A Major Project Report On

Mutual Authentication on Java Card

Submitted in partial fulfillment of the requirements

for the award of the degree of

MASTER OF TECHNOLOGY
IN
SOFTWARE ENGINEERING

By

Om Prakash
(Roll No. 2K13/SWE/10)

Under the guidance of

Dr. Daya Gupta
Professor

Co-guide
Ms. DivyashikhaSethia
Assistant Professor
Department of Software Engineering
Delhi Technological University, Delhi

DELHI TECHNOLOGICAL UNIVERSITY
Department of Computer Engineering
Delhi Technological University, Delhi
2013-2015

CERTIFICATE

ACKNOWLEDGEMENTS

I feel immense pleasure to express my heartfelt gratitude to **Dr. Daya Gupta** and **Ms. DivyashikhaSethia** for her constant and consistent inspiring guidance and utmost co-operation at every stage which culminated in successful completion of my research work.

I also would like to thank the faculty of Software Engineering Department, DTU for their kind advice and help from time to time.

I owe my profound gratitude to my family which has been a constant source of inspiration and support.

Om Prakash

Roll No. 2K13/SWE/10

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

PLAID was introduced by Centrelink in 2009 and in a short time it has become one of the top choices for mutual authentication on Java smart cards. The protocol is cryptographically stronger, faster and more private, than most or all equivalent protocols currently available either commercially or via existing standards. PLAID is designed to perform a high strength mutual authentication in the 150-300 millisecond range, making it suitable for a range of mission critical contact-less applications.

This work is about the complete implementation of the protocol including the applet which runs on the card and the reader application which runs on the host machine. The prime focus has been on the implementation of the protocol and to get the real performance evaluation of the protocol. It discusses about the implementation processes and also the difficulties with solutions steps and processes which are carried out. On breaking the cumulative timings of the operations of the protocol in this implementation, it could be confirmed that the claims made by the protocol to carry out a cryptographically strong mutual authentication within 300 milliseconds can be easily achieved using the PLAID protocol.

Keywords: Mutual Authentication, java card, PLAID, OpenSSL

Chapter 1. Overview

1.1 Introduction

Mutual authentication, also called 2-way authentication is the process in which a client's identity is verified by the server and the same way the identity of server is verified by the client. Mutual authentication verifies that the message from the client reaches the intended server and in similar way the message from the server reaches the intended client. Generally mutual authentication is required before any type of access of a certain service. It is commonly used in the scenarios where a specific service is used by a number of clients. Mutual authentication facilitates in the verification process that the service is used by the intended user and not some malicious identity with the hidden intentions of destruction of the service. On the other way the clients need to verify that the services to be used are provided by the intended service provider not by some fraudulent server.

Java card in the basic terms is, adaptation of the Java platform to be used on smart cards. Java cards are subsets of the set of smart cards [2,12], plastic cards with embedded Integrated Chip (IC) on it. These smart cards can be classified in to categories connected and connectionless. Mobile SIM cards, credit cards, driving licenses and vehicle registration certificates (RC) and metro cards are some of the commonly available forms of smart cards in India. These smart cards are embedded with ICs which consists tiny processor, RAMs, EEPROMs etc. to provided computation capability. On the software front these cards have card OS over which a Java Card Virtual Machine (JCVM) and Java Card Runtime Environment (JCRE) works. On the top of that, Java card applications called 'Applet' works. A card could have a no. of Applets at a time but communicates with only one at a time.

Protocol used here in the thesis work for the mutual authentication is PLAID [8,9,10,13,14] (Protocol for Lightweight Authentication of Identity).We abbreviate the version number of the protocol and it would be just denoted by PLAID in the rest of this thesis report. The protocol was proposed by **Centrelink** in the year 2009 and in a sort time, it has become one of the top choices for mutual authentication on Java smart cards. It is a standard for smart card based mutual authentication in Australia. The protocol is cryptographically stronger, faster and more private, than most or all equivalent protocols currently available either commercially or via existing standards. PLAID is a candidate for broad usage in Physical and Logical Access Control Systems (PACS /LACS) where the requirement for a fast, private, ID leakage proof, secure, free and extensible smart card authentication protocol has grown significantly. PLAID is designed to perform a high strength mutual authentication in the 150-300 millisecond range, (0.15 to 0.3 of a second), making it suitable for a range of mission critical contact-less applications.

1.2 Motivation

Without any security measures in place, smart card based Java card might give away its information too easily. In a world where many applications utilize these cards in the system for identification and if these share information too easily, may lead to loss of privacy. In mutual authentication, the reader and cards does not reveal its identity unless some criteria has been met. Usually this type of authentication is proven through the knowledge of secret information shared between two entities. In addition to sharing a secret information, an affordable way to encrypt the data must be found as well as appropriate measures must be taken to prevent other types of attack such as message replay, eavesdropping etc. Now keep in mind the above things, we need to search for a protocol which meets our requirements. Several authentication schemes have been proposed and discussed, however our focus is on finding those which are more suitable for smart cards which are resource-constrained devices. Being resource-constrained devices, the amount of computation and the memory requirements should be kept as small as possible. The search for the authentication protocol ended at above discussed protocol more known as PLAID.

1.3 Thesis goals

In the thesis, we set out to implement the above said mutual authentication protocol PLAID, on our 'MicroSD embedded Java cards'[33]. The objective is to to find an optimal mutual authentication protocol and to implement the protocols specification as closely as possible the Java card, we have, which are made available by the smart card manufacturer and distributer Go-Trust, Taiwan. As mentioned in the official documentation and made available by the **Centrelink** [8,9], the protocol has been designed to get the authentication done within a time frame of 150 - 300 seconds, which has been tried to achieve on the card available with us.

The main contribution of the work is research of optimal mutual authentication protocols and implementation of PLAID mutual authentication on Java card (applet as well as a reader), Java Card devices run a subset of the Java language, tailored to suit resource-constrained devices. However, aside from the advantages of code portability and multi-application support, the use of an interpreted language does not come without a performance penalty. For selection we considered mutual authentication protocols such as EAP-PSK, EAP-TLS, EAP-FATS, EAP-IKEv2, PLAID and OPACITY. Of these, PLAID was found to be most suitable for mutual authentication scheme for cards (justification provided in section 2.5). Some benefits with PLAID are, its fast i.e. transaction for PLAID authentication is 150-300 millisecond, it is cryptographically strong, uses hybrid nature cryptography (RSA, AES, SHA1, SECURE_RANDOM), designed for smart cards so use APDU for encapsulating the data, could be used with different key sizes (16, 24, or 32 bytes), provides privacy protection and could work in three different modes such as 1-factor (normal), 2-factor(PIN hash) and 3-factor (minutiae).

1.4 Dissertation Organization

In this chapter we provide a brief work overview in terms of introduction and the thesis main goals. The remainder of this dissertation is organized as follows.

In Chapter 2 and chapter 3, we cover the topics and terms introduced in the above sections which need to be explained for better understanding of the dissertation. Topics such as mutual authentication and Java cards are explained with the required details so that reader could get all the related things in single go and does not feel to look around to get introduced to terms used in the dissertation work.

Chapter 4 covers the PLAID protocol used for authentication process. As this dissertation is all about PLAID so it’s quite fair to put it into a separate chapter.

Chapter 5 covers the aspects related to the implementation such as public key cryptography, RSA, symmetric key cryptography AES and hashing SHA1 used in the work to accomplish the requirement specifications of the PLAID. The end of the chapter contains the details related to the implementation work which consists breaking the problem in modules and achieving each module and finally aggregating the modules to achieve the goals. It also carries the challenges and their respective solutions, regarding an implementation on a Java Card platform.

Chapter 6 covers the details about the final results and observation which are encountered in this dissertation work.

Chapter 2. Mutual Authentication

In the following sections we will look at different authentication schemes which are built upon symmetric and public-key cryptosystems. The reader is not required to fully grasp the internal behavior of the cryptographic functions used, nevertheless, it is important to get familiarized with the computational costs and implementation challenges involved. We will cover the essential background; readers who want a deeper understanding of cryptographic theories, have at their disposal several good books devoted to cryptography, such as *Cryptography and Network Security* by William Stallings [6]. In the current chapter, we include some background details on mutual authentication process and technique. We would also provide some brief introduction about terms used throughout this dissertation work.

2.1 Introduction

Mutual authentication is technique through which a claimant (usually one who claim to identity) to show a verifier (the one who verifies the claim) that he is who he claims to be, in other words, an authentication scheme allows someone's identity to verified and hence any attempt of impersonation could be prevented. The term 'mutual' associated with the authentication signifies that it is a two-way process such as both the parties actively participate in the process of authentication. Identity of the Claimant is verified by the verifier and also the identity of the verifier is verified by the claimant. The terms Identification and Authentication are used interchangeably, however, the former refers to the process of establishing an identity, while the latter refers to process of linking the claimed identity to someone. For example, identification involves a claim of identity to which it requires to get associated “I am SRK”, while the authentication would be verification of that claim. Therefore, it could be considered that the Authentication is wider and inclusive for the term Identification. Found a suitable definition for the terms as:

Identification is the means by which a user provides a claimed identity to the system. Authentication is the means of establishing the validity of this claim [4].

The objectives of identification protocols have been listed as [5]:

1. In the case of honest parties A and B, A is able to successfully authenticate itself to B, i.e., B will complete the protocol having accepted A’s identity.

2. *(transferability)* B cannot reuse an identity exchanged with A so as to successfully impersonate A to a third party C.

3. *(impersonation)* The probability is negligible that any party C distinct from A, carrying out the protocol and playing the role of A, can cause B to complete and accept A’s identity. Here negligible means “is so small that it is not of practical significance”.

4. Points 1 to 3 remain true even if a (polynomial order) large number of previous authentications between A and B have been observed; the adversary C has participated in previous protocol executions with either or both A and B; and multiple instances of the protocol, possible initiated by C, may be run simultaneously do not even reveal any partial information which makes C’s tasks any easier whatsoever.

To provide a proof of identity, authentication can be based on several different factors, which can be used alone or combined. Some commonly used factors are:

1. Something *known*: something which is known to the individual. This secret information can be, for example, a password, a personal identification number (PIN), or a cryptographic key.

2. Something *possessed*: something that the individual owns. Tokens such as magnetic-stripe cards or smart cards are commonly used and of which Java smart cards would be discussed in the subsequent sections of this report.

3. Something *inherited*: something that the individual is, which usually refers to biometric data (e.g., handwritten signatures, facial features, fingerprints, retinal patterns, voice, etc.).

There are several characteristics of authentication protocols that must be addressed, such as:

1. *Reciprocity*: either unilateral or mutual authentication is possible, provided that

only one, or both entities provide a proof of identity, respectively.

2. *Computational efficiency*: the number of operations required to be executed per authentication process of the protocol.

3. *Communication efficiency*: this includes the number messages need to be exchanged between entities, as well as the bandwidth required (total number of bits transmitted) to complete the authentication.

The most widely used password authentication schemes, such as fixed password schemes and one-time password schemes, have deep roots in the areas of authentication, but they are still vulnerable to a variety of threats such as replay attacks and dictionary attacks, mentioned later in the chapter. In password authentication, the claimant proves her identity by demonstrating that she knows a secret, the password. However, revealing the secret makes it susceptible to interception by the adversary. Therefore, we focus on the cryptographic mechanisms available in smart cards which allow us to design stronger authentication schemes.

Available authentication techniques can be broadly divided into two categories, one is Challenge-Response technique and the other is Zero-Knowledge proof. These can also be categories as unilateral (unidirectional) Authentication and Mutual Authentication (bi-directional Authentication) techniques. As the name suggests, in unilateral Authentication only the claimant is verified and in same way in the latter both claimant as well as verifier are verified.

2.2 Challenge-Response techniques

In the Challenge-Response Authentication techniques, the claimant need not to present the secret to the verifier but needs to demonstrate the knowledge of the secret by correctly responding to the challenge thrown by the verifier. The response here could simply be a function of secrets and the challenge. Since every challenge is different similarly the response differs for the same secret or set of secrets. Even on monitoring the communications, the response from one execution of the authentication protocol could not provide an adversary with useful information for a subsequent authentication, as subsequent challenges will differ, thereby precluding replay attacks.

Challenge-Response based authentication techniques use different challenges by keeping the variations in time or using some randomly generated nonces, which changes for every subsequent authentication process. Without time-variant parameters, protocols are vulnerable to replay attack, interleaving attacks as well as chosen-text attacks. There are three main classes of time-variant parameters that can be used: random numbers, sequence numbers, or timestamps. The presence of random number or sequence number or timestamps provide some uniqueness (every attempt can be distinguished from the other), so attempts of replay attack can be easily captured). These are classified based on clock (timestamp and random or sequence). Both, these have their advantages and disadvantages; since the majority of smart cards lack internal time source therefore, they are not adequate for time stamp-based protocols.

Some challenge-Response based Authentication mechanisms also utilize the symmetric-key as well as the public-key cryptosystems. In symmetric key cryptosystems the secret key could be pre-shared between the involved entities and while the authentication process the challenge as well as the response is encrypted using the pre-shared keys. In the public-key based cryptosystems for challenge-response based authentication, claimant demonstrates the knowledge of its private key in any of two ways: the claimant made to decrypts a challenge encrypted using its public key or made to digitally sign a challenge which is verified by the verifier using the public of the claimant.

2.3 Zero-Knowledge Techniques

In zero-knowledge interactive proofs for Authentication, the claimant only needs to demonstrate the knowledge of the secret, and not anything else that might reveal or endanger the confidentiality of the secret. An interactive proof is said to be a proof of knowledge if it has both the properties of completeness and soundness.

A modern day example of zero-knowledge proof can be seen when a smart phone has been lost. Now Victor has found the smart phone and the device is locked with a pass-code or pattern. Peggy approaches and claims to be the owner of the phone (statement). Victor needs to prove the ownership and ask Peggy to unlock the device. Peggy does not want to reveal her pass-code or pattern, so she turns her back to Victor and unlocks the phone.

Seeing the unlocked phone, Victor now has proof that Peggy is the rightful owner. Just as Peggy has provided zero knowledge of her personal pass-code while proving ownership.

Completeness If the statement is true, the honest verifier will be convinced of this fact by an honest prover.

Soundness If the statement is false, no cheating prover can convince the honest verifier that it is true, except with some small probability.

Zero-knowledge If the statement is true, no cheating verifier learns anything other than this fact.

The general structure of zero-knowledge protocols is the following:

$$A \to B : \text{witness}$$

$$B \to A : \text{challenge}$$

$$A \to B : \text{response}$$

The entity claiming to be A selects a random number from a predefined set, as its secret commitment, from which he computes the witness. This mechanism provide randomness which allows to distinguish different protocol runs. Upon reception of the witness, B issues a challenge to which only the legitimate party A can provide a correct response. To decrease the probability of successful cheating, the protocol is iterated if necessary.

2.4 Authentication Protocols

2.4.1 EAP-PSK

The EAP Pre-shared key is a mutual authentication protocol defined in RFC 4764[36]. Below figure 1 shows a representation of the message exchange in EAP-PSK between a server and a peer (Note: these terms are just the names that are used in the standard). The ID_S and ID_P is the identity information of the server and peer respectively. The RAND_S and RAND_P fields are the two random fields for the dynamic challenges.

