Leveraging UICC with Open Mobile API for Secure Applications and Services

by

Ran Zhou

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Fakultät für Elektrotechnik, Informatik und Mathematik
Heinz Nixdorf Institut und Institut für Informatik
Fachgebiet Softwaretechnik
Warburger Straße 100
33098 Paderborn

Leveraging UICC with Open Mobile API for Secure Applications and Services

Master's Thesis

Submitted to the Software Engineering Research Group
in Partial Fulfillment of the Requirements for the
Degree of

Master of Science

by
Ran Zhou
Am Rippinger Weg 1
33098 Paderborn

Thesis Supervisor:
Jun.-Prof. Dr. Christoph Sorge
and
Prof. Dr. Franz Josef Rammig

Paderborn, August 2012

Declaration

(Translation from German)

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original Declaration Text in German:

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

City, Date	Signature

Acknowledgement

This work would not have been possible without the support and help of many people. I wish to express my gratitude to my supervisor, Jun.-Prof. Dr. Christoph Sorge who has offered invaluable assistance and guidance with great patience. Deepest gratitude are also due to Prof. Dr. Franz Josef Rammig.

I would also give my deepest gratitude to Carsten Rust, who has helped me from the very beginning until the last second. Without his inspiration and numerous advices, this work would not be possible. The same gratitude is also for Andreas Leicher, who has inspirited me and helped me to achieve the goal.

Special thanks also to all my graduate colleges in the company Morpho e-Documents. They have always given me the most professional advices and the most thoughtful helps.

Abstract

Nowadays smartphones become as the cornerstone of the mobile information society. The growing market has strongly attracted hackers which bring serious security threats on the smartphone. The universal integrated circuit card (UICC) serves as the security anchor of the mobile network operator (MNO), and is contained in almost all mobile phones. The GSM association has stated that the UICC is the strategically best alternative as a secure element for mobile devices. Due to the architecture of smartphones, no secure element API is provided to application developers. In order to fill the gap between mobile applications and secure elements, SIMalliance has proposed and standardized Open Mobile API. Using this open API, a dual application architecture can be realized. This work is based on the use of Open Mobile API as the communication channel in the dual application architecture to implement a local authentication server of the Smart OpenID protocol. The implemented demo and testing show the possibility of using the UICC to enhance the security in the mobile world.

Contents

List of Figures

List of Tables

1 Introduction

Nowadays smartphones become as the cornerstone of the mobile information society. They allow the traditional personal information manager (PIM) usage and new applications to access enterprise applications at anytime and anywhere. Furthermore, transaction-based applications, such as the new ID card (nPA) and the non-contract electronic payment (NFC), would also be served [ASS12].

The growing market has strongly attracted hackers to make the potential for serious security threats on smartphones a reality [Lea11]. The 2011 mobile treats report from Juniper networks mobile threat center shows evidences of a significant increase of mobile malwares and other attacks [Net11]. Thousands of malwares: among them worms, Trojan horses and other viruses, have been unleashed against the smartphone [Hyp06], and this number increases rapidly according to the report from McAfee's Lab [McA12].

Although the smartphone contains plenty of sensitive information, users always underestimate the importance of security softwares on it [Kas11]. In the meantime, another survey reveals that the smartphone users are three times more vulnerable to phishing attacks [Boo11]. Since more people tend to use smartphone to handle business issues and banking transactions, these security threats need to be seriously considered and properly addressed.

The universal integrated circuit card (UICC) serves as the security anchor of the mobile network operator (MNO). It is used to identify and authenticate the subscriber into the mobile network. According to the statistics from the International Telecommunication Union (ITU), at the end of 2011, there were 6 billion mobile subscriptions, which is equivalent to 87 percent of the world population. With the Java Card technology, the UICC can provide even more additional functions such as digital signature and cryptography. However, until now the mobile application developers and today's operating system providers have largely ignored the potential of the UICC which is contained in almost all mobile phones.

The GSM association has stated that the UICC is the strategically best alternative as a secure element for mobile devices, which can be used in payment or other secure services [Sma09]. Security features of the UICC such as secure storage, cryptographic and secure update with OTA (over-the-air) make it a suitable element to enhance the security level of mobile applications and services [SIM11c].

The most smartphones have an architecture composed of two physically isolated subsystems, namely the application processor and the baseband processor for the UICC. The operating system runs on the application processor. Through a proprietary interface, the system software accesses the UICC to obtain the mobile network information or dial a number. For this reason, no application programming interfaces are provided by the smartphone to application developers for utilizing the functionality on the UICC.

1.1 Solution Idea

In order to fill the gap between mobile applications and secure elements, SIMalliance has proposed and standardized Open Mobile API. This open API enables mobile application developers to communicate with secure elements on the smartphone, such as the UICC. Therefore, the mobile application can store its sensitive data on the UICC and invoke cryptographic functions such as encryption, decryption, signature, and hashing.

A dual application architecture is introduced by GSMA for the near field communication (NFC) service in [Ass11]. In this architecture, the mobile application provides the user interface and other non-security sensitive functions, while the applet on secure elements provides the actual logic and secure functions. As the communication channel between both components, Open Mobile API is used in this master thesis.

To demonstrate the use of the UICC in security sensitive applications and services, a UICC-based authentication application is implemented using the dual application architecture. This application is a component of the Smart OpenID protocol [LSS12], which uses our implemented application as an authentication server on the smartphone.

The example implementation shows the common usage pattern of Open Mobile API on the smartphone. The most frequently used features on the UICC such as secure storage, atomic operation, cryptographic functions are covered. Therefore, the implementation of the local authentication server can be referenced for other secure applications and services.

1.2 Structure

Firstly, fundamental technologies are introduced in the second chapter to provide basic knowledge for the following parts. The state of the art is presented in the third chapter which includes an analysis of the security status in the mobile world. In the fourth chapter, the UICC-based dual application architecture, which uses Open Mobile API as the communication channel, is introduced as the framework of using the UICC to improve the security level of mobile applications. Two UICC-based authentication protocols are presented and analyzed to show the effect of integrating the UICC into security sensitive applications and services. One of these protocols is Smart OpenID, which contains a local authentication server on the smartphone. The last two chapters implement and test the UICC-based authentication server to show the feasibility of Smart OpenID protocol and the usability of the dual application architecture.

2 Fundamental Technologies

The universal integrated circuit card (UICC) is a smart card with an operating system that is compliant with ISO/IEC 7816 standards and optimized for applications in the mobile telecommunication sector [RE10]. Therefore, the introduction of the UICC is based on the characteristics of the smart card. As a popular operating system for the UICC, Java Card is the second part of this chapter. To utilize the UICC for applications on mobile phones, SIMAlliance has introduced and specified Open Mobile API. The introduction of this secure element access API contains the architecture as well as the usage pattern. As the host of Open Mobile API, the Android operating system is chosen and introduced.

2.1 UICC

2.1.1 Overview

The UICC is the trusted anchor of the mobile network operator (MNO). It resides on the mobile phone and has the functionality to authenticate the user into the mobile network [CGRS09]. As the identity of the subscriber, the UICC is identified by the international mobile subscriber identity (IMSI), which is a unique identification in the mobile network. Besides network access applications (NAA) such as SIM and USIM, the UICC also supports other applications, for example, smart card web server and Toolkit applications. The physical and logic characteristics of the UICC are specified in the ETSI TS 102.221 specification.

2.1.2 Smart Card

The characteristic feature of the smart card is an integrated circuit embedded in the card, which has components for transmitting, storing and processing data [RE10]. According to components and functionalities, smart cards can be divided into different groups. Memory cards, which contain EEPROM, ROM and a security logic module, are optimized for the prepaid telephone card and the health insurance card. Microprocessor cards, which contain EEPROM, ROM, RAM, I/O interface and a microprocessor, are optimized for hosting a single or multiple applications.

In microprocessor smart cards, data can only be accessed via a serial interface that is controlled by the operating system. Therefore, the sensitive data can be written and stored preventing from being accessed by unauthorized parties. The

Part	Title
1	Cards with contacts: Physical characteristics
2	Cards with contacts: Dimensions and location of the contacts
3	Cards with contacts: Electrical interface and transmission protocols
4	Organization, security and commands for interchange
5	Registration of application providers
6	Interindustry data elements for interchange
7	Interindustry commands for Structured Card Query Language(SCQL)
8	Commands for security operations
9	Commands for card management
10	Cards with contacts: Electronic signals and answer to reset for synchronous cards
11	Personal verification through biometric methods
12	Cards with contacts: USB electrical interface and operating procedures
13	Commands for application management in multi-application environment
15	Cryptographic information application

Table 2.1: ISO/IEC 7816 Standards

computation of cryptographic algorithms is also supported on smart cards. These security features allow the smart card to be implemented as a secure element.

The UICC is a microprocessor smart card that supports multiple applications.

2.1.3 ISO/IEC 7816 Standards

ISO/IEC 7816 is a family of international standards for the integrated circuit cards (ICC) with 14 parts specifing different aspects and characteristics of smart cards. Table 2.1 shows the whole standard.

2.1.4 Smart Card Operating System

The initial usage of smart cards was for special applications, and there was no actual operating system on it. The smart card used in the C-Netz (the German precursor of the GSM system) starting in 1987 had an operating system that was optimized for one application and included a specific transmission protocol, special commands, and a tailored file system [RE10]. This is the start of the evolution of the smart card operating system.

The modern smart card operating system supports multiple applications, file management, memory management and also third-party programs. Such operating systems are called open platform. Java Card is one of these operating systems conforming to the ISO/IEC 7816-4 standard.

2.1.5 Global Platform

Global Platform (GP) is an international association to standardize technologies for multi-application smart cards. The GP specification has become the de facto standard for loading and managing Java-based applications in the Java Card operating system [RE10]. In addition, the specification also defines the notion of secure communication over and above that defined in ISO/IEC 7816 standards.

2.1.6 Basic Encoding Rules

Although the smart card has a much bigger memory capacity compared to the magnetic strip card, it is still never enough. A good data encoding mechanism can help saving on memory usage and improving data transmission. One popular data encoding rule used in smart cards is the basic encoding rules (BER), which is a part of the abstract syntax notation one (ASN.1) standard [IT02].

The data object, which is created according to this rule, is called BER-TLV-coded data object. TLV is the abbreviation for tag-length-value structure, where T defines the type of the object, "L" is the length of the data and "V" represents the actual data. An end-of-content part can be included and placed after the data, when the length requires. The BER-TLV-coded data object should be encoded in octets (one 8-bits unit). Figure 2.1 depicts the structures of two possible encodings.

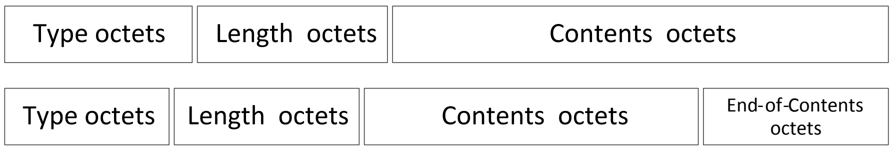


Figure 2.1: Structures of BER Encoding

2.1.7 PIN

A personal identification number (PIN) is a secret number used to authenticate the cardholder. Regarding the UICC, ETSI TS 102.221 defines the types of PIN that shall exist, namely the universal PIN, the application PIN, the local PIN and the administrative PIN.

2.1.8 File System

Files are stored and managed hierarchically in smart cards. The modern smart card file management system uses an object-oriented file structure. All the information about the file are stored in the file itself. The information are organized as the file control parameters (FCP) template and stored in the file header, while the file body, which is linked to the file header using a pointer, stores the actual data. Each FCP template is a BER-TLV object that contains a list of TLVs.

The smart card file system contains two categories of files: dedicated files (DF), which can be considered as the directory file, and elementary files (EF) holding the actual user data. The master file (MF) is a special type of the DF as the root of the file system. Instead of using physical addresses, files are identified with the 2-byte file identifier as the logical name. File identifiers can be reserved by standards. For historical reason, the MF has always the identifier “3F00”. Figure 2.2 depicts an example of the tree-like file system.

Regarding the UICC, ETSI TS 102.221 specifies the application dedicated file (ADF) as a special DF that contains all DFs and EFs for an application. The ADF can be referenced either with the 2-byte file identifier or with the application identifier (AID). In contrast to the DF, the ADF cannot be placed under the MF, which makes ADFs similar to the MF.

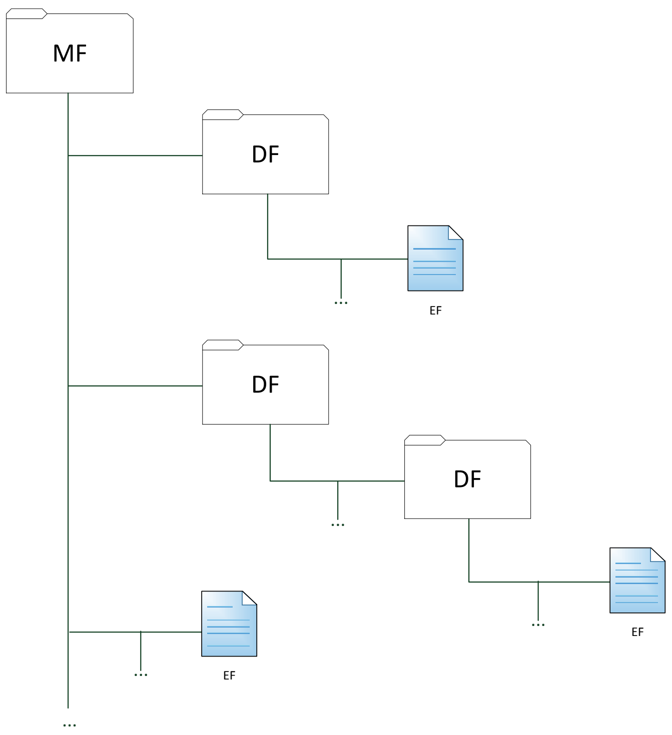


Figure 2.2: File System in Smart Cards

2.1.9 File Access Conditions

As part of the object-oriented structure, all files include information that governs access to the file in the context of file management [RE10]. The UICC adopts command-oriented access conditions. An access table, which contains information that specifies which commands must be successfully executed for each type of access, is stored in the file's FCP template.

2.1.10 Commands

In the realm of smart card operating systems, commands are instructions to the smart card to perform specific actions [RE10]. ISO/IEC 7816-4 standard defines the commands for smart cards in general. Global Platform specification defines the commands for application management. In the telecommunication sector, the commands are specified for GSM (3GPP TS 51.011 and 3GPP TS 51.014) and USIM (ETSI TS 102.221 and ETSI TS 102.222). Figure 2.3 shows the commands in the standards and specifications described above. For the specific application areas, only subsets of these commands are implemented in the smart card operating system.

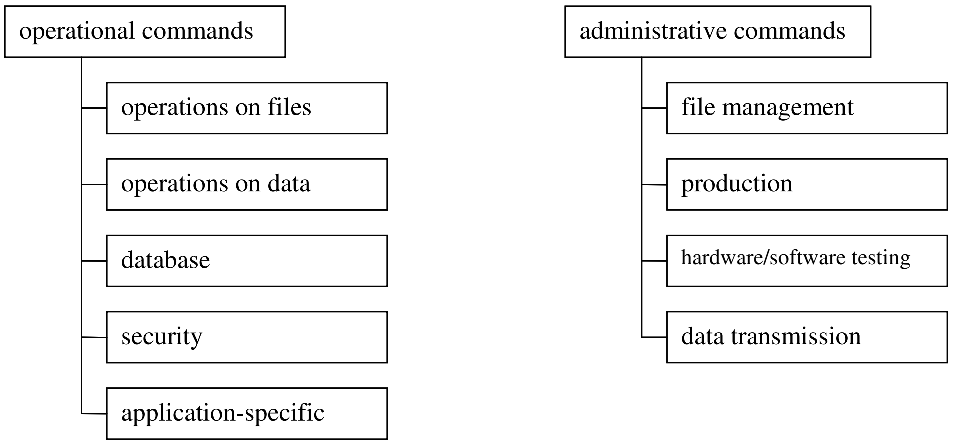


Figure 2.3: Typical Smart Card Commands [RE10]

2.1.11 Application Protocol Data Unit

To exchange commands and data between the terminal and the smart card, application protocol data units (APDU) are used. APDUs can be divided into two categories: command APDU being sent to the smart card and response APDU being returned from the smart card. The structure of the APDU is defined in the

ISO/IEC 7816-4 standard. Table 2.2 shows the structure of the command-response APDUs.

The coding of fields in the UICC is compliant with the above mentioned standard. The concrete definition of CLA, INS and SW bytes is specified in the ETSI TS 102.221 specification.

Field	Description	Number of bytes
Command header	Class byte denoted CLA	1
	Instruction byte denoted INS	1
	Parameter bytes denoted P1-P2	2
Lc field	Absent for encoding Nc = 0, present for encoding Nc >0	0, 1 or 3
Command data field	Absent if Nc = 0, present as a string of Nc bytes if Nc >0	Nc
Le field	Absent for encoding Ne = 0, present for encoding Ne >0	0, 1, 2 or 3
Response data field	Absent if Nr = 0, present as a string of Nr bytes if Nr >0	Nr (at most Ne)
Response trailer	Status bytes denoted SW1-SW2	2

Table 2.2: Structure of Command-Response APDUs

2.1.12 Applications

Before the introduction of the UICC, the SIM is a smart card with the specialized application designed for the GSM network. By contrast, the UICC supports multiple applications that enable mobile phones to be authenticated not only in the GSM network but also in the UMTS or CDMA network. To enable the GSM network access, a SIM application must be present on the UICC, while the USIM application is used to access the UMTS network. The SIM and USIM are called network access applications (NAA). Depending on the capability of the mobile phone, only one of them could be selected.

Besides the NAA, other types of application are also present on the UICC. For example, the mobile network operator provides value-added services with applications based on the toolkit technology. The near field communication (NFC) application is developed and deployed on the UICC to provide contactless payment functionality. Applications from the third party can also be installed on the UICC to extend its usability.

2.1.13 Logical Channels

In microprocessor smart cards, multiple applications can be present. The logical channel enables the exchange of commands and data between the terminal and multiple applications. The required channel number is encoded in the CLA byte of the SELECT command. Once the application is selected, the subsequent APDUs will be dispatched to the application in the opened logical channel. The smart card that supports logical channels must have one basic channel (channel number is zero), which must be permanently available, and at least one additional logical channel. The interleaving of command-response pairs is prohibited, which means that only one channel can be active at a time.

Regarding the UICC, the basic channel is occupied by the SIM or the USIM application, while logical channels can be occupied by other applications.

2.1.14 Cryptography

The cryptographic techniques have a big significance in smart cards for two motivations [Che00]:

- To ensure application security as well as data transmission security between the smart card and the terminal
- To function as a security module through which the security of other systems can be enhanced

The four objectives of cryptography are: authenticity, confidentiality, integrity and non-repudiation [RE10]. Consider the first motivation of using cryptographic regarding these objectives:

- **Authenticity**: in which the involved parties must be validated to be who they claim they are. For example, the mobile payment applet stores the electronic currency of the cardholder. Before a payment is actually made, a mutual authentication must be carried out to confirm that the merchant and the customer are genuine. In addition, the authenticity of data can also be ensured by using cryptographic functions such as message authentication code (MAC).
- **Confidentiality**: in which messages or data being transmitted are not viewable by the unauthenticated entity. For example, after the mutual authentication, the mobile payment applet makes a payment by sending the sum of money, the bank account number and the TAN-code. To ensure the privacy of the customer, the data must be encrypted before transmitting.
- **Integrity**: in which the data cannot be modified undetectably. Regarding the mobile payment applet, the transferred sum of money must be kept unmodified, otherwise, the merchant or the customer will suffer from loss.

To ensure the integrity of data, the message authentication code can be used. The client applet computes the MAC of the encrypted messages and includes it in APDUs. Upon receipt of the APDUs , the merchant must compute the MAC itself and compare it with the one supplied by the customer.

- **Non-repudiation**: in which the involved parties of a transaction cannot deny its participation. Regarding the mobile payment example, the customer cannot deny her/his confirmed payment, while the merchant cannot deny the received payment. Public key cryptography can be used to ensure this property as well as the confidentiality. In case of sending files or Emails, digital signatures can be used to establish non-repudiation.

Smart cards can also be used as the secure module, for example, the UICC in mobile phones or the smart card in TV set-top-boxes. Such application scenarios include a network server that provides services and a client that requires services. To get access to these services, the user must be authenticated with the corresponding smart card. The UICC contains the international mobile subscriber identity (IMSI) and an authentication key (Ki) for the GSM network. After the mobile phone is powered on, it obtains the IMSI from the UICC and sends it to the mobile network Operator (MNO). The MNO searches for the associated Ki for the claimed IMSI and generates a challenge in the form of a random number, and then it signs this random number with the found Ki, the result is called *signed response 1*. Upon receipt of the challenge, the UICC signs it with the Ki to get the *signed response 2*, and then sends it to the MNO. Then the MNO compares the received response with the *signed response 1*, if they are identical, then the UICC is authenticated.

