contains the session key that is to be used between the client and the application server.

15. The E-Bank server is expected to decrypt the ticket using it's secret key and then extract the session key that can now be used to communicate with the client.

16. The E-Bank server also sends back a message to the client. This message is now encrypted with the session key that is shared between the client and the E-Bank server and the client needs to decode the message. This is achieved by the `decodeSecureMessage()` method that is implemented in the `KerberosEBank` class. The code below gives details of this implementation in MIDlet application.

```
String decodedMessage = kc.decodeSecureMessage(rspMessage,
                        tk.getKey());
```

Variable `rspMessage` was assigned to message contents that are returned by `sendSecureMessage()` method.
When `decodeSecureMessage()` method does not returns null, then it indicates that the message exchange with the E-Bank server was successful and the results of the exchange are displayed in the MIDlet using the `decodedMessage` variable.

Figure 2.3 below shows a break down of these messages and their sequence of occurrence in the E-Banking application.

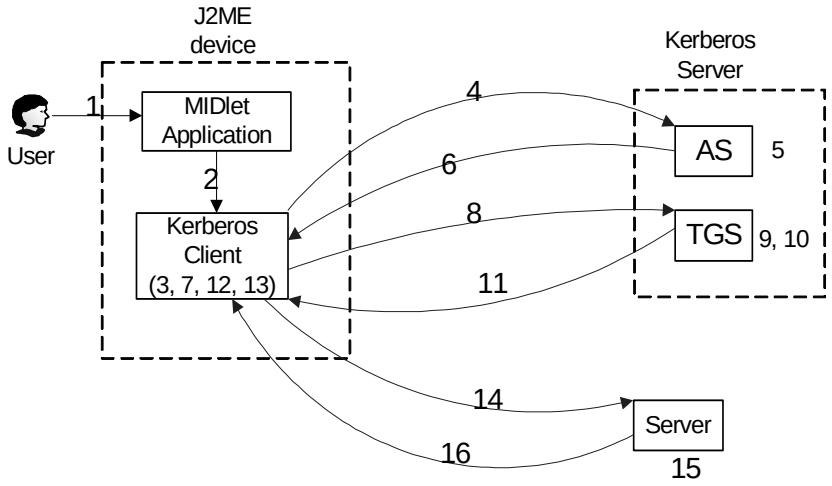

Figure 2.3: *Kerberos Message exchanges in E-Banking application*

2.2.2 E-Banking Application message exchanges

This section presents messages that are exchanged in a Kerberos dialogue involving the E-Banking client, Authentication Server, Ticket Granting Server and the Application Server. The messages presented here are derived from a study of the application source code and where source code cannot be available (for example in KDC implementations), we derive returned and expected message contents from the way in which the application code processes the replies . In circumstances where there is no explicit information that can be derived from the source code, we depend on Kerberos RFC 1510 to know the compulsory fields that the KDC is expected to set and return in the messages.

The symbols that are used in this section and their essence are described in section 2.1.1. The following abbreviations are used in the description of the protocol:

```
C = Client
AS = Authentication Server
TGS = Ticket Granting Server
V = Application Server (Services Server)
```

Authentication service exchange to obtain ticket-granting ticket

In this section, we present the message exchanges that are used to obtain a Ticket-Granting Ticket(TGT) from the Authentication Server(AS).

Client request message to Authentication Server :

1. In this section we present the information that the Client sends as a request to the Authentication Server. The messages presented in this section are based on the information we described from the application and presented in sub section "Authoring a TGT request"(2.2.1).

   $C \rightarrow AS : Pvno||Msg-type_1||Kdc-options||Name_C||Realm_C||Name_{AS}||$
   $Till||Nonce_1||Etype_C$

   Authentication Server reply message to Client request :

   This reply is opaque to us, but we can deduce the type of message that the application implementation expects from the Authentication Server based on how the response in (1) above is being processed in the application code and the message structure based of Internet RFC 1510 defined sub section 2.1.1.

2. $AS \rightarrow C : Pvno||Msg-type_2||Realm_C||Name_C||Tkt-vno||Realm_{AS}||Name_{AS}||$
   $Ticket_{TGS}||EncryptedData_1$

   $EncryptedData_1 = E_{K_{C,AS}}[K_{C,TGS}||Last{-}req||Nonce_1||Key{-}Expiration||Flags||Authtime||$
   $Starttime||Endtime||Endtime||Renew-till||Realm_{TGS}||Name_{TGS}||Caddr]$

   $Ticket_{TGS} = E_{K_{AS,TGS}}[Flags||K_{C,TGS}||Realm_C||Name_C||$
   $Authtime||Starttime||Endtime||Renew-till||Caddr]$

Ticket-granting service exchange to obtain service-granting ticket

In this section, we present the message exchanges that are used to obtain a Service-Granting Ticket from the Ticket-Granting Server(TGS).

Client request message to Ticket-Granting Server :

3. We present the information that the Client sends as a request to the Ticket-Granting Server. The messages presented in this section are based on the information we described from the application and presented in sub section "Authoring a Service Ticket request"(2.2.1).

   $C \rightarrow TGS : Pvno||Msg - type_3||Padata||Req - body$

   $Padata = Padata - type||Pvno||Msg - type_3||Ap - options||Tkt - vno||Realm_{AS}||$
   $Name_{AS}||Ticket_{TGS}||Authenticator_C$

   $Ticket_{TGS} = E_{K_{AS,TGS}}[Flags||K_{C,TGS}||Realm_C||Name_C||$
   $Authtime||Starttime||Endtime||Renew - till||Caddr]$

   $Authenticator_C = E_{K_{C,TGS}}[Auth{-}vno||Realm_C||Name_C||Cksum||Cusec_C||Ctime_C]$

   $Req - body = Kdc - options||RealmTGS||Name_{TGS}||Till||Nonce_2||Etype$

   Ticket-Granting Server reply message to Client request :

   The reply from the TGS is opaque to us but we take some assumptions based on the message the message that was sent to the TGS in (3) above and the Kerberos message specification in sub section 2.1.1. We also use the information that was extracted by the `getTicketResponse()` method to deduce the fields that the application expects to process from the Ticket-Granting Server. This information was discussed in sub section "Processing Service Ticket request"(2.2.1)

4. $TGS \rightarrow C : Pvno||Msg - type_4||Name_C||Tkt - vno||Realm_{TGS}||$
   $Name_{TGS}||Ticket_V||EncryptedData_2$

   $Ticket_V = E_{K_{TGS,V}}[Flags||K_{C,V}||Realm_C||Name_C||$
   $Authtime||Starttime||Endtime||Renew - till||Caddr]$

   $EncryptedData_2 = E_{K_{TGS,C}}[K_{C,V}||Last - req||Nonce_2||Key - expiration||$
   $Flags||Authtime||Starttime||Endtime||Renew - till||Realm_V||Name_V, Caddr]$

Client-Server authentication exchange to obtain service

In this section, we present the message exchanges that are used to obtain a service from the Application Server (V).