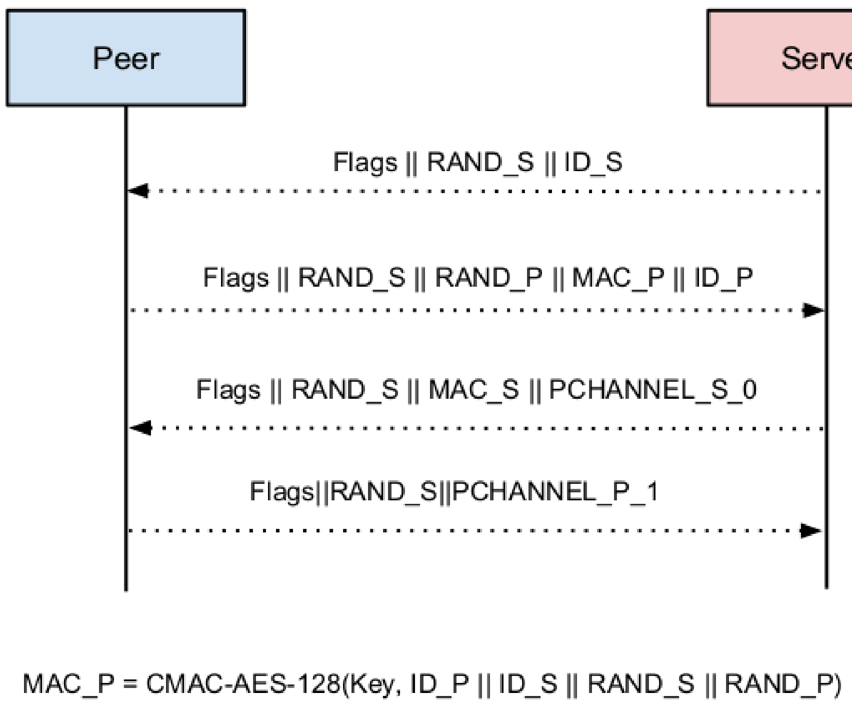

Peer
Server
Flags || RAND_S || ID_S
Flags || RAND_S || RAND_P || MAC_P || ID_P
Flags || RAND_S || MAC_S || PCHANNEL_S_0
Flags||RAND_S||PCHANNEL_P_1
MAC_P = CMAC-AES-128(Key, ID_P || ID_S || RAND_S || RAND_P)

Figure 1 Representation of EAP-PSK message exchange

T

he responses that accomplish the authentication are named MAC_S (for the server) and MAC_P (for the peer). Figure 1 shows how these response values are calculated. The CMAC-AES-128 function is a MAC based on the AES encryption algorithm with a 128 bit key size. The server and peer can verify the received MAC_P and MAC_S respectively by calculating the values themselves and compare it to the received one. If the received and calculated are values match, the authentication is successful.

Additional information (PCHANNEL_S_0 and PCHANNEL_S_1) is exchanged in order to set up the secure channel that can be used after the authentication is completed.

2.4.2 EAP-TLS

EAP Transport Layer Security (EAP-TLS) [37] is a PKI based authenticated scheme which uses X.509 certificates. EAP-TLS is based on TLS which is widely used in the Internet to provide a secure channel in HTTP, FTP and e-mail connections. EAP-TLS provides mutual authentication, protected cipher suite negotiation, privacy and automatic key management. Key management refers to the automatic establishment of a key(s) for the symmetric encryption algorithm.

Figure 2 illustrates the message exchange between the client and a server. The client first sends a ClientHello which includes a random value (which is later used as challenge) and a list of supported cipher suites. The server responds by sending five messages.

The ServerHello contains the server's challenge and a selection from the list of cipher suites that was sent by the client. The Certificate message contains the X.509 certificate of the server. The ServerKeyExchange is optionally sent when the certificate does not contain enough information for the client to send information confidentially. The CertificateRequest message is sent if the server wishes to authenticate the client (mutual authentication). The ServerHelloDone indicates that the server has sent all its messages.

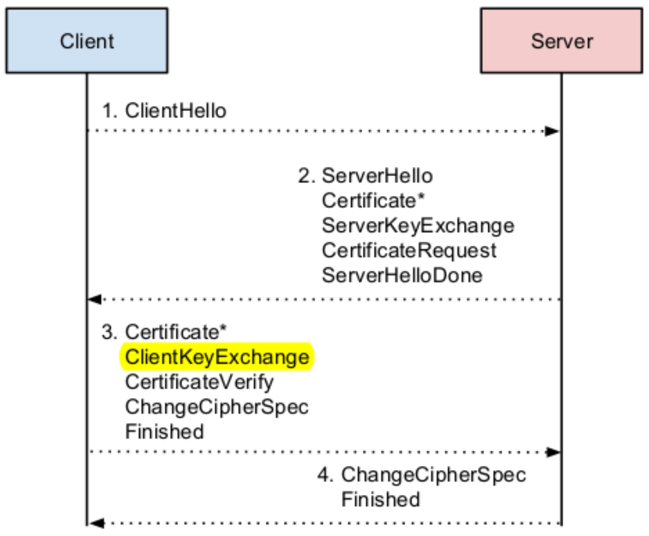


Figure 2 Representation of EAP-TLS message exchange.

The client processes the server's response. Most importantly, it will verify the trustworthiness of the received certificate. If the server requested it, the client will send its certificate. The ClientKeyExchange contains a pre-master secret. The pre-master secret consist of another random value generated by the client and of the TLS version number. The server and client both calculate the master secret from the pre-master secret and random values that were exchanged in the hello messages. The master key is used as a session key in the secure channel.

The CertificateVerify message authenticates the client to the server. This is done by hashing and encrypting all data that is sent until now (this includes the random value /

challenge) with the client’s private key. The ChangeCipherSpec message indicates that all subsequent traffic will be secured with the negotiated parameters (a secure channel with the selected cipher suite and calculated master key). The Finish message is the first message in the secure channel which allows to server to verify that it is working properly.

The server responds with a ChangeCipherSpec and a Finish message. In case of the server, the purpose of the Finish message is authentication to the client by showing that it was able to decrypt the pre-master secret.

Optionally, EAP-TLS provides privacy. This is accomplished by executing the TLS handshake twice. In the first handshake, the server only authenticates to the client. When this authentication is successful, the resulting secure channel is used to perform a second handshake in which both peers mutually authenticate. This makes it impossible for an unauthorized party to obtain the client’s certificate. In [75] an alternative approach for privacy in EAP-TLS is suggested which leverages the TLS extension mechanism. This approach is more efficient because it does not require the handshake to be executed twice.

2.4.3 EAP-TTLS

The EAP Tunneled Transport Layer Security (EAP-TTLS) [37] is similar to EAP-TLS and thus PKI-based but it provides backwards comparability with password based authentication.

This scheme describes mutual authentication between a user and authenticator. Only the authenticator has a certificate. The user verifies this certificate in order to make sure that the authenticator is legitimate after which a secure channel is set up as normally done in TLS. Subsequently, this secure channel is used to send the user’s password to the authenticator as if it was application data after which the user is authenticated.

2.4.4 EAP-IKEv2

The EAP Internet Key Exchange Protocol is defined in [38] and based on IKEv2 which is defined in [39]. IKE is used in IPsec in order to perform mutual authentication and setting up a security association.

EAP-IKEv2 is similar to EAP-TLS as it provides similar functionality. It supports X.509 certificate for authentication which are either pre-shared or distributed using domain name systems (DNS). Furthermore, it supports cipher suite negotiation, automatic key management and identity confidentiality (depending on the mode of operation).

A number of extensions to IKEv2 are documented which provide additional functionality. Session resumption allows the continuation of a session between a peer and authenticator after a failure without going through to the complete set up process [40]. IKE redirect enables the redirection of incoming request to other servers in order to perform load balancing [41]. EAP-IKEv2 extends IKEv2 by including support for password based authentication.

2.4.5 PLAID

PLAID [8, 9, 10, 13] has been found most suitable for mutual authentication on Java card, in the work PLAID has been used as the authentication method has been discussed in detail in chapter 4.

2.4.6 OPACITY

Open Protocol for Access Control, Identification, and Ticketing with privacy (OPACITY) [42] is an open authentication protocol targeted at smart cards. Its application include access control and also ticketing in public transport systems. OPACITY is compliant with many smart card related standard/recommendation documents including ISO 24727. It offers mutual authentication (depending on the mode), support for multiple cipher suites, end-to-end confidentiality and a key-agreement protocol that supportsforward secrecy.

As opposed to PLAID, support for multi factor authentication is not included. Furthermore, support for the exchange of authorization based information (PLAID enables this by using the ACS record as explained in chapter 4) is not included.

OPACITY relies on PKI in order to provide authentication. Every card and terminal are pre-loaded with a unique signed Card Verifiable Certificate (CVC), a corresponding private key and a number of trusted root certificates. The CVC contains the identity of the card or terminal and is defined according to the X.509 standard. The authentication process is similar as explained in Section 5.3.2: the certificates are exchanged and the signature is validated. Subsequently, a challenge/response mechanisms is used to verify that a node has the private key corresponding to the certificate that it presented.

OPACITY also supports identity privacy meaning that an attacker cannot read the identity (CVC) from the card. This is accomplished by encrypting the card's CVC symmetrically before it is sent to the terminal. The symmetric key is derived using a key agreement protocol called Elliptic curve Diffie-Hellman (ECDH). The exact operation of this protocol is not discussed here but this protocol makes sure that only a valid terminal can read the CVC.

Persistent binding is an optimization feature. The idea is to save time in the key agreement phase. When a particular card is presented to a particular terminal for the first time, the session key is derived and stored in both the card and the terminal together with one time card identifier (referred to as the PB record). This key and identifier can be used the next time the card is presented to the same terminal which will speed up the transaction time.

Two modes of operation are defined:

1. **OPACITY-FS** - This mode is designed for use in applications with high security requirements. Therefore, it supports mutual authentication, end-to-end confidentiality and forward secrecy.

2. **OPACITY-ZKM** - This mode is designed for applications where short transactions times are required. It does not offer mutual authentication: the terminal will only authenticate the card and not vice versa which decreases the transaction time but comes at the cost of security.

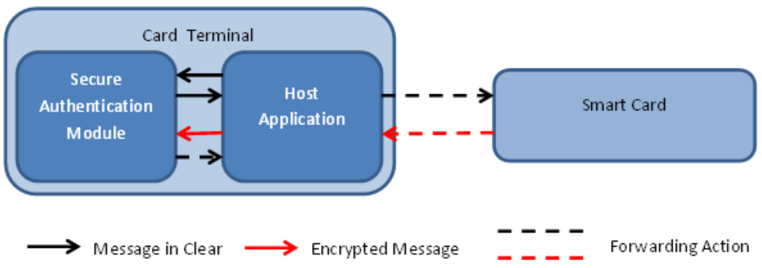


Figure 3 Architecture of OPACITY based system

The architecture of OPACITY based system is shown in Figure 3. The terminal consists of the Secure Authentication Module (SAM), which is responsible for authentication and cryptographic functions, and the client application which implements the service/business logic.

This architecture allows for the easy integration of OPACITY in existing client applications. During the authentication process, the client application acts as an interface between the card and the SAM meaning that it simply relays the traffic between these two entities. After the authentication, the client application can communicate with the card over a secure channel by letting the SAM encrypt and decrypt the data.

To the best of our knowledge, OPACITY is currently not widely deployed and therefore its quality is not proven in the field. Hence, we depend on scientific publications that evaluate this protocol. A cryptographic analysis of OPACITY is given in [43]. In this work it is concluded that there are restrictions to the privacy guarantees (identity privacy). Furthermore, according to [43], it is not recommended to use OPACITY-ZKM mode for deployment due to weak authentication and identity privacy in this mode.

In [13], three limitations of OPACITY are described.

1. **High transaction times** - The asymmetric encryption used is slow compared to symmetric encryption due to limited processing capabilities in smart cards. This results in higher transaction times.

2. **No CVC revocation list** - Typically, in PKI-based systems a list of revoked certificates is maintained in a certificate revocation list (CRL). Certificates on this list should not be trusted any more. OPACITY does not use a CRL or similar mechanism. Effectively, this means that when a smart card is stolen, there is no way to mark a card as untrusted. This will make it difficult to prevent that an attacker with a stolen card gains access illegally.

3. **No support for multi factor authentication** - In an environment where security requirements are strict, multi factor authentication is often desirable. The lack of support for this means that OPACITY cannot be used in an environment where this feature is required.

2.5 Comparison of protocols

From all of the above protocols, PLAID and OPACITY are the only one designed specifically to be used with smart cards. As these are designed for cards so use APDU for data encapsulation. Both the protocols, standards mention explicit protection against MiTM, (mentioned in the following section). PLAID and OPACITY also do not use passwords. Hence, it can be concluded that these protocols are protected against dictionary attacks. EAP-PSK is protected against dictionary attacks because no passwords are used in this protocol. PLAID provides high privacy protection. EAP variants are not specifically designed, but these are made compliant to be used on smart cards. The performance of the open source java card implementation of EAP-TLS is discussed in a master thesis [44]. The TLS handshake takes 10.5 seconds to complete in this implementation (for ’typical’ cipher suite TLS_RSA_WITH_RC4_128_SHA). This is high compared to the authentication times that are common in smart card specific protocols such as PLAID (300 milliseconds). Therefore, it is concluded that TLS-based protocols do not satisfy this requirement. EAP-PSK, PLAID and OPACITY do not support cipher suite negotiation. EAP- PSK does not provide any “End-user identity protection”.
It is found that PLAID and OPACITY are most compliant to be used with smart cards. From these two PLAID comes out be better as compared to OPACITY. OPACITY has higher transaction time, no CVC revocation list i.e. it does not provide any mark for card that is untrusted, in case card stolen or lost, it does not provide multiple operation. So, in these terms PLAID even found to better then OPACITY. So PLAID was chosen for implementation in this dissertation work.

2.6 Malicious Attacks

Let’s look at some of the common attacks that can be launched on smart card based systems. We would also describe the nature of the different attacks, as well as discuss how PLAID protects the Java card based authentication system from these attacks.

Physical attacks: A physical attack includes attempting to model the behavior of the card by probing the physical components of the card, or trying to clone the card in other mechanical ways. Cards are designed to physical temper proof and unclonable in nature. So, there is no way that a card could mimic the nature of some other card.

Replay attack: As discussed earlier, the other form of active attack is a replay attack. An adversary may try to impersonate an authenticated tag or reader in order to gain some sort of access. PLAID protocol used messages are of no use immediately after they used once, so a replay of a message will not be useful to an attacker. An active adversary may succeed in tracking the card, as he can send a request to the card, and the card will reply with response encrypted by RSA key at one of the supported keysetIDs or with some random “RSA Shill key”. If it is not known that what keysetIDs the card supported even a several billions of responses could not provide any significant information.

Man in the middle attack (MiTM): In this kind of attack, attacker sits between both the involved parties and relays and possibly alters the communication between them. Parties in between authentication (communication) process communicate with the attacker but it is not deduced by both these parties. Attackers could get all the sensitive information which it could be used maliciously.