2.1.15 Secure Channel Protocol

Secure channel protocol (SCP) is a secure communication protocol that provides a set of secure services. The Global Platform specification defines the secure channel protocol with the following secure services [Glo11]:

- **Entity authentication**: in which the card or the off-card entity proves its authenticity to the other entity through a cryptographic exchange

- **Integrity and authentication**: in which the receiving entity (the card or off-card entity) ensures that the data being received from the sending entity (respectively the off-card entity or card) actually came from an authenticated entity in the correct sequence and has not been altered

- **Confidentiality**: in which data being transmitted from the sending entity (the off-card entity or card) to the receiving entity (respectively the card or off-card entity) is not viewable by an unauthenticated entity

Different versions of the secure channel protocol (for example, SCP02, SCP10 and SCP80) are defined according to the concrete implementation of the secure services.

2.1.16 Over-the-Air

Over-the-air (OTA) is a technology that manages data and applications in smart cards after they are issued. Figure 2.4 shows the OTA infrastructure. The back-end system includes the mobile network operator, the content provider, the billing system and so on. They use OTA to maintain the issued smart cards, provide the value-added services. The OTA gateway receives the service data from the back-end system, transforms the service data into the secured packet format, and transports it using the corresponding bearer such as SMS.

GSM 03.48 specification defines the secured packet structure in the SMS for the SIM. ETSI TS 102.225 (SCP80) specification generalizes the secured packet structure for the UICC and extends the bearer from the SMS to the HTTPS. The contents of secured packets are encrypted commands for the UICC. The UICC stores the received secured packets in a particular buffer and restores them to the correct sequence. The commands can be divided into two groups:

- **Remote file management (RFM)**: in which files can be selected, read, updated, invalidated and rehabilitated.
- **Remote applet management (RAM)**: in which applets can be loaded, installed and deleted.

2.2 Java Card

2.2.1 Overview

Java Card stands for an open operating system for the smart card. As an ISO/IES 7816 compatible platform, Java Card provides access to the common smart card file system. Furthermore, Global Platform specification defines the standard for loading and managing Java-based applications in Java Card.

2.2.2 Java Card Language

Due to the limitation of the memory capacity, the Java Card language is a subset of the Java language. All the supported features in Java Card also exist in Java and have the same behaviors. Table 2.3 depicts an overview of the Java Card language.

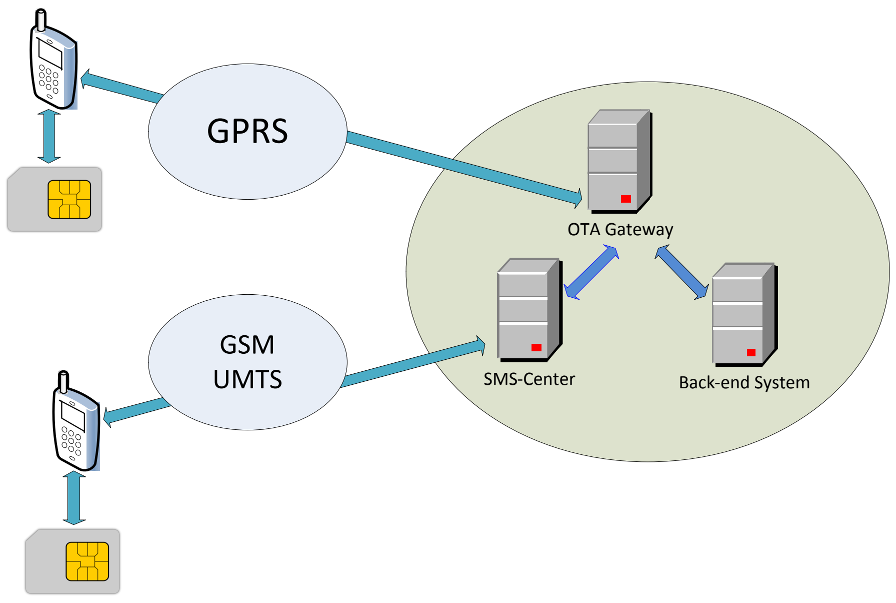


Figure 2.4: OTA Infrastructure

2.2.3 Java Card Virtual Machine

Due to the limitation of system resources in smart cards, the Java Card virtual machine (JCVM) applies a split model (Figure 2.5). The code, which is written in the Java Card language, is firstly converted using the off-card converter on a PC or a workstation. During the converting, the static check is performed. At the end, a CAP (converted applet) file is generated. The Java Card off-card installer transmits the executable bytecode in a CAP file to the on-card installer that writes the bytecode in the smart card memory and links it with the needed class. At the end, the on-card interpreter executes the Java Card bytecode.

2.2.4 Java Card Runtime Environment

The Java Card platform runtime environment contains the Java Card virtual machine (VM), the Java Card application programming interface (API) classes (and industry-specific extensions), and support services [SM06]. Therefore, it can be considered essentially as the smart card's operating system. Figure 2.6 shows the JCRE in smart cards.

System classes, which are analogues to the core of the operating system, are on the basis of the JCVM and native methods. The Java Card framework classes define the API for the development of Java Card applets. The industry-specific extensions are used to provide additional services or interfaces for specific domains. The installer is provided to enable download and installation of applets onto the issued smart card.

Supported Java Features	Unsupported Java Features
• Small primitive data types: boolean, byte, short, one-dimensional arrays, Java packages, classes, interfaces, and exceptions • Java object-oriented features: inheritance, virtual methods, overloading and dynamic object creation, access scope, and binding rules • The int keyword and 32-bit integer data type support are optional • The Garbage Collector	• Large primitive data types: long, double, float • Characters and strings • Multidimensional arrays • Dynamic class loading • Security manager • Threads • Object serialization • Object cloning

Table 2.3: Overview of the Java Card Language Subset[Che00]

2.2.5 Java Card APIs

The Java Card APIs contain the mandatory core packages and optional extension packages (with an "x"). The packages for specific domains can also be included for application developers, for example, the file access API defined in the ETSI TS 102.241 specification.

2.2.6 Java Card Memory Model

Three types of memory are used in smart cards: ROM, EEPROM and RAM. ROM is used to store the operating system and relevant data during the manufacture process, afterwards it cannot be modified anymore. EEPROM and RAM are rewritable. The most significant difference between them is: EEPROM does not lose the content in case of a power loss (nonvolatile). Additionally, EEPROM is low priced compared to RAM, while RAM is 1,000 times faster by write operations [RE10]. For these reasons, EEPROM typically has a greater capacity than RAM on the smart card. EEPROM is normally used to save the downloaded applets and data, while RAM is for the intermediate results of different applets.

According to the memory model in smart cards, Java Card supports two types of objects: persistent objects and transient objects. Persistent objects, which are stored in EEPROM, are able to exist beyond the execution time of a process.

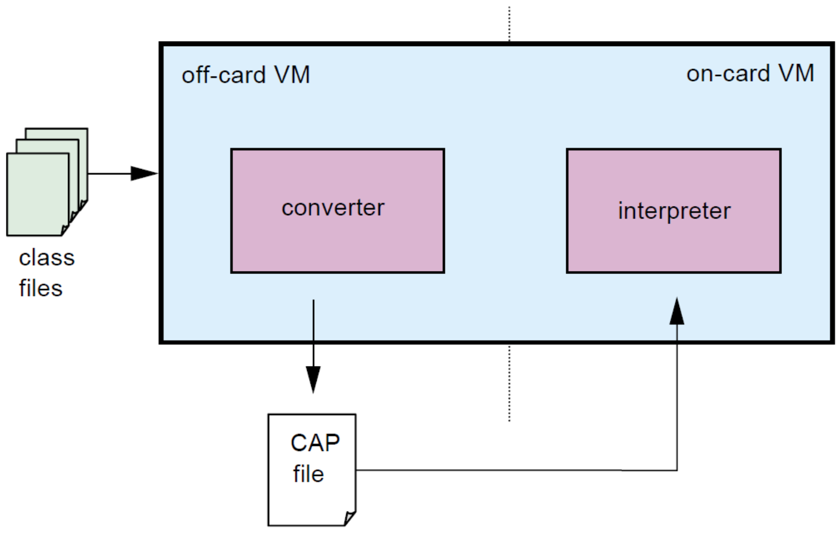


Figure 2.5: Java Card Virtual Machine [Che00]

Transient objects contain essentially two parts: the data field in RAM and the object reference in EEPROM. Transient objects are normally used for fields that need frequently write operations. Figure 2.7 shows transient objects in Java Card.

2.2.7 Transaction Integrity

The data integrity is essential in smart cards. A sudden power loss might cause the integrity problem of a uncompleted transaction. The JCRE provides the atomic operation mechanism to ensure the transaction integrity.

The transaction model in Java Card is similar to the data base transaction, which uses *begins*, *commit* and *rollback* routines to ensure the atomicity. The class *JCSystem* from package *javacard.framework* provides the methods to build the Java Card transaction routine:

```
// Starts a transaction
JCSystem.beginTransaction();

...
// Commits a transaction
JCSystem.commitTransaction();
```

Only when the *JCSystem.commitTransaction()* is reached, all changes to the data can be committed. Otherwise, all changes will be aborted, and the *TransactionException* will be thrown.

2.2.8 Java Card Applets

The Java Card applet must extend the class *Applet* from the package *javacard.framework*, and implement the *install*, *process*, *select* and *deselect* methods.

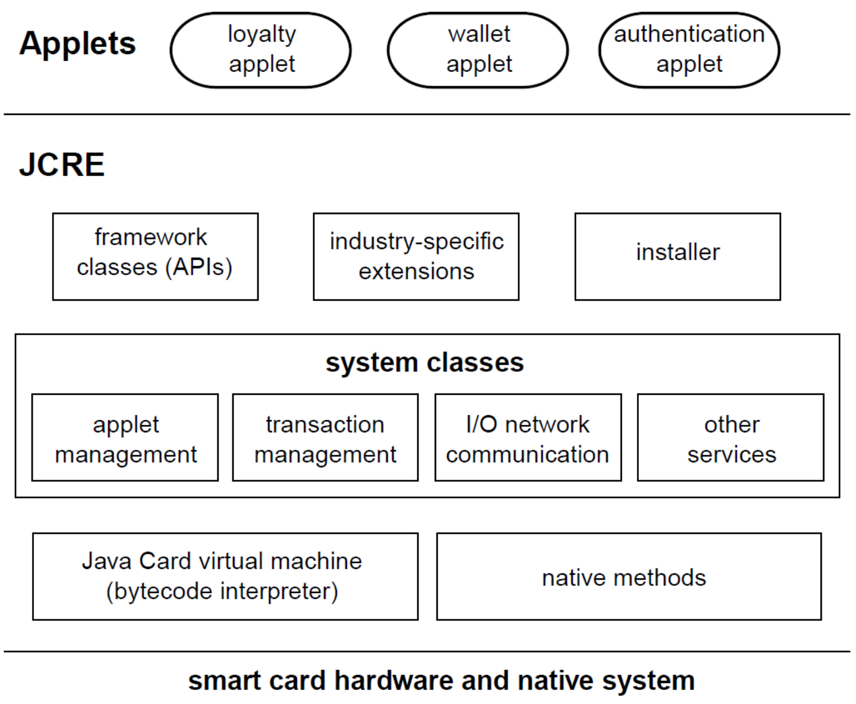


Figure 2.6: On-card System Architecture [Che00]

2.2.9 APDUs in Java Card

The application protocol data unit, which is used in smart cards to transmit data between the terminal and the card, is supported in Java Card with the class *javacard.framework.APDU*. Upon receipt of APDUs from the terminal, the JCRE creates the APDU object and dispatches it to the current selected applet. The *process* method in the applet handles the received APDU object and creates the response to the terminal. An example of the *process* method is as follows:

```
/**
 * Processes an incoming APDU.
 * @see APDU
 * @param apdu the incoming APDU
 * @exception ISOException with the response bytes per ISO 7816-4
 */
 public void process(APDU apdu)
 {
     // Returns the APDU buffer byte array
     byte buffer[] = apdu.getBuffer();
     // Reads the data into the APDU buffer
     short bytesRead = apdu.setIncomingAndReceive();

     ...
     // Sets the data transfer direction to terminal
     apdu.setOutgoing();
     // Sets the length of the response data
     apdu.setOutgoingLength((byte)2);
     // Sends the data from the offset with the given length
     apdu.sendBytes((short)0, (short)2);
 }
```

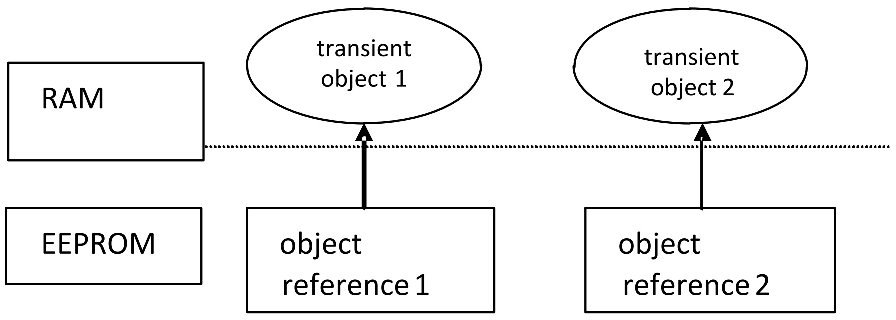


Figure 2.7: Transient Objects in Java card [Che00]

2.2.10 Java Card Cryptography APIs

The Java Card cryptography APIs are designed according to the Java cryptography architecture (JCA). This architecture is designed to separate the interface from the implementation. Therefore, the JCRE provider can tailor the implementation and only provides the necessary code to save the memory in smart cards.

The *jacacard.security* package provides definitions of algorithms and factories for building cryptographic keys. The most important classes are depicted in Table 2.4.

Class	Description
KeyBuilder	This class is a key object factory.
MessageDigest	This class is the base class for hashing algorithms such as MD5 and SHA.
RandomData	This class is the base class for random number generation.
Signature	This class is the base class for signature algorithms such as RSA.
KeyAgreement	This class is the base class for key agreement algorithms such as Diffie-Hellman and EC Diffie-Hellman.

Table 2.4: Classes of Java Card Cryptography APIs

2.3 Open Mobile API

2.3.1 Overview

The SIMalliance OpenMobileAPI is a software interface between the application on mobile phones and the applet on secure elements. A three-layer architecture is defined in [SIM11a] (Figure 2.8):

- **Transport layer** provides general access to secure elements using APDUs. This layer forms the foundation for the other layers.
- **Service layer** abstracts the available functions, such as cryptography and authentication, on the secure element. The application developer can utilize the services provided by the secure element without concerning about the underlying transport API.
- **Application layer** represents the various applications that uses Open Mobile API.

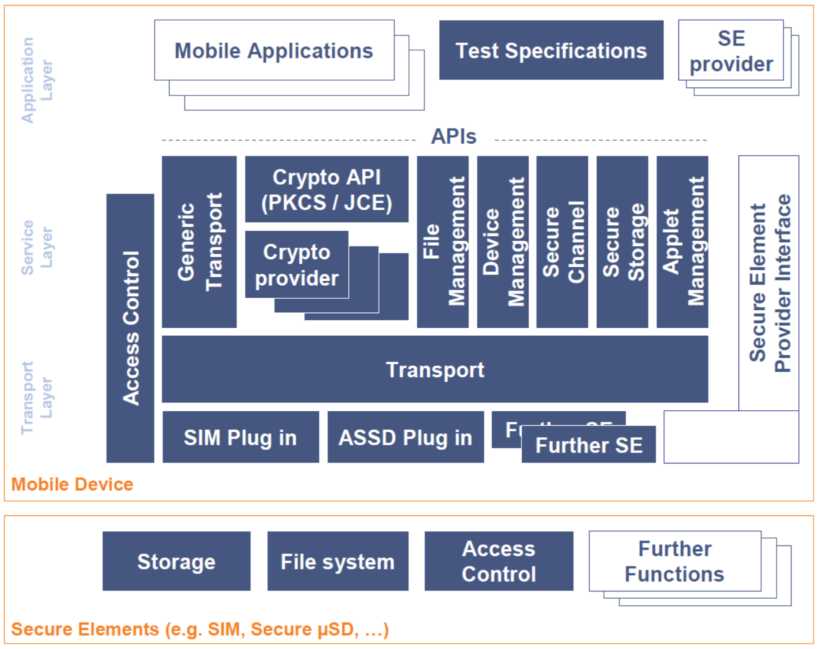


Figure 2.8: Architecture of SIMalliance Open Mobile API [SIM11a]

2.3.2 Transport Layer API

As the interface between the secure element and the mobile application, the transport layer is responsible to transmit command-response APDUs. Table 2.5 depicts the classes of the transport layer.

2.3.3 Service Layer API

The service layer abstracts the available services and functionalities provided by the secure element. Different types of secure elements exist and can be used with Open Mobile API; therefore this layer depends on the concrete type of the secure element, and should be specified by the corresponding organization.

Class	Description
SEService	This class is the entry point of Open Mobile API.
SEService: CallBack	This class receives call-backs when the service is connected.
Reader	This class represents the available secure element readers connected to the mobile phone.
Session	This class represents a session with the secure element.
Channel	This class represents a channel (ISO/IEC 7816-4) opened with the secure element.

Table 2.5: Classes of the Transport Layer [SIM11a]

2.3.4 Usage Pattern

To access the secure element via Open Mobile API, the transport layer API should be used. The following steps describes a general usage pattern:

1. The mobile application creates an object of the *SEService* class to represent the connection between the application and the subsystem implementing the secure element access functionality. An object implementing the *SEService.Callback* is passed as the parameter. The callback method is called, when the connection to the secure element has been established.

2. Using the *SEService* object, all the available readers are enumerated. The application selects a reader and opens a session with the included secure element.

3. With the session, the mobile application retrieves and validates the answer-to-reset (ATR), which contains various parameters related to the secure element.

4. After validating the ATR, the mobile application opens a logical channel with the applet by specifying the identifier.

5. In the opened channel, the mobile application transmits APDUs with the applet.

6. At then end, the mobile application closes the opened channel, the session as well as the connection to the secure element.

2.4 Android

Android is an open source operating system for mobile devices. The kernel of Android is based on the Linux kernel with architecture changes by Google. The operating system is programmed in C, while the most applications are developed with Java and run on the virtual machine Dalvik.

2.4.1 Android Runtime

The Android runtime environment consists of the Dalvik virtual machine and the core libraries [MK09]:

- **Dalvik virtual machine**: the main difference between the Java virtual machine and the Dalvik virtual machine is that, the Dalvik VM has a register-based architecture, while the JVM is a stack machine. Due to the limited resources on mobile devices, the Dalvik VM is optimized to be run on a slower CPU with less memory.

- **Core Libraries**: The core libraries of the Android runtime are a subset of the Apache Harmony Java implementation [Fou11], which is an open source Java SE. Several APIs from the Java SE are removed (for example, javax.print), since these functionalities are not really relevant to the mobile device. The javax.crypto and javax.security classes are included to provide cryptographic functionalities.