Client request message to Application(Service) Server :

5. In this section we present the information that the Client sends as a request to the Application Server. The messages presented in this section are based on the information we described from the application and presented in sub section "Authoring Service Request"(2.2.1).

   $C \rightarrow V : ThisMech||Pvno||Msg - type_5||Ap - options||Tkt - vno||Realm_{TGS}||$
   $Name_{TGS}||Ticket_V||Authenticator_C$

   $Ticket_V = E_{K_{TGS,V}}[Flags||K_{C,V}||Realm_C||Name_C||$
   $Authtime||Starttime||Endtime||Renew - till||Caddr]$

   $Authenticator_C = E_{K_{C,V}}[Pvno||Realm_C||Name_C||ChecksumBytes||Cusec_C||$
   $Ctime_C||K_{C,V}||Seqnumber]$

   Application(Service) Server reply message to Client request :

   In this section present the message that is expected to be received from the Application Server. The message exchange represented in this section is based on the information that we collected from the application and presented in sub section "Processing Service Reply and Sending Service Request"(2.2.1)

6. $V \rightarrow C : Pvno||Msg - type_6||E_{K_{C,V}}[Ctime_C||Cusec_C||K_{C,V}||Seqnumber]$

2.2.3 Automated Verification with AVISPA

In this section we present and discuss the results that were obtained in the verification phase. In order to perform verification of security properties for the implemented protocol, we performed a sequence of steps that are shown in figure 2.4 below. This model is generic and can be applied to any other system that is to be verified using AVISPA tool.

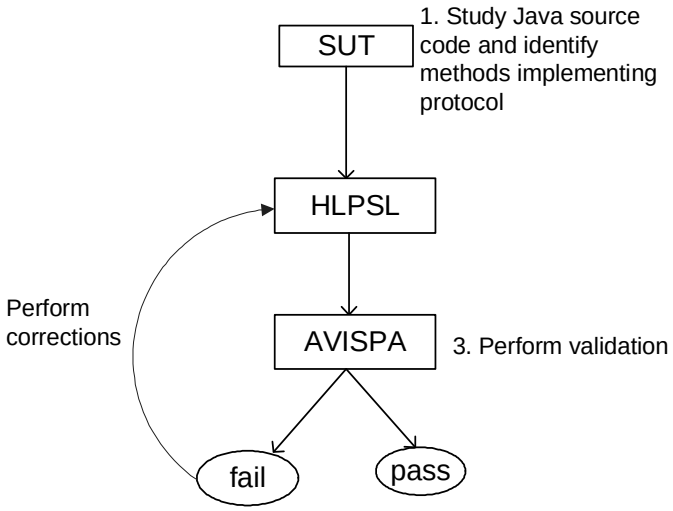

Figure 2.4: *Steps taken in verifying Java Card application in Avispa*

In order to perform verification of the implemented protocol, we needed to obtain an HLPSL model of the implemented protocol by first studying the source code for the application. After studying the source code (subsection 2.2.2 refers), the HLPSL model for the protocol implementation was developed to represent the desired security properties of the application and checked using AVISPA backend model checkers for syntax correctness. All corrections for corresponding syntax errors were subsequently done to finally obtain a model representing the protocol implemented in the E-Banking application.

The AVISPA tool consists of four model checkers[arm05], and we used two of them to perform automated verification of the implementation of Kerberos network authentication protocol in the E-Banking application. The two models used include the On-the-Fly Model Checker (OFMC) [BMV05] and SAT Model Checker (SATMC)[AL04] known as SATMC. The choice of these two model checkers was based on speed of analysis for OFMC and functionality(ability to perform step wise analysis) for SATMC.

The SATMC model checker was used to verify executability of all the states that are present in the modelled protocol. In figure 2.5 below, we present a framework for performing automated validation of the model using AVISPA. The HLPSL model is translated into an intermediate format(IF) using a translator known as hlpsl2if. The intermediate language IF is transparently read into the backend model checker.

The SATMC model checkers was invoked with the option `--check_only_executability=true`, so that it goes through the model to check whether all states are reachable. The action of checking whether all steps/stages in a model are executable helps to eliminate chances of obtaining false negatives which are usually a result of some parts of the protocol having not

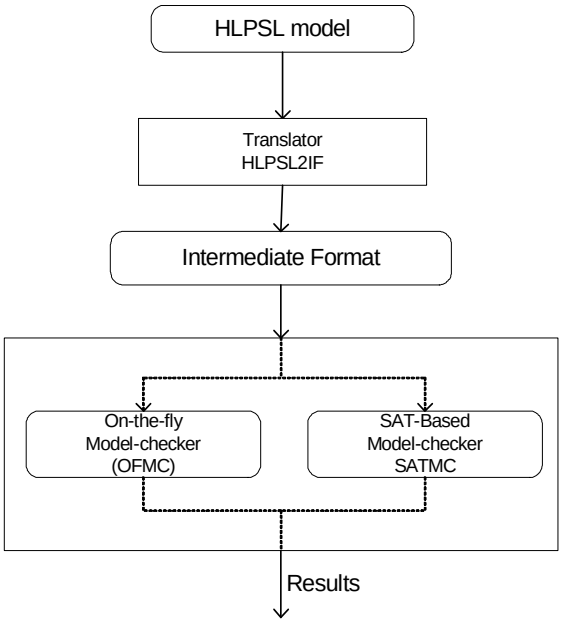

Figure 2.5: *Testing framework for passport applet*

been executed.

After confirming that the model is wholly executable, we invoked the On-the-Fly Model Checker (OFMC) to perform the actual validation work. In case any attacks were present in the HLPSL model, OFMC model checker would inform us and even provide the corresponding attack trace.