Dictionary Attack: This kind of attack is carried out in the systems where password are utilized as the security procedure. The attack is carried out by utilizing the English dictionary for guessing the password.

Chapter 3. Smart cards

3.1 Introduction

Smart cards are pocket-sized devices such as credit cards, driving licenses, vehicle registration certificates (contact-based), Delhi metro cards (connection-less) include an embedded integrated circuit chip (ICC) which typically hosts a micro-controller with internal memory and secured storage. The micro-controller and memory components present makes it capable of storing and processing information through the electronic circuits embedded in the plastic substrate of its body. The enormous progress in microelectronics in the 1970s created the path for smart card creation, making it possible to integrate data storage and processing logic on a single silicon chip. With the advances in chip technology and modern cryptography, smart cards areas of application widened to the telecommunications (GSM networks: 1991), credit cards (EMV specifications: 1994) and electronic signatures (European directive: 1999).

From a software developer's perspective, smart card software was initially rigid and monolithic, with closed proprietary systems that made the process of application development lengthy and difficult. Nevertheless, the success of open smart cards like MultOS [21] and Java Card, became an important milestone in the history of smart cards. These brought flexible and inter-operable mechanisms, by which multiple applications could be installed after the card had been issued.

Smart cards store information securely in a tamper-resistant security system, protecting the confidentiality and integrity of sensitive data such as keys and personal data from known and anticipated attacks. The micro-controllers offer a secure computing environment for the execution of algorithms to carry out on-card functionalities, typically providing security services. Cards without a micro-controller known as memory cards, depend on the security of the card reader for processing and thus offer lower security services than the ones which perform on-card processing.

Figure 4 Smart card devices commonly

Smart cards are usually categorized into *contact* and *contactless* devices. Contact smart cards must be inserted into a smart card reader to function. They have a contact area (of around 1 square centimeter) comprising several gold-plated contact pads, which will become in direct contact with the reader. The reader provides power to the card through these pads. All communication between the card and the reader is also accomplished through these contact plates.

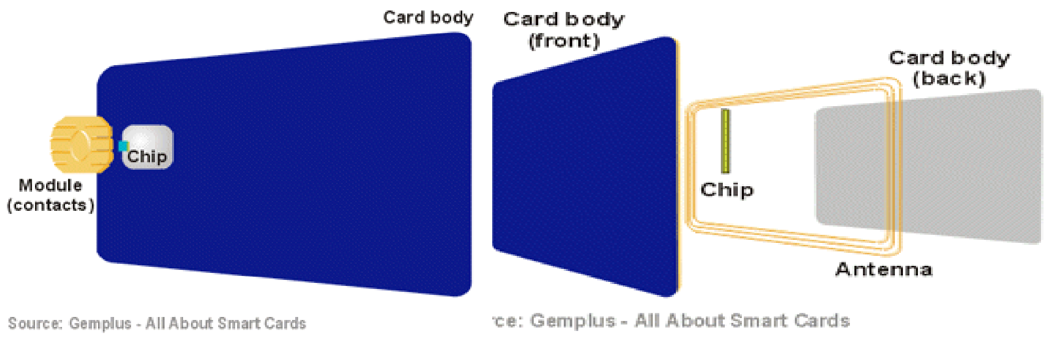

Card body
Chip
Module (contacts)
Source: Gemplus - All About Smart Cards
Card body (front)
Card body (back)
Chip
Antenna
ce: Gemplus - All About Smart Cards

Figure 5 Contact smart card and contact less smart card

Contactless smart card requires only close proximity to a reader and communicates through antennae using radio frequencies (RF). As with contact card, the reader powers the card, but this time the power is transferred through RF induction technology.

3.2 Advantages of using Smart cards

Some advantages of the smart card technology are listed below:

- Security: smart cards are tamper-resistant devices with embedded chip which allow data storage, processing and personal key management in a secure way. Unlike magnetic stripe cards, smart card exploitation requires not only the physical possession of the card, but also intimate knowledge of the smart card hardware, software and specialized equipment.

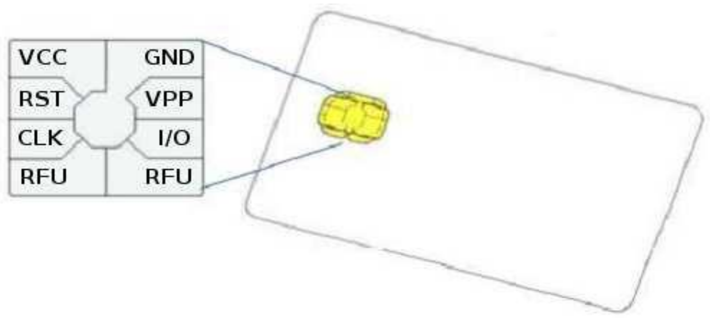
- Multi-application support: multiple applications can reside on a single card. Moreover, these can be installed and removed after the card has been issued, without compromising the security of the various applications.

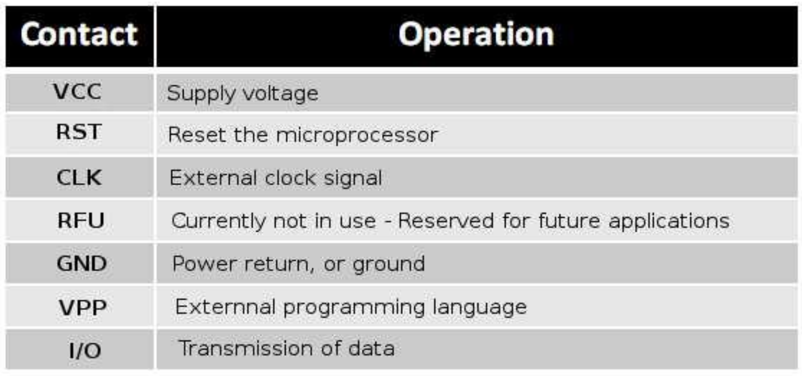
- Standardized features: standards such as the EMV, ISO7816, ISO 14443 and GSM ensure interoperability between different card manufacturers and different card readers.

- Cryptographic support: faster microprocessors and bigger storage capacity allow the exploitation of complex cryptographic algorithms. Current smart cards provide both symmetric and public-key cryptography through AES [11] and RSA [15], respectively, hashing (SHA-1 [16], MD5) and digital signature schemes such as DSA and ECDSA. Furthermore, these can be used to develop more advanced cryptographic protocols and provide security schemes (e.g., authentication).

3.3 Smart card hardware

Smart card contact points and architecture are depicted in figure 3. The Smart card architecture, depicted in figure 4, is made of one embedded CPU; three types of memory: electrical erasable program read-only memory (EEPROM), random access memory (RAM) and read-only memory (ROM); and may also have a coprocessor for mathematical computations.





Contact	Operation
VCC	Supply voltage
RST	Reset the microprocessor
CLK	External clock signal
RFU	Currently not in use - Reserved for future applications
GND	Power return, or ground
VPP	Externnal programming language
I/O	Transmission of data

Figure 6 Smart card contact points

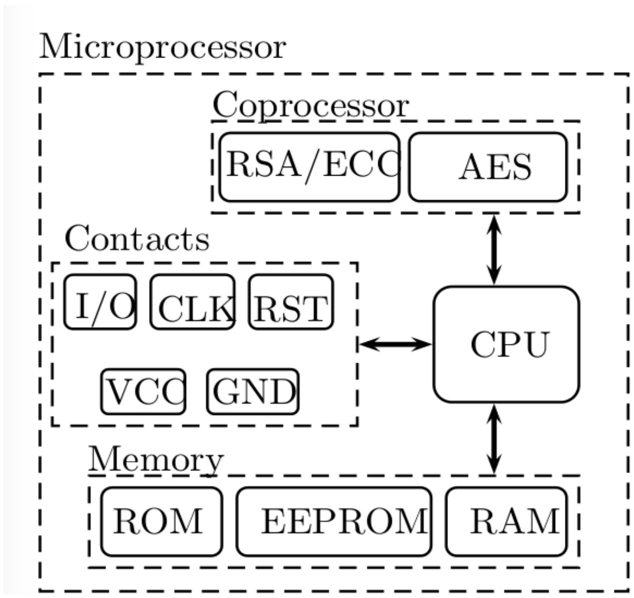


Figure 7 Smart card architecture

•

- Smart Card Contact Points: As presented in the fig. 6 a smart card has eight points of which allow it to communicate with a CAD: Vcc, RST, CLK, GND, I/O and RFU [18].

- Processor: Most of the CPUs on the smart cards are 8-bit size. However those with a 16-bit or 32-bit micro-controller exists and likely to become more common in the future. The clock signal is supplied externally as smart card processors usually do not have internal clock generators. Even though the standards restrict the clock signal to a range of 1-5 MHz, an internal clock multiplier allows card to operate at higher frequencies.

- Coprocessor: Smart cards are very limited in terms of resources, and most are without a specialized mathematical coprocessor, some cryptographic operations would otherwise be infeasible. Security applications which involve modular arithmetic and large-integer calculations commonly resort to the coprocessor (e.g., RSA [15], ECC [20]).

- Memory System :

  - ◆ Read-only-memory (ROM): This persistent memory can only be programmed once, by the manufacturer, and usually includes the operating system routines, cryptographic algorithms and transmission protocols.

  - ◆ EEPROM: Most data of the smart card applications and operating system parameters is stored in this type of memory. It can hold data after the power supply is switched off and also be erased electronically and rewritten. However, writing this memory is considerably slower than RAM, which should be carefully taken into consideration when designing and implementing applications.

  - ◆ RAM: RAM is the fastest and most scarce type of memory in a smart card, meant to store and modify temporary data. The data is stored temporarily in RAM, being immediately lost when the power supply is switched off.

  - ◆ Depending on the application area, memory capacity may range from 16 to 400KB of ROM, 1 to 500KB of EEPROM and 256 bytes to 16KB of RAM [19].

3.4 Smart card Communication Models

In order to communicate with a computer, a card interacts with a Card Acceptance Device (CAD). This device can either be a reader, which in turn communicates with the computer it is connected to, or a terminal if it comprises both the tasks of a reader and a computer (e.g., ATM machine). In this communication model (fig. 8), the applications that communicate with the smart card are called host applications. Smart cards adopt client-server paradigm, being smart card the server and the host application the client. Due to the client-server paradigm, communication between a host and a CAD is half duplex, therefore data can only be sent in one direction at a time. This is achieved through a request-response protocol fig. 5, in which application protocol data units (APDUs) are exchanged. It is the host who initiates the communication, which he does by sending to the card a command APDU (C-APDU), thereafter the smart card replies with a response APDU (R-APDU).

The structure of APDUs are illustrated in the fig. 6. In the C-APDU the mandatory fields are, CLA byte identifies the class of instructions and the INS byte further specifies the specific instruction, while bytes P1 and P2 provide extra parameters. The SW1 and SW2 bytes form the status word, which is used to provide feedback about the execution of the C-APDU. Several status words are predefined in the ISO 7816 standard [32]; examples of status words are “0x9000”, which means that the command was successfully executed and “0x6D00” for an invalid INS value.

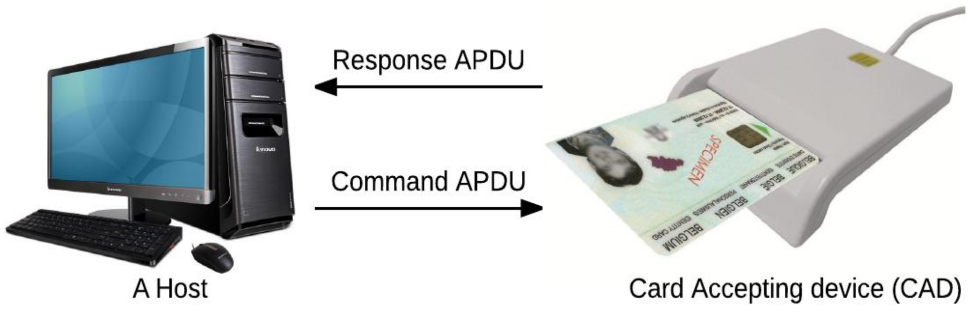


Figure 8 Smart card communication model

The remaining fields are optional: the data field may contain up to 255 bytes, where Lc defines the number of data bytes in the C-APDU and Le the maximum number of bytes expected in the R-APDU data field.

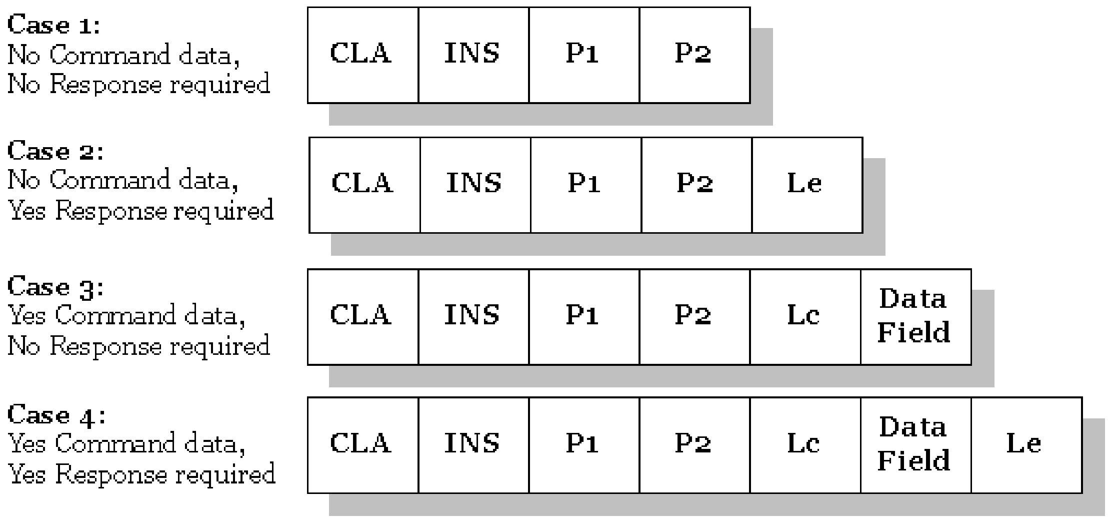


Figure 9 APDU structure

When powered up, or after receiving a RST command, a smart card sends out an answer to reset (ATR) message to the host. ATR message contains various parameters related to the transmission protocol; card hardware parameters but also allows the host to identify the card as cards from the same family share the same ATR.

Transmission protocols are designated as “T=” (for ’transmission protocol’) plus a sequential number. The two most used protocols are called T=0 and T=1, where the former is asynchronous, half-duplex, byte oriented, and the latter is asynchronous, half-duplex, block oriented.

3.5 Standards and Specifications

Many standards and specifications have been developed over the years to ensure the interoperability between smart card systems. Along the way, many projects have born, evolved, fused together or even dropped. Here we mention the most commonly known terms in the context of the smart cards:

- **ISO/IEC 7816** [31]: The most important standard regarding smart cards, it defines multiple aspects of smart cards, such as physical characteristics, transmission protocols and their security architecture are defined by this international standard.
- **ISO/IEC 14443** [32]: Describes the properties and operation modes of contactless smart cards with a range of approximately 10cm.