2.4.2 Android Application Framework

In Android, each application has an *AndroidManifest.xml* file presenting the essential information of the application to the operating system. The manifest file contains the information of the package and various components such as activities, services, broadcast receivers, and Intent filters. It also declares the permissions allocated to the application. The permissions are used to limit the access to specific components and resources of the operating system. An example of the *AndroidManifest.xml* file is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
   xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.morpho.soid.nx" android:versionCode="1"
   android:versionName="1.0">
   <uses-sdk android:minSdkVersion="10" />

   <application android:icon="@drawable/icon" android:label="@string/app_name">
      <activity android:name=".SOID_nxActivity" android:label="@string/app_name">
      <intent-filter>
        ...
      <intent-filter>
      </activity>
   </application>
   <uses-sdk android:minSdkVersion="10" />
   <uses-permission android:name="org.simalliance.openmobileapi.SMARTCARD">
   </uses-permission>
</manifest>
```

An application can contain multiple activities. An activity is designed as an encapsulation of a specific action that the user performs. Activities can optionally associate with a user interface. One of these activities is specified as the main activity presented to the user, when the application is launched.

2.4.3 Activity Lifecycle

An activity in the Android application must extend the *android.app.Activity* class, and implement the callback methods that are called during the states transition. The possible states of an activity are [MK09]:

- **Active (running)**: the activity is visible and has the user focus.
- **Paused**: a new activity comes to the foreground and gets the user focus, but its user interface is partially transparent or does not cover the whole screen. In this case, the former activity is paused and still maintains system states.
- **Stopped**: the stopped activity is completely invisible, but it still maintains system states.
- **Destroyed**: when the system resources must be released for other activities, the paused or stopped activities could be destroyed by the operating system.

To handle the states transition, the corresponding callback methods are called by the system, they are:

- **onCreate(Bundle savedInstanceState)**: the activity is being created.
- **onStart()**: the activity is about to be visible.
- **onResume()**: the activity is visible (it is in the state "active").
- **onPause()**: the activity is about to be paused, while another activity is taking the user focus.
- **onStop()**: the activity is about to be stopped (not visible).
- **onDestory()**: the activity is about to be destroyed.

2.4.4 Intent

In Android, the Intent is a mechanism enabling the interaction between components, such as activities, services, and broadcast receivers, in the same or different applications. An Intent object describes the action to be performed, and the data to be operated on. The most important usage of Intent is to start an activity. Two types of Intents exist in Android, they are:

- **Explicit Intent**: the target component is specified explicitly by the name.
- **Implicit Intent**: no target component is specified. The system performs the Intent resolution to find the best component.

The parameters for initializing an Intent object include component name, action, data, category, extras and flags. Some of the most important actions are [Gui12]:

- **ACTION_CALL**: initiate a phone call.
- **ACTION_EDIT**: display data for the user to edit.
- **ACTION_MAIN**: start up as the initial activity of a task, with no data input and no returned output.
- **ACTION_SYNC**: synchronize data on a server with data on the mobile device.
- **ACTION_VIEW**: display data to the user.

The data is specified with the uniform resource identifiers (URI). Actions are paired with specific types of data, for example, the action *ACTION_VIEW* can be operated on an *http*, which is referred with the data URI.

The operating system must know the capabilities of an application to support the Intent resolution. Therefore, the Intent filters are included in the *AndroidManifest.xml* file in form of *intent-filter* elements. A filter specifies the action, data and category, which will be checked during the Intent resolution.

2.4.5 Securtiy

As stated in [Inf], the Android security features can be divided into 2 groups:

- System and kernel level security
- Android application security

At the operating system level, the Android platform provides the security of the Linux kernel, application sandbox, system partition and safe mode, file system permissions and encryption, password protection, device administration, and memory management.

The third-party applications are becoming the key component on modern smartphone platforms [Top12]. Therefore, a secure application management mechanism is necessary to protect the user from threats caused by malicious applications. The Android application security includes: permission model, interprocess communication, device metadata, application signing, digital rights management, and android updates. The Android permission model is the core of the application security.

To ease the development of the third-party application, APIs are provided to developers. There are APIs relating to sensitive functionalities such as camera, GPS and bluetooth. Since a malicious application can use these functionalities to track the user and violate the privacy, their usage must be controlled strictly. Some APIs are cost-sensitive, such as SMS/MMS, network/data and NFC. The misuse of these APIs may generate additional cost. To use the protected APIs, an application must include the relevant permissions in its *AndroixManifest.xml* file. Before installing the application, the user will be asked whether to grant these permissions or not.

2.4.6 UICC Access in Android

Smartphones are usually composed of two physically isolated subsystems, which are connected only through the well-defined interfaces [VLP11]. One of the subsystems is the so-called application processor running the operating system and applications, while the other one is the baseband (radio) that implements the interface to the mobile network.

Android provides the telephony services to monitor the basic phone information. The radio layer interface (RIL) is an abstract layer between the telephony services and the baseband processor.

The RIL consists of two primary components [Gui]:

- **RIL daemon**: which initializes the vendor RIL, processes all communication from Android telephony services, and dispatches calls to the vendor RIL as solicited commands.
- **Vendor RIL**: which is specified for the baseband, processes all communication with the baseband processor, and dispatches calls to the RIL daemon through unsolicited commands.

The communication between the vendor RIL and the baseband is in the form of Hayes AT commands [3GP12a], as depicted in Figure 2.9.

If a customer application wishes to communicate with the UICC through the RIL, the need AT commands must be supported by the RIL. However, there are two restrictions in the Android operation system for accessing the UICC by a customer application:

1. The Android telephony framework does not provide the needed APIs to access the UICC.
2. The RIL does not support all the needed AT commands.

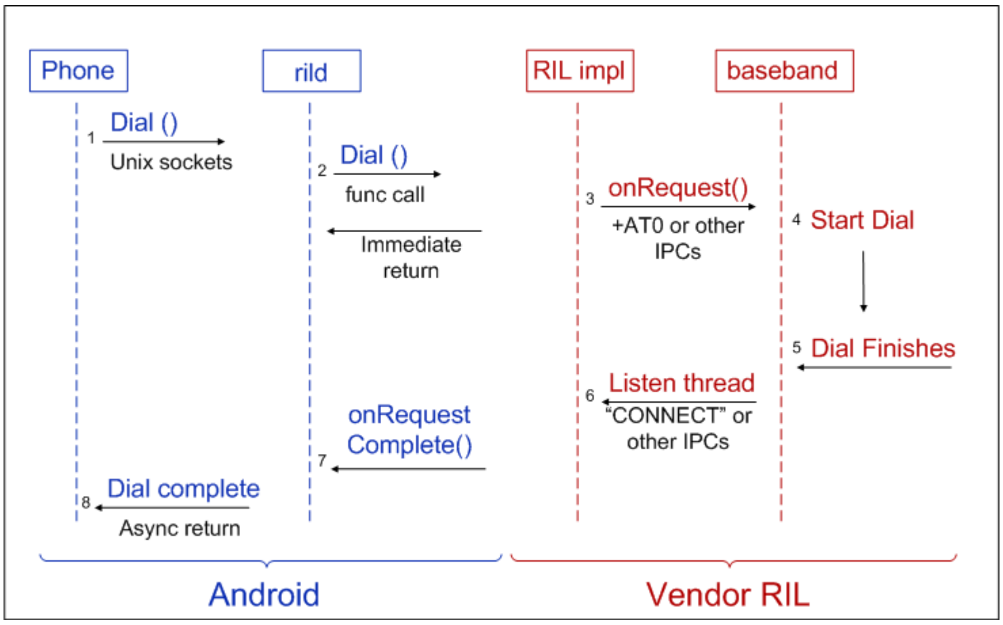


Figure 2.9: A Solicited Call in Android[Gui]

3 State of the Art

The security in mobile environment has been relatively easy to manage, since mobile phones were traditionally used for voice communication and messaging. Feature phones are advanced mobile phones that offer more functionalities such as digital camera, gaming, music and video streaming. These applications are preloaded and managed by manufacturers; therefore the security of feature phones could still be managed. A smartphone is a mobile phone that offers more advanced computing ability and connectivity than a contemporary basic feature phone [JKLW11]. The advanced smartphone operating system allows a huge developer community to build advanced applications and services [IA06]. As smartphones become the handheld computer and the personal assistant, applications and services offered, such as cloud storage and payment, become more security sensitive. In the meantime, the software as a service (SaaS) grows rapidly as a new model of software distribution for business applications such as accounting and enterprise resource planning (ERP). Users can access SaaS with the application on smartphones as a thin client, while the business logic and data are hosted externally by the vendor.

Since the value of data being stored on smartphones and transferred across the web grows, it also highlights significant challenges as the malware, virus and hacking threats of the desktop and Internet go mobile [SIM11b]. For this reason, the vulnerability of mobile devices as well as the mobile Internet should be carefully analyzed and addressed to protect the user's assets from being attacked.

Identity management is an important element of the most network-based applications and services. A lot of attacks are intended to violate the identity and then get rewards by gaining access as the victim. The focus of this chapter is to present the vulnerability when using the conventional username/password authentication scheme. The HTTP authentication and OpenID are chosen to be analyzed. Both protocols use password as the authentication factor, which is vulnerable to certain attacks. In the next chapter, the discovered vulnerability will be addressed with UICC as another authentication factor.

3.1 Security in Mobile World

3.1.1 Assets of Smartphone

Assets of mobile phones are mainly private information and the device itself. Private information include the address book, the calling history and SMS messages. The device consists of the connectivity and resources, for example, CPU, RAM and battery.

A smartphone is an advanced mobile phone, and therefore possesses the above mentioned assets. In addition, applications on smartphone can also be defined an asset [JKLW11]. After being launched by the smartphone operating system, applications occupy system resources, such as CPU and RAM, and the connectivity, such as wireless network. Some applications need to access the private information, such as the address book, personal location, and Emails. Furthermore, applications are able to store the username's identifier and password. Therefore, applications on the smartphone should be managed carefully to protect assets of the user.

3.1.2 Secure Mobile Applications

Applications on mobile phones can be categorized according to their platforms, domains, and issuers. Mobile network operators (MNO) deliver their applications on the UICC, while application providers allow users to download applications and install them on the smartphone. Secure mobile applications could be grouped into the following categories [Gem08]:

- **Financial applications**: take several forms, such as online-banking applications, payment applications, loyalty applications and tickets applications.
- **Access applications**: are any applications that access service via authentication before service is granted, for example, virtual private network (VPN) applications. The software as a service (SaaS) also belongs to this group, since the user needs to log in with the thin client on the smartphone, before the service is provided by the server.
- **Content protection applications**: are used to protect the intellectual property rights of authors and producers of contents, such as ring tones, games, music and video. In addition, the sensitive data of the user should also be protected with such applications.
- **Authentication applications**: refer to applications that authenticate users, user equipments (UE) to network-based services, for example, the generic bootstrapping architecture (GBA), public key infrastructure (PKI) and certificate based authentication.

The smartphone provides the developer a more powerful and friendly development environment compared to the UICC. Applications on smartphones have graphical user interface, and can access the service on the server with faster connection, such as wireless network. However, the modern smartphone operating system and the distributed application architecture are vulnerable to a set of threats.

3.1.3 Threats in Mobile Environment

The support of third-party applications provides more functions, such as entertainment and mobile business, on the smartphone. In the meantime, the malware and virus, which exist normally in the desktop environment, go mobile and increase dramatically [McA12]. The most common malicious activities include collecting user information, sending premium-rate SMS messages, credential theft and SMS spam [FFC+11]. Similarly to solutions on the desktop, the smartphone operating system provides security mechanisms to defend against these threats. For example, Android uses the permission model to warn the user when installing malicious applications.

For mobile phones, which have the mobile network connectivity, such as GPRS and Wi-Fi, different types of attacks exist:

- **Eavesdropping attack**: the attacker monitors traffic on the network for data such as password in plaintext form. The virtual private network or transport layer security could be used to defend against this attack.

- **Man-in-the-middle attack**: makes it appear that two entities are communicating with each other, when actually they are sending and receiving data with an entity between them, or the "man-in-the-middle". The attacker captures the data that is being transmitted, records it, optionally alters it, and then sends it on to the original recipient without its presence being detected [Cia11]. Mutual authentication should be applied to distinguish malicious parties from genuine ones.

- **Replay attack**: is similar to a passive man-in-the-middle attack. Whereas a passive attack sends the transmitted data immediately, a replay attack makes a copy of the transmission before sending it to the recipient [Cia11]. For example, an attacker captures a login credential from the victim, and then uses it repeatedly to get logged again. One-time token can be included in the transmitted data to prevent from this attack.

- **Phishing attack**: is the process of enticing victims into visiting fraudulent websites and persuading them to give identity information such as passwords and any further sensitive information that can be made to seem plausible [MC07]. Phishing attack can be carried out by Emails which include links to malicious websites under the attacker's control. For distributed authentication systems such as OpenID, phishing can happen when the victim visits a malicious website [BGH+12].

3.1.4 Countermeasures

Countermeasures against threats in the mobile world depend on platforms, connectivities, types of applications, and so forth. For example, the anti-malware software should be used to defend against malware attacks. For the network-based software and application, security needs to be maintained by all participants. Service providers must prevent attacks on their servers and databases that contain information of customers. The user must detect rogue websites before giving identity information such as passwords. To avoid eavesdropping, the transport layer security should be used between the server and the client. Before granting access to the client, the service provider must authenticate the claimed user. The security of authentication relies not only on the network infrastructure and smartphone environment, but also on the protocol itself. Because of the importance of authentication in secure applications, the status of the identity management will be carefully analyzed in the following sections.

3.2 Identity Management

3.2.1 Overview

Identity management is the precondition for other security elements, such as resource access control, data and message security, non-repudiation, and service availability [Ben06]. Digital identity is the computer representation of users, programming agents, hosts or network devices. The management of a digital identity includes [BBB10]:

- **Identification**: associates an identity with a subject ("Who are you?")
- **Authentication**: establishes the validity of an identity ("Are you the entity you claim to be?")
- **Authorization**: associates rights or privileges with a subject ("What rights do you have?")

Identification forms a unique identity that distinguishes an individual. Authentication is the process that validates the identity. Authorization verifies the permission associated with the identity.

3.2.2 Identity Management Model

Identity management is classified into three models in [JFH$^+$05]:

- **Isolated identity management**: the service provider has its own identity management domain. The user gets separate identifiers from different service providers.
- **Federated identity management**: identifiers from different service providers are linked and associated. If the user is authenticated by one service provider, she is authenticated by other associated service providers as well.

- **Centralized identity management**: there is an independent identity provider providing identifiers and credentials. The service provider does not manage its own identity management system, instead, it relies on the identity provider.

The first model is simple from the service provider's point of view, but the growing number of identifiers becomes a burden for the end user. The federated model requires the service provider to establish additional trust relationships with other parties. The last model moves the identity management from the service provider to the independent party.

3.2.3 Authentication Factor

The user gets identifiers from different identity management models. Each identifier is associated with a credential that can be verified by the authentication system. There are three methods, which are called authentication factors, to present a credential to the computing system [PP03]:

- Something the user knows: password, personal identification number (PIN)
- Something the user has: secret token (for example, smart card)
- Something the user is: biometric (for example, finger print)

The authentication system uses one or a combination of several authentication factors to authenticate the identity. Services that are under stringent security constraints could use multi-factor authentication, for example, the automatic teller machine (ATM) uses PIN together with the bankcard as a two-factor authentication.

3.3 HTTP Authentication

In the isolated identity management model, the user gets the identifier after registering by the service provider. A password is chosen by the user as the credential. To get authenticated by the service provider, the user must provide the identifier and the password. The HTTP authentication is the most common protocol used in this model. Two schemes exist: basic authentication and digest authentication [FHBH$^+$99].

3.3.1 Basic Authentication

The basic authentication uses the first authentication factor, which is a password in plaintext form, to authenticate the identity [BLFF96]. Upon receipt of an unauthorized request for a URI within the protection space, the server challenges the client. The client provides the identifier and the password in unencrypted form as the response. This authentication scheme is non-secure, because it is based on the assumption that the connection between the server and the client can be regarded as trusted, which is not generally true in an open network.

3.3.2 Digest Authentication

The digest authentication is proposed as an improved authentication scheme for the HTTP protocol in [FHBH+97]. Like the basic authentication, the digest scheme is also based on the challenge-response paradigm. After receiving the unauthorized request, the server challenges the client with a nonce value. A valid response from the client must contain a digest of the identifier, the password, the given nonce value, the HTTP method, and the requested URI. In this way, the password is never sent in plaintext form. The digest of the client's response is computed as follows:

$$\textit{Response} = \textit{KD(H(A1), nonce-value ``:'' (A2))}$$

where

- ***H(data):*** is the string obtained by applying the digest algorithm to the data, for example, *MD5(data)*
- ***KD(secret, data):*** is the string obtained by applying the digest algorithm to the data concatenated with the secret, for example, *MD5(concat(secret, data))*