The results of this validation stage with AVISPA can be found in appendix A, which shows also the time it took to perform the validation for the implemented protocol. The results show that all the steps in the protocol were executable when checked by the SATMC model checker.

The results in appendix A also show that the HLPSL model for the implementation was correctly modelled and that all the steps could be executed. The results further indicate that the Kerberos version 5 implementation in the E-Banking application is safe and free of attack traces when checked with the OFMC model checker.

Chapter 3

E-Passport

In the previous chapter we saw the work that was done in order to perform verification of the Kerberos protocol implemented in the E-Banking application; in this chapter we present the E-Passport system (a form of Machine Readable Travel Document) that we used to analyse and perform conformance testing.

Travel documents like passports are used to ascertain identities of people that travel across borders. People with various interests always try to forge passports to beat the security mechanisms at border points of entry. The International Civil Aviation Organisation (ICAO)[1] spear headed the development of biometric passports in order to provide a more sophisticated solution for verifying authenticity of passports.

ICAO envisaged integration of biometrics in Machine Readable Travel Documents (MRTDs) as a means of providing better authenticity[ICA04c]. Biometrics are the automated use of physiological and behavioural traits in recognising the identity of an individual. Biometric passports also commonly known as E-Passports are basically passports equipped with a chip that contains biometric information of the holder. In February 2002, ICAO recommended face, iris and fingerprint as the biometrics applicable to MRTDs.

MRTD promise better security since regulating authorities can store biometric information of the passport holder on a smart card that is hard to counterfeit. Additionally, implementation of modern public key infrastructure (PKI) schemes and use of Digital Signatures enhances further the security of MRTDs. MRTDs are expected provide better mechanisms for verifying authenticity of passport than the non machine readable documents, that many people have successfully forged in the past.

The rest of this chapter is organised as follows. In section 3.1, the free implementation of MRTD that used in conformance testing is presented. In section 3.2 the inspection flow process that is specified by ICAO is discussed within the scope of this research. The basic access control protocol that is tested for conformance to the ICAO specification is discussed detail in subsection 3.2.1. Section 3.4 ends the chapter with the conformance testing procedure and results in subsections 3.4.1 and 3.4.2 respectively.

[1] http://www.icao.int/mrtd/Home/Index.cfm

3.1 JMRTD

JMRTD[2] is a free implementation by Radboud University Security of Systems research group[3] of Machine Readable Travel Documents that are informally specificed by the International Civil Aviation Organisation[4]. JMRTD is used in this thesis as a reference application for performing conformance testing of security protocols implemented in Java Card applications.

JMRTD simulates a typical environment in which an E-Passport can be authenticated at the border point of entry. The real border environment would consist of a terminal reader that is used to read information from physical E-Passports presented by their holders. The terminal reader (host side) also has a computer program that contains the logic for communicating with the E-Passport applet. E-Passports have smart cards embedded in them as earlier discussed. The smart cards contains PKI credentials, biometric information of the passport holders and an applet that contains the logic for communicating with the terminal. In JMRTD, the passport host API (terminal) is implemented in Java and the passport applet implemented in Java Card.

The architecture presented in figure 3.1 below, shows the use of Java Card Open Platform(JCOP)[5] as an emulator for a smart card communicating with a terminal performing basic access control (explained in detail in sub section 3.2.1) during the inspection flow process.

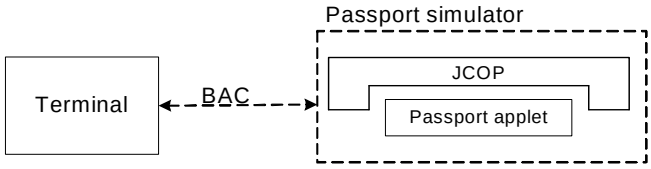

Figure 3.1: *E-Passport Architecture*

3.2 Inspection Flow Process

The inspection process flow is concerned with verification process of the passport and its holder. For our discussion in this section, the inspection terminal is sometimes abbreviated as IFD and the passport's chip referred to in abbreviation as ICC. These notations are adapted from the abbreviations from ICAO documents [ICA04b, ICA04a].

The following steps have to be carried out when a holder of a passport is at international boarders. The description of the measures presented in this section are discusses in detail in the PKI technical report [ICA04b] from ICAO. Some of the steps are optional, but the ICAO PKI technical report describes the sequence of steps to be taken, even when optional checks are implemented. The checks presented below are in the order expected to be followed at the inspection point of entry.

[2]http://jmrtd.sourceforge.net/
[3]http://www.cs.ru.nl/sos/
[4]http://www.icao.int
[5]http://www.zurich.ibm.com/jcop/

3.2.1 Basic Access Control(optional)

Basic Access Control(BAC) mechanism is specified as optional in the ICAO technical report so the descriptions do not apply to all Machine Readable Travel Document (MRTD)[ICA04b]. The chip in a travel document that is protected by basic access control, which denies access to chip contents until the inspection system proves that it is authorised to access the chip. A challenge response protocol is used to prove that the inspection system has knowledge of the MRTD-individual keys, K_{enc} and K_{mac} that are derived from information on the Machine Readable Zone (MRZ).

Basic access control is define by ICAO as a means of preventing skimming (unauthorised distant reading of the passport's chip) since the passport needs to be opened for having the MRZ read) as well as eavedropping on the communication channel between the passport and the inspection terminal.

Basic Access Control proves that the keys K_{enc} and K_{mac} present on the passport's chip are the same as those derived from the passport's MRZ. That's if the chip is authentic, so is the MRZ.

The messages presented in the enumeration below are exchanged between the terminal (IFD) and the passport chip (ICC) during basic access control.
The notation is in such a way that the party on the left handside of the right arrow sends a message to the party on the right handside of the right arrow and what follows after the full colon is the message that is sent by the party on the left hand side. The double bar(||) is used to represent concatenation of message entities in the exchange.