3.6 Card Operating Systems

The smart card's Chip Operating System or sometimes Card Operating System (COS) is a sequence of instructions, permanently embedded in the ROM, that allow user applications to be stored from an outside development system and provide resources for their execution.

Today's smart card's COSs are nothing like the monolithic first-generations of smart cards, which did not allow the management and execution of third-party applications. As a result, early smart cards were inflexible and failed to provide portability, since applications needed to be developed with a single microprocessor in mind.

It was the introduction of open smart card platforms, namely MultOS [21] and Java Card [12] that allowed both hardware abstraction and multi-application deployment. Later on, other technologies emerged, such as Windows for Smart Cards (WfSC), BasicCard and smart card.NET.

While MultOS and Java Card remain the most widely used smart card platforms, the Java Card appears to be the one enjoying the widest acceptance amongst security researchers. WfSC was intended to be an alternative to Java Card, however, due to lack of acceptance by the smart card industry, the project was abandoned by Microsoft. On the other hand, BasicCard and SmartCard.NET platforms appear to be growing in popularity, specially the latter one, taking into consideration the growing numbers of published articles involving these smart cards.

The basic functions of an operating system that are common across all smart card products are:

1. Management of interchange between the card and the outside world, primarily in terms of interchange protocol.
2. Management of local files and data held in memory.
3. Access control to information and functions.
4. Management of card security and cryptographic algorithm procedures.

More information on the above mentioned technologies can be found in, since a detailed comparison of these platforms is out of the scope of this thesis. As we are going to use the Java card based smart card for our thesis work, the next section provides a detailed information about the card technology.

3.7 Java Cards

Despite its major hardware constraints, its portability, tamper resistance and capability to execute most of the popular security protocols and algorithms, remain one of the mobile computing devices of choice. Java Card technology [12] enables programs written in a subset of Java programming. One of the most widely used multi-application smart card platforms , Java Card, is one of major reason behind the success of smart cards as the ”*write-once-run-everywhere*” concept of Java brought even to the smart card application developers more development flexibility and platform independence. Applications written for the Java Card platform are referred to as applets. Due to the smart card hardware limitations, only a subset of the features of the Java can be supported. Furthermore, Java virtual machine (JVM) must be distributed between the smart card and the workstation.

The Java Card runtime environment (JCRE) manages card resources, network communications, applet execution and security. It also makes sure that different applets do not interfere through a security mechanism referred to as applet firewall.

Table 1 Supported and unsupported Java features

Supported	Unsupported
Boolean, byte, short	int, long, double, float, char
One-dimensional arrays	String, multi-dimensional arrays
Packages, classes, interfaces, exceptions	Dynamical class loading
objects	Object cloning, object serialization
Garbage collection	threads
	Security manager

The bottom layer of the JCRE contains the Java Card virtual machine (JCVM) and native methods. The JCVM executes bytecodes, controls memory allocation, manages objects, and enforces runtime security. Unlike the Java virtual machine (JVM), the Java Card Virtual Machine is split between the card and the workstation. The former is referred to as the on-card VM, or interpreter, and the latter as off-card VM, or converter. While the converter loads and processes the class files and outputs a CAP (converted applet), the interpreter executes the CAP, by which we mean that it executes bytecode instructions and ultimately executes applets. Another component of the JCVM is the Java Card application framework classes (APIs) which provides functions coded in the native instructions of the target processor. Since this can yield a considerable increase in processing speed, APIs should be used as much as possible (e.g., cryptographic operations).

The steps for creating and downloading a Java Card application are summarized as follows:

1. At the workstation, the application programmer writes the Java source code and compiles it, with a standard Java compiler, creating a class file and an export file. At this point, the process is identical to Java programming for PCs.

2. The class file is then transferred to the Java Card Converter (the off-card portion of the VM), which performs static tests and, if all these tests are passed successfully, delivers a second export file and a card application file (CAP file).

3. The applet is loaded into the smart card in the form of a CAP file, which is often carried out using Global Platform.

4. On card VM (interpreter) tests and interprets the bytecode line by line and generates machine instructions for smart card processor from bytecode.

We would be using the terms such as Java smart cards, smart cards and Java cards interchangeably for our Java card embedded on the Micro-SD card provided by Go-Trust, Taiwan [33].

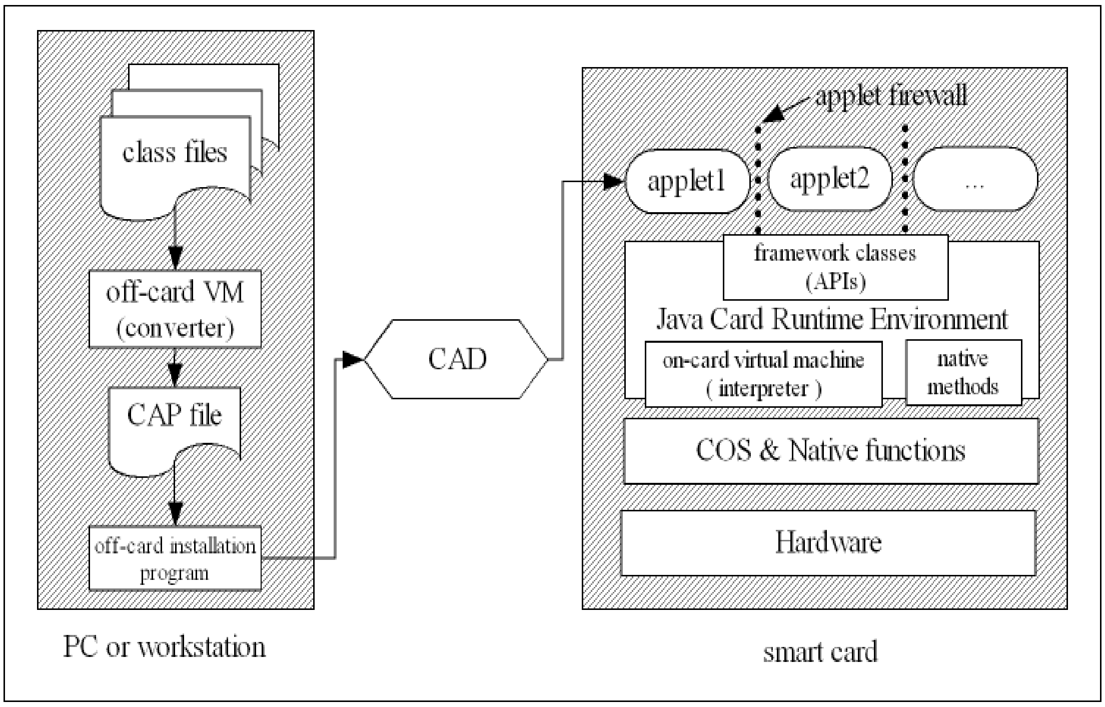


Figure 10 communication model of Java card

Java Card offers several advantages which could explain why it has enjoyed such a wide acceptance from programmers: extensive documentation, development tools, cards and code portability. On the other hand, Java is an interpreted language, which comes at a performance price. Nevertheless, it is relatively difficult to make fair comparisons between assembler or C programs and Java. We can only assume that with proper use of the Java API, the execution time will be approximately 50% longer than for a comparable implementation in C or assembler.

3.7.1 Java Card Progress

The Java Card concept was born in Schlumbereger Austin Texas, with the primary target to bring smart card development into the mainstream. API s were first introduced in 1996, after which Bull and Gemplus joined Schlumberger to cofound the java card Forum, as they stated on their website, the aim of this forum is to promote and develop Java as the preferred programming language for multiple-application smart cards.

The first set of specifications was pretty much rigid and the platform was not extensible. The specification only included standard classes and APIs that allow secure and chip-independent execution of Java applets to run directly on a standard ISO-7816 compliant card. The minimum requirements to run such applets were 512 bytes of RAM, 12Kb ROM and 4Kb EEPROM.

In November 1997, Sun Microsystems Inc. released Java Card 2.0 specification, which evolved from the work of Integrity Arts (a spinoff of Gemplus that specialized in Smart Card virtual machines and operating systems) in collaboration with the Java Card Forum members. The specification contained more concrete details on both API specifications and the Java Card Virtual Machine (JCVM). However, portability and interoperability of applets was still not addressed even on a source code level. The class file conversion, download process, and executable instructions were still in the responsibility of the Java Card implementer, and the API lacked the necessary details for a reasonably complex Java Card application to be portable from one card to another [31]. Typical minimum execution requirements suggested in the specifications w ere 512 bytes of RAM, 16Kb ROM and 8Kb EEPROM.

Java Card 2.1 was released in March 1999, and consisted of 3 specifications [2]:

- The Java Card Virtual Machine (JCVM) Specification providing the instruction set of the Java Card Virtual Machine, the supported subset of the Java language, and the file formats 25used to install applets and libraries into Java Card technology enabled devices.

- The Java Card Runtime Environment (JCRE) Specification defining the necessary behavior of the runtime environment in any implementation of the Java Card technology. The RE includes the implementation of the Java Card Virtual Machine, the Java Card API classes, and runtime support services such as the selection and de-selection of applets.
- APIs for the Java Card Platform complementing the Java Card RE Specification, and describing the application programming interface of the Java Card technology. It contains the class definitions required to support the Java Card VM and the Java Card RE.

The JCVM architecture and applet-loading format specifications in Java Card 2.1, allowed for true applet interoperability. Version 2.2 of the specification was released June 2002, followed with incremental updates 2.2.1 (October 2003) and 2.2.2 (March 2006). This version included additional features such as:

- support for contactless and ID cards,
- remote method invocation (RMI),
- up to 4 logical channels support,
- improved interoperability for cards with multiple communication interfaces,
- richer cryptography and security features such as AES and elliptic curves algorithms,
- Standardized biometry support.

3.7.2 Java Card 3.0

By the year 2008, smart card hardware technology was producing 32 bit RISCs microprocessors with Gigabytes of Flash (form of EEPROM) and a full speed USB 2.0 communications. Such configurations could support multiple communication interfaces each capable of running with independent co-resident applications on a single device [32]. The application potential was enormous, and it was becoming feasible to encrypt/decrypt a real-time video conference or securely run a web server on the card. Java Card 3 technology was released March 2008 and introduced a completely new architecture. The specification became available in two separate, yet coherent versions:

- **Classic Edition**, which is based on an evolution of the Java Card Platform, Version 2.2.2, introducing several incremental changes to ensure alignment with smart card and security standards. The specification is still made up of 3 documents as previous versions, JCVM, JCRE and API specifications.

- **Connected Edition**, featuring a significantly enhanced execution environment and a new virtual machine. It includes new network-oriented features, support for web applications with new Servlet API s, and support for applets with extended and advanced capabilities.

Both Editions are backward-compatible with existing applications, and share key security features. Version 3.0.1 was announced March 2009, and introduced minor API updates and clarifications.

The Connected Edition specification is targeted at less resource-constrained devices (minimum 32-bit CPU with 24Kb RAM, 256Kb ROM and 128K EEPROM), which would typically have a high-speed full-duplex contacted physical interface, such as a USB interface, as well as additional I /O interfaces such as I SO 7816-4 contact connections and ISO 14443 contactless physical interfaces. The specification makes it possible for a device to handle multiple concurrent communications to such interfaces.

The introduced features include support for web applications and applets with extended and advanced capabilities, offering the possibility of a multitude of applications in various industries. For example a device supporting Java Card 3 can act as a secure network node, providing security services such as resource access control.

The specification [32] includes the following documents:

- **Runtime Environment Specification** describing the runtime environment required for interoperable execution of Java Card technology-based servlets and applets with extended/advanced capabilities.

- **Java Servlet Specification** describing the requirements for interoperable Java Card technology-based servlet execution.

- **Application Programming Interface** defining a set of classes upon which Java Card technology-based servlets and applets with extended/advanced capabilities can be constructed.

- **Virtual Machine Specification** describing the new virtual machine for the Connected Edition of the Java Card Platform.

- **Sample Structure of Application Modules** showing the sample structure of the distribution format and the use and syntax of the application descriptors.

Sun Microsystems illustrate the Connected Edition Java Card platform using the following fig 8. The right hand side column represents the Classic Edition based on Java Card 2.2.x. This is still part of the Java Card 3 specification, kept for backwardcompatibility as w ell as low-end hardware.

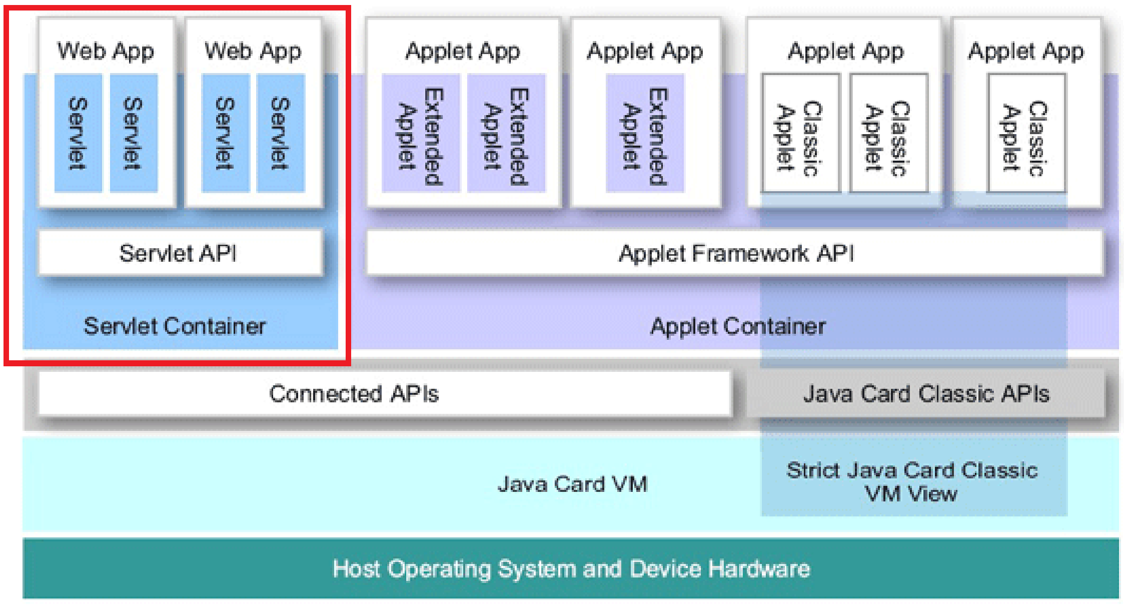


Figure 11 Java card 3.0 architecture

The Connected Edition virtual machine introduced a number of new features, including:

- class file loading from a Java Archive (JAR) file (in classic edition, loading, linking, and verification functions are performed by the off-card converter tool)

- support for on-card class file verification assuring type safety at runtime

- Java SE language features such as generics and annotations
- support for multi-threading and concurrent execution of applications
- automatic garbage collection (GC) freeing unused temporary session data
- Framework for end-to-end secure connectivity.