$$\textit{A1 = user-ID ``:'' realm-value ``:'' password}$$

where

- ***realm:*** is a string to be displayed to users so they know which identifier and password to use

$$\textit{A2 = method ``:'' digest-uri-value}$$

where

- ***method:*** is defined in [FGM+99] as the token indicating the method to be performed on the resource identified in the request URI such as GET, POST and PUT
- ***digest-uri-value:*** is the value of the request URI

In [FHBH+99], the digest authentication is improved by introducing several optional security enhancements such as quality of protection (qop), nonce counter, and client-generated nonce. If additional security options are used, i.e., the "qop" directive's value is "auth-int" (with integrity protection), then the digest of the response is computed as follows:

Response = KD(H(A1), nonce-value ":" nc-value ":" cnonce-value ":" qop-value ":" H(A2))

where

- ***nc-value:*** is the hexadecimal count of the number of responses that the client has sent with this nonce value
- ***cnonce-value:*** is the string value provided by the client and used by both client and server to defeat chosen plaintext attack

A1 = user-ID ":" realm-value ":" password

A2 = method ":" digest-uri-value ":" H(entity-body)

Upon receipt of the response from the client, the server computes the digest with the same procedure and algorithm. If the computed digest is the same as the received one, then the user is authenticated; otherwise, the user will receive the "401 unauthorized" message.

3.3.3 Security Analysis of Digest Authentication

The strength of the digest authentication is that, eavesdroppers cannot directly obtain the password from the user. Therefore, the password is protected from the network sniffing. Other possible attacks against the digest authentication include [FHBH+99]:

Replay attack
The attacker eavesdrops on the conversation between the user and the server to intercept the transmitted data. After the session between the victim and the server is closed, the attacker uses the eavesdropped authentication response repeated to get authenticated by the server as the victim.

The use of the nonce value protects against the replay attack. As stated in [FHBH+97], the nonce contains a digest of the client IP, the time stamp, and the private server key. The attacker has an IP different from the client. It must convince the server that the request is coming from a false IP address. Furthermore the attack can only succeed in the period before the time stamp expires.

Regarding applications that are not tolerated to any replay attacks, the server can use one-time response digest. This causes the overhead of the server remembering which responses have been used. In this way, the replay attack is avoided.

Man-in-the-middle attack
As stated in [TAN05], the digest authentication is vulnerable to the man-in-the-middle attack, even when it happens in the secure tunnel such as TLS [DR08]. The combination of the digest authentication with a secure tunnel is intended

to make the authentication process more secure and reliable. Before the actual authentication, the client establishes a secure tunnel with the server. Authentication messages are transferred in the secure tunnel; therefore they are protected from any eavesdroppers. However, the tunneled digest authentication protocol is vulnerable to the MITM attack, as discussed in [ANN05], because:

- The client uses legacy authentication protocol in other environments.
- The client cannot properly authenticate the server during the TLS handshake.

The MITM attack against the tunneled digest authentication protocol proceeds as follows:

1. The attacker compromises the DNS resolution and waits for the client to starts an authentication session. For the legacy client, an authentication without the secure tunnel is used; otherwise, a secure tunnel with the attacker is established.
2. If the client fails to recognize the attacker during the TLS handshake, it requests an authentication with the identifier.
3. The attacker impersonates the victim to the legitimate server and receives the challenge from it.
4. The attacker forwards the challenge to the victim.
5. If the basic authentication is used, the attacker receives the password from the victim in plaintext form. In case of digest authentication, the client computes the digest of the response and sends it to the attacker.
6. Upon receipt of the response, the attacker closes the session with the victim and then uses the response to get authenticated by the legitimate server as the victim.

The root cause of this vulnerability is that the client cannot verify whether the server, which established the secure tunnel with it, is the one, which initializes the authentication request; while the server is not able to verify whether the client, which established the tunnel with it, is the one, which sends the challenge response. The recommended solution is to enable a mutual authentication between the client and the server as suggested in [ANN05]. A prerequisite to the mutual authentication is the existing of a shared secret between both parties. With the help of the shared secret, a cryptographic binding between the authentication session and the secure tunnel session can be established. An example of using the cryptographic binding in the real world is the authentication and key agreement (AKA) protocol in the UMTS network [3GP12b].

Chosen plaintext attack
If the DNS resolution is compromised, the attacker could choose to uncover the password instead of simply using the response to get authenticated as the victim for only once. In this case, the attacker challenges the client with a self-generated nonce value. Upon receipt of the challenge, the client computes the digest of the response, which includes the received nonce value and the password. This attack is called chosen plaintext attack. If the attacker combines the dictionary attack, which uses the precomputed table such as rainbow table, the cryptanalysis of a password will become much easier [KR95]. The contents of the precomputed table are digests of challenge responses, which consist of passwords from the dictionary and the already known nonce value. After receiving the response from the victim, the attacker looks up the precomputed table to search for the password that yields the same digest. If a match is found, then the password of the victim is uncovered.

The client-generated nonce value, which is introduced in [FHBH$^+$99], is the countermeasure against this attack. Using a random number from the client, the digest of the response varies for each authentication, even the nonce value from the attacker remains the same. In this way, the chosen plaintext attack is defended.

3.4 OpenID

OpenID is an open standard that allows the user to sign in all the OpenID-enabled websites with one identifier. OpenID technologies are promoted, protected and nurtured by the non-profit organization OpenID Foundation (OIDF). OpenID 2.0 is the current version, finalized in December 2007.

3.4.1 Protocol Overview

OpenID is an authentication protocol that authenticates the user in a decentralized manner, i.e., OpenID-enabled service providers delegate the authentication service to OpenID identity providers. The server, which provides OpenID identifiers and the authentication service, is called the OpenID provider (OP). The service provider, which supports OpenID as the authentication mechanism, is called the replying party (RP). The entity, which wants to assert its identity, is called the end user. The following steps describe the OpenID 2.0 protocol ([Ope07]) as depicted in Figure 3.1:

1. The end user visits the RP, which is OpenID-enabled, and then submits the OpenID identifier *user.myopenidexample.com* to request authentication. The OpenID identifier is either a URL or a XRI, which is chosen by the end user from the OP.

2. The RP performs discovery on the user-supplied identifier to establish the OP endpoint URL *www.myopenidexample.com* that the end user uses for

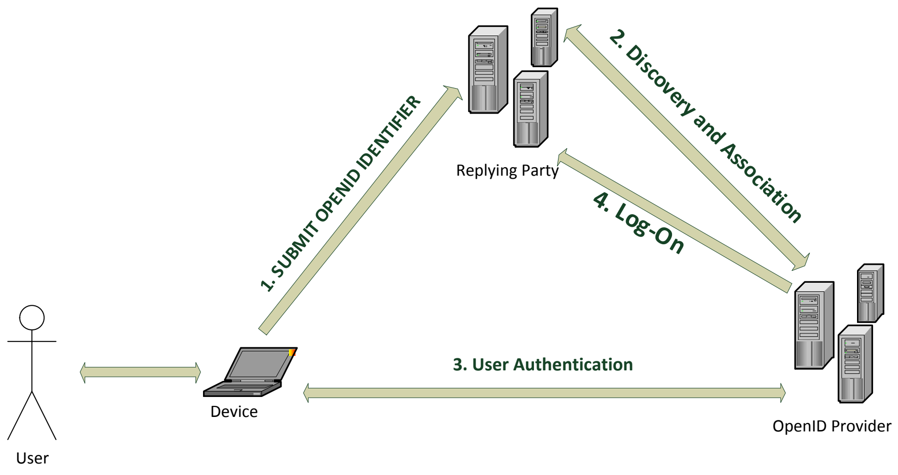


Figure 3.1: OpenID Protocol

authentication.

3. The RP requests an association session with the discovered OP by sending an association request. Upon receipt of the request, the OP generates a shared cryptographic secret and an association handle. The shared secret is a symmetric key for the keyed-hash message authentication code algorithm HMAC [KBC97], with which the OP signs the authentication assertion using the shared secret. Two HMAC algorithms are supported in OpenID: HMAC-SHA-1 and HMAC-SHA-256. The RP should specify the algorithm in the association request. The association handle, which is a string of 255 ASCII characters or less, is used to refer to this association in subsequent messages. Using Diffie-Hellman key exchange ([Res99]), the shared secret is exchanged between the RP and the OP.

4. Once the shared secret has been exchanged, the RP redirects the end user to the OP by sending an authentication request. This request should contain the URL of the OP and the association handle from last step to refer to this authentication in subsequent messages.

5. The end user authenticates by the OP using the credential. At the end of the authentication, the end user should decide whether the RP is to be authorized.

6. After the end user has successfully authenticated and the RP is authorized, the OP redirects the end user to the RP with an authentication assertion. The assertion is identified with the association handle. A unique response nonce, which is a string of 255 characters or less, should be included to avoid replay attacks. A signature of the assertion, which consists of the URL of the OP, the identifier, the response nonce, the association handle,

and possibly other data, is computed with the shared secret using the RP specified HMAC algorithm and included in the authentication assertion.

7. Upon receipt of the authentication assertion, the RP verifies it. The URL of the OP and the end user's identifier must be identical with the discovered values. The response nonce value must be checked and accepted only once to protect from replay attacks. At the end, the RP generates the signature of the authentication assertion with the same procedure and algorithm that the OP followed. If the computed signature matches the one from the OP, the assertion is validated and the end user is authenticated.

3.4.2 Security Analysis

The end user can use a single OpenID identifier to sign in all OpenID-enabled service providers. For this reason, the security issues regarding OpenID must be seriously considered.

Phishing attack

Phishing is attempting to impersonate a trustworthy entity to acquire sensitive information such as password from the victim. The decentralized manner of OpenID makes it highly susceptible to phishing attacks. Phishing is also documented in the OpenID 2.0 specification [Ope07] as the rogue relying party proxying attack.

The general idea of phishing is to redirect the end user from the malicious RP to a phished OP instead of the legitimate one as depicted in Figure 3.2. The common process of a phishing attack is as follows:

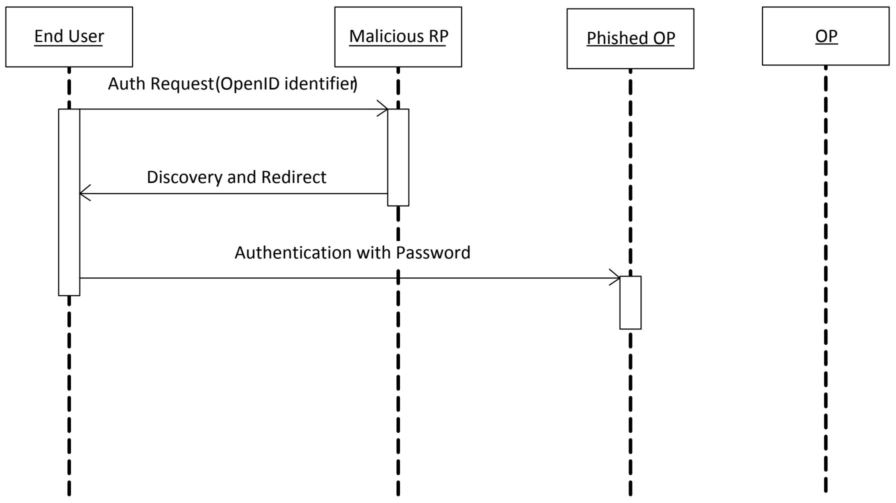


Figure 3.2: Phishing Attack in OpenID

1. The end user visits a malicious RP that attempts to compromise the victim's OpenID credential.

2. The end user submits the OpenID identifier as usual and waits for the redirection to the legitimate OP.

3. The malicious RP performs discovery on the received identifier to discover the URL of the legitimate OP. An impersonation of the OP can be previously prepared by saving the login page and style sheet of the legitimate OP. After discovery, the RP redirects the end user to the impersonated OP.

4. If the end user does not detect that the OP is the impersonated one and enters the password, then the attacker phishes the credential of the victim successfully.

There are a lot of suggestions to prevent phishing attacks, such as educating users, personal icons, and bookmark login[mar]. The central idea of these countermeasures is that the end user should be alert of phishing attacks and be able to identify the impersonation of the OP. However, these are not real solutions. Instead of using password for authentication by the OP, other authentication factors could provide the possibility to avoid phishing attacks.

Man-in-the-middle attack
If the DNS resolution is compromised, then the MITM attacker can impersonate the OP to issue its own association response and the following authentication assertion as depicted in Figure 3.3. In this way, the attacker uses the victim's OpenID identifier and log in the RP without authentication by the legitimate OP.

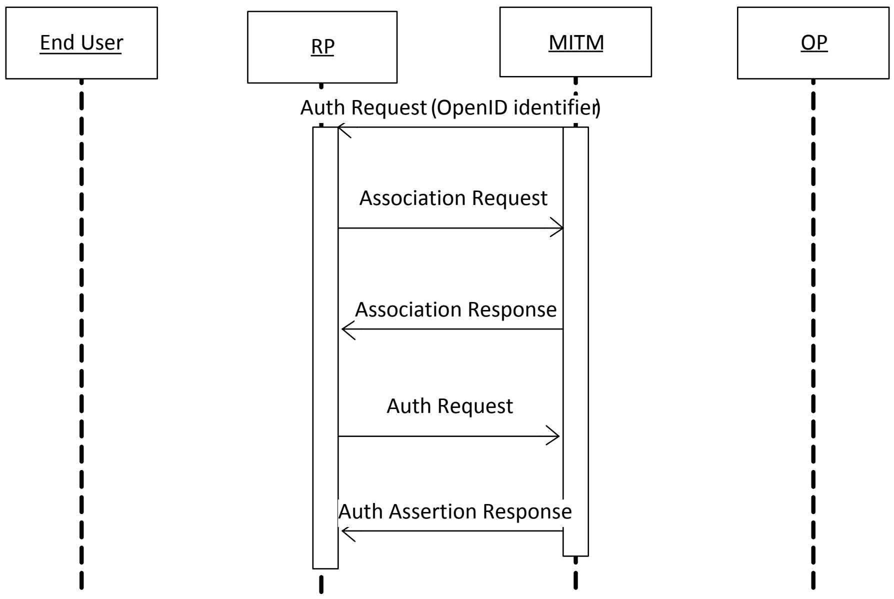


Figure 3.3: MITM Attack in OpenID

The countermeasure against the mentioned MITM attack is to use the transport layer security with certificates signed by a trusted authority [Ope07]. After receiving the result of the DNS lookup, the RP must verify the certificate of the found OP before starting the TLS handshake. Once the validity of the certificate has been established, tampering is not possible. Since authentication messages are transferred in the secure tunnel, they are protected from any eavesdroppers. Therefore, the DH-exchange is no more needed for the exchange of the shared secret.

Session swapping attack
The OpenID protocol does not suggest a mechanism to bind the OpenID session to the end user's browser, letting an attacker force the browser of the victim to complete the authentication as the attacker [BJM08]:

1. The attacker visits the RP and finishes the authentication by the OP as usual.

2. The attacker intercepts the redirection URL from the RP and sends it to the victim.

3. If the victim uses the received redirection, then the RP completes the authentication process and stores a session cookie in the victim's browser.

4. At the end, the victim is logged in the RP as the attacker. If the victim gives sensitive information on the RP, then the attacker is able to uncover them afterwards.

The root cause of this attack is the lack of association between the end user's browser, which initializes authentication, and the RP. To prevent from this attack, the RP must generate a fresh nonce value after receiving the authentication request. The nonce should be stored in the cookie of the end user's browser and included in the association request. The OP should also include this nonce in the authentication assertion. Upon receipt of the authentication assertion, the RP must check that the included nonce is identical with the one stored in the cookie. In this way, the whole authentication process is guaranteed to be completed with a single browser.

Replay attack
The attacker eavesdrops on the conversation between the RP and the OP to intercept the authentication assertion. If the assertion is positive, then the attacker attempts to replay it and get logged in the RP with the same assertion as the victim.

The response nonce value, which is included in the authentication assertion, must be verified by the RP to avoid the replay attack. A positive assertion can only be accepted once, all subsequent attempts should be rejected.

3.5 Summary

The security analysis in 3.3.3 and 3.4.2 shows the vulnerability when using username/password authentication. Some attacks can be avoided by improving the authentication protocol, for example, using the nonce value to prevent from replay attacks. However, several issues remain unresolved, when using password as the sole authentication factor. Digest authentication is vulnerable to the man-in-the-middle attack, since no mutual authentication can be realized. OpenID is vulnerable to the phishing attack, if the end user is not able to identify the malicious OP. Therefore, a multi-factor authentication should be taken into consideration to improve existing authentication protocols.

4 Mobile Security with UICC

The vulnerability of existing authentication protocols presented in the last chapter is a great threat to the network-based applications and services on mobile phones. The lack of ability to securely store sensitive data such as private key is another danger. In the mean time, the UICC, which is contained on every mobile phone, is a trust anchor of the mobile network operator. The UICC is able to store secrets from being tampered with by unauthorized parties, and performs cryptographic operations. For this reason, the UICC is an ideal element to enhance the security level of applications and services on mobile phones. Using SIMAlliance Open Mobile API, a communication channel between the UICC and the mobile phone can be established. The dual application architecture is based on the ability of the mobile application to communicate with the UICC applet. This architecture will be presented and applied for the UICC-based authentication protocols and other security sensitive applications.

4.1 The Role of the UICC

4.1.1 Overview

The UICC is the trusted mobile network operator (MNO) anchor in mobile networks. On mobile phones, the UICC is the security module and the bearer of the subscriber identity. In addition, the UICC is also the platform for the MNO's value-added services. The MNO utilizes the OTA platform for remote management of issued UICCs. During an OTA session, the UICC lies at the end of the secure channel, in which keys and sensitive data are transmitted. The MNO is a natural owner of the root certificate in mobile networks. Therefore, the MNO could become the certificate authority (CA) that issues digital certificates to subscribers.

4.1.2 Secure Element

The logical and physical feature of the UICC makes it a secure element. The object-oriented file structure and the file access control mechanism provide a secure storage environment for sensitive data. Cryptographic operations such as hashing, key derivation and signing can be performed inside the UICC, without sending critical data out of the protected environment. The Java Card UICC can be extended with the customer specific algorithms for new applications.

4.1.3 Secure Channel

The secure channel protocol 80 (SCP80) is used in the OTA platform for secure communication between the UICC and back-end systems. Using the secure channel, credentials and sensitive data can be transmitted in a secure way.

4.1.4 Wireless PKI

The UICC represents the unique identity of the mobile network subscriber. By subscription, a process of recording and verifying personal data of the subscriber is performed. Therefore, it is reasonable for the MNO to become a CA. By providing digital certificates and keys, a wireless public key infrastructure (PKI) can be established [Whi11].

4.2 Dual Application Architecture

4.2.1 Overview

In order to utilize the UICC to improve the security level of applications and services on mobile phones, the dual application architecture can be applied. This architecture is introduced by GSMA for the near field communication (NFC) service in [Ass11].

Figure 4.1 depicts this dual application architecture. The user interface is provided by the mobile application as the consumer facing component, while the application logic resides in the secure element to enhance the security of the whole architecture. The communication channel between both components is provided by the secure element access API such as Open Mobile API.

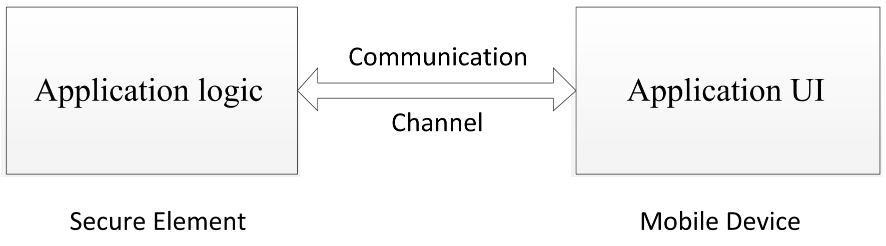


Figure 4.1: Dual Application Architecture

4.2.2 Access Control Mechanism

In the dual application architecture, access to the secure element should be controlled carefully to prevent threats from malicious applications. A signature-based access control mechanism is proposed by GMSA in [Ass11]. An access control module, which is implemented according to this mechanism, should be included in the secure element access API as depicted in Figure 4.2.

The access control module is built up with three sub-modules:

- Access control filtering in the secure element access API
- Access control rules database & engine in the secure element access API
- Access control data in the secure element

The access control filtering listens to the access request, which contains the signature of the application's certificate and the identifier (AID) of the target applet, from the mobile application. Upon receipt of the request from the filtering, the engine looks up the applicable access control rule in the rules database. At the beginning, the database is empty. When no suitable rule is found, the engine checks the access control data on the secure element to decide, whether to build a new rule or reject the access request. If a new rule is built, it will be stored in the database for future use. Therefore, the database can be considered as a cache of applicable rules, which are built from data on the secure element. The format of the access control data depends on the underlying file structure of the secure element, for example, the PKCS#15 structure could be used on the UICC.