1. Terminal request for a challenge by sending the GET CHALLENGE command.
   $IFD \rightarrow ICC : GETCHALLENGE$

2. Chip generated a random nonce(RND.ICC) and sends it to the terminal.
   $ICC \rightarrow IFD : RND.ICC$

3. Terminal sends a MUTUAL AUTHENTICATION command to the passport chip. The MUTUAL AUTHENTICATION command is sent in the form of a message that is encrypted with a key E_{Kenc} that was generated by the terminal from the passport's MRZ. The encrypted message contains a random nonce(RND.IFD) that is generated by the terminal, a nonce RND.ICC that was earlier sent by the chip in the passport and the keying material (K.IFD).
   The terminal also computes a checksum of the encrypted material and concatenates it(checksum) with the cryptogram.
   $IFD \rightarrow ICC : E_{Kenc}[RND.IFD||RND.ICC||K.IFD]||$
   $MAC_{Kmac}[E_{Kenc}[RND.IFD||RND.ICC||K.IFD]]$

4. The chip verifies the checksum of $E_{Kenc}[RND.IFD||RND.ICC||K.IFD]$,
   decrypts $E_{Kenc}[RND.IFD||RND.ICC||K.IFD]$ and extracts RND.ICC to verify that IFD returned the correct value. The following message is sent back to the terminal

$ICC \rightarrow IFD : E_{Kenc}[RND.ICC||RND.IFD||K.ICC]||$
$MAC_{Kmac}[E_{Kenc}[RND.ICC||RND.IFD||K.ICC]]$

5. The terminal also verifies the checksum of $E_{Kenc}[RND.ICC||RND.IFD||K.ICC]$, decrypts $E_{Kenc}[RND.ICC||RND.IFD||K.ICC]$ and extracts RND.IFD to check that ICC returned the correct value.

The messages exchanged during basic access control protocol is presented in a message sequence diagram in figure 3.2 below.

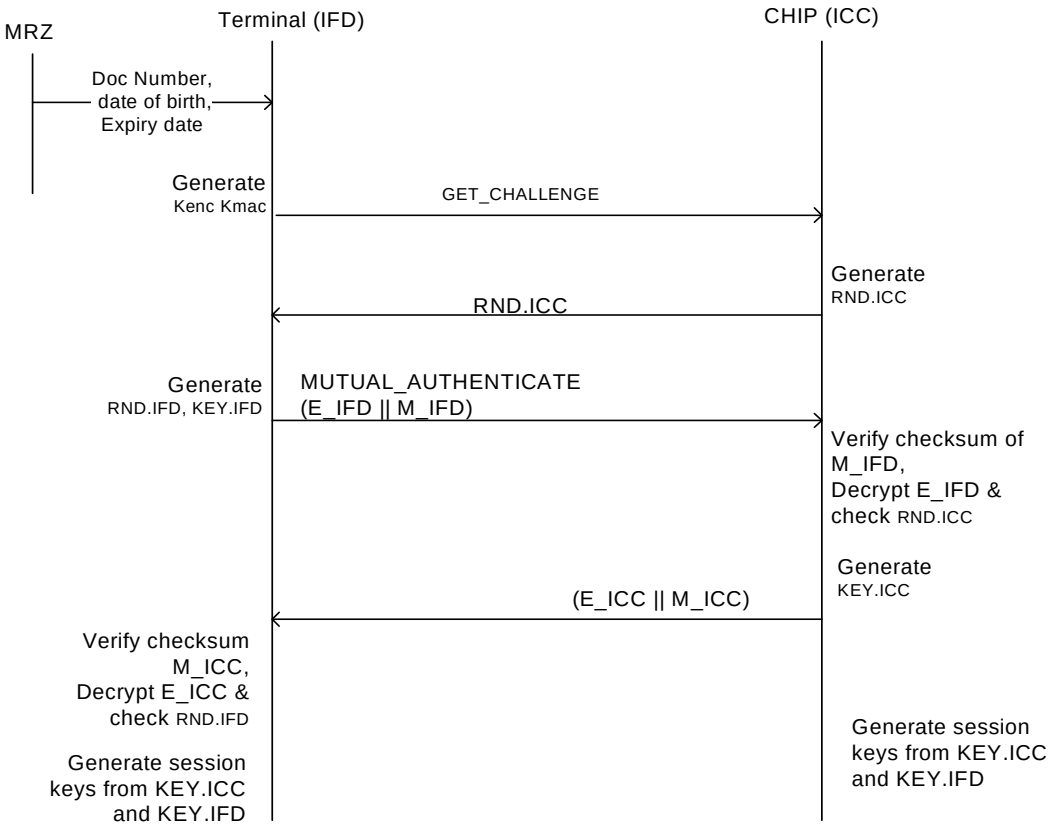

Figure 3.2: *Basic Access Control*

3.2.2 Passive Authentication (mandatory)

This a process of checking the signatures on the security document in the chip and serves the purpose of validating authenticity of the data on the chip, i.e., of the so-called Logical Data Structure (LDS) [ICA04a]. Passive authentication ensures that the unique passport number on the chip is authentic.

3.2.3 Active Authentication (optional)

Active authentication, the inspection mechanism that is used to detect a clone of the original passport.
This stage is initiated by the inspection terminal sending an INTERNAL AUTHENTICATION command to the passport chip. The INTERNAL AUTHENTICATION command takes a random nonce as input. The chip then encrypts the received nonce with a private key Kpr_{AA}, which resides in the read-protected memory of the passport chip. The encrypted nonce is sent back to the inspection system using secure messaging, since BAC was already performed. Having read the LDS during passive authentication, the inspection system has in particular read and authentified the public key Kpu_{AA}, which it can use to decode the chip's message. Clones of the existing passport cannot have the private key Kpr_{AA} from the chip's protected memory and will fail the above challenge response protocol.

3.3 Basic Access Control Validation

The validation of basic access control(BAC) protocol in the passport was done by Dr. Rusu[6] based on the specification information from ICAO. The specified protocol was found to be safe after validation with AVISPA[7].
In section 3.4, that follows, we present the conformance testing work that was done on the E-Passport(JMRTD).

[6] http://www.irisa.fr/vertecs/Equipe/Rusu/
[7] http://www.avispa-project.org

3.4 E-Passport Conformance Testing

In conformance testing process, we were interested in testing behaviour of the E-Passport during the Basic Access Control(BAC) process. The conformance testing process was performed to verify adherence of the passport applet to the specifications provided by the Internation Civil Aviation Authority [ICA04b].

In the sections that follow, we provide details of our finds and experiences of the conformance testing process for the E-Passport.

3.4.1 Automated Testing with TorX

Formal conformance testing was done using TorX to provide automatic test generation, test implementation, test execution and analysis[TB03]. In figure 3.3 below, we show a schematic diagram for the process of performing conformance testing.