The Connected-specific API s in the next layer include support for:

- concurrent interfaces and multi-threading, allowing concurrent APDU and TCT/I P based communications
- networking classes from the Generic Connection Framework (GCF) providing IP connectivity for faster communication and also allowing for card-initiated connections (not possible with APDU - based ISO 7816)
- enhanced security features to support a more complex environment, including, flexible role-based and permission-based security as w ell as end-to-end secure connections
- More Java programming language support and API s, such as Collection classes, Localization and Internationalization support generics, annotations, auto boxing, typed enums etc.

An enhanced version of the classic APDU-driven Applet container is provided, w hereby extended applet applications can make use of the connected API s. Such applets can be truly concurrent applications through multi-threading, concurrently processing APDU commands received over different I /O interfaces.

The Servlet container is a completely new feature of Java Card 3, supporting a subset of the Java Servlet API Specification, providing an embedded web server with full HTTP support. This allows smart cards to host web-based applications, interacting with off-card clients via a HTTP and HTTPS request/response protocols over TCP/I P, typically using high speed interfaces like USB. As described in [12], the Java Card platform’s Web application container manages the life cycle of the web applications, the communication services required, and the security of access to these applications and their resources.

3.7.3 Java card Security

Java is well known as a secure programming language, and the Java Card platform inherits most of the Java integrity and security features, adapting them to the ‘memory-limited’ environment, while adding other security features and services which are only applicable to the smart card technology.

Like Java, Java Card compilers are strongly typed, and make extensive stringent error checking when the program is compiled. All primitive types have a specific size and all operations are performed in a designated order.

Byte-code verification is performed at compile time controlling access to all methods and variables. Access to methods and instance variable can be defined through access modifiers, different levels of access controls. Class file verification is made off-card, before code is downloaded into the card, after which the code is signed. On-card byte-code verification is also done at runtime in later versions of the Java Card runtime environments (mandatory in Java Card 3).Java abstracts away any pointers to direct memory locations and does not allow malicious programs snooping around inside memory.

The card architecture and the security features of Java Card make it possible for multiple applets to coexist safely on a card. This offers the possibility of a single smart card to safely host multiple applets such as electronic pure, authentication and health care program developed and managed by different service providers. The Java Card Runtime Environment (JCRE) allow s applet isolation through a firewall mechanism which controls access between different applications on the card, and communication can only occur in specified and controlled ways.

The JCRE also includes ability to execute code in a transactional context. Such set of updates to the card state are guaranteed to execute atomically, i.e. all updated are effected or none, to avoid inconsistent or corrupted data on card.

CHAPTER 4. PLAID

4.1 Introduction

PLAID is an acronym for Protocol for Lightweight authentication and Identification is an authentication protocol targeted at smart cards. It was developed and standardized [8, 14] by Centrelink, an agency of the Australian government’s Department of Human Services (DHS). The protocol is open and supports both physical access control and logical access control. According to its developers claim, the protocol is supposed to provide a cryptographically strong, fast, and private protocol for physical and logical access control, without exposing “card or card holder's identifying information or any other information which is useful to an attacker.

PLAID was initially proposed for use in the internal ID management of employees at Centrelink. However, the intended scope of applications has since significantly broadened to include the whole of DHS and the Australian Department of Defence. Indeed, the protocol’s promoters aspire to broader commercial and governmental deployment, including international levels. With the technological advances, protocol was optimized "to bridge the gap between existing RFID - based technologies that offer speed but lack the necessary security features, and PKI - based authentication, which is cryptographically secure but lacks the speed necessary in many contactless smart card scenarios".

Another strategy that is being actively pursued is standardization [14]. PLAID was previously registered as the Australian standard AS-5185-2010 and was then entered into the ISO/IEC standardization process via the fast track procedure. At the time of writing, the current ISO/IEC version is draft international standard (DIS) 25185-1.2 and is currently in the “Inquiry phase” 40.60 (close of voting). However, the latest version of the PLAID specification supports part 3 and 6 of ISO 24727, a standard that lists a number of requirements for smart cards used as identification cards. This means that it is on the right track for approval as an ISO standard. Minor changes in the original protocol to match the international standard have been applied. Reference implementations, based on the Australian standard, are available both from the Australian Department of Human Services (in Version 8.04) and of the Australian Department of Defense (in draft version

1.0.0).

4.2 Protocol Data Dictionary

PLAID mutual authentication is supported and implemented using both symmetric and asymmetric encryption. The complete authentication process takes less than 0.3 seconds in ideal conditions. Furthermore, it supports multi factor authentication meaning that multiple authentication factors, i.e. something the user knows (such as a PIN) or something the user has (or possess as a unique characteristic e.g. such as a fingerprint), are used in conjunction with the card itself.

Before going ahead with the operations of the PLAID authentication protocol, it would be good that we go through the different data objects of PLAID. The following table sets out the size and details of PLAID data objects.

Table 2 Data dictionary PLAID protocol

Object Name	**Purpose**	**Size Bytes**	**Data type**	**Comments**
ACSRecord	Access Control System Record	varies	Alpha-Numeric	An Access Control System record for each supported Operational Mode Identifier for the purpose of authorization by back office PACS or LACS access control systems. This record is mapped by the OpModeID to the particular back office numbering system the protocol is supporting. This record is returned by the Final Authenticate command response.
DivData	Symmetric Key Diversification Data	8	Binary	A number (or salt) which is set at PLAID instantiation for use in the key diversification algorithm to ensure that loss of an individual card symmetric key cannot result in a breach of the system master keys. This salt is determined by the issuer and should preferably be both random and unique per PLAID invocation AND per system.

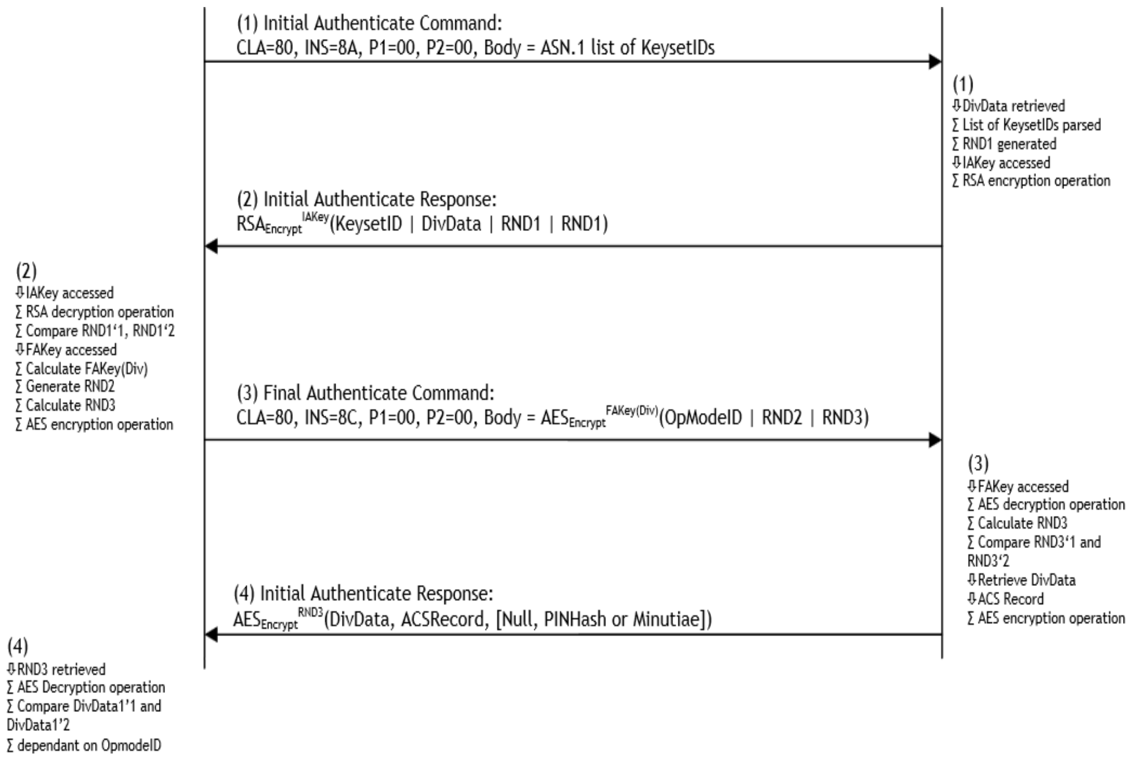
Object Name	**Purpose**	**Size Bytes**	**Data type**	**Comments**
FAKey	Undiversified Final Authenticate Key (AES)	16	Binary	A number (or salt) which is set at PLAID instantiation for use in the key diversification algorithm to ensure that loss of an individual card symmetric key cannot result in a breach of the system master keys. This salt is determined by the issuer and should preferably be both random and unique per PLAID invocation AND per system.
FAKey(DIV)	Diversified Final Authenticate Key (AES)	16	Binary	An instance of a Final Authenticate key that has been diversified against an ICC's diversification data. There are only 3 distinct key sizes allowable by AES.
KeySetID	Uniquely identify a keyset	2	Binary	One or more 2-byte identifiers sent in a list to the ICC in the Initial Authenticate command so as to determine and/or negotiate the key set to be used for authentication.
Minutiae	Fingerprint Minutiae data stored on-card	Variable	Binary	Minutiae template is extracted as raw data and evaluated by the IFD. At this version we are looking to understand if this is sufficient data for operational systems. We are explicitly seeking comment as to whether additional minutiae data should be designed into the specification or whether minutiae should be by individual finger etc.
OpModeID	Operation Mode Identifier	2	Binary	An identifier sent to the ICC in the Final Authenticate command that determines which ACSRecord record is served up in the final authentication response from the ICC.
PIN	PIN	8	Alpha-Numeric	The PIN Global to the ICC.
PIN Hash	PIN Hash	20	Binary	The SHA1 hash value of the PIN which is served up in the final authentication response from the ICC.

Object Name	**Purpose**	**Size Bytes**	**Data type**	**Comments**
RND1	Random Number 1	16	Binary	Random number generated by the smartcard using its TRNG.
RND2	Random Number 2	16	Binary	Random number generated by the IFD or back office system using a TRNG.
RND3	Random Number 3	16	Binary	String generated by the IFD and ICC separately calculation SHA[RND1][RND2].
Secure ICC	Secure the ICC	1	Binary	Flag to hold initial state of the PLAID application. 0 = unsecured, 1 = secured.
SessionKey	Session Key	16	Binary	String generated by the IFD and ICC separately calculating RND3. There are only 3 distinct key sizes allowable by AES.
ShillKey	Shill Key (AES)	Varies	Binary	A shill key is randomly generated by the ICC during PLAID instantiation but is only known to the ICC. A shill key is generated for both the Initial Authenticate (RSA) and the Final Authenticate (AES) commands. ShillKey is used by the ICC in place of the actual key when an inconsistency is detected, thereby removing any indication to a potential attacker that an inconsistency has been detected.
VersionNo	Version number	1	Binary	Implementation version number, starting at zero and incrementing by one for each release.

4.3 Protocol operations

Before the first protocol operation it is required that card is initialized with the keysetId, keys against the ids, ACS record, divData and other things such as PIN and minutiae. Then the card is instantiated to the secure mode, which means that the card is ready for the further protocol operations. We will cover this card initialization process in the implementation chapter of the dissertation. The OpModeID field is a two byte number on the card which indicates the supported mode of operation. For example, aOpmodeID could indicate that a PIN code is used for authentication. The card contains a ACS Record which can be used for authorization, i.e. it indicates the role or level of permission that a legitimate user has. The divData is a random number provided by the claimant, which is used by the verifier for the diversification of the AES key using the divData as a salt.

Figure 9 illustrates the subsequent steps of authentication process. The terms 'interface device' (IFD) is used in this figure which is same as the card reader. It illustrates the messages exchanged between the IFD and the ICC (integrated circuit card).



(1) Initial Authenticate Command:
CLA=80, INS=8A, P1=00, P2=00, Body = ASN.1 list of KeysetIDs
(1)
⇩DivData retrieved
∑ List of KeysetIDs parsed
∑ RND1 generated
⇩IAKey accessed
∑ RSA encryption operation
(2) Initial Authenticate Response:
$RSA_{Encrypt}^{IAKey}$(KeysetID | DivData | RND1 | RND1)
(2)
⇩IAKey accessed
∑ RSA decryption operation
∑ Compare RND1'1, RND1'2
⇩FAKey accessed
∑ Calculate FAKey(Div)
∑ Generate RND2
∑ Calculate RND3
∑ AES encryption operation
(3) Final Authenticate Command:
CLA=80, INS=8C, P1=00, P2=00, Body = $AES_{Encrypt}^{FAKey(Div)}$(OpModeID | RND2 | RND3)
(3)
⇩FAKey accessed
∑ AES decryption operation
∑ Calculate RND3
∑ Compare RND3'1 and RND3'2
⇩Retrieve DivData
⇩ACS Record
∑ AES encryption operation
(4) Initial Authenticate Response:
$AES_{Encrypt}^{RND3}$(DivData, ACSRecord, [Null, PINHash or Minutiae])
(4)
⇩RND3 retrieved
∑ AES Decryption operation
∑ Compare DivData1'1 and DivData1'2
∑ dependant on OpmodeID

Figure 12 Operations in PLAID protocol

1. In the initial authentication request, the IFD send a list of KeySetIDs that it supports in order of preference.

    - Note: ASN1 encoding has been used for the initial card data setup process and initial authentication process.

2. The ICC will extract the list of received KeySetIDs and select the first key, IAKey, which is supported. Subsequently, a string, STR1, is calculated from a

randomly generated string, RND1, as follows:

STR1 = KeySetID | DivData | RND1 | RND1

The random string RND1 is included twice such that it can be used as a checksum. Before being sent to the IFD, STR1 is encrypted asymmetrically using the IAKey.

3. The IFD will decrypt the received STR1 by trying the supported keys. A successful decryption is indicated by the presence of two equal numbers (RND1) in the decrypted string. The IFD generates a new random number, RND2 and calculates a new random number, RND3 by using SHA as a hash function:

   RND3 = SHA (RND1 | RND2)

   RND3 is used as a session key to encrypt all traffic symmetrically

   STR2 = OpModeID | RND2 | RND3

STR2 is encrypted using a symmetric key, referred to as the final authentication key FAKey, which is calculated from a shared master key diversified by using DivData.

4. The ICC receives the encrypted STR2 and decrypts it by calculating FAKey. After 68this it verifies RND3 by calculating this number itself. If it matches the received one, it indicates that IFD was able successfully decrypt STR1, containing RND1 and thus the IFC is authenticated to the ICC. If the authentication is indeed successful, the ICC will respond with STR3, symmetrically encrypted using RND3 as a key.

   STR3 = DivData | ACSRecord | (Null, PINHash and/or Biometric Minutiae)

The last object optionally contains information for the multi factor authentication such as a hashed PIN code or biometric information.

5. Upon reception, the IFD decrypts the message and obtains STR3. It verifies that the DivData is the correct value. After this the authentication is complete.

6. The authentication also results in a secure channel between the IFD and ICC:

RND3 is used as a session key to encrypt all data. After authentication, the card behaves as a secure memory: data can read and written from/to the card over the secure channel with the get data and set data commands. This allows the card to be used in a variety of other applications than physical access control. For example, in a payment application the secure memory could be used to store the balance of a user and update it after a payment.