4.2.3 Application Scenarios

Different roles of the UICC allow it to be used in a broad range of scenarios. The traditional UICC-based applet can be implemented with the dual application architecture without affecting security, while the graphical user interface of the mobile application provides a more user friendly experience. The UICC-based private key infrastructure can be utilized for enterprise level applications such as the secure Email application, encryption and authentication of documents, and mobile signatures. As a secure element, the UICC can be integrated into the authentication protocol as the second authentication factor. The UICC-based two-factor authentication prevents the credential from being attacked by various network attacks such as man-in-the-middle and phishing. Therefore, the dual application architecture is an ideal framework for the thin client of the software as a service (SaaS).

4.3 UICC-Based Digest Authentication

The digest authentication is introduced and analyzed in 3.3.2. This authentication scheme is vulnerable to the man-in-the-middle attack, even the transport

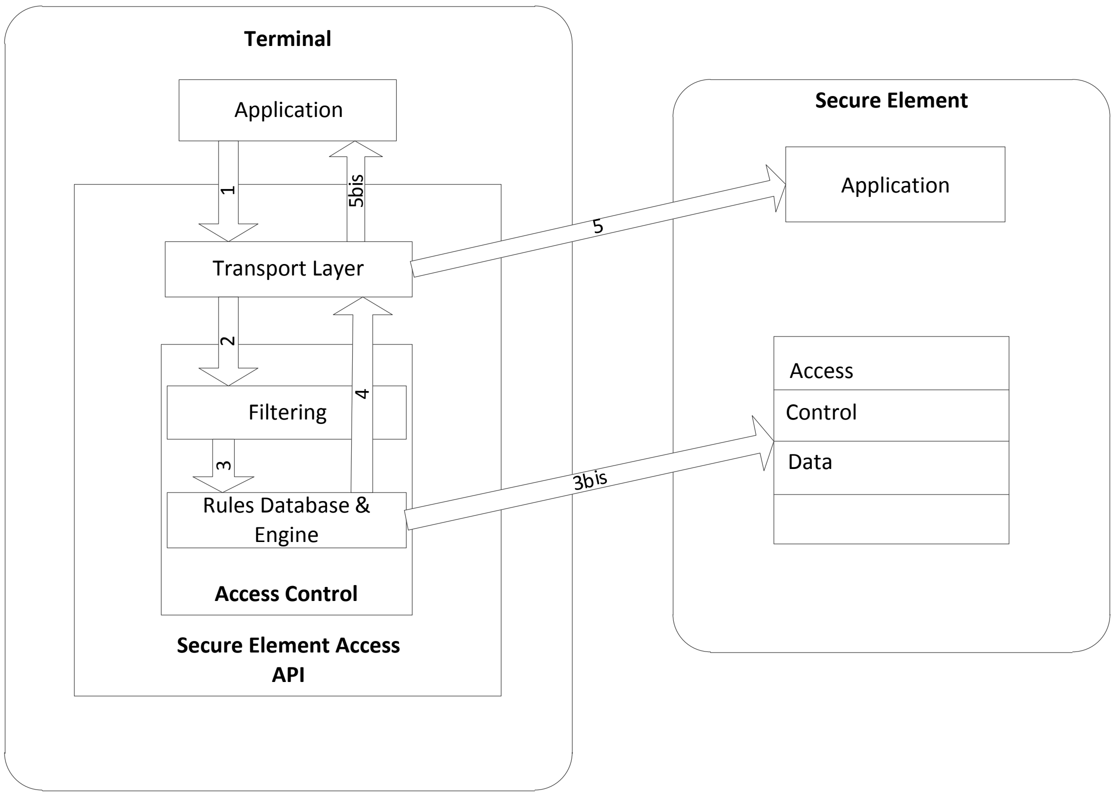


Figure 4.2: Access Control Mechanism [SIM11a]

layer security is used. The root cause is that the server lacks the ability to validate whether the authentication occurring inside a tunnel originated at the same endpoint as the secure tunnel itself [PLPS03]. To enforce the validation, a cryptographic binding of the authentication session to the secure tunnel session should be established. In this way, the client can distinguish the legitimate server from the attacker [ANN05]. This section introduces a UICC-based digest authentication including the cryptographic binding to avoid the mentioned vulnerability.

4.3.1 Protocol Overview

The UICC-based digest authentication is a two-factor authentication protocol including a PIN verification in addition to the digest authentication. The user registers by the service provider, which supports this authentication scheme, and get an applet installed on the UICC with OTA. The applet contains the user supplied identifier and a long-term secret (LTS) as the credential. Cryptographic algorithms, such as digest algorithm and key derivation function, are also implemented in it. In addition, a PIN is specialized for this applet.

The registered user must log in before using services of the service provider. The authentication process involves two participants:

- **Client:** is a consumer facing mobile application such as a browser, and the

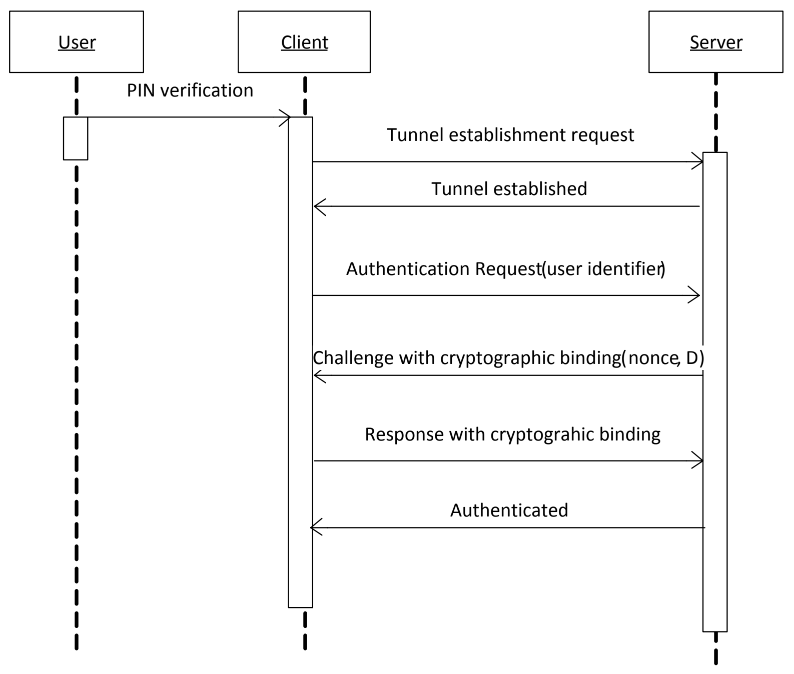

User
Client
Server
PIN verification
Tunnel establishment request
Tunnel established
Authentication Request(user identifier)
Challenge with cryptographic binding(nonce, D)
Response with cryptograhic binding
Authenticated

Figure 4.3: UICC-based Digest Authentication Protocol

UICC applet from the service provider. This combination follows the dual application architecture, and uses Open Mobile API as the interface between both components. The mobile application is in charge of communicating with the service provider as well as the UICC to transmit the data between them.

- **Server:** is a service provider that provides the identifier and performs the authentication on its own. The server shares the credential with the client. Distribution and management of the UICC applet as well as the credential can be delegated to a mobile network operator, which provides the OTA platform.

The following steps describe the UICC-based digest authentication protocol depicted in Figure 4.3:

1. The mobile application part of the client is a browser, with which the user visits a website that supports the UICC-based digest authentication. Before actual authentication, the server informs the browser about the identifier of the needed UICC applet. Upon receipt this information, the browser requires the user to verify PIN. In order to transmit the PIN value from the browser to the UICC applet, a logical channel between them is opened. After successful PIN verification, the UICC applet is activated for the subsequent operations.

2. The browser starts the handshaking procedure to establish a secure tunnel (for example, TLS) with the server.

3. The server responses to the request and finishes the tunnel establishment with the browser. At the end, a shared master secret S is generated for this session.

4. In the established secure tunnel, the browser sends the user-supplied identifier to request authentication by the server.

5. Upon receipt of the authentication request, the server derives a cryptographic key K from the *LTS* belonging to the claimed identifier and the master secret S using the key derivation function PBKDF2 [Kal00]:

   $$K = \mathit{PBKDF2(PRF,\ LTS,\ S,\ c,\ dkLen)}$$

   where

   - ***PRF***: the pseudorandom function such as HMAC-SHA-1
   - ***LTS***: the long-term secret as the master password
   - ***S***: the master secret of the TLS session as the salt
   - ***c***: the iteration count (suggested to be 1000)
   - ***dkLen***: the length of the derived key K

   The key derivation function is an approach to produce the suitable cryptographic key for the subsequent cryptographic operations. HMAC [KBC97] could be used as the pseudorandom function. The HMAC's key is the master password of PBKDF2, while the HMAC's text is the salt of PBKDF2. The iteration of the pseudorandom function increases the cost of the exhaustive search by the attacker.

   To challenge the client, the server generates a fresh nonce value. The digest of the nonce value and the derived key is computed as follows:

   $$D = \mathit{KD(K,\ nonce)}$$

   where

   - ***K***: the derived key
   - ***KD(secret, data)***: the string obtained by applying the digest algorithm to the data concatenated with the secret, for example, *MD5(concat (secret, data))*

   The digest implies a cryptographic binding of the authentication session and the secure tunnel session, since the nonce value and the master secret S vary for each authentication process.

6. The browser transmits the received challenge to the UICC in the logical channel. The UICC applet derives the key *K'* using the same key derivation function as follows:

   $$K' = PBKDF2(PRF, LTS, S, c, dkLen)$$

   The digest of the nonce value and the derived key *K'* is computed by the UICC applet with the same algorithm as follows:

   $$D' = KD(K', nonce)$$

   If the computed digest *D'* is the same as the received *D*, the server is authenticated. That means, the server and the client reside in the same secure tunnel, and the server holds the same *LTS* as the UICC applet. And then the applet computes the challenge response similar to the procedure in 3.3.2:

   $$Response = KD(H(A1), nonce\text{-}value\ \text{“:”}\ (A2))$$

   $$A1 = user\text{-}ID\ \text{“:”}\ realm\text{-}value\ \text{“:”}\ K'$$

   $$A2 = method\ \text{“:”}\ digest\text{-}uri\text{-}value$$

   If additional security options are used, i.e., the “qop” directive’s value is “auth-int” (with integrity protection), then the response is computed:

   $$Response = KD(H(A1), nonce\text{-}value\ \text{“:”}\ nc\text{-}value\ \text{“:”}\ cnonce\text{-}value\ \text{“:”}\\ qop\text{-}value\ \text{“:”}\ H(A2))$$

   $$A1 = user\text{-}ID\ \text{“:”}\ realm\text{-}value\ \text{“:”}\ K'$$

   $$A2 = method\ \text{“:”}\ digest\text{-}uri\text{-}value\ \text{“:”}\ H(entity\text{-}body)$$

   The browser receives the response digest from the UICC, and sends it to the server as the challenge response.

7. Upon receipt of the response, the server computes the digest with the same procedure and algorithm. If the computed digest is the same as the received one, then the client is authenticated; otherwise, the browser will receive the “401 unauthorized” message.

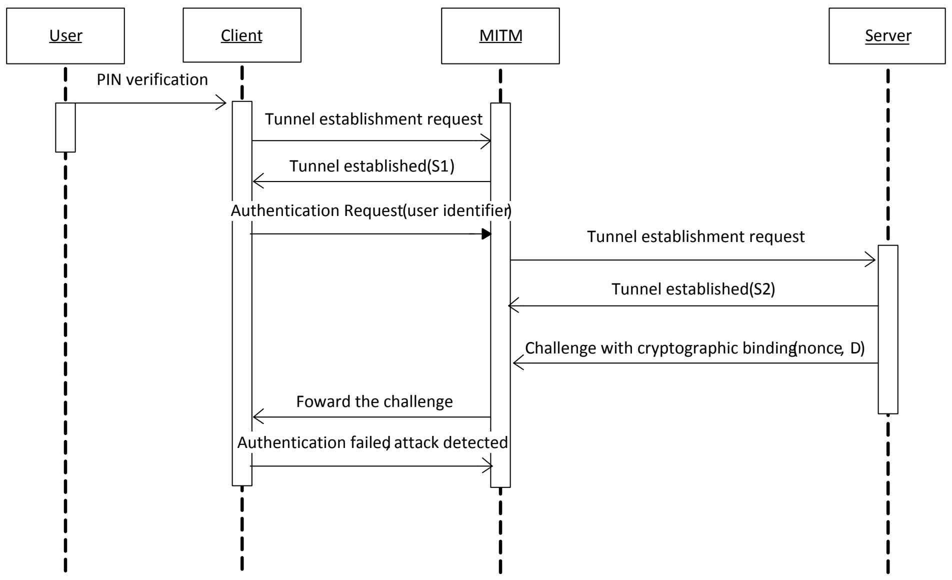

User
Client
MITM
Server
PIN verification
Tunnel establishment request
Tunnel established(S1)
Authentication Request(user identifier)
Tunnel establishment request
Tunnel established(S2)
Challenge with cryptographic binding(nonce, D)
Foward the challenge
Authentication failed, attack detected

Figure 4.4: UICC-based Digest Authentication Under MITM Attack

4.3.2 Security Analysis

The introduction of the UICC into the digest authentication enables a two-factor authentication. The PIN verification shows what the user knows, while the credential in the UICC applet represents what the user owns. The applet PIN has a retry counter. By a positive comparison, the counter is reset to zero. If a negative comparison occurs, the counter is incremented by one. If the retry counter reaches its maximum number, then the PIN is blocked. The user does not need to memorize the long-term secret; therefore it could be set long enough to defend against cryptanalysis. Furthermore, the *LTS* could be updated by the service provider after a predefined validity time.

This UICC-based protocol is based on the digest authentication introduced in 3.3.2; therefore attacks against the digest authentication should also be considered for this protocol.

Man-in-the-middle attack

As described in 3.3.2, the digest authentication is vulnerable to the MITM attack. In the UICC-based protocol, a cryptographic key is derived from the credential of the claimed identifier and the master secret of the secure tunnel session. The digest of the nonce value and the derived key binds the authentication session and the secure tunnel session cryptographically. This binding addresses the vulnerability as stated in [ANN05].

Figure 4.4 depicts an attack scenario in our protocol. Assuming that the MITM attacker masquerades the legitimate server to the victim client. The attacker has a similar URL to the legitimate one. By mistake, the user inputs the URL of the masqueraded server, and fails to recognize the attacker. After PIN verification, the browser establishes the secure tunnel with the attacker. The master secret of the secure tunnel session is denoted as *S1*. Upon receipt of the authentication request from the browser, the attacker establishes a secure tunnel with the legitimate server (master secret *S2*) and forwards the authentication request. The server generates a challenge, which contains a nonce value as well as the digest of the nonce and the derived key. The legitimate server derives the key from the master secret *S2* and the *LTS* belonging to the claimed identifier. Upon receipt of the challenge, the attacker forwards it to the victim. Using the master secret *S1* and the *LTS*, the client computes the digest to authenticate the server. Since *S1* is not equal to *S2*, the authentication fails, and the MITM attack is detected by the client.

Replay attack
This protocol requires the establishment of the secure tunnel prior to the actual authentication; therefore the eavesdropping is avoided. The replay attack is not possible, since the authentication response is transmitted in encrypted form.

Chosen plaintext attack
In order to uncover the credential of the victim, the attacker initializes authentication by the server as the victim. The challenge from the server includes a digest computed from the nonce value, which is together sent in plaintext form, and the derived key, which is computed from the master secret and the victim's credential. To reveal the *LTS*, the attacker must recover the derived key firstly.

The dictionary attack cannot be used, because the derived key is computed with the pseudorandom function. The inputs of the key derivation function are the *LTS* and the master secret of the secure tunnel session. Since the master secret varies for each session, no useful dictionary could be precomputed.

MD5 is used as the default digest algorithm for the digest authentication and the UICC-based protocol. Since no way to analyze the MD5 one-way function used by digest is currently known [FHBH$^+$99], the derived key must be exhaustive searched. The GPU-based brute force cracker for MD5 ([Gol09]), which is considered to be more efficient than the CPU solution, utilizes the parallel computation ability of the GPU and produces about 4 billion (approx. 2^{32}) MD5 operations per second. If HMAC-SHA-1 is used as the pseudorandom function in the key derivation function PBKDF2, then the derived key has a length of 160 bits. The nonce value is known to the attacker. An exhaustive search of the 160-bit key would cost about 2^{128} seconds using the GPU-based cracker. In the mean time, the server could update the *LTS* after a predefined validity time, for example, one year. In this way, the *LTS* can be considered safe.

4.3.3 Usage Scenarios

To use the UICC-based digest authentication, the mobile application must be implemented to be able to communicate with the UICC applet. Therefore, the current browser must be extended with Open Mobile API. In addition, service providers could develop their own client softwares, which are Open Mobile API enabled, and provide the user a more secure way to use their services, for example:

- **Cloud storage**: with the increasing data transfer rate over the telecommunication network and wireless network, the cloud storage becomes an important service that extends the storage capacity of the mobile phone. The cloud storage management application is based on the dual application architecture, and applies the UICC-based digest authentication. The mobile application provides the user interface, file storage management. The UICC applet stores the identifier and the credential of the user. Secure logic, such like key derivation function and signature algorithm, is also implemented in the applet.
- **Online-banking**: a client application of the user's primary bank can provide an optimized layout of the user interface on the mobile phone. The client follows the dual application architecture. The UICC applet, which stores the identifier and the credential of the user, can be considered as a digital bankcard. Using the OTA platform, the bank can distribute and manage the TAN-codes in the UICC applet.

4.4 Smart OpenID

The UICC-based digest authentication utilizes the UICC's security functionalities and provides a two-factor authentication. For each service provider, which uses this protocol as the authentication scheme, an applet containing the credential belonging to the identifier should be installed on the UICC. However, due to the limitation of the memory space on the UICC, only limited number of applets can be installed. To address this restriction, an combination of the UICC and the OpenID protocol is introduced, namely Smart OpenID [LSS12]. Instead of using the identifier/password authentication, a two-factor authentication is realized.

4.4.1 Protocol Overview

The central idea of Smart OpenID is to setup a local OpenID server on the mobile phone, which acts as the delegate of the network OpenID provider. Instead of using the browser to authenticate on the website of the OpenID provider, a mobile application is used. The local OpenID server follows the dual application architecture, which includes the UICC applet to handle the secure logic and the consumer facing mobile application. Figure 4.5 depicts the protocol of Smart OpenID. The participants are:

- **End user**: refers to the entity, which uses the Smart OpenID identifier to authenticate.

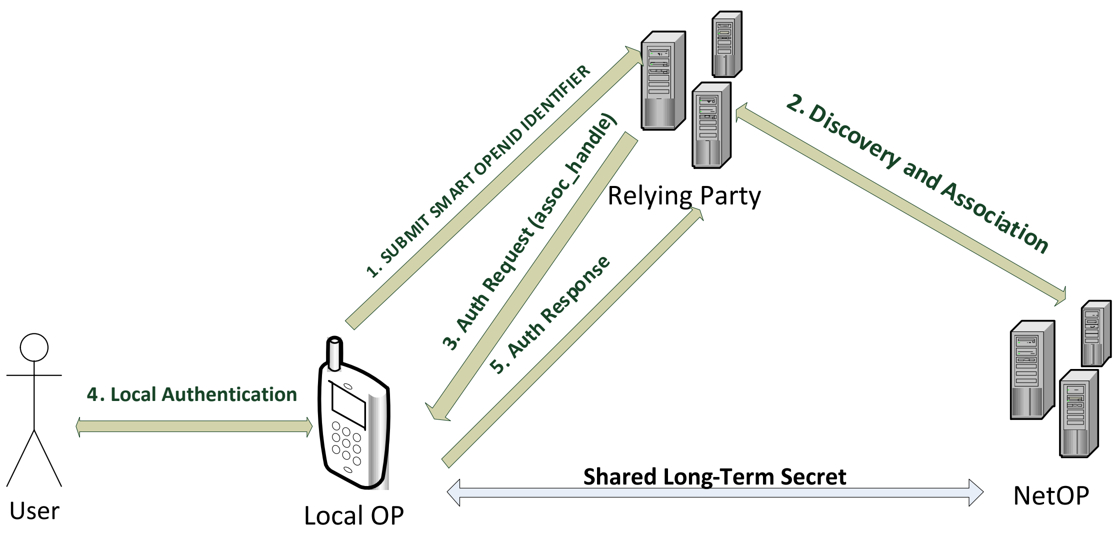


Figure 4.5: Smart OpenID Protocol

- **Relying Party (RP)**: refers to the Smart OpenID acceptor, which provides a means to specify Smart OpenID as the authentication method.
- **Local OpenID Server (LocalOP)**: refers to the delegate of the network OpenID provider (NetOP). It consists of the mobile application and the UICC applet. The mobile application provides the user interface and HTTP interface towards the browser, while the UICC applet stores the Smart OpenID identifier and the credential (long-term secret), which is shared with the NetOP.
- **Network OpenID Provider (NetOP)**: refers to the Smart OpenID provider, which manages the identifier, credential and the authentication.

The following steps describe the authentication process with Smart OpenID as depicted in Figure 4.6:

1. The end user visits the RP, which is Smart OpenID-enabled, and then submits the Smart OpenID identifier *user.mysmartopenid.com* to initialize the authentication. The Smart OpenID identifier has the same form as the OpenID identifier, which is either a URL or a XRI.
2. The RP performs discovery on the user-supplied identifier, and finds the NetOP endpoint URL *www.mysmartopenid.com.* The RP establishes an association session with the discovered NetOP by sending an association request.
3. Upon receipt of the association request, the NetOP generates an association handle *(AH)*, which is a string of 255 ASCII characters or less, to refer to this association in subsequent messages. Using the key derivation function

PBKDF2 [Kal00], a cryptographic key is derived from the *LTS* belonging to the claimed identifier and the *AH*:

$$S = PBKDF2(PRF,\ LTS,\ AH,\ c,\ dkLen)$$

where

- ***PRF***: the pseudorandom function such as HMAC-SHA-1
- ***LTS***: the long-term secret as the master password
- ***AH***: the association handle as the salt
- ***c***: the iteration count (suggested to be 1000)
- ***dkLen***: the length of the derived key S

The key derivation function is an approach to produce the suitable cryptographic key for the subsequent cryptographic operations. HMAC [KBC97] is used as the pseudorandom function. The HMAC's key is the master password of PBKDF2, while the HMAC's text is the salt of PBKDF2. The iteration of the pseudorandom function increases the cost of the exhaustive search by the attacker. The LocalOP will use the same procedure and algorithm to derive the key and sign the authentication assertion using HMAC in step 6. The RP will use the received key to verify the authentication assertion from the LocalOP in step 8. Two HMAC algorithms are supported by Smart OpenID: HMAC-SHA-1 and HMAC-SHA-256. The RP should specify the algorithm in the association request. The derived key *S* and the *AH* should be exchanged between the RP and the NetOP using Diffie-Hellman key exchange ([Res99]).

4. After receiving the association response, the RP redirects the end user to the LocalOP by sending an authentication request. This request contains the *association handle* from the last step to refer to this authentication in subsequent messages.

5. Upon receipt of the redirection from the RP, the mobile application is opened and requires the end user to verify PIN. In order to transmit the PIN value from the mobile application to the UICC, a logical channel between them is opened. After successful PIN verification, the UICC applet is activated for the subsequent operations. The mobile application collects the to be signed fields, which include the claimed identifier, the association handle and so on, and then encodes them into a hexadecimal string *HS*. A fresh response nonce, which is a string of 255 characters or less, should be included to avoid replay attacks. In the opened logical channel, the encoded data are transmitted to the UICC applet in the form of APDUs.

6. The UICC applet checks the claimed identifier included in received data. If the identifier matches the one stored in the applet, a derived key *S'* is computed as follows:

   $$S' = PBKDF2(PRF, LTS, AH, c, dkLen)$$

   And then the UICC applet signs the *HS* with the *S'* using the RP specified HMAC algorithm:

   $$sig = HMAC\text{-}SHA\text{-}1(S', HS) \text{ or } HMAC\text{-}SHA\text{-}256(S', HS)$$

   At the end, the *sig* is transmitted to the mobile application in the form of APDUs.

7. The mobile application constructs the authentication assertion with the *sig* and the other parameters such as the identifier, the response nonce, the URL of the NetOP, the association handle and so on. The response is sent through the HTTP interface of the mobile application to the RP.

8. Upon receipt of the authentication assertion, the RP must compare the included URL of the NetOP and the identifier with the values that the RP discovered. The response nonce must be checked and accepted only once in order to prevent the replay attack. At the end, the RP computes the signature of the authentication assertion with the same procedure and algorithm that the LocalOP followed. If the computed signature matches the received one, the authentication assertion is validated and the end user is authenticated.

4.4.2 Extension for Mutual Authentication