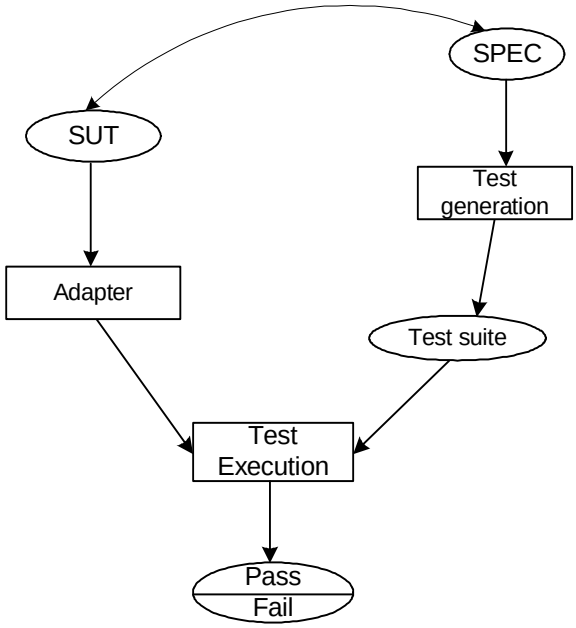

Figure 3.3: *Conformance testing process*

This figure is based on the conformance testing theory that we discussed in section 1.5.1 in which a formal specification of the system under test (SUT) is developed and used in the automatic generation of test cases.

Conformance Testing Process

In order to perform conformance testing with TorX, and adapter was developed to connect the SUT to the testing engine which compares behaviour of the SUT to the specification and then a deduction is made as to whether the implementation passed or failed the test.

In model-based testing we are required to have a specification of the implementation, which is expected to represent the behaviour of the SUT. The Promela specification for the SUT was developed to provide input to TorX that is used by the testing engine to automatically derive test cases.

The implementation of the E-Passport system is considered to be a blackbox which when presented with a stimulus is expected to give some form of response. TorX then observes the response after each stimulus and deducts whether the system conforms to the specification based on the expected out that was provided in the formal model.

Figure 3.4 below presents the testing framework that we used for conformance testing. The JMRTD API is used to connect the model-based testing tool (TorX) to the E-Passport via the JCOP Java Card framework. The Adapter in the testing framework is specific to

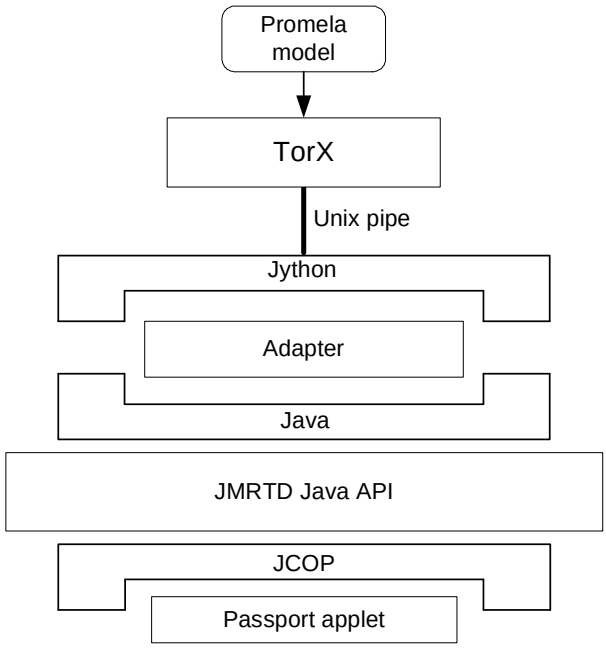

Figure 3.4: *Testing framework for passport applet*

the SUT and provides a connection to TorX testing engine. The adapter is responsible for sending or receiving inputs and outputs from TorX to the SUT and vice versa. The adapter performs encoding or decoding of messages from the SUT to TorX and vice versa. The process of encoding and decoding messages for communication between the SUT and TorX, clearly makes the Adapter specific to both the specification language and the SUT.

Promela models

A formal model(specification) of the system under test (SUT) was developed in Promela[8] based on the ICAO specification[ICA04b]. The behaviour of the E-Passport that is specified by ICAO is presented in figure 3.5 below. The behaviour is presented in form of a finite state machine, in which GC represents the GET CHALLENGE command and MA represents the MUTUAL AUTHENTICATION command. The label ok is used as a representation for success or 9000 and fail represents 6982 as specified by ICAO. We carried out initial tests using TorX with the model presented in figure 3.5 and the SUT passed all the test cases that TorX generated.

After the SUT had passed all the possible tests, the Promela specification was extended to check whether the E-Passport exhibited extra behaviour. This process was done step

[8]http://spinroot.com/spin/Man/promela.html

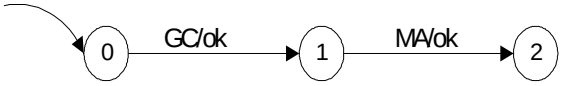

Figure 3.5: *BAC ICAO specification*

wise by changing small parts in the promela specification and then observing test results with TorX. A series of test that were performed are presented the figure below.

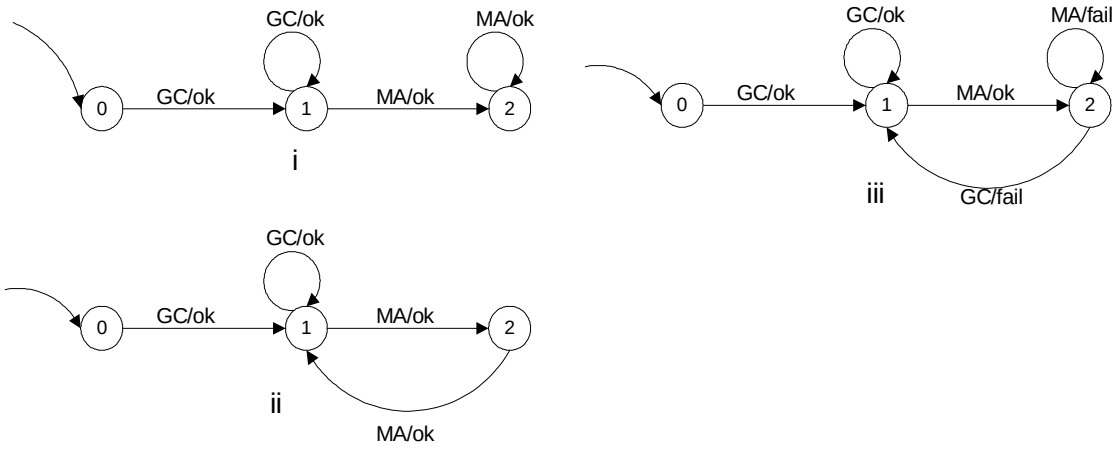

Figure 3.6: *Specifications for which some tests with the SUT failed*

i. In figure 3.6 TorX generates a verdict `fail` for the SUT because, specification `i` indicates that an observation of success is expected in state 2 after performing a MUTUAL AUTHENTICATION command (MA) yet TorX observes a fail from the SUT in state 2.