To the best of our knowledge, PLAID is currently not widely deployed and therefore its quality is not proven in the field. Hence, we depend on scientific publications that evaluate this protocol.

5.4 Advantages with PLAID

We have mentioned advantages of PLAID in introduction section lets elaborate it here

- An advantage with PLAID protocol is the use of hybrid cryptography, i.e. use both asymmetric and symmetric encryption. In operations the second operation of PLAID protocol card sends RSA encrypted response to the reader, except this all other communications are AES encrypted which is faster than asymmetric encryption or decryption.

- In terms of transaction time, it takes 300 milliseconds for authentication. It is better compared to asymmetric encryption based protocols (Asymmetric encryption requires considerably more processing power compared to symmetric encryption). It is known from experience that PSK based systems are easy to set up but do not scale well whereas for PKI this is vice versa. The fact that PLAID uses a hybrid model poses the question about the set up cost and scalability of PLAID.

- The hybrid model used in PLAID does not require the setup of certificate authorities since no certificates are used. This means lower set up cost. PLAID uses key sets (a set of multiple public and corresponding private keys).

- A key set can be used for multiple cards, meaning that no shared secret is required for each card as is done with PSK. This improves scalability compared to using

PSK. However, an increase in card numbers in the system results in an increase in the number of keys in the key set. This means that the reader has to try more keys before finding the right one when decrypting STR1 which may take too long. Thus, this limits the scalability. The conclusion is the hybrid model of PLAID can be placed between PSK and PKI in terms of scalability and set up cost.

- PLAID protocol does not let you know if some step in the authentication process is not approved then it response some random data encrypted using random key called “shill key”. Its response is uniform in all the situations, which makes it more resilient to attacks, as attacker won't be able to guess whether the previous step was a success or failure.

- PLAID could work in different modes. The modes are corresponding to 1-FACTOR, 2-FACTOR, and 3-FACTOR mode. In first mode, the authentication takes place without any added security. In 2-factor authentication the authentication takes place with added security of pin hash and in 3-factor the biometric minutiae based security is also added.

5.5 Limitations of PLAID

As claimed in by unpicking PLAID [45], their “results show that PLAID has significant privacy weaknesses. The shill key attack and the keyset fingerprinting attack reveal card identifying information and, via access authorizations, information about the card holder. As for entity authentication and the secrecy of established keys for subsequent communication, in several places the design of PLAID follows some uncommon strategies and reveals potential attack vectors, such as the lack of forward security. The case of PLAID also shows that standards should specify details thoroughly, in order to avoid vulnerable implementations. An example here is the ISO/IEC 9797-1 non-compliant CBC padding in PLAID, which could potentially be exploited”. It recommend the indiscriminate usage of PLAID in its current form, especially not for privacy-critical scenarios. The PLAID description promises that the protocol should be scrutinized by “the most respected cryptographic organizations, as well as the broader cryptographic community”. Unfortunately, we are not aware of any available documents in this regard.

Indeed, standardization processes in general would benefit if supporting material, arguing the security of a proposal, was available at the time of evaluation.

Chapter 5. Implementing PLAID protocol

In this chapter we address the strategies used while implementation, issues faced and the choices we made to tackle the issues which arrived. The implementation can be divided into two separate process as implementation of the PLAID applet on the card and the other is implementation of the reader for the card.

First of all let us gets some briefs about the well-known algorithms used in the implementation work. We have seen that in total we need RSA, AES, for random number generation (used secure random algorithm for that) and for hashing purpose SHA1 has been used in the implementation.

5.1 RSA

In this section we describe the components implemented for RSA cryptosystem. The RSA cryptosystem is a public-key cryptosystem that offers both encryption and digital signatures (for authentication). It uses a public modulus n, product of two large prime numbers p and q, a public exponent e, less than n and relatively prime to $\varphi(n) = (p-1)\cdot(q-1)$, and a private exponent $d = e -1 \ (mod\ \varphi(n))$. The values of e and d are called the public and private exponents, respectively. The public key is the pair (n, e), while (n, d) is the private key. The factors p and q that constitute n must be kept secret, and allow us to use the Chinese Remainder Theorem (CRT) to speed up decryption/signing.

The keys and other required parameters are computed externally, in the workstation with the *java.math.BigInteger* class, and loaded into the card at applet instantiation time. Once the RSA cryptosystem is set up, i.e., the modulus and the private and public exponents are determined and the public components have been published, both the operation of signing and verification can be performed with the computation of a modular exponentiation, M^e (mod n). The digital signature is created by exponentiations: s = m d mod n, where d and n are the signatory's private key, and m the message to be signed. The validation/verification of the signature is performed by m = s e mod n, where e and n are the signatory's public key.

5.2 AES

In this section we describe the main classes implemented for AES cryptosystem. The AES algorithm is iterative and every round operates on an entire data block called State. The input to the encryption and decryption algorithms is fixed-sized block (usually 128 bits, but AES is easily adaptable for a multiple 32-bit size block, such as 192 and 256 bits). Data is processed as a square matrix of bytes. However, the Java Card specifications do not support multi-dimensional arrays, which forces us to represent the state matrix as an array of 128/8 = 16 Bytes. In order to speed up the transformations performed upon the state variable, we store the data as a transient byte array.

A working implementation of AES must support all the four transformations: substitution, permutation, mixing and key adding. The four main methods that are used for the encryption process are the SubBytes, ShiftRows, MixColumns and AddRoundKey functions; these operations are depicted in Figure 5.1. As for decryption, the sequence of method invocation is reversed, except for the AddRoundKey, whose inverse transformation is identical to the forward transformation, because the Xor operation is its own inverse.

In our implementation of the protocol the javax.smartcardio supports AES of three different key lengths: 128, 192 and 256 bits. The cipher key is loaded into the card at applet instantiation time and stored in static memory. The AES class constructor is invoked when the AES cipher is created at applet instantiation time; it takes a key as parameter and performs all the necessary memory allocation, key expansion and set up of the algorithm.

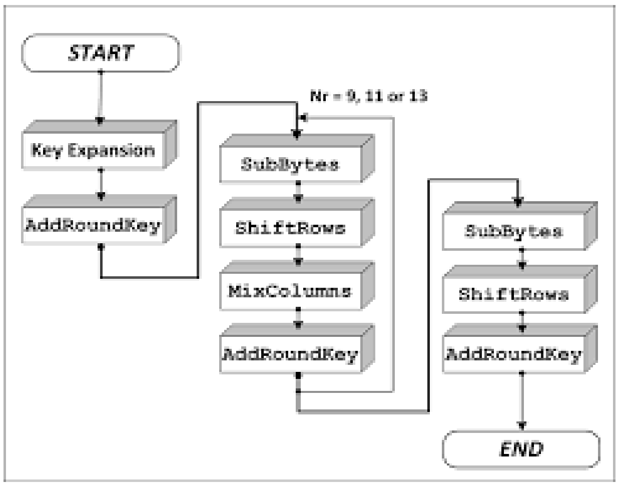


Figure 13 AES Data flow for encryption

SubBytes

The SubBytes transformation performs a simple byte substitution on each byte of the State using a substitution table, the substitution box (sBox), which contain the permutations of all 256 8-bit values. These boxes are constructed using defined transformation of values in GF (2^8) and are loaded into the card as static final byte arrays (EEPROM) at applet instantiation time. Each of this tables is stored in the EEPROM and requires 256 bytes of memory.

The implementation of the inverted SubBytes function is straightforward, as is it processed exactly in the same way as the SubBytes operation, with the exception that the inverse substitution Box (invsBox) table is used. Consequently, a total of 512 bytes are needed for storing the S-BOX and the inverted S-BOX table, which is almost negligible for modern smart cards.

ShiftRows

The ShiftRows transformation consists of circular byte shifts, where each row is shifted over a different number of positions. The inverse shift row transformation (invShiftRows) performs the circular shifts in the opposite direction.

MixColumns

MixColumns is the most expensive operation, since it involves matrix multiplication in GF(2^8). GF(2^8) multiplication is defined with a carefully selected primitive polynomial $x^8 + x^4 + x^3 + x + 1$ to speed up computation.

In practice, Mix Columns can be implemented by expressing the transformation of each column as four equations to compute the new bytes for that column. The computation only involves shifts, XORs and conditional XORs (for the modulo reduction). However, the decryption is slower due to the computation requiring the use of the inverse matrix, which has larger coefficients.

We can achieve a significant speed up by using lookup tables with all the precomputed multiplications in GF(2^8). The additional 1536 bytes of EEPROM memory required to store the lookup tables does not comprise a problem, taking into account the considerable amount of memory currently available in smart cards. Lookup tables not only improve the performance of the AES algorithm but also make it more secure by making it less prone to timing and power attacks.

AddRoundKey

The AddRound function is straightforward: the 128 bits of State are bitwise XORed with the 128 bits of the round key.

KeyExpansion

The AES key expansion algorithm takes as input the key and produces the expanded key array, which ranges from 176 bytes to 240 bytes, depending on the size of the input key. The expanded key could be expanded each time it is being used, or be expanded once and stored in the static memory. We have followed the latter approach, since storing the key in static memory has a low memory footprint and decreases the time needed to perform either encryption or decryption.

The round constant array (Rcon) contains the values given by x^{i-1} , with x^{i-1} being powers of x in the field GF(2^8). This array is stored in persistent memory and is depicted in Table 3.

Table 3 Rcon array

i	0	1	2	3	4	5	6	7	8	9
x^i	01	02	04	08	10	20	40	80	1b	36

5.3 Random Number

In simple, a random number is a number generated through a process, whose outcome cannot be predicted i.e. its outcomes should be completely unpredictable in nature and also which cannot be produced sub sequentially with any reliability.

Random number generation requires random events represented in bits, these *n* random bits are obtained by gathering "physical events" which should be unpredictable, as far as physics are concerned. Usually, timing is used: the CPU has a cycle counter which is updated several billions times per second, and some events occur with an inevitable amount of jitter (incoming network packets, mouse movements, key strokes...). The system encodes these events and then "compresses" them by applying a cryptographically secure hash function such as SHA-256 (output is then truncated to yield our *n* bits). What matters here is that the encoding of the physical events has enough *entropy*: roughly speaking, that the said events could have collectively assumed at least *2^n* combinations. The hash function, by its definition, should make a good job at concentrating that entropy into an *n*-bit string.

Once we have *n* bits, we use a PRNG (Pseudo-Random Number Generator) to crank out as many bits as necessary. A PRNG is said to be cryptographically secure if, assuming that it operates over a wide enough unknown *n*-bit key, its output is computationally indistinguishable from uniformly random bits.

In the context of cards, the random number generation for implementation of the applet, we have used the enriched API provide along with the java card sdk. The RandomData class of the javacard.security package has been used to get the random numbers, which are required in the protocol. The getInstance method, which is used to initialize the object with the desired algorithm types and the generateData method is used to get the random bytes.

5.4 Hash

Cryptographic hash functions simply the hash function are algorithms that take data input (often called the 'message') and generate a fixed-size result (often called the 'hash' or 'digest'). The process of generating the hash of a message is hashing. For example**:**

"administrator" => "b3a5a92c793ee039b4a9b0a5f5fc056e05140df3"

An ideal hash functions make it very difficult to get the original message back from the digest or the hash, it should be reasonably easy to compute a hash for a given message, it must be infeasible to generate a message with a given hash, also infeasible to modify a message without changing the resultant hash, and final it should be infeasible to find two messages with the same hash.

While there is hardly an ideal function exists, but functions which aim for these properties can prove to be very useful. A classic example for the usage of cryptographic hash function is their use in storage of passwords. When you sign up for a website, your data is usually stored in a database on their servers. The issue is that if your password is stored on the server as regular text (often called 'plaintext') and somebody hacks into the server, your password is completely compromised. If your password is hashed on the server, however, an attacker shouldn't be able to formulate your password from the stored value.

This concept may leave some wondering how a password entered at a later date could then be compared to the stored value to check if the login information is correct, but in fact this is extremely simple. The inputted password is simply hashed using the same function, and this digest is simply compared to that in the database -- if they match, the inputs were the same, and thus the user has inputted the correct password. Situations like this are also where it being "infeasible to find two messages with the same hash" becomes very important. If two values generate the same hash (these situations are called **collisions**, and are something that pretty much all hash functions are vulnerable to), somebody could input an incorrect password yet it could validate as correct.

The irreversibility is not actually as impossible as it might first sound -- the tough bit comes in compromising this with all the other ideal properties. The trick is the split the message into a number of blocks, and then have these messed up and interact with each other to get some final value pop out. The interactions are often in the form of bitwise AND, OR, and XOR operations which mean a loss of information to the original input, and this combined with the mixing of blocks means that there simply isn't enough information to get back to the original input via working the algorithm back.

For hashing, the java card *javacard.security* provides supports for different algorithms. The hashing function supported are SHA, SHA256, SHA348, SHA-512, and MD5. In cards context, for hash generation we need to call *getInstance()* method to get a *messageDigest*object to initial for further use. Then the call of the *dofinal*or the update methods does the hashing of the message.

5.5 DER

Abstract Syntax Notation One (ASN.1) defines the following rule sets that govern how data structures that are being sent between computers are encoded and decoded.

- Basic Encoding Rules (BER)
- Canonical Encoding Rules (CER)
- Distinguished Encoding Rules (DER)
- Packed Encoding Rules (PER)

The original rule set was defined by the BER specification. CER and DER were developed later as specialized subsets of BER. PER was developed in response to criticisms about the amount of bandwidth required to transmit data using BER or its variants. PER provides a significant savings [23].

As DER has been used in the first communication as well as in the initialization from reader to the card it would be good to provide some introduction about the notation standard. In the encoding the basic unit is a byte. The different data types are assigned different **tag bytes** for example an octet string is assigned 0x04 as a tag byte. Format for sending an **octet string** in DER encoding is given in the table below.

Table 4 DER format octet string data type

Tag Byte	Content (Octet String) Size	Content(Octet String)
04	0A	00 E1 04 06 2D 03 5A 85 41 09

As in the table the first byte is the tag byte, next one is the size of the content i.e. the number of octet strings which follows. The encoding for other data types would be similar. Let’s also discuss one more type used in the protocol that is **Sequence** or **Sequence of**, which is encoded as in the table.

Table 5 DER format Sequence data type

Tag Byte	content Size	Content(sequence of Octet String)
10	11	04 01 08 04 02 11 11 04 08 85 41 09 08 93 12 7D 5B

The table shows the encoding of the sequence, as given the tag byte for the sequence is 10 (i.e. 16 in decimals) and size is the 17 (content carries 17 octets). The content itself is the sequence of octet strings. The content consists three octet strings. It satisfies the DER for octet strings. The first octet string starts with 04 (which is tag byte for the octet string), followed by the size which is 01 and then the content which is 08 similarly the next the octet string which is of size 02 and content as 11 11 and then the last octet string follows that is of length 08 [24].

5.6 Implementing the Applet

Before discussing about the applet implementation, it would be good if we get introduced about the development environment. The development environment for the applet development is different from the development environment for the reader. We will discuss here only the environment for the applet development. The development environment for the reader will be discussed in the section related to the reader development.