The authentication request from the RP contains an association handle, which is generated by the NetOP, to identify the association and the following authentication session. The association handle is a string of 255 ASCII characters or less. Therefore, the LocalOP is not able to authenticate the request, which might be generated and sent by a malicious RP.

Adding an association secret, which is shared between the NetOP and the LocalOP, can enable a mutual authentication. The LocalOP encrypts the generated association handle with the association secret before sending it to the RP. Upon receipt of the authentication request, the LocalOP decrypts the encrypted association handle, and then uses the plaintext one for the key derivation function PBKDF2.

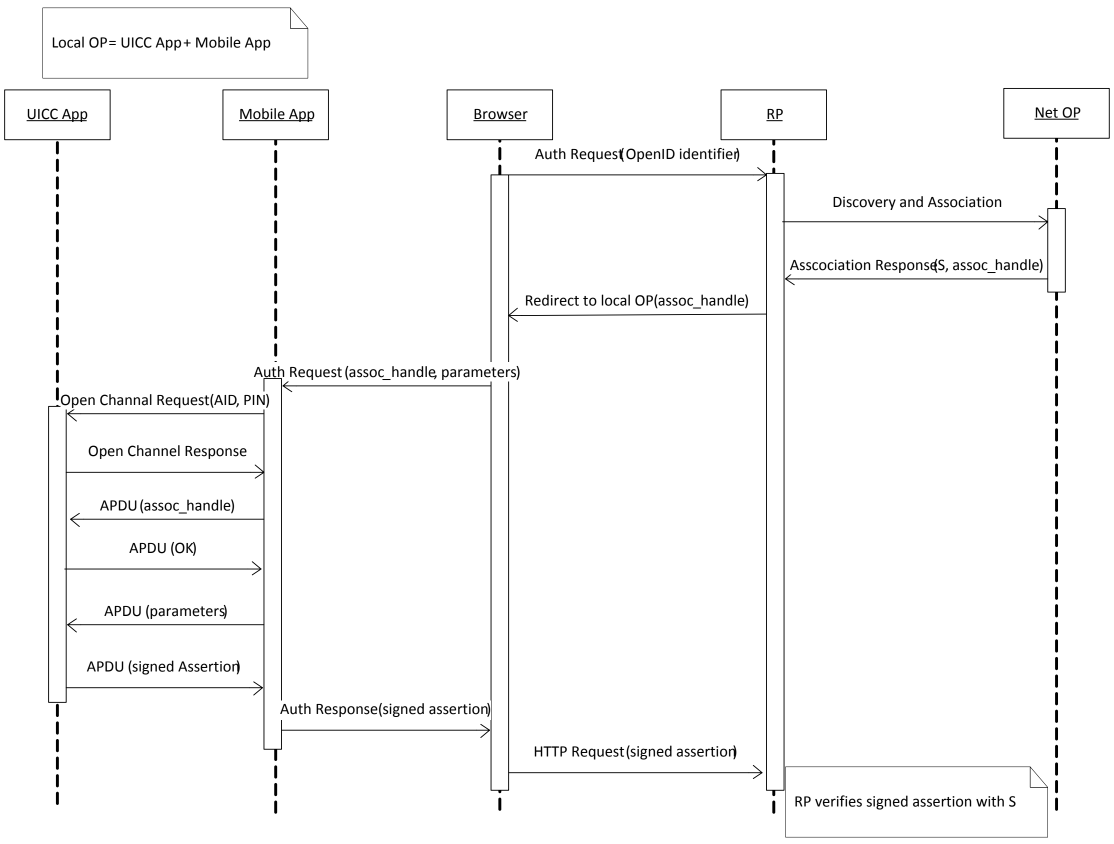

Local OP= UICC App+ Mobile App
UICC App
Mobile App
Browser
RP
Net OP
Auth Request(OpenID identifier)
Discovery and Association
Asscociation Response(S, assoc_handle)
Redirect to local OP(assoc_handle)
Auth Request (assoc_handle, parameters)
Open Channal Request(AID, PIN)
Open Channel Response
APDU (assoc_handle)
APDU (OK)
APDU (parameters)
APDU (signed Assertion)
Auth Response(signed assertion)
HTTP Request (signed assertion)
RP verifies signed assertion with S

Figure 4.6: Smart OpenID Authentication Process

4.4.3 Security Analysis

The combination of the UICC and OpenID enables a two-factor authentication. The PIN verification shows what the user knows, while the credential on the UICC represents what the user owns. The user does not need to memorize the credential; therefore it can be set long enough to defend against cryptanalysis. To provide maximal security of the credential, the service provider could update the long-term secret after a predefined validity time.

In Figure 4.7, an attack tree is built to show possible attacks which intend to compromise authentication in Smart OpenID. The following analysis goes through all identified threats in the attack tree to check the security status of Smart OpenID. The extension in 4.4.2 is not included in this analysis.

Phishing attack

In Smart OpenID, the LocalOP shares the long-term secret with the NetOP. For each authentication, a cryptographic key is derived from the the long-term secret and a random generated association handle by both parties separately. The NetOP sends the derived key to the RP, which uses it to verify the signed authentication assertion from the LocalOP. Therefore, the long-term secret is never transmitted

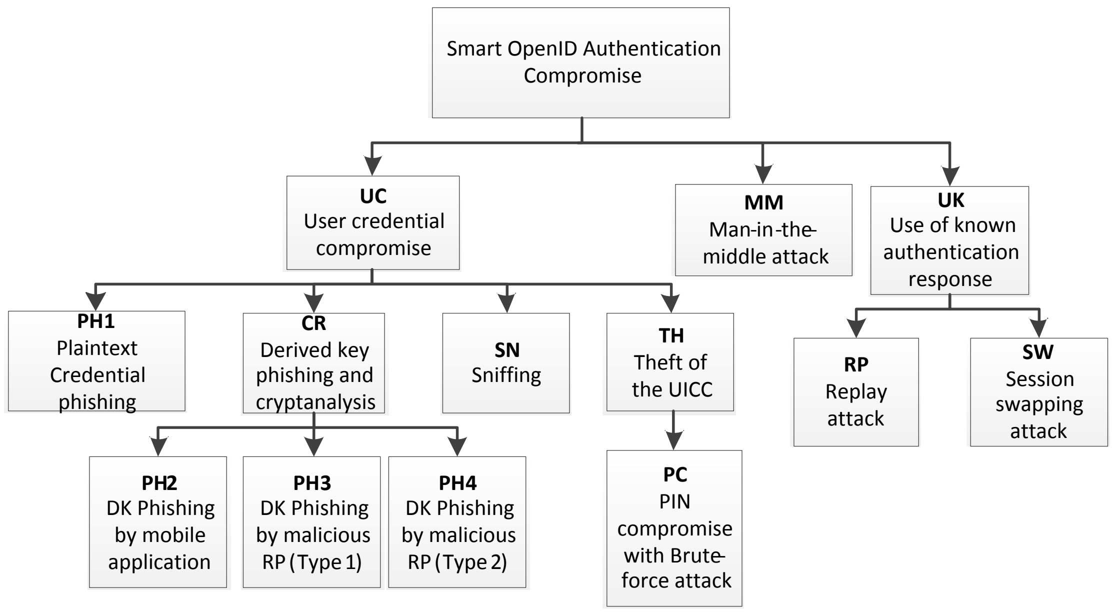

Smart OpenID Authentication Compromise
UC User credential compromise
MM Man-in-the-middle attack
UK Use of known authentication response
PH1 Plaintext Credential phishing
CR Derived key phishing and cryptanalysis
SN Sniffing
TH Theft of the UICC
RP Replay attack
SW Session swapping attack
PH2 DK Phishing by mobile application
PH3 DK Phishing by malicious RP (Type 1)
PH4 DK Phishing by malicious RP (Type 2)
PC PIN compromise with Brute-force attack

Figure 4.7: Attack Tree in Smart OpenID

directly. For this reason, Smart OpenID is immune to the normal phishing attack (*PH1*).

To reveal the long-term secret of the victim, the attacker must firstly recover the derived key, which is used by the HMAC algorithm as the key to sign the authentication assertion, and then do cryptanalysis of the recovered key. The following analysis describes three attack scenarios for this purpose as well as the countermeasures in Smart OpenID.

1. Figure 4.8 depicts a scenario of the attack *PH2*, in which the attacker is a malicious mobile application that sends authentication requests with the self-generated association handle to the UICC applet. In Smart OpenID, the association handle is a string of 255 ASCII characters in plaintext form. Therefore, the UICC applet is not able to verify the originate of the received association handle. If the end user finishes the PIN verification, the UICC applet will handle this request as it is from the legitimate mobile application. At the end, the attacker receives the signature of the authentication assertion. Since the plain text of the signature is known to the attacker; therefore it is also the known-plaintext attack. To protect against this attack, the access control mechanism of Open Mobile API is applied. Only the mobile application, which certificate is signed by the authority, can be granted to access the UICC applet.

2. In the scenario of Figure 4.9, the attack *PH3* is shown. The malicious RP is the attacker that sends authentication requests with the self-generated association handle to the LocalOP. If the end user does not notice that the

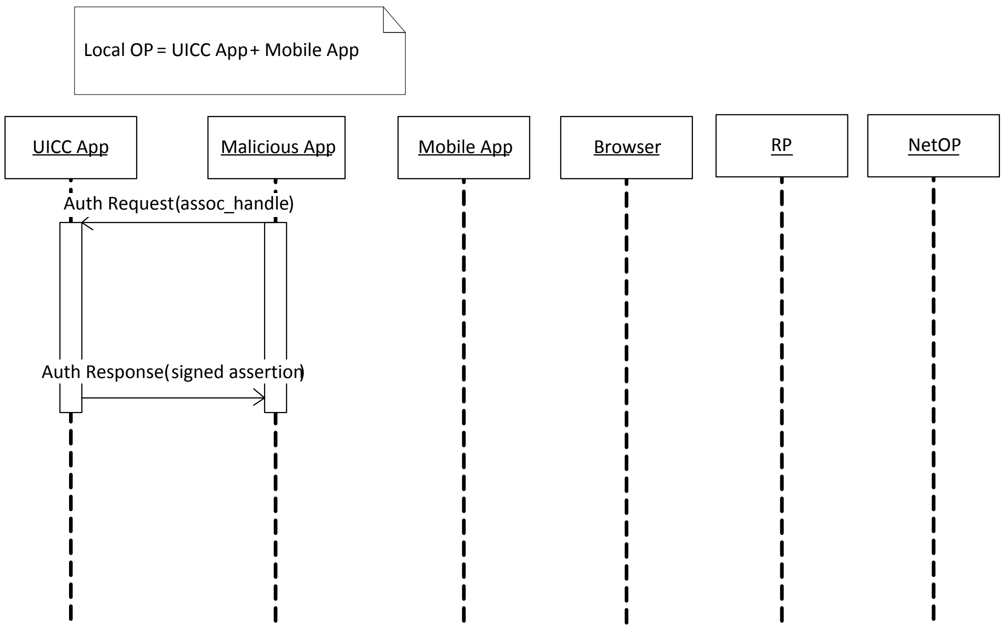

Local OP = UICC App+ Mobile App
UICC App
Malicious App
Mobile App
Browser
RP
NetOP
Auth Request(assoc_handle)
Auth Response(signed assertion)

Figure 4.8: Phishing Attack Scenario 1

RP is malicious and finishes the PIN verification, the LocalOP will handle this request as it is from the legitimate RP. At the end, the attacker receives the signature of the authentication assertion. Unlike the first scenario, there is no access control mechanism between the LocalOP and the RP. To uncover the long-term secret, the attacker must firstly recover the derived key, which is used by the HMAC algorithm to sign the authentication assertion. According to [BCK96], if the underlying hash algorithm is secure, then the HMAC algorithm is also secure. In Smart OpenID, the underlying hash function of the HMAC algorithm is either SHA-1 or SHA-256, which are specified as the secure hash algorithms by NIST (National Institute of Standards and Technology) in [PUB12]. Therefore, the state of the art shows a recovery of the derived key is computationally infeasible.

3. The attack *PH4* is that the malicious RP directly associates with the NetOP as depicted in Figure 4.10. In this case, the attacker starts an association session with the NetOP as the victim. If the NetOP does not detect that the RP is malicious, the association session will be performed normally. At the end, the attacker receives the derived key and the association handle. The key is derived from the long-term secret belonging to the victim's identifier and the known association handle using PBKDF2 algorithm. In Smart OpenID, the pseudorandom function of PBKDF2 is either HMAC-SHA-1 or HMAC-SHA-256. The long-term secret is used as the key to the HMAC, while the association handle is used as the text. As stated in the second scenario, the HMAC algorithm is secure, when the secure hash algorithm is used as the underlying hash function. For this reason, the recovery of the

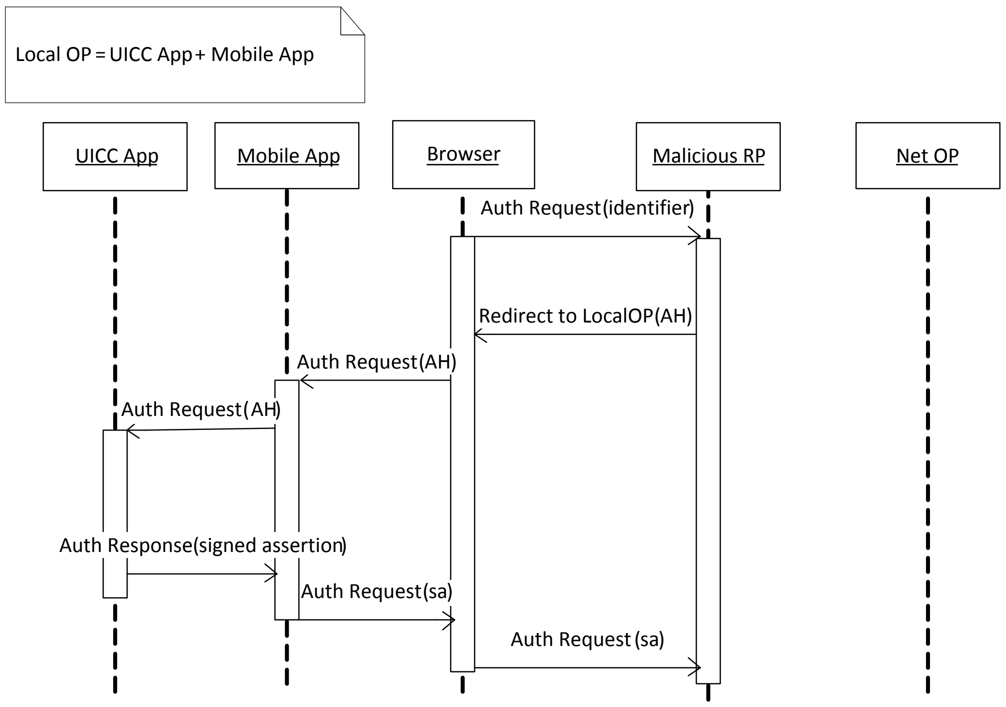


Figure 4.9: Phishing Attack Scenario 2

long-term secret is computationally infeasible. Furthermore, the exhaustive search of the long-term secret is also impractical. The 64 bytes credential has a key space of 2^{512}. For each PBKDF2 operation, the pseudorandom function iterates for a predefined number, which is suggested to be 1000 in [Kal00]. The GPU-based brute force cracker for SHA-1 ([Gol09]), which is more efficient than the CPU-based one, utilizes the parallel computation ability of the GPU and produces about 1 billion SHA-1 operations per second. An exhaustive search using this cracker costs about 2^{493} seconds, which is computational impractical.

Sniffing attack
The sniffing attack *SN* is that the attacker eavesdrops on the communication between the RP and the NetOP. In Smart OpenID, the transport layer security (TLS) is required and used to protect the sensitive data from being tampered.

PIN Compromise
In case of loss or theft, the UICC might be obtained by the attacker. This is the attack *PC* in the attack tree. The applet PIN has a retry counter. By a positive comparison, the counter is reset to zero. If a negative comparison occurs, the counter is incremented by one. If the retry counter reaches its maximum number, then the PIN is blocked. A 4-digit PIN is used in Smart OpenID, while the retry counter is set to 3. Therefore, the chances of having PIN compromised by the brute-force attack are very small.

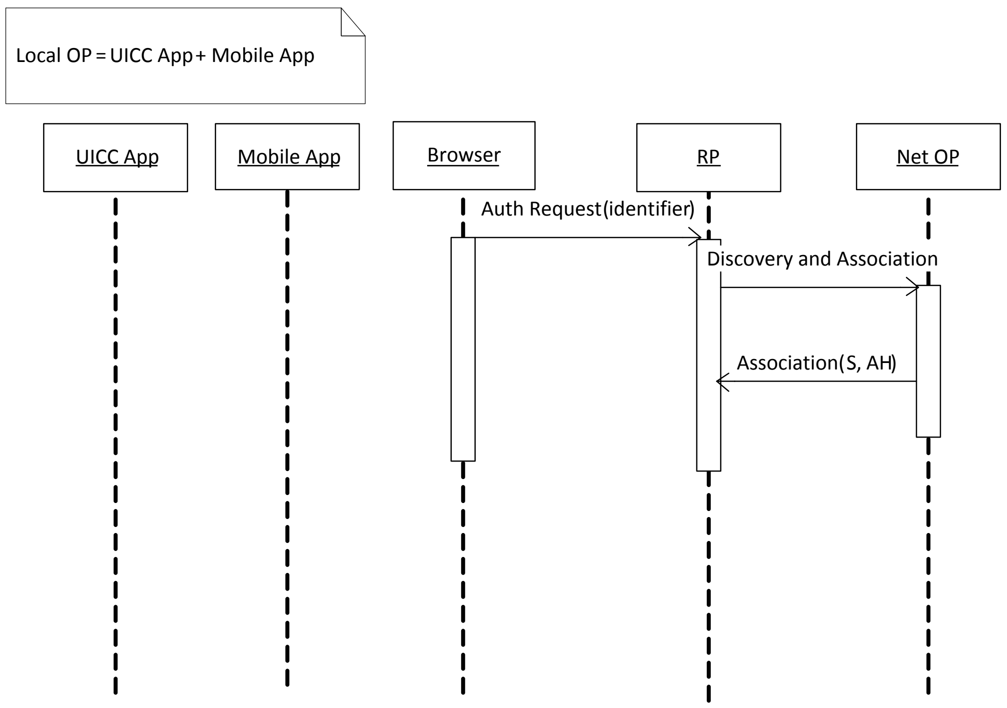


Figure 4.10: Phishing Attack Scenario 3

Man-in-the-middle attack
In the attack *MM*, which is depicted in Figure 4.11, the attacker impersonates the NetOP as well as the LocalOP. Using the identifier of the victim, the attacker spontaneously initializes an authentication by the RP. If the DNS resolution is compromised, the MITM attacker could impersonate the legitimate NetOP to the RP. If the RP fails to recognize the attacker and considers it as the legitimate one, then it sends the association request to the attacker. The attacker responses with a self-generated derived key and an association handle. Since the derived key and the other data for signing the authentication assertion are known to the attacker, it signs the authentication assertion and sends the response to the RP. Using the same derived key, the RP verifies the authentication assertion successfully. At the end, the attacker is authenticated as the victim by the RP.

The cause of this vulnerability is that the RP lacks the ability to authenticate the NetOP. The NetOP's certificate, which is used by the handshake procedure of TLS, should be signed by a trusted authority [Ope07]. After receiving the result of the DNS look-up, the RP must validate the certificate of the discovered NetOP. Once the validity of the certificate has been established, tampering is not possible.

Session swapping attack
Regarding the session swapping attack (attack *SW*), the countermeasure is the same as for OpenID. The solution relies on the implementation of the RP and

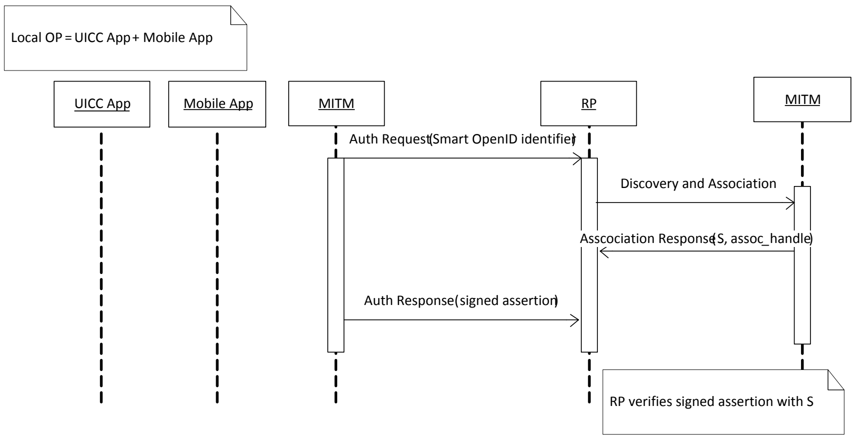

Local OP = UICC App + Mobile App
UICC App
Mobile App
MITM
RP
MITM
Net OP
Auth Request(Smart OpenID identifier)
Discovery and Association
Asscociation Response(S, assoc_handle)
Auth Response(signed assertion)
RP verifies signed assertion with S

Figure 4.11: Smart OpenID Under MITM Attack

leads to a binding of the session to the initialized browser [BJM08]. For each authentication request, the RP must generate a fresh nonce value and store it in the browser's cookie. Upon receipt of the signed assertion from the LocalOP, the RP must validate the nonce included in the authentication assertion and the one stored in the browser's cookie. In this way, the authentication is ensured to be completed with the same browser.

Replay attack
Regarding the replay attack (attack *RP*), the countermeasure is the same as for OpenID. The response nonce included in the authentication assertion must be verified by the RP. A positive assertion can only be accepted once, all the subsequent attempts must be rejected.

4.4.4 Threats Modeling

1. **Assets**
   Smart OpenID identifier:

   - Authenticity (AS01): the identifier and the credential (long-term secret, PIN) must be created by the claimed issuer and assigned to the claimed user
   - Integrity(AS02): the credential for the identifier must be kept unchanged from the unauthorized parties

   User:

   - Identity(AU01): the credential must not be stolen or uncovered by the unauthorized parties

   LocalOP:

- Availability(AL01): the authentication service must be assured for and only for the authorized user

NetOP:

- Availability(AN01): the authentication service must be assured for and only for the authorized user

RP:

- Availability(AR01): the authentication service must be assured for and only for the authenticated user
- Authenticity(AR02): the RP and the user must be who they claim to be

2. **Threats**
   Smart OpenID identifier:
   - TS01: fake identifier and credential from the unauthorized issuer
   - TS02: compromise of the credential by the attacker

   User:
   - TU01: phishing of credential
   - TU02: loss or theft of the credential (UICC)

   LocalOP:
   - TL01: denial of service

   NetOP:
   - TL01: denial of service
   - TN02: man-in-the-middle

   RP:
   - TR01: session swapping
   - TR02: replay
   - TR03: man-in-the-middle
   - TR04: malicious URL
3. **Vulnerabilities**
   V01:
   - Related threat: TS01 (fake identifier and credential from the unauthorized issuer)
   - Affected asset: AS01 (authenticity)
   - Description: the attacker issues the fake identifier to cheat the RP and make it believe that the user with the fake identifier is the claimed one.

- Countermeasure: the identifier and credential must be prearranged on the UICC using either personalization or OTA.

V02:

- Related threat: TS02 (compromise of the credential by the attacker)
- Affected asset: AS02 (integrity)
- Description: the credential in Smart OpenID includes long-term secret and PIN. The attacker tries to compromise the UICC or eavesdrop the communication to uncover the credential.
- Countermeasure: the UICC security mechanism ensures that only the authorized party can access the sensitive data on the UICC. Additionally, the access control mechanism of Open Mobile API ensures that only the authorized mobile application, which certificate is signed by the authority, is granted to access the UICC applet.

V03:

- Related threat: TU01 (phishing of credential)
- Affected asset: AU01 (identity)
- Description: the attacker phishes the user to uncover the credential.
- Countermeasure: in Smart OpenID, the long-term secret is never transimitted separately.

V04:

- Related threat: TU02 (loss or theft of the credential)
- Affected asset: AU01 (identity)
- Description: in case of loss or theft of the mobile phone, the UICC might be occupied by the attacker. Since the credential is stored on the UICC, the attacker tries to uncover the credential and violate the identity of the user.
- Countermeasure: the PIN verification is used to prevent this attack. The length of PIN must be from 4 digits to 6 digits and has a retry counter. In case of losing the UICC, the user can contact the MNO immediately to block the UICC as the trusted anchor for the Smart OpenID identifier.

V05:

- Related threat: TL01 (denial of service)
- Affected asset: AL01 (availability)
- Description: a malicious mobile application sends authentication requests repeatedly to the UICC applet. This threat is harmful, because the UICC applet would lose its availability for the legitimate mobile application.

- Countermeasure: Open Mobile API requires the access control mechanism, which validates the certificate of the mobile application. Only the application, whose certificate is signed by the authority, can be granted to access the UICC applet.

V06:

- Related threat: TN01 (denial of service)
- Affected asset: AN01 (availability)
- Description: different from V06, the availability of the NetOP is threatened by the rogue RP, which sends association or authentication requests repeatedly.
- Countermeasure: the NetOP faces the same threat of denial of service as the OpenID provider. The banning techniques or blacklist could be used.

V07:

- Related threat: TN02, TR03 (man-in-the-middle)
- Affected asset: AU01 (identity), AR02 (authenticity)
- Description: by compromising the network DNS, the MITM attacker can impersonate the NetOP to the RP. The attacker can compromise the authenticity of the RP by violating the identity of the victim.
- Countermeasure: the transport layer security (TLS) with certificate signed by a trusted authority must be used to guarantee the identity of the NetOP.

V08:

- Related threat: TR01(session swapping)
- Affected asset: AR02(authenticity), AU02 (privacy)
- Description: the user is forced to log into the RP with the account controlled by the attacker. If the user does not realize that the account is controlled by the malicious party, then the private information might be given to the RP. This attack affects the authenticity of the RP, since the claimed user changes.
- Countermeasure: the root cause of this attack is the lack of binding between the RP and the browser, which initialized the authentication. The implementation of the RP is responsible to store a nonce value for this session in the cookie of the browser. Upon receipt of the authentication assertion, the RP must check the nonce included in the response and the one stored in the cookie.

V09:

- Related threat: TR02(replay attack)

- Affected asset: AR02(authenticity), AU01(identity)
- Description: the attacker sniffs the positive assertion from the LocalOP and replays it against the same RP to log in as the victim.
- Countermeasure: to protect against replay attacks, the RP is required to check the nonce included in the authentication assertion.

V10:

- Related threat: TR04 (malicious URL)
- Affected asset: AR01 (availability)
- Description: the Smart OpenID identifier is a URL, with which the RP discovers the endpoint URL of the NetOP. An attacker can submit a malicious URL and lead to harmful attacks [TT07], such as exploit of internal scripts, third site scan and denial of service.
- Countermeasure: the RP must assume all the inputs are malicious until proven and perform validation against them. The techniques are: black listing, white listing, data type conversion, regular expression and XML validation [JOG07].

4.5 Summary

UICC-based authentication protocols can address the vulnerability in existing protocols, while the user experience will not be impacted. Using the dual application architecture and the two-factor authentication, network-based applications can be implemented for the financial domain and enterprise usage. The mobile network operator can also extend its business model as the certificate authority and the identity provider.

5 Implementation

To demonstrate the feasibility of the Smart OpenID protocol, a proof of concept system is implemented. The Smart OpenID protocol includes not only relying parties or Smart OpenID providers on the Internet, but also local servers on the mobile phone and the UICC. As the interface between the UICC applet and the mobile application, the use of Open Mobile API is the focus of the implementation. Instead of implementing the full functional RP and the NetOP, a modified RP is developed to test the LocalOP. The association session is omitted by using a set of precomputed authentication requests and expected responses. Upon receipt of the authentication assertion from the LocalOP, the RP compares the response with the expected value to validate the assertion. Primary aims of the demo include:

- Testing the usability of Open Mobile API
- Testing the behavior and the performance of the UICC applet
- Testing the mobile application implemented in Android
- Testing the whole system as the feasibility assessment of the protocol

5.1 SIMply ON

The *SIMply ON* product [eDa] from Morpho e-Documents is chosen as the UICC platform. The applied UICC complies with the Java Card 2.2.2 specification and the latest 3GPP standards. OTA remote file and application management (according to the 3GPP 23.047 standard) is also supported. The full Java Card interoperability of the UICC enables the use of Java Card cryptographic functionalities and customer specific algorithms.

5.2 SEEK-for-Android

The *SEEK-for-Android* (secure element evaluation kit for the Android platform) [fA] is a project which implements the SIMAlliance Open Mobile API on the Android platform. It is an open source project under the Apache license 2.0. The current version *Smartcard API* 2.3.2 has implemented the transport layer of Open Mobile API 2.0 [SIM11a].

The telephony framework, which is the interface between the RIL and the customer application, provides the location for holding the secure element access

API from the SEEK-for-Android project. Hayes AT commands, which are used to transmit data and commands between the RIL and the baseband processor, are not completely included in the RIL. Therefore, a patch is applied to add the required AT commands in the RIL to support the transmission from the mobile application, via telephony framework, to the UICC which is attached to the baseband.

5.3 Utility Class

The SEEK-for-Android project has only implemented the transport layer of Open Mobile API. Therefore, application developers, which make use of functions in secure elements, need to construct all APDUs from scratch before transmitting them with the transport API. For this reason, the application developer must have knowledge about the UICC as well as the Android platform. To ease the development of the mobile application in the dual application structure, a utility class *UICCService* is implemented on top of the transport layer. Figure 5.1 is the UML diagram of the *UICCService* class. The constructor includes *Context* and *Callback* as parameters to initialize the underlying *SEService* object, while *aid* specifies the identifier of the target UICC applet:

```
public UICCService(Context context, CallBack callback, String aid)
```

The *startSession* method encapsulates steps, from opening the session to select the specified applet:

```
session = readers[0].openSession();
logicalChannel = session.openLogicalChannel(aid);
```

The *signature* and *PIN* methods represent the available services in the UICC applet. By calling them, all parameters should be given in the form of an ASCII string. Inside the method, the parameter is encoded into hexadecimal code and concatenated with other fields of the APDU. Using the opened logical channel, the command and response APDUs are transmitted in the form of byte array. An example of the *verifyPIN* method is as follows:

```
byte[] cmd = converter.hexStringToByteArray("A0200000"
             + converter.dezToHexString(pin.length())
             + converter.charStringToASCIIStringInHex(pin));
String result = converter.byteArrayToHexString(logicalChannel.transmit(cmd));
```

For methods that have long parameters, the parameter must first be divided into substrings, which contains maximal 250 characters, and then transmitted in the form of APDUs successively.

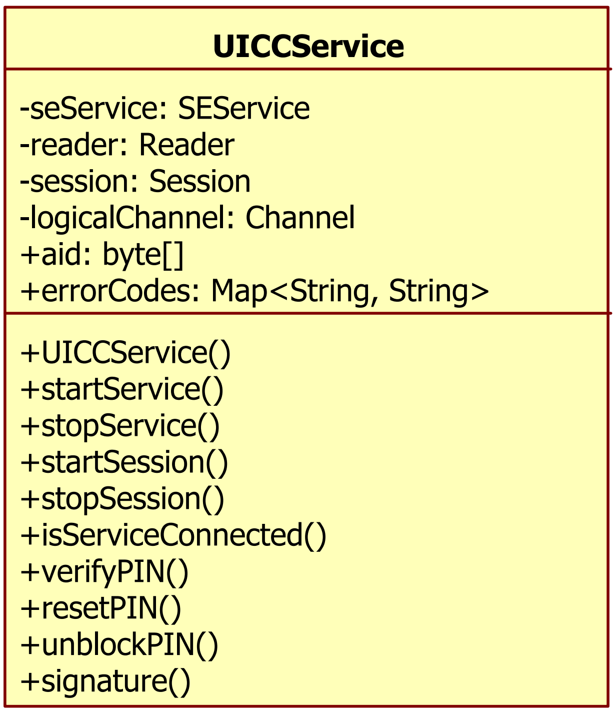


Figure 5.1: UICCService Class

5.4 UICC Applet

In the LocalOP, the UICC applet provides the secure logic and stores credentials. All methods provided in the utility class have a corresponding handler in the applet. Java Card API 2.2.2 provides the AES and the HMAC method, while the PBKDF2 function is self-implemented according to the specification [Kal00].

5.4.1 Classes

Two classes are included in the applet as depicted in Figure 5.2:

- ***SmartOpenID***: is the main class of the applet which extends *javacard.framework.Applet* and implements *install*, *select*, *deselect*, *process* method.
- ***Crypto***: includes the PBKDF2 implementation and methods for other cryptographic operations such as signing the authentication assertion.

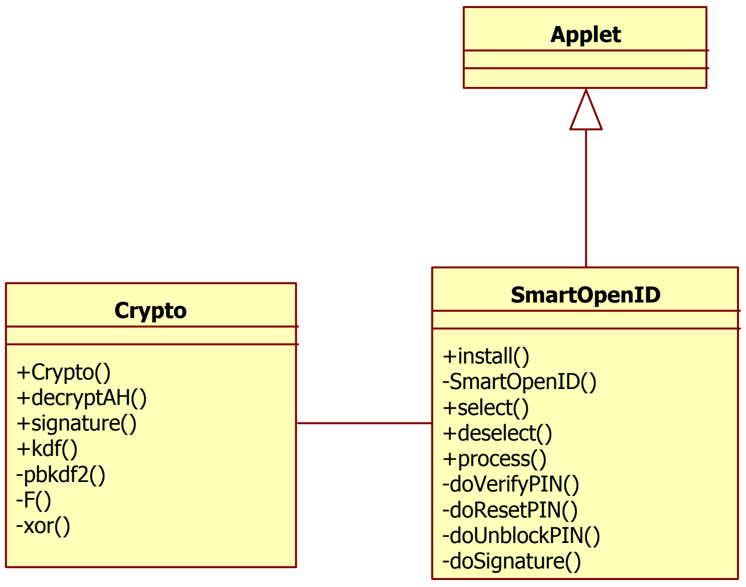


Figure 5.2: Class Diagram of UICC Applet

5.4.2 Commands

A set of supported commands by the applet are defined in the form of APDUs in table 5.1:

Command	CLA	INS	P1	P2	Lc	Data
VERIFYPIN	A0	20	0	0	4	PIN
UNBLOCKPIN	A0	21	0	0	6	PUK
RESETPIN	A0	22	0	0	8	Old PIN and new PIN
SIGNATURE	A0	32	1-8	1-8	1-255	Message in hexadecimal

Table 5.1: Supported Commands in UICC Applet

Upon receipt of the command APDU, the applet uses the instruction filed to invoke the corresponding handler as follows:

```
switch (buffer[ISO7816.OFFSET_INS]) {
   case (byte) INS_VERIFYPIN:
        doVerifyPIN(apdu);

        break;
   case (byte) INS_RESETPIN:
        doResetPIN(apdu);

        break;
   case (byte) INS_UNBLOCKPIN:
        doUnblockPIN(apdu);
```

```
            break;
        case (byte) INS_SIGNATURE:
            doSignature(apdu);

            break;
        default:
            ISOException.throwIt(ISO7816.SW_FUNC_NOT_SUPPORTED);
    }
```

5.4.3 Error Codes

Error codes are used to identify the location and the cause of an exception in the applet. The try-catch pattern in Java is kept in Java Card. When error happens, the error code, which is assigned in the catch block, will be thrown to inform the exception. Table 5.2 shows the defined error codes in the applet.

Description	Error code
SW_PIN_VERIFICATION_REQUIRED	0x6300
SW_PIN_VERIFICATION_FAILED	0x63C0
SW_PIN_BLOCKED	0x6983
SW_PIN_LENGTH_WRONG	0x6985
SW_UPIN_VERIFICATION_FAILED	0x63C1
SW_UPIN_BLOCKED	0x6984
SW_UPIN_LENGTH_WRONG	0x6986
SW_ASSOCIATION_HANDLE_LENGTH_WRONG	0x6303
SW_DATA_LENGTH_WRONG	0x6304
SW_KDF_ERROR	0x6305
SW_SIGNATURE_ERROR	0x6306
SW_DECRYPTION_ERROR	0x6307
SW_PARAMETER_ERROR	0x6308

Table 5.2: Error Codes

5.4.4 PIN Methods

The PIN verification is used to authenticate the user locally. Only after the user has successfully verified PIN, the *signature* method in the applet can be invoked to sign the authentication assertion. The *javacard.framework.OwnerPIN* class is used for this purpose. The *OwnerPIN* offers services to verify and update PIN. In case of unsuccessful attempts, the retry counter will be increased by one. During initialization, the size of PIN and the maximal value of the retry counter are given:

```
pin = new OwnerPIN(PIN_TRY_LIMIT, MAX_PIN_SIZE);
pin.update(pinValue, (short) 0, PIN_SIZE);
```

```
upin = new OwnerPIN(UPIN_TRY_LIMIT, MAX_UPIN_SIZE);
upin.update(upinValue, (short) 0, UPIN_SIZE);
```

Before verifying, the retry counter and the length of the supplied PIN value must be checked:

```
if(pin.getTriesRemaining() == (byte)0) {
   ISOException.throwIt(SW_PIN_BLOCKED);
}

if((byte)buffer[ISO7816.OFFSET_LC] != PIN_SIZE) {
   ISOException.throwIt(SW_PIN_LENGTH_WRONG);
}

if (!pin.check(buffer, ISO7816.OFFSET_CDATA, PIN_SIZE)) {
   ISOException.throwIt(SW_PIN_VERIFICATION_FAILED);
}
```

If an exception happens, then the user will be informed about the problem. For example, if the user types in the wrong PIN three times, then a "PIN blocked" warning will be given.

5.4.5 Decryption

The extension of Smart OpenID protocol in 4.4.2 is included in the implementation to enable the mutual authentication. The association handle must be encrypted by the NetOP before being sent to the RP. Upon receipt of the authentication request from the RP, the LocalOP decrypts the association handle to authenticate the NetOP. 128-bit AES is used as the algorithm. The key is shared between the NetOP and the LocalOP as the association secret.

The applet uses the factory method to request an AES service object implemented by the JCRE provider:

```
cipher = Cipher.getInstance(Cipher.ALG_AES_BLOCK_128_ECB_NOPAD, true);
```

The instance returned from the factory method must first be initialized with the shared key before doing the actual decryption operation.

5.4.6 PBKDF2

In Java Card API 2.2.2, no key derivation functions are provided. Therefore, the PBKDF2-HMAC-SHA-1 algorithm is implemented as a method according to the specification [Kal00]:

```
private byte[] PBKDF2(byte[] P, byte[] S, short c, short dkLen)
```

where

- $\boldsymbol{P}$: is the master password, a hexadecimal string

- ***S***: is the salt, a hexadecimal string
- ***c***: is the iteration count
- ***dkLen***: is the length of the derived key

The applied pseudorandom function is HMAC-SHA-1 which is supported by the Java Card cryptographic APIs. The key for the HMAC is the master password P, while the text for the HMAC is the salt S. In Smart OpenID, P is the shared long-term secret and S is the decrypted association handle.

The iteration of the pseudorandom function increases the cost of the exhaustive search by the attacker. By derivation a key from the user password, a value of 1000 is suggested as the iteration count [Kal00]. However, one thousand times HMAC-SHA-1 operations cost more than one minute on the current UICC. The long-term secret has a length of 64 bytes, which has a obviously larger key space than the normal user password; therefore a smaller iteration count such as 64 is used. Using the key derivation function, the length of the derived key can be easily changed to fit the change of the authentication protocol.

5.4.7 Assertion Signing

In Smart OpenID, HMAC-SHA-1 and HMAC-SHA-256 are applied as the algorithm for signing the authentication assertion. The mobile application receives the authentication request from the RP and composes the authentication assertion which contains the decrypted association handle, the identifier, the URL of the NetOP and so on. This authentication assertion should be signed by the UICC applet with the derived key using PBKDF2 algorithm. The mobile application passes the whole assertion as the parameter to the *signature* method in the utility class. In APDUs, the data field can hold maximal 255 bytes. Therefore, the assertion must be divided into parts of 255 bytes or less. The sum of all APDUs is included in the parameter byte P1, while the current number is included in P2. All APDUs are sent using the utility class via the transport API to the selected applet one after another.

The applet has a buffer to store the data from the received APDUs. The size of the buffer determines the maximal length of the authentication assertion. In the demo, the buffer has a capacity of 2000 bytes. Upon receipt of a new APDU, the parameter byte P1 and P2 must be compared to decide the next action:

- If P2 is smaller than P1, then the data is added to the buffer. If the buffer is full, then an exception with the error code *SW_DATA_LENGTH_WRONG* will be thrown.
- If P2 is equal to P1, and the data can be added to the buffer without overflow, then the *signature* method in the applet is invoked.

- If P2 is greater than P1, then an exception with the error code *SW_ PARAMETER_ ERROR* will be thrown.

The *signature* method in the applet takes the whole assertion in the buffer as parameter:

```
public byte[] signature(byte[] message, short paramOffset,  short paramLength)
```

The key for HMAC-SHA-1 is initialized with the derived key from the PBKDF2-HMAC-SHA-1:

```
hmacKey = new HMACKeyImpl(KeyBuilder.TYPE_HMAC, KeyBuilder.LENGTH_HMAC_SHA_1_BLOCK_64);
hmacSignature = new HmacSignature(Signature.ALG_HMAC_SHA1, false);
hmacKey.setKey(derivedKey, (byte) 0, DERIVATION_SECRET_LENGTH);
```

At the end, the authentication assertion is signed as follows:

```
hmacSignature.init(hmacKey, Signature.MODE_SIGN);
hmacSignature.sign(message, (short) paramOffset,
                  + (short) paramLength, signature, (short) 0);
```

5.4.8 Transient Objects

The variables *association handle*, *derived secret* and *signature* are defined as transient objects:

```
private byte[] signature= null;
private byte[] associationHandle = null;
private byte[] derivedKey = null;

this.signature = JCSystem.makeTransientByteArray(HLEN_HMAC_SHA1,
                  + JCSystem.CLEAR_ON_DESELECT);
this.associationHandle = JCSystem.makeTransientByteArray(ASSOCIATION_HANDLE_LENGTH,
                  + JCSystem.CLEAR_ON_DESELECT);
this.derivedKey = JCSystem.makeTransientByteArray(DERIVATION_SECRET_LENGTH,
                  + JCSystem.CLEAR_ON_DESELECT);
```

After the UICC applet has been deselected, contents in transient objects will be cleared. This is a security precaution so that the other applet cannot discover the sensitive data.

The variables *U_r*, *U_i*, *T* and *DK*, which are used in the PBKDF2 method, are also declared as transient objects, since RAM has a much faster write cycle time then EEPROM. In this way, the performance of the PBKDF2 method with a large number of iteration can be improved.

5.5 Android Application

The Android application utilizes the Intent mechanism to handle all Smart OpenID authentication requests from the RP. It is implemented with the Android SDK 2.3.5.

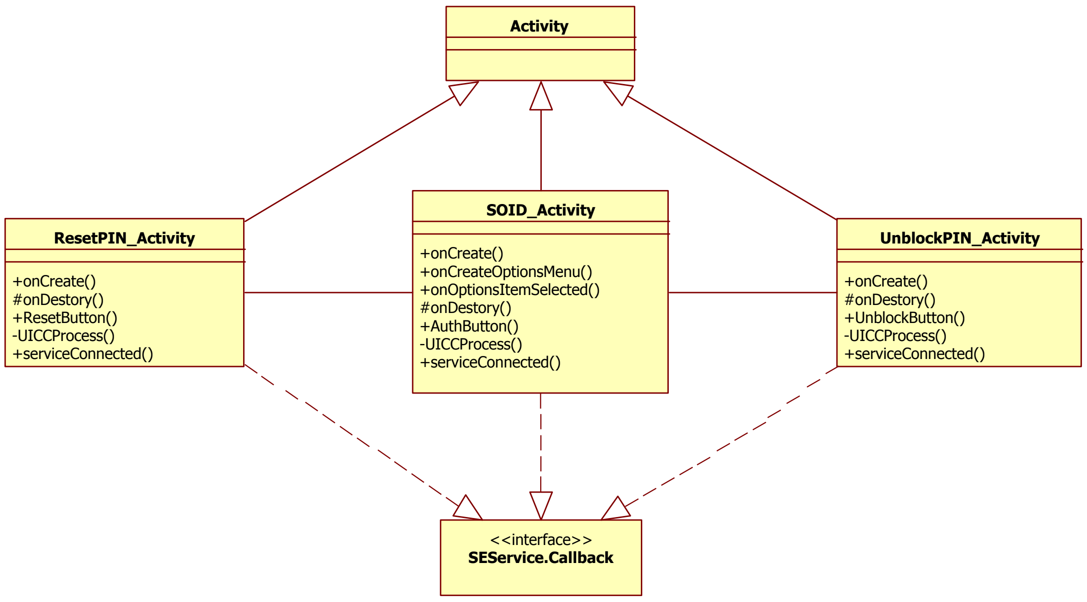


Figure 5.3: Class Diagram of Android Application

5.5.1 Classes

The application consists of three classes: *SOID_Activity*, *ResetPIN_Activity* and *UnblockPIN_Activity*. All classes extend the *android.app.Activity* class and contain their own graphical user interface. The *SOID_Activity* is the main activity which will be activated when the application is launched by the operating system. The *ResetPIN_Activity* and the *UnblockPIN_Activity* can be considered as menu components of the main activity. The class diagram of the application is shown in Figure 5.3.

5.5.2 Android Manifest

The *AndroidManifest.xml* describes the construction of the application. In the *SOID_Activity*, an Intent filter is defined to inform the Android operating system its ability of processing the Smart OpenID authentication request. The permission of using Open Mobile API is also included in it.