ii. TorX generates a `fail` verdict for the SUT with specification `ii` because of the reasons similar to `i` above. TorX observes a fail after MA from the SUT yet the specification indicates success.

iii. TorX generates a `fail` verdict for the SUT with specification `iii` because TorX observes successful execution of GET CHALLENGE command (GC) in state 2 yet the specification indicates that success is not expected.

In figure 3.7 below, the SUT passed all the tests that were performed. This was because the observed and expected behaviour of the SUT matched.

The final promela specification that was used to test the E-Passport is presented in Appendix B. The final Promela specification also considers testing passport behaviour for a passport that has wrong credentials from the machine readable zone just in case they were altered or forged.

The complete observed behaviour of the E-Passport is presented in figure 3.8 below as a finite state machine.

The message sequence chart presented in figure 3.9 shows that 13,123 steps were executed while performing automated testing with TorX. At that point we were satisfied with that the implementation (SUT) passed the test since we can not perform an test indefinitely.

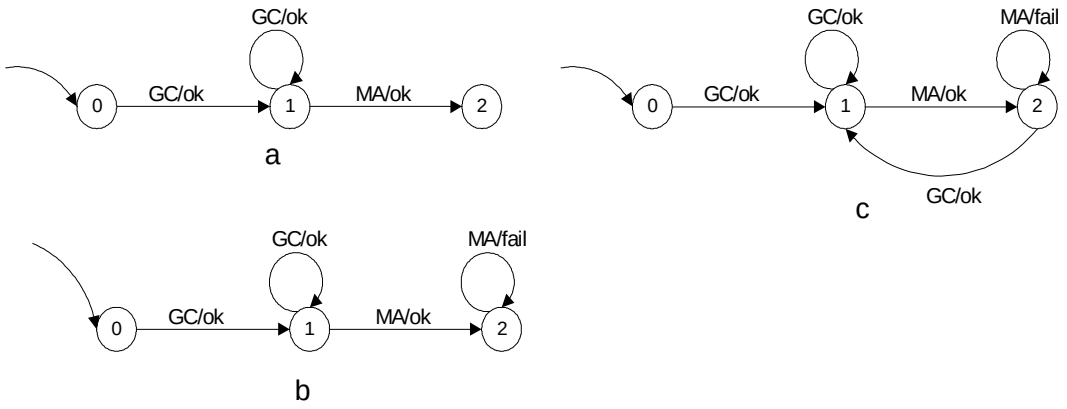

Figure 3.7: *Specifications for which tests with the SUT all passed*

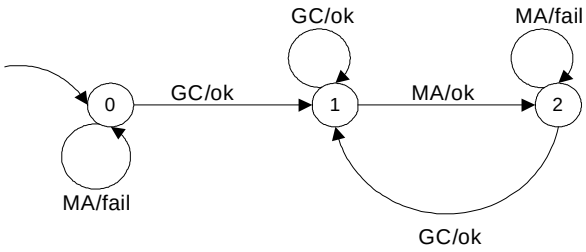

Figure 3.8: *Finite state machine model of observed passport behaviour*

3.4.2 Results Discussion

The final model of the E-Passport represented in figure 3.8 was obtained after performing experiments with models and observing behaviour using TorX and then making the necessary changes to the specification.

Since TorX is based on the ioco testing theory to define correctness as discussed earlier in section 1.6.2, we represent the ICAO specification of the E-Passport in figure 3.5 and the implementation of the E-Passport(i) in figure 3.8 as a labelled transition system in figure 3.10 below.

Implementation i which is represented as a labelled transition system in figure 3.10 is `ioco` correct with respect to the specification s since for all possible behaviours of the specification ($\forall\ \sigma\ \in\ Straces(s)$), any output x produced by the implementation ($x \in out(i\ after\ \sigma)$) can also occur as an output of the specification ($x \in\ out(i\ after\ \sigma)$). In otherwords, the definition of `ioco` ($i\ ioco\ s \Leftrightarrow_{def} \forall\ \sigma\ \in\ Straces(s) :\ out(i\ after\ \sigma) \subseteq out(s\ after\ \sigma)$) discussed in section 1.6.2 holds for the implementation of the E-Passport and ICAO specification presented in figure 3.10.