5.6.1 Applet development environment

For the development and testing of smart card application, we opted for the host computer an Intel® Core™ i3 CPU M 380 @ 2.53GHz × 4 PC with 3GB of RAM under Windows 7 ultimate of OS type 64-bit. Several tools are available to develop and load Java Card applets, and, even though there are interesting proprietary tools, we focused only on publicly available and open source software. The development environment has the following configuration:

- A GoTrust Micro-SD embedded java card based smart card is used in this project. It is Java Card 3.0.1 and Global Platform [33] 2.1.1 compliant and has an approximate available memory size of 80K. Multiple cryptographic algorithms are supported, such as RSA (up to 2048 bit), AES (128, 192 and 256 bits) and SHA1. True random number generation and real garbage collector (JC 2.2.1 specification) are also available.

- As it is Micro-SD embedded smart card so we do not need any specific hardware for reader. A normally available SD card reader can be used perform the tasks of a CAD. We would require an adapter to use the Micro-SD into the SD card reader.

- The workstation is a Windows/PC, which is used to develop the Java Card applets as well as running the host applications. Java code source file (.java) can be compiled via the Java Development Kit (JDK) into a class file (.class). Sun's Java Card Development Kit (JCDK) version 2.2.1 [1] is used for converting class files into converted applet files (.cap). The JCDK does not provide a visual development environment, therefore, the Eclipse IDE is used for developing Java Card applets through the JCDE [3] plugin, which integrates the functionality of the JCDK. The Eclipse integrated Java Card development Environment is depicted in Figure 6.1.

- In order to deploy the application, the CAP file must be loaded into the smart card, which can be achieved through the jSDSCTool v.1.3.4.

- Our implementation of the AES cryptosystem for the Java Card smart card supports three key sizes: 128, 192 and 256 bits. However, the standard Java SDK does not support the 192 and 256 key sizes due to export restriction policies. In order to access all sizes in the challenge-response, Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files need to be downloaded.

- The javax.smartcardio package was included in Java 6 and allows Java programs to communicate with a Java Card smart card, using ISO/IEC 7816-4 APDUs. With the Java Smart Card I/O API it is also possible to detect card insertion/removal as well as to establish connection with a reader

5.6.2 The applet development

Prior to deployment, applications may be tested in the PC/Windows workstation, using the two simulators provided by the JCDK: JCWDE and CREF. Card simulator reveals several advantages, such as speeding up the development process of applets and not wearing out the card. However, these simulators also have several limitations; for instance, several cryptographic algorithms are not or only partially available (e.g. RSA keys are limited to 512 bits and NOPAD mode is not available). Moreover, accurate benchmarking cannot be performed because the code is executed by the simulator running on a 0x86 CPU and can only provide estimate values. Emulators, on the other hand, provide more accurate results as their behavior is similar to the physical cards. However these are only available on proprietary tools, such as JCOP tools.

As we have already gone through the operations of the protocol. An implementation has been done keeping the specification. Got the sample implementation which worked as reference for our implementation. We have made the required changes in the reference applet code so that we could get the data and keys and verification of the data we set to the card. Also used a tweak to get the status of the card at different stages of the protocol operations. This helped to get the status of the card at different stages of the protocol, which helps to debug the applet code.

In the protocol, firstly the card receives the keysetID in plain text format, which is encoded using the DER ASN.1 [23, 24], the first operation uses the RSA in the first operation for which I got reference for the code at Oracle forum on java card, where one can get many code references for the RSA as well as for the AES. As specified in the PLAID protocol specifications, I have used 1024 bits key size for RAS encryption and 16 bytes (128 bits) (16,24,32 in the specifications) length keys for AES encryption/decryption. The algorithm used for RAS is “ALG_RSA_PKCS1”, algorithm used for AES is “ALG_AES_BLOCK_128_CBC_NOPAD”, algorithm used for random number generation is “ALG_SECURE_RANDOM” and the algorithm for hash is “ALG_SHA”. Brief introduction about these algorithms has already provided in the above sections.

5.7 Implementing the Reader

Again, let's go through the development environment used for the implementation of the reader. For the PLAID to communicate with the host machine we required a reader.

As we have Micro-SD embedded Java card from GoTrust it does not require any other card reader except the SD card reader, which is available in most of the machines out in the market. The API provided along with the card provides us the methods for sending the commands APDU and get the response.

5.7.1 Reader development environment

For the development of smart card reader applications, we opted for the host computer, an Intel® Core™ i3 CPU M 380 @ 2.53GHz × 4 PC with 3.0 GB of RAM under Ubuntu 14.04 LTS of OS type 32-bit. As the card API provided for interfacing is in C language, so it was decided to stick to C language for the reader application. The reader for the most part is written in C language.

- *Code::Blocks* worked as the primary Integrated Development Environment further refereed as IDE for the reader application [25].
- As the thesis work require multiple cryptographic functionalities, I opted to go with the most popular and open source library, **OpenSSL**v.1.0.2a [7, 26].
- For the AES implementation we have used the Java language, so environment for the Java language the complier and the runtime environment also need to be setup [27].

5.7.2 Reader development

Going through the operations of the protocol the first operation which takes place is the sending the key set IDs which is supported by the reader. As we have the theSdk made available along with the card, utilizing it provides us the functionalities to communicate to the card using APDUs.

Now if the card supports the key set ID it would send reply encrypted with RSA public key. If the card does not support the key set ID even then it sends some reply which basically consists random bytes encrypted with some unknown random public RSA key.

Now as the card supported the key set ID replies with the RAS encrypted data which is required to be decrypted. For the decryption process the reader utilizes the private decrypt function available with the OpenSSL library [7, 28]. The function returned the decrypted data. The success of the decryption process has been confirmed on the fact that the last 32 bytes consisted a single 16 byte data repeated twice. As the card does not communicate anything about the key set IDs it supported but simply encrypted the data with one of the supported key set, so it was required to determine the key set ID with which it has been encrypted. It is done by trying to decrypt with all the available key sets and checking the repeated 16 bytes at the end.

The next operation at the reader end is to generate a 16 byte random number. For the random generation the OpenSSL library has been utilized [7, 29]. Also we required to generate hash of the RAND1 (from the card) and RAND2 (from reader) to get another 16 byte random number RAND3. For the hash generation the again the OpenSSL library function has been used [7, 30].

Now in the third communication between the card and the reader the reader sends RAND2 and RAND3 encrypted using the AES. The AES key used here is the key from the determined key set ID. For the encryption process the Java has been used which encrypts the data and returns the result to the reader which is then packed into APDU and then transmitted to the card. RAND3 is now set as the session key which functions as the AES key for further communication in the process.

Chapter 6. Evaluation and Results

To get results of the implementation of the protocol we are required to initialize the card with various data. In the chapter, we first discuss the card initialization so that we get the card initialized and then carry out the operations to get the results.

6.1 Card initialization

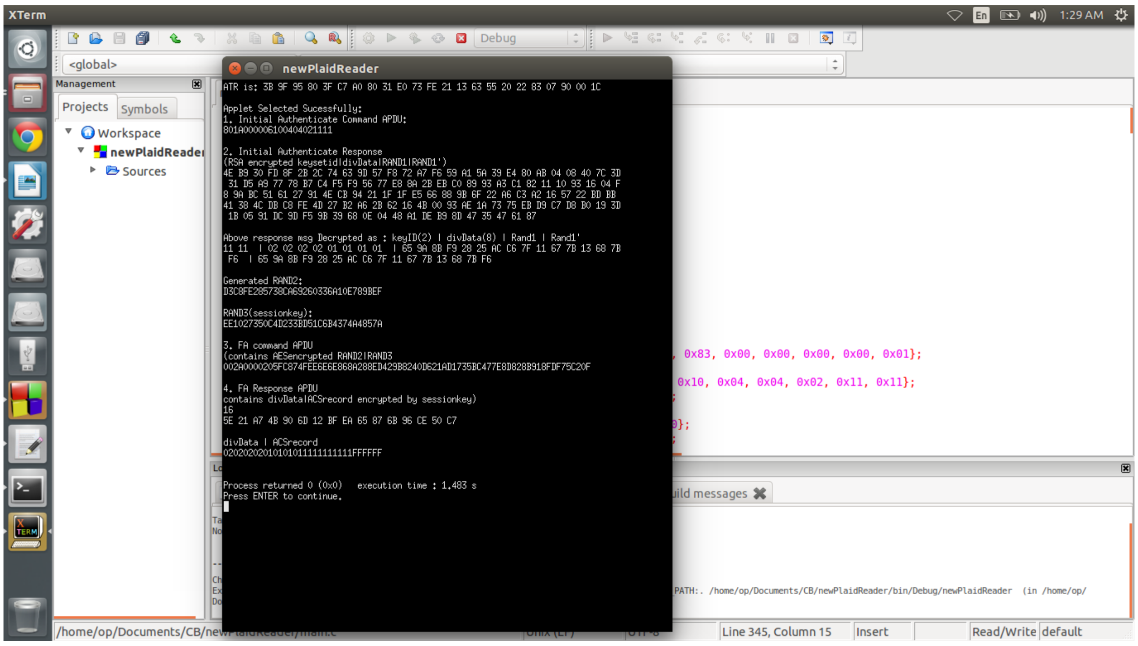
After all the implementation, we are required to initialize the card with data so that I could test out the implementation of the PLAID protocol. The initialization process basically consists adding various data such as keysetIDs along with the keys, ACSrecord and divData to the card to make it operational. For all the initialization related work we can utilize the jSDSCTool provided for loading applet to the card or we could also use the initialization application created for the initialization process only.

With the jSDSCTool for the initialization process we are required to send a set of command APDUs, which basically set data to the card. Sometimes we could be required to send a number of command APDUs to the card. All the command APDUs are DER ASN.1 type encoded.

Other method for the initialization of the card data is the application which is written just to initialize the card data. Basically in the application all the command APDUs we were required to send using the jSDSCTool are now put into a single application which transmits the data to the data in a sequential manner.

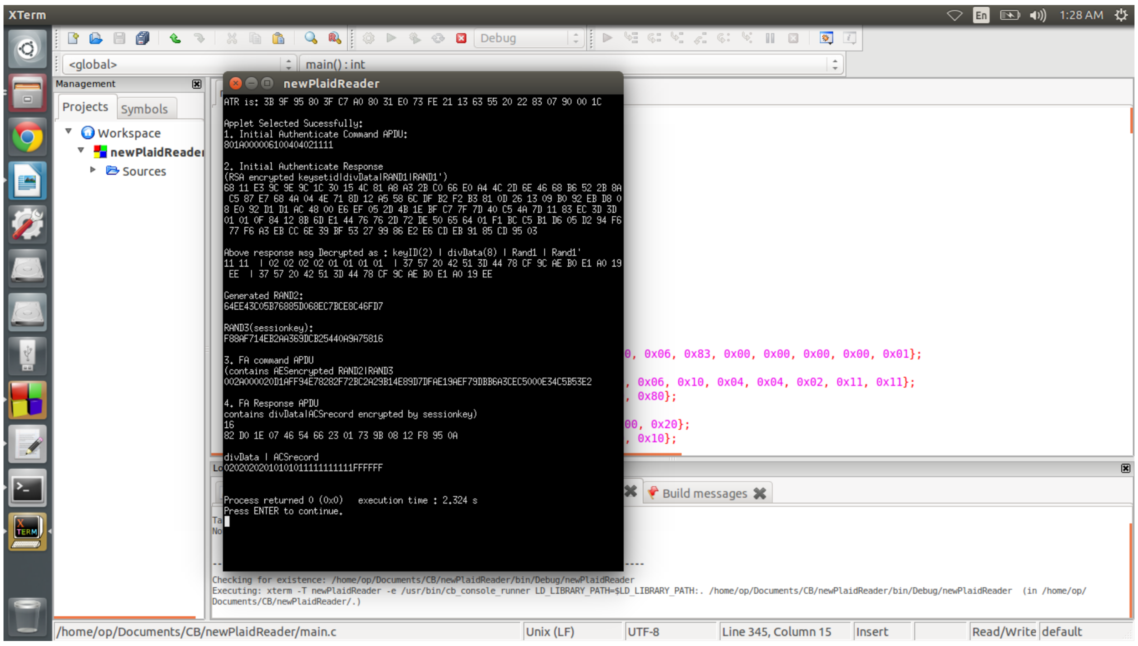
After initialization of the card with the data card is required to put into the secure mode so that card cannot be accessed in any other way except the discussed operations of the protocol.

The screen shots below present the three runs of the reader application.



```
ATR is: 3B 9F 95 80 3F C7 A0 80 31 E0 73 FE 21 13 63 55 20 22 83 07 90 00 1C

Applet Selected Sucessfully:
1. Initial Authenticate Command APDU:
801A0000061004040211111

2. Initial Authenticate Response
(RSA encrypted keysetid|divData|RAND1|RAND1')
4E B9 30 FD 8F 2B 2C 74 63 9D 57 F8 72 A7 F6 59 A1 5A 39 E4 80 AB 04 08 40 7C 3D
 31 D5 A9 77 78 B7 C4 F5 F9 56 77 E8 8A 2B EB C0 89 93 A3 C1 82 11 10 93 16 04 F
8 9A BC 51 61 27 91 4E CB 94 21 1F 1F E5 66 88 9B 6F 22 A6 C3 A2 16 57 22 BD BB
41 38 4C DB C8 FE 4D 27 B2 A6 2B 62 16 4B 00 93 AE 1A 73 75 EB D9 C7 D8 B0 19 3D
 1B 05 91 DC 9D F5 9B 39 68 0E 04 48 A1 DE B9 8D 47 35 47 61 87

Above response msg Decrypted as : keyID(2) | divData(8) | Rand1 | Rand1'
11 11  | 02 02 02 02 01 01 01 01  | 65 9A 8B F9 28 25 AC C6 7F 11 67 7B 13 68 7B
 F6  | 65 9A 8B F9 28 25 AC C6 7F 11 67 7B 13 68 7B F6

Generated RAND2:
D3C8FE285738CA69260336A10E789BEF

RAND3(sessionkey):
EE1027350C4D233BD51C6B4374A4857A

3. FA command APDU
(contains AESencrypted RAND2|RAND3
002A0000205FC874FEE6E6E868A288ED429B8240D621AD1735BC477E8D828B918FDF75C20F

4. FA Response APDU
contains divData|ACSrecord encrypted by sessionkey)
16
5E 21 A7 4B 90 6D 12 BF EA 65 87 6B 96 CE 50 C7

divData | ACSrecord
0202020201010101111111111FFFFFF


Process returned 0 (0x0)   execution time : 1.483 s
Press ENTER to continue.
```

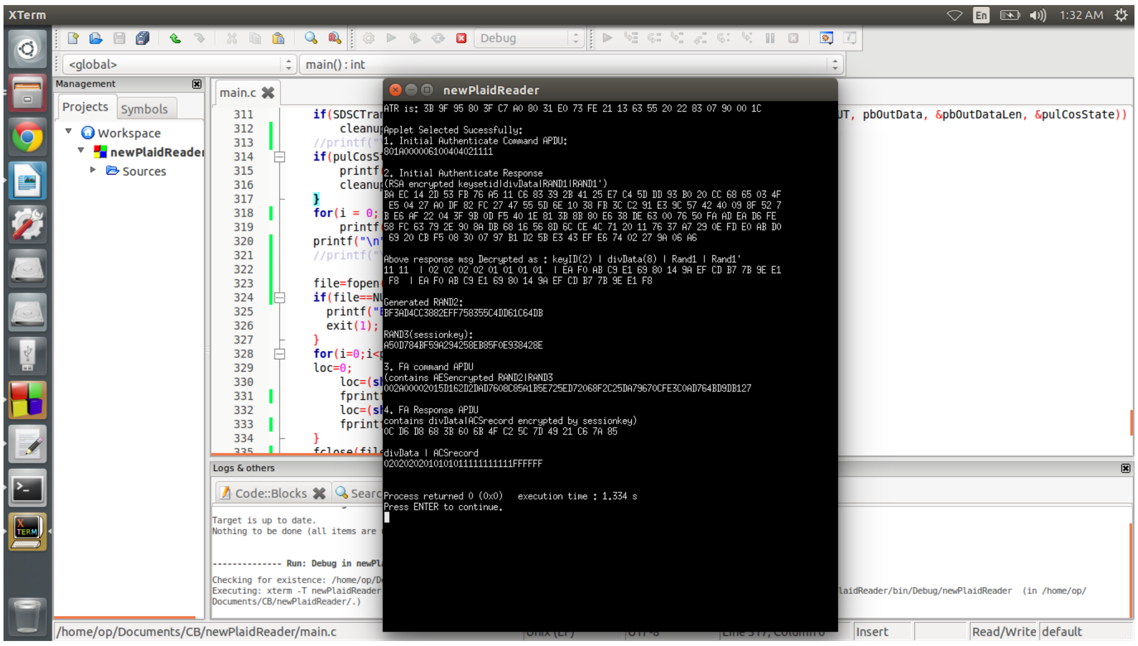
Figure 14 Run1 for the reader of the PLAID protocol



```
ATR is: 3B 9F 95 80 3F C7 A0 80 31 E0 73 FE 21 13 63 55 20 22 83 07 90 00 1C

Applet Selected Sucessfully:
1. Initial Authenticate Command APDU:
801A0000061004040211111

2. Initial Authenticate Response
(RSA encrypted keysetid|divData|RAND1|RAND1')
68 11 E3 9C 9E 9C 1C 30 15 4C 81 A8 A3 2B C0 66 E0 A4 4C 2D 6E 46 68 B6 52 2B 8A
 C5 87 E7 68 4A 04 4E 71 8D 12 A5 58 6C DF B2 F2 B3 81 0D 26 13 09 B0 92 EB D8 0
8 E0 92 D1 D1 AC 48 00 E6 EF 05 2D 4B 1E BF C7 7F 7D 40 C5 4A 7D 11 83 EC 3D 3D
01 01 0F 84 12 8B 6D E1 44 76 76 2D 72 DE 50 65 64 01 F1 BC C5 B1 D6 05 D2 94 F6
 77 F6 A3 EB CC 6E 39 BF 53 27 99 86 E2 E6 CD EB 91 85 CD 95 03

Above response msg Decrypted as : keyID(2) | divData(8) | Rand1 | Rand1'
11 11  | 02 02 02 02 01 01 01 01  | 37 57 20 42 51 3D 44 78 CF 9C AE B0 E1 A0 19
 EE  | 37 57 20 42 51 3D 44 78 CF 9C AE B0 E1 A0 19 EE

Generated RAND2:
64EE43C05B76885D068EC7BCE8C46FD7

RAND3(sessionkey):
F88AF714EB2AA369DCB25440A9A75816

3. FA command APDU
(contains AESencrypted RAND2|RAND3
002A000020D1AFF94E78282F72BC2A29B14E89D7DFAE19AEF79DBB6A3CEC5000E34C5B53E2

4. FA Response APDU
contains divData|ACSrecord encrypted by sessionkey)
16
82 D0 1E 07 46 54 66 23 01 73 9B 08 12 F8 95 0A

divData | ACSrecord
0202020201010101111111111FFFFFF


Process returned 0 (0x0)   execution time : 2.324 s
Press ENTER to continue.
```

```
Checking for existence: /home/op/Documents/CB/newPlaidReader/bin/Debug/newPlaidReader
Executing: xterm -T newPlaidReader -e /usr/bin/cb_console_runner LD_LIBRARY_PATH=$LD_LIBRARY_PATH:. /home/op/Documents/CB/newPlaidReader/bin/Debug/newPlaidReader  (in /home/op/
Documents/CB/newPlaidReader/.)
```

Figure 15Run2 for the reader of the PLAID protocol



```
ATR is: 3B 9F 95 80 3F C7 A0 80 31 E0 73 FE 21 13 63 55 20 22 83 07 90 00 1C

Applet Selected Sucessfully:
1. Initial Authenticate Command APDU:
801A000006100404021111

2. Initial Authenticate Response
(RSA encrypted keysetid|divData|RAND1|RAND1')
BA EC 14 2D 53 FB 76 A5 11 C6 83 39 2B 41 25 E7 C4 5D DD 93 B0 20 CC 68 65 03 4F
 E5 04 27 A0 DF 82 FC 27 47 55 5D 6E 10 38 FB 3C C2 91 E3 9C 57 42 40 09 8F 52 7
B E6 AF 22 04 3F 9B 0D F5 40 1E 81 3B 8B 80 E6 38 DE 63 00 76 50 FA AD EA D6 FE
58 FC 63 79 2E 90 8A DB 68 16 56 8D 6C CE 4C 71 20 11 76 37 A7 29 0E FD E0 AB D0
 69 20 CB F5 08 30 07 97 B1 D2 5B E3 43 EF E6 74 02 27 9A 06 A6

Above response msg Decrypted as : keyID(2) | divData(8) | Rand1 | Rand1'
11 11  | 02 02 02 02 01 01 01 01  | EA F0 AB C9 E1 69 80 14 9A EF CD B7 7B 9E E1
 F8  | EA F0 AB C9 E1 69 80 14 9A EF CD B7 7B 9E E1 F8

Generated RAND2:
BF3AD4CC3882EFF758355C4DD61C64DB

RAND3(sessionkey):
A50D784BF59A294258EB85F0E938428E

3. FA command APDU
(contains AESencrypted RAND2|RAND3
002A00002015D162D2DAD7608C85A1B5E725ED72068F2C25DA79670CFE3C0AD764BD9DB127

4. FA Response APDU
contains divData|ACSrecord encrypted by sessionkey)
0C D6 D8 68 3B 60 6B 4F C2 5C 7D 49 21 C6 7A 85

divData | ACSrecord
02020202010101011111111111FFFFFF


Process returned 0 (0x0)   execution time : 1.334 s
Press ENTER to continue.
```

Figure 16 Run3 for the reader of the PLAID protocol

It can be seen from screen shots, data related to the operations are numbered. The first operation of the protocol is sending the supported key set IDs by the reader to the card. Next operation is the response of the first step. Both the encrypted response and the response after the decryption are presented. Next the generated random numbers are presented. In the next command the AES encrypted RAND2 and RAND3 are sent to the card. It concludes the PLAID authentication process. In the next step, divDataand theACSrecord encrypted using the AES session key are decrypted and presented. This communication has been through the generated session key, it shows that the session key has been determined and agreed upon has been generated at both the ends successfully.

Timings Breakdown

The protocol implementation, on an average the PLAID protocol takes an average time of 1.42 seconds (1420 milliseconds), execution time present in the screenshots above fig 11, 12 and 13, for complete run of the application. This time includes checking and listing the available card readers, selecting a card reader, establishing connection to the card, resetting the Java card, applet selection, PLAID authentication and finally disconnecting the communication channel with the applet. The most of the execution time is consumed in the Java function calls for the AES encryption and decryption process. Two Java class file calls for execution takes 600 milliseconds on an average. First two operations of the protocol that is sending the keysetID and then decrypting the response takes about 75 milliseconds on an average. The random number generation and hashing takes less than a millisecond (400 microseconds) on the host machine. Card reset and applet selection processes takes 90 millisecond on an average. The combined timing for device listing, selecting a device and establishing connection to the device takes about 310 millisecond on an average. Making card ready for first communication takes about (310 + 90) 400 milliseconds.

Considering only the PLAID authentication timings only then it comes close to 300 milliseconds on an average, which is in the acceptance limits by the proposed execution timings of the protocol.

Preferences

[1] Java card development kit. http://java.sun.com/javacard/devkit/. [Online; accessed February, 2014].

[2] Sun Microsystems Inc. Java Card Platform Specifications. http://www.oracle.com/technetwork/java/javacard/specs-138637.html [Online; accessed February 2015].

[3] Sourceforge.net: Eclipsejcde. http://eclipse-jcde.sourceforge.net/. [Online; accessed February , 2014].

[4] NIST, editor. An Introduction to Computer Security : The NIST Handbook, Special Publication 800-12, October 1995.

[5] Paul C. van Oorschot Alfred J. Menezes and Scott A. Vanstone. Handbook of Applied Cryptography. Crc Press Series on Discrete Mathematics and Its Applications, 1997.

[6] William Stallings. Cryptography and Network Security: Principles and Practice. Prentice Hall, 4th edition, 2005.

[7] Pravir Chandra, Matt Messier, John Viega . Network Security with OpenSSL . O'Reilly , June 2002

[8] Centrelink. Protocol for Lightweight Authentication of Identity (PLAID) — Logical Smartcard Implementation Specification PLAID Version 8.0 - Final, December 2009.

[9] Hideki Sakurada. Security evaluation of the PLAID protocol using the ProVerif tool. http://crypto-protocol.nict.go.jp/data/eng/ISOIEC_Protocols/25185-1/25185-1_ProVerif.pdf, September 2013.

[10] Standards Australia. AS 5185-2010 Protocol for Lightweight Authentication of IDentity (PLAID). Standards Australia, 2010

[11] Announcing The Federal. Federal information processing standards publication 197. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, November 2001. [Online; accessed March, 2015].

[12] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[13] Koh Ho Kiat and Lee Yong Run. An analysis of OPACITY and PLAID protocols for contactless smart cards. Master's thesis, Naval Postgraduate School, Monterey, CA, USA, September 2012.

[14] ISO. DRAFT INTERNATIONAL STANDARD ISO/IEC DIS 25185-1.2 Identification cards - Integrated circuit card authentication protocols - Part 1: Protocol for Lightweight Authentication of Identity. International Organization for Standardization, Geneva, Switzerland, 2014.

[15] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM, 21:120–126, February 1978.

[16] Federal Information Processing Standards Publications. Secure Hash Standard (SHS). Technical Report FIPS PUB 180-3, National Institute of Standards and Technology, October 2008.

[17] Knuth, D., The Art of Computer Programming, Vol. 2, 2nd ed., Addison-Wesley, Reading, (1981).

[18] W. Rankl and W. Effing. Smart Card Handbook. John Wiley & Sons, Inc., New York,NY, USA, 3 edition, 2003.

[19] Wolfgang Rankl and Kenneth Cox. *Smart Card Applications: Design models for using and programming smart cards*. John Wiley & Sons, Inc., New York, NY, USA, 2007.

[20] Patrick Gallagher, Deputy Director Foreword, and CitaFurlani Director. Fips pub 186-3 federal information processing standards publication digital signature

standard (dss), 2009.

[21] Keith Mayes and KonstantinosMarkantonakis. Smart Cards, Tokens, Security and Applications. Springer Science+Business Media, 2008.

[51] NIST, "Advanced Encryption Standard (AES)," Tech. Rep. FIPS PUB 197, National Institute of Standards and Technology (NIST), Gaithersburg, Maryland, United States, 2001.

[23] A Layman's Guide to a Subset of ASN.1, BER, and DER (An RSA Laboratories Technical Note) By Burton S. Kaliski Jr. Revised November 1, 1993 http://luca.ntop.org/Teaching/Appunti/asn1.html (cited February 2015)

[24] Introduction to ASN.1 Syntax and Encoding https://msdn.microsoft.com/en-us/library/windows/desktop/bb648640(v=vs.85).aspx (cited February 2015)

[25] Code::Blocks IDE http://www.codeblocks.org/home (cited March 2015)

[26] OpenSSL Library home page https://www.openssl.org/ (Online accessed March 2015).

[27] Ubuntu community page. Java (JDK and JRE) for ubuntu https://help.ubuntu.com/community/Java (accesed online, March 2015)

[28] OpenSSL RSA documentation page, for public decryption https://www.openssl.org/docs/crypto/rsa.html (accesed online, March 2015)

[29] OpenSSL random bytes documentation. for random number generation, https://www.openssl.org/docs/crypto/RAND_bytes.html (accesed online, April 2015)

[30] OpenSSL sha1 https://www.openssl.org/docs/crypto/SHA1_Init.html (accesed online, April 2015)

[31] Government International Standards for Business and Society. Iso/iec 7816. http://

www.iso.org/iso/search.htm?qt=7816&published=on&active_tab=standards.
[Online; accessed March, 2015].

[32] Government International Standards for Business and Society. Iso/iec 14443. http://www.iso.org/iso/search.htm?qt=14443&published=on&active_tab=standards. [Online; accessed March, 2015].

[33] Micro-SD embedded Java card. http://www.go-trust.com/products/microsd-java/ [Online accessed Jan, 2015]

[34] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc., New York, NY, USA, 1993.

[35] J. Beusink, "Secure access control to personal sensor information in federations of personal networks," 2012.

[36] F. Bersani and H. Tschofenig, "The EAP-PSK protocol: A pre-shared key extensible authentication protocol (EAP) method." http://tools.ietf.org/html/rfc4764.

[37] M. Badra and P. Urien, "Adding identity protection to eap-tls smartcards," in Wireless Communications and Networking Conference, 2007.WCNC 2007. IEEE, pp. 2951–2956, March 2007.

[38] P. E. Y. Sheffer, H. Tschofenig, "An extension for eap-only authentication in ikev2." http://tools.ietf.org/html/rfc5998.

[39] P. Eronen, C. Kaufman, Y. Nir, and P. Hoffman, "Internet key exchange protocol version 2 (IKEv2)." http://tools.ietf.org/html/rfc5996.

[40] Y. Sheffer and H. Tschofenig, "Internet key exchange protocol version 2 (ikev2) session resumption." http://tools.ietf.org/html/rfc5723.

[41] V. Devarapalli and K. Weniger, "Redirect mechanism for the internet key exchange protocol version 2 (ikev2)." http://tools.ietf.org/html/rfc5685.

[42] ActivIdentity, "Open protocol for access control identification and ticketing with privacy specifications," tech. rep., ActivIdentity, Silicon Valley, California, 2011.

[43] M. Fischlin and C. Onete, "A Cryptographic Analysis of OPACITY," Darmstadt University of Technology, Germany, pp. 1–46.

[44] G. Bernabé, *TLS embedded in smart card*. PhD thesis, University of Plymouth, 2012.

[45] Jean Paul Degabriele, Victoria Fehr, Marc Fischlin, Tommaso Gagliardoni, Unpicking PLAID, Information Security Group, Royal Holloway, University of London Cryptoplexity, Technische Universität Darmstadt, Germany