```
<application android:icon="@drawable/icon" android:label="@string/app_name">
   <activity android:name=".SOID_Activity" android:label="@string/app_name">
      <intent-filter>
         <data android:scheme="soid.scheme" />
         <action android:name="android.intent.action.VIEW" />
         <category android:name="android.intent.category.BROWSABLE" />
      </intent-filter>
   </activity>
   <activity android:name=".UnblockPIN_Activity" />
   <activity android:name=".ResetPIN_Activity" />
</application>
```

```
<uses-permission android:name="org.simalliance.openmobileapi.SMARTCARD"></uses-permission>
<uses-permission android:name="android.permission.INTERNET"></uses-permission>
```

5.5.3 Using Open Mobile API

In order to access the UICC from the Android application, the utility class in 5.3 is used. After the application has been launched, the *onCreate* method in the main activity is called by the operating system. Inside this method, the *UICCService* object is initialized as:

```
uiccService = new UICCService(this, this, "0101010101010101");
```

The first parameter *this* represents the activity itself as the context for the *SEService*, while the second parameter represents the class, which implements the *SEService.Callback* interface. The last parameter is the identifier of the UICC applet.

Before transmitting command APDUs, the *startSession* method must be called to open a logical channel with the UICC applet. After the channel has been opened, the activity can invoke *verifyPIN*, *resetPIN*, *unblockPIN* and *signature* method to require services from the UICC applet. Construction of the command APDU is automatically performed inside the utility class. Upon receipt of response APDUs, the status word must be examined to decide the next action, for example:

```
result = uiccService.verifyPIN(password);

if (result.substring(0, 4).equals("9000")) {
   signature = uiccService.Signature(dataToBeSigned, encryptedAH).substring(4, 44);
}
```

After all operations in the UICC have been finished, the *stopSession* method must be called to close the logical channel.

5.5.4 Intent

The Intent mechanism is used to enable interaction between components in Android system. The main activity *SOID_Activity* has an Intent filter *soid.scheme*. Once the RP redirects the web browser with the authentication request `soid.scheme://www.mysmartopenid.com/associationhandle/message`, the operating system detects the *soid.scheme* within the request, and then starts the *SOID_Activity* to handle it. After the activity has been started, it parses the received Intent object to get more information:

```
Uri data = getIntent().getData();
List<String> params = data.getPathSegments();
encryptedAH = params.get(0);
message = params.get(1);
```

In order to transmit the authentication response to the RP, the *SOID_Activity* generates another Intent:

```
Intent myIntent = new Intent(Intent.ACTION_VIEW,
                + Uri.parse("http://www.mysmartopenid.com/check.html?"
                + authenticationresponse));
startActivity(myIntent);
```

The operating system detects the type of data in the Intent object, and then starts the web browser to redirect to the RP with the authentication response.

5.5.5 User Interface

The computation in the UICC applet might last several seconds. To show the user the progress of the computation in the UICC, a separate progress dialog should be displayed. This is called asynchronous task which contains a computation running on a background. In Android, this is realized with the *AsyncTask* class.

An *AsyncTask* class is an inner class of an activity. An asynchronous task contains four steps namely *onPreExecute*, *doInBackground*, *onProgressUpdate* and *onPostExecute*. In the demo, it is implemented as follows:

```
private class AuthTask extends AsyncTask<String, Void, String> {
   protected void onPostExecute(String result) {
      dialog.dismiss();

      if (result.subSequence(0, 4).equals("9000")) {
         Intent myIntent = new Intent(Intent.ACTION_VIEW,
                + Uri.parse("http://www.mysmartopenid.com/check.html?"
                + authenticationresponse));
         startActivity(myIntent);
      } else {
         pinEdit.setText("");

         Toast.makeText(SOID_nx_webActivity.this,
         result.subSequence(4, result.length()),
         Toast.LENGTH_SHORT).show();
      }
   }

   protected String doInBackground(String... params) {
      return UICCProcess(params[0], params[1], params[2]);
   }
}
```

All communications between the UICC happen in the *doInBackground* step. When the UICC applet has finished its computation, the *onPostExecute* step is executed, in which the progress dialog is dismissed and the Intent is created and sent. Figure 5.4 depicts the progress dialog during the UICC operation.

5.6 Relying Party

Since the association session in Smart OpenID is the same as in OpenID, it is omitted in this implementation. Therefore, no NetOP is implemented in the demo. Instead, the RP contains a set of precomputed association responses, which consists

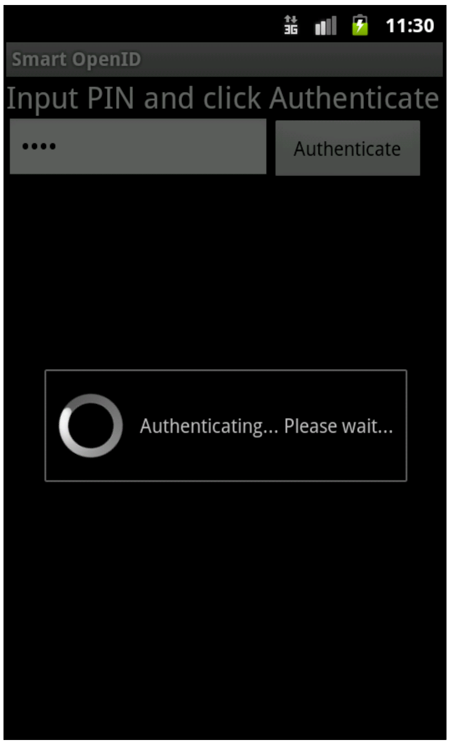

11:30
Smart OpenID
Input PIN and click Authenticate
••••
Authenticate
Authenticating... Please wait...

Figure 5.4: Progress Dialog

of the encrypted association handle and the derived key. The key for encryption is shared between the UICC applet, while the long-term secret for key derivation is also the same as the one in the LocalOP. The Android application is implemented to only transmit the data between the RP and the UICC applet, thus not able to compose the authentication assertion from the authentication request of the RP. For this reason, the RP randomly generates a message for each precomputed association response. This message together with the encrypted association handle build an authentication request. For each authentication request, the RP prepares the expected authentication response, which contains the signature of the message and the corresponding association handle, and stores all test data in an array as test vectors. When the end user visits the RP and requires an authentication, then the RP randomly chooses a test vector from the array and sends it with the prefix *soid.scheme* to the web browser. A part of the RP implementation is as follows:

```
var testArray = new Array();
testArray[0] = ...
...
testArray[10] = ...

function authRequest(){
   var randomnumber = Math.floor(Math.random()*10);
   var testParameter = "soid.scheme://soid.com/" + testVectors[randomnumber];
   window.location =  testParameter;
}
```

Upon receipt of the response from the LocalOP, the RP looks up the expected response according to the association handle. If the received response is identical to the expected one, then the end user is logged into the RP.

6 Testing

Testing of the implemented demo consists of three phases: unit testing, integration testing and system testing. Unit testing is used to validate the functionality of the implemented PBKDF2 method in the UICC applet as well as the performance of the PBKDF2 and the HMAC-SHA-1 method. Integration testing seeks to verify the communication between the Android application and the UICC applet via Open Mobile API. System testing refers to a test of the complete system which consists of the UICC applet, the Android application, and the web-based RP.

6.1 Unit Testing

In unit testing, only the UICC applet is tested. Two PC/SC (personal computer/smartcard) card readers are used to transmit data between the UICC and the software on a test computer. Test cases are composed in the form of APDU script, which contains command APDUs and expected responses. The test environment consists of:

- **UICC**: SIMply ON product from Morpho e-Documents introduced in 5.1.
- **Card reader**: OMNIKEY 3121 USB Desktop Reader and ACR card reader (Morpho e-Documents)
- **Software**: Smart Scripter (Morpho e-Documents) [eDb]

6.1.1 PBKDF2 Testing

In [Jos11], six test vectors are defined to test the PBKDF2 algorithm (using HMAC-SHA-1 as the pseudorandom function), for example:

```
Input:
       P = "password" (8 octets)
       S = "salt" (4 octets)
       c = 1
       dkLen = 20

Output:
       DK = 0c 60 c8 0f 96 1f 0e 71
            f3 a9 b5 24 af 60 12 06
            2f e0 37 a6 (20 octets)
```

The fouth test vector has an iteration count of 16777216 rounds, which takes about 280 hours on the UICC. Therefore, an APDU script is composed of the remaining five test vectors. With the Smart Scripter, the script is transmitted to the UICC applet, which uses the PBKDF2 method to derive the key with the received data. The response from the applet is compared with the expect value, which is included in the script itself. At the end, all test vectors have successfully passed the test.

6.1.2 Performance Testing

The performance of the PBKDF2 and the HMAC-SHA-1 method is measured with the ACR card reader. Test scripts are composed to test the PBKDF2 and the HMAC-SHA-1 method with different parameters. The time units are all in milliseconds.

In Figure 6.1, the performance measurement of the PBKDF2 method is depicted. The pseudorandom function is HMAC-SHA-1, while the master key has a length of 64 bytes and the length of the salt is 240 bytes.

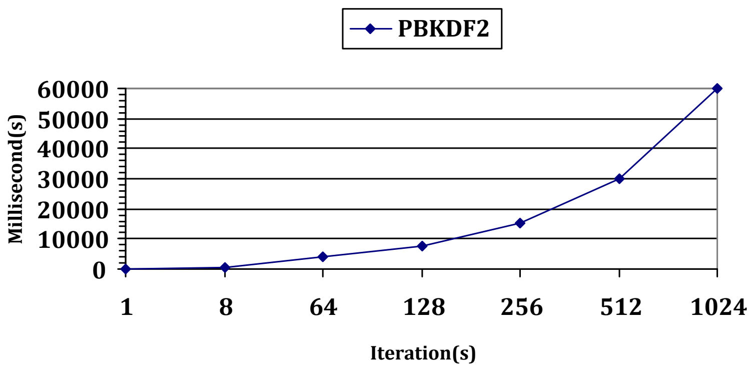


Figure 6.1: First Performance Testing for PBKDF2

When the iteration count is fixed at 64 rounds and the salt has a length of 240 bytes, the time consuming measurement of the PBKDF2 method, which has master keys of different lengths, is depicted in Figure 6.2. The result shows that the length of the master key has only a subtle influence on the performance.

The performance measurement of the HMAC-SHA-1 method is depicted in Figure 6.3. The secret key has a length of 20 bytes, while the length of messages varies from 255 bytes to 2000 bytes. The result contains not only the time consuming measurement of the method, but also the measurement that includes the duration of the APDU transmission.

6.2 Integration Testing

In integration testing, the dual application architecture is tested. In order to test the functionality of Open Mobile API as the secure element access interface, an extra Android application is implemented. The PC/SC card reader is used to

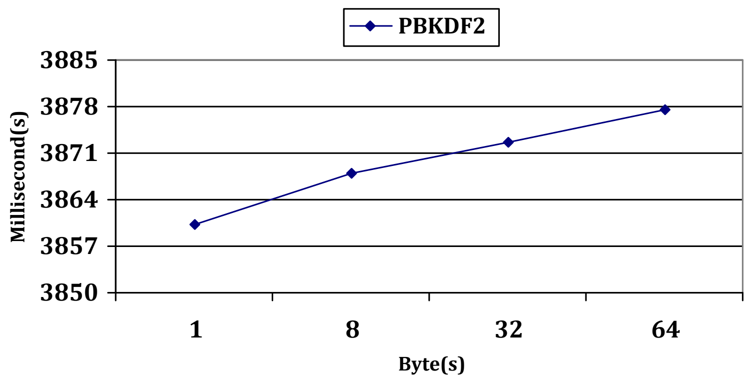


Figure 6.2: Second Performance Testing for PBKDF2

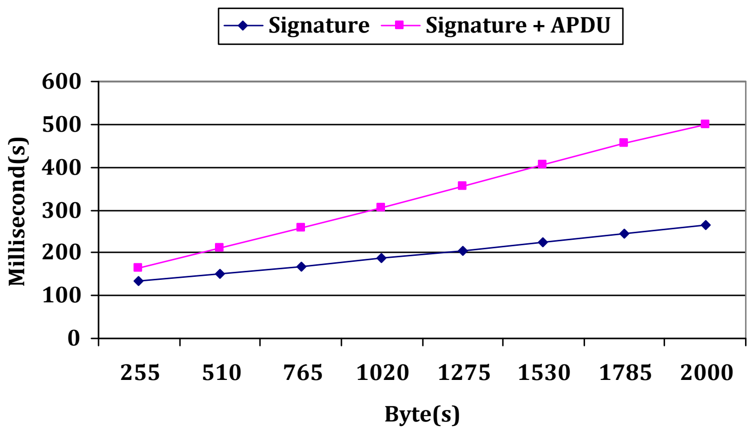


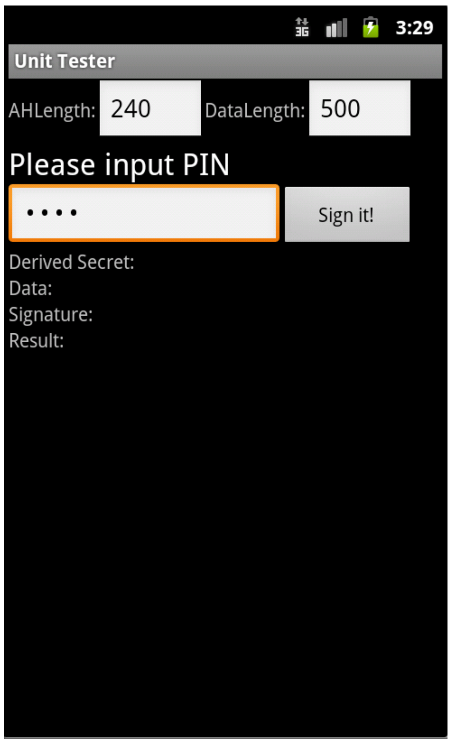
Figure 6.3: Performance Testing for HMAC-SHA-1

transmit APDUs between the UICC applet and the Android application which runs on the emulator. The test environment consists of:

- **UICC**: SIMply ON product from Morpho e-Documents introduced in 5.1.
- **Card reader**: OMNIKEY 3121 USB Desktop Reader
- **Emulator**: Android emulator 2.3.5 with UICC and RIL plugins from SEEK-for-Android project
- **Open Mobile API**: Smartcard API 2.3.2 from SEEK-for-Android project

6.2.1 Test Application

The Android application is able to generate association handles and messages with the given length. An authentication request can be simulated with the randomly generated association handle and the message. Using Open Mobile API, the pseudo authentication request is transmitted to the UICC applet. Furthermore, the test application has implemented the same key derivation function and the signature method. Upon receipt of the response from the applet, the application verifies it by computing the result with the same data and procedures. The UI of the test application is shown in Figure 6.4.



AHLength: 240 DataLength: 500

Please input PIN

••••

Derived Secret:
Data:
Signature:
Result:

Figure 6.4: UI of Test Application

6.2.2 Testing Process

The association secret and the long-term secret are shared between the UICC applet and the test application. Using the association secret and the 128-bit AES algorithm, the association handle is encrypted before being sent to the UICC applet. Parameters of the test application have the following configuration:

- ***Association secret***: 16 bytes hexadecimal number
- ***Long-term secret***: 64 bytes hexadecimal number
- ***Association handle***: 240 bytes ASCII characters (from 33 to 126)
- ***Message***: 1 - 2000 bytes ASCII characters (from 33 to 126)

Two sets of test cases are defined. The first set uses encrypted association handles of different lengths, while all messages have the length of 500 bytes. Only test cases, in which the association handle has 240 bytes, should be accepted by the UICC applet. The second set contains messages of different lengths, while all encrypted association handles have the length of 240 bytes. Only messages, which length is between 1 and 2000 bytes, should be accepted by the UICC applet.

If the UICC applet has accepted the test data, then it decrypts the association handle with the shared association key. Using the decrypted association handle and the long-term secret, the applet derives the key for the signature method. The computed signature is sent to the test application, and then being compared with the value, which is computed by the test application with the same procedure. The result of the comparison is displayed on the UI of the test application. If the test data is denied by the UICC applet, then an exception code should be given and displayed. All test cases are shown in Table 6.1.

TestID	Description	Expected result
AH01	Association handle: 0 byte	Exception: Wrong length
AH02	Association handle: 1 byte	Exception: Wrong length
AH03	Association handle: 239 bytes	Exception: Wrong length
AH04	Association handle: 240 bytes	Success: Signature is generated
AH05	Association handle: 241 bytes	Exception: Wrong length
ME01	Message: 0 byte	Exception: Wrong length
ME02	Message: 1 byte	Success: Signature is generated
ME03	Message: 1999 bytes	Success: Signature is generated
ME04	Message: 2000 bytes	Success: Signature is generated
ME05	Message: 2001 bytes	Exception: Wrong length

Table 6.1: Test Cases

6.2.3 Test Result

All test cases in Table 6.1 have been executed and successfully passed. The result is shown in Table 6.2.

6.3 System Testing

In system testing, the cooperation between the LocalOP, which consists of the UICC applet and the Android application, and the RP is tested. The PC/SC card reader is used to transmit APDUs inside the LocalOP. The RP is hosted in the local Apache server which can be visited by the Android web browser via the local IP address. The test environment consists of:

- **UICC**: SIMply ON product from Morpho e-Documents introduced in 5.1.
- **Card reader**: OMNIKEY 3121 USB Desktop Reader

TestID	Result
AH01	Error code 6303: association handle length is wrong
AH02	Error code 6303: association handle length is wrong
AH03	Error code 6303: association handle length is wrong
AH04	Status word: 9000. Computed signature is identical to the result from the test application.
AH05	Error code 6303: association handle length is wrong
ME01	Error code 6304: Message length is wrong
ME02	Status word: 9000. Computed signature is identical to the result from the test application.
ME03	Status word: 9000. Computed signature is identical to the result from the test application.
ME04	Status word: 9000. Computed signature is identical to the result from the test application.
ME05	Error code 6304: Message length is wrong

Table 6.2: Testing Result

- **Emulator**: Android emulator 2.3.5 with UICC and RIL plugins from SEEK-for-Android project
- **Open Mobile API**: Smartcard API 2.3.2 from SEEK-for-Android project
- **Web server**: Apache Server 2.2

6.3.1 Authentication Testing

In authentication testing, a set of association responses, which consists of the encrypted association handle and the derived key, are precomputed using the same algorithms as the UICC applet. The key for encryption is the same as the one stored in the LocalOP. For each association response, there is a randomly generated message. This message is a simple form of the authentication assertion. Furthermore, the expected signature of the message is computed using the HMAC-SHA-1 method with the corresponding derived key. Therefore, a test vector consists of the encrypted association handle, the derived key, and the expected signature. The RP stores the precomputed test vectors in an array. The testing process is as follows:

1. The end user visits the RP and submits the test identifier `rzhou.mysmartopenid.com`. After the login button has been clicked, the RP chooses a test vector randomly from the array. An authentication request is constructed with the encrypted association handle and the message. The prefix *soid.scheme* tells the Android operating system that this is a Smart OpenID authentication request.

2. The Android operating system detects the *soid.scheme* in the received authentication request, and then sends an Intent object to the *SOID_Activity*. After the Android application has been launched, the end user inputs PIN and clicks the authentication button.

3. The *SOID_Activity* opens a logical channel with the UICC applet and uses the *UICCService* object to require the *verifyPIN* and the signature service.

4. Upon receipt of the computed signature from the UICC applet, the *SOID_Activity* generates an Intent with the URL of the RP, which is appended with the signature and the encrypted association handle. The Android operating system catches the Intent and launches the default web browser with the URL of the RP.

5. According to the encrypted association handle, the RP looks up the expected signature of the message. If the received signature is identical to the expected one, then the end user is logged into the RP.

In Figure 6.5, screenshots of the authentication testing are depicted.

6.3.2 UnblockPIN Testing

In this test, the *unblockPIN* method in the UICC is tested. Using the menu in the Android application, the end user can input the PUK number and call the method. The testing process is as follows:

1. The end user first clicks the menu button, and then clicks the "Unblock PIN" button in the menu list. The *SOID_Activity* generates an Intent object to start the *UnblockPIN_Activity*.

2. The end user submits the PUK number, and then clicks the "Unblock" button.

3. The *UnblockPIN_Activity* opens a logical channel with the UICC applet, and then uses the *UICCService* object to require the *unblockPIN* service.

4. After the UICC applet has finished the operation, the *UnblockPIN_Activity* closes the opened channel and informs the user about the result.

In Figure 6.6, screenshots of the *unblockPIN* testing are depicted.

6.3.3 ResetPIN Testing