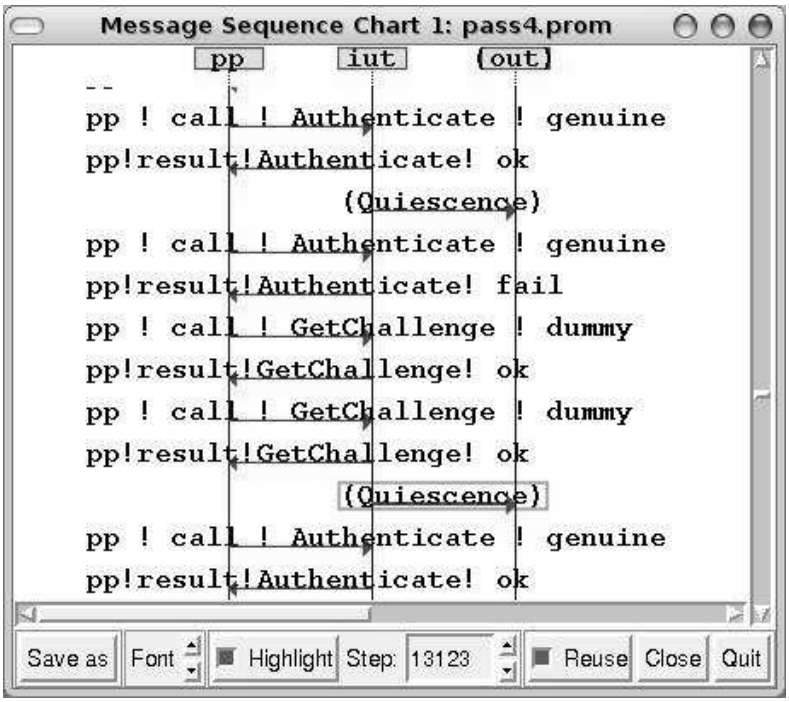

Figure 3.9: *TorX Message Sequence Chart*

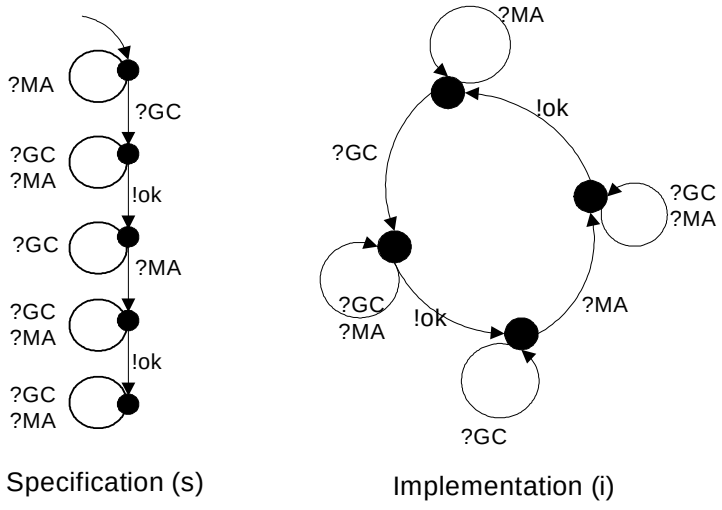

Figure 3.10: *E-Passport specification(s) and implementation (i)*

Chapter 4

Conclusions and Remarks

In this research, we used two real life implementations(E-Banking application and E-Passport) to investigate automated verification and model-based testing of network security protocols in Java Card applications. In our experience, we found benefits in using model-based testing and challenges as well that we present in this chapter. We noticed the following advantages in the use of model based testing:

i. It provides an organised and structured analysis of the system under investigation. The testing models show exactly the sequence of actions that take place in the system. This means that a tester or developer can use these models to either extend the application or tests for the system. Model-based testing provides a testing framework which is easy to extend in case the application has been changed.

ii. As indicated by different researchers in model-based testing, we also realised that model-based testing facilitates automatic generation of large quantities of tests cases that can be sent to the System Under Test.

iii. The model can be used to test for extended behaviour of the system. As in the E-Passport conformance testing stage, we extended the model to discover additional behaviour features that we presented in the finite state machine in section 3.4.2.

After this practical experience, we concluded that model-based testing can be done for network security protocols. We were also interested in the effect of model-based testing of Java Card applications to the testing process. Apparently, testing of applications based on Java Card technology did not pose any setbacks to the project work.

During this research, we appreciated the fact that, it is a good idea to perform validation of choices that have to be made before actually implementing the security protocol. This would require integrating model-based validation and verification in the system development life cycle.
In the example of large protocol like Kerberos, it would cost lots of development effort to redevelop a system that has been verified and found that is violated some security properties. The best option would be to first verify the implementation choices that have been made to be sure that the protocol is safe and that there are no security properties that have been violated.

We also realised that verification alone cannot prove that a system does not have a certain defect. However, formal verification can prove that a system has a certain property.

We were able to prove using formal verification that properties of secrecy of keys and authenticity existed in the E-Banking application. However, performing a successful formal verification did not mean that the product was correctly functional. Conformance testing was necessary to prove further correctness of the system.

A combination of formal verification and conformance testing, is good method for analysing the quality of a software product. The process of formal verification can prove that a system does not have a certain defect or does have a certain property based on a specification of the system. However, the specification says nothing about the behaviour of the actual implementation of the system in the environment in which it is supposed to operate. That is where conformance testing complements formal verification, since testing involves experimenting with the actual implementation of the system.

4.1 Outlook

During this research project, we encountered challenges and possible areas of improvement that we present in this section as possible areas of improvement and research.

After performing formal verification, it was not possible to re-use these models to perform conformance testing. Existing tools do not have the ability to use a particular formal verification model written in a language like HLPSL for conformance testing. We imagine a scenario in which a model which has been used to formally verify a system, being used by an automatic testing tool like TorX to generate test cases for the implementation.

Development of the Adapter to be used in automatic testing with TorX is still laborious. The process of encoding and decoding messages exchanges is still far from trivial. It is not yet clear how this task can be simplified to probably make the Adapter less application specific.

4.2 Remarks and experiences

In this research project, we gained experience in implementations of theoretical knowledge from cryptography. We also gained experience in systems migration and integration. Where developers had implemented the systems in Windows, we tried to migrate the systems in Unix environment to have an experience of how everything works.
It was interesting to see that even though systems like Java are platform independent, deployment of a system on either windows or Unix means that it may rely on platform specific services which could break the platform independence. This makes migration from one system onto another non trivial. An example was the E-Banking system, for which we were required to be members of a domain in order for the application to work. After installing everything on the Unix environment, setting up a domain controller and removing some of the bugs that were found in the system, limitations of the J2ME frame work could not allow us to run the system. The Security and Trusted Services API[1] which enables use of extended cryptography is only currently supported on windows.
The work around for this problem would have been running SATSA with a wine tool in Linux. However, wrong developer decoding of Abstract Syntax Notation One(ASN.1) messages from the Key Distribution Center(KDC) Server rendered the application unusable.

[1] http://java.sun.com/products/satsa

We could have continued debugging the application to make it usable, but this proved to be a wrong direction to follow for a tester.
Despite the challenges of the project, it was fun going through everything that we did. It was a very huge learning experience for the bug fixing part and studying code written by other people, performing validation and conformance testing.

Appendix A

Verification Results

The listing below shows the process and results of checking for executability of the steps in the HLPSL model. If any of the steps could not be executed then the SATMC model checker would tell us that the given step cannot be executed. Executability of these steps cannot be checked by the OFMC model checker

```
$ avispa EBank_Kerberos.hlpsl --satmc --check_only_executability=true

%% EXECUTABILITY TEST:

%% step_0: this rule can be executed..
%% step_1: this rule can be executed..
%% step_2: this rule can be executed..
%% step_3: this rule can be executed..
%% step_4: this rule can be executed..
%% step_5: this rule can be executed..
%% step_6: this rule can be executed..
```

The listing below shows results of validation using the On-the-Fly Model Checker(OFMC). The `SUMMARY` section in the listing show that the protocol is safe. If the protocol had violation of the desire security properties, then the attack trace would be printed out instead.

```
avispa EBank_Kerberos.hlpsl --ofmc
% OFMC
% Version of 2006/02/13
SUMMARY
  SAFE
DETAILS
  BOUNDED_NUMBER_OF_SESSIONS
PROTOCOL
  /home/results/EBank_Kerberos.if
GOAL
  as_specified
BACKEND
  OFMC
COMMENTS
```

```
STATISTICS
  parseTime: 0.00s
  searchTime: 4.76s
  visitedNodes: 330 nodes
  depth: 10 plies
```

Appendix B

Final promela specification

```
#ifdef TROJKA
#define OBSERVABLE observable
#else
#define OBSERVABLE
#endif

mtype = {
  dummy,
  call, result,
  ok, fail,
  genuine, invalid,
  Reset,
  Authenticate,
  GetChallenge }

chan pp = [0] of { mtype, mtype, mtype } OBSERVABLE;

proctype echo()
{
s0:     if
          :: pp?call(GetChallenge, dummy);
             pp!result(GetChallenge,ok); goto s1

          :: pp?call(Authenticate,invalid);
             pp!result(Authenticate,fail); goto s0

          :: pp?call(Authenticate,genuine);
             pp!result(Authenticate,fail); goto s0
        fi;

s1:

        if
          :: pp?call(GetChallenge, dummy);
```

```
            pp!result(GetChallenge,ok); goto s1

          :: pp?call(Authenticate,invalid);
             pp!result(Authenticate,fail); goto s1

          :: pp?call(Authenticate,genuine);
             pp!result(Authenticate,ok); goto s2
        fi;

s2:
        if
          :: pp?call(GetChallenge, dummy);
             pp!result(GetChallenge,ok); goto s1

          :: pp?call(Authenticate,invalid);
             pp!result(Authenticate,fail); goto s2

          :: pp?call(Authenticate,genuine);
             pp!result(Authenticate,fail); goto s0
        fi;
}

#ifndef TROJKA
proctype environment() {
}
#endif

/* --------------------------------- init ---------------------------------- */
init    {
atomic {
        run echo()
#ifndef TROJKA
        ; run environment()
#endif
        }
}
```

List of Figures

Bibliography

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425, 1990.

[AL04] Armando Alessandro and Compagna Luca. SATMC: a SAT-based Model Checker for Security Protocols. *Lecture Notes in Computer Science*, 4, 2004.

[arm05] The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. *17th International Conference on Computer Aided Verification, CAV*, pages 281–285, 2005.

[ASNa] Abstract Syntax Notation One. *http://www.itu.int/ITU-T/asn1/index.html*.

[ASNb] Abstract Syntax Notation One faqs. *http://www.faqs.org/faqs/kerberos-faq/general/section-13.html*.

[BHK05] Y. Boichut, P.C. Heam, and O. Kouchnarenko. Automatic Verification of Security Protocols Using Approximations. *Writing*, page 10, 2005.

[BMV05] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.

[CCJ03] H. Chen, JA Clark, and JL Jacob. Automated design of security protocols. *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, 3, 2003.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

[che03] A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols. *Proc. SAPS*, 4, 2003.

[GL98] Bella Giampaolo and Paulson Lawrence. Kerberos version iv: Inductive analysis of the secrecy goals. *Lecture Notes in Computer Science*, 1998.

[Gui04] ISO/IEC Guide. Standardization and related activities – General vocabulary . 2, 2004.

[HL03] R. Heckel and M. Lohmann. Towards Model-Driven Testing. *Electronic Notes in Theoretical Computer Science*, 82(6), 2003.

[Hol97] G.J. Holzmann. The Model Checker Spin. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 23(5):279, 1997.

[ICA04a] Development of a logical data structure (LDS) for optional capacity expansion technologies. *Technical Report LDS 1.7200405-18, Revision 1.7, International Civil Aviation Organization.*, May 2004.

[ICA04b] PKI for machine readable travel documents offering ICC read-only access. *International Civil Aviation Organisation, Technical Report, Machine Readable Travel Documents, PKI Task Force*, 2004.

[ICA04c] T.A.G.M. ICAO. NTWG, Biometric deployment of machine readable travel documents. Technical report, Technical Report Version 2.0 21, May 2004.

[KN93] J. Kohl and B.C. Neuman. The Kerberos network authentication service (version 5). *Internet Request For Comment RFC-1510, September*, 1993.

[Knu03] J. Knudsen. *Wireless Java: Developing with J2ME.* Apress, 2003.

[KU00] J. Kabat and M. Upadhyay. RFC2853: Generic Security Service API Version 2: Java Bindings. *Internet RFCs*, 2000.

[Lin] J. Linn. The Kerberos Ver. 5 GSS-API Mechanism. Technical report, RFC 1964,.

[Net06] SUN Developer Network. Java Card Technology. *http://java.sun.com/products/javacard/*, 2006.

[Ste88] Kerberos: An Authentication Service for Open Network Systems. *Proc. Winter USENIX Conference*, 1988.

[TB03] J. Tretmans and E. Brinksma. TorX: Automated model based testing. *1st European Conference on Model Driven Software Engineering*, pages 31–43, 2003.

[Tor02] TorX Test Tool Information. http://fmt.cs.utwente.nl/tools/torx/introduction.html. 2002.

[TPHT] R. Terpstra, L.F. Pires, L. Heerink, and J. Tretmans. Testing theory in practice: A simple experiment. *COST*, 247:168–183.

[Tre96] Jan Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[Tre99] J. Tretmans. Testing concurrent systems: A formal approach. *CONCUR*, pages 99–10, 1999.

[Tre04] Jan Tretmans. Testing techniques lecture notes. Radboud University Nijmegen, 2004.

[Wik06a] Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/formal_verification. 2006.

[Wik06b] Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/security_protocol. September 2006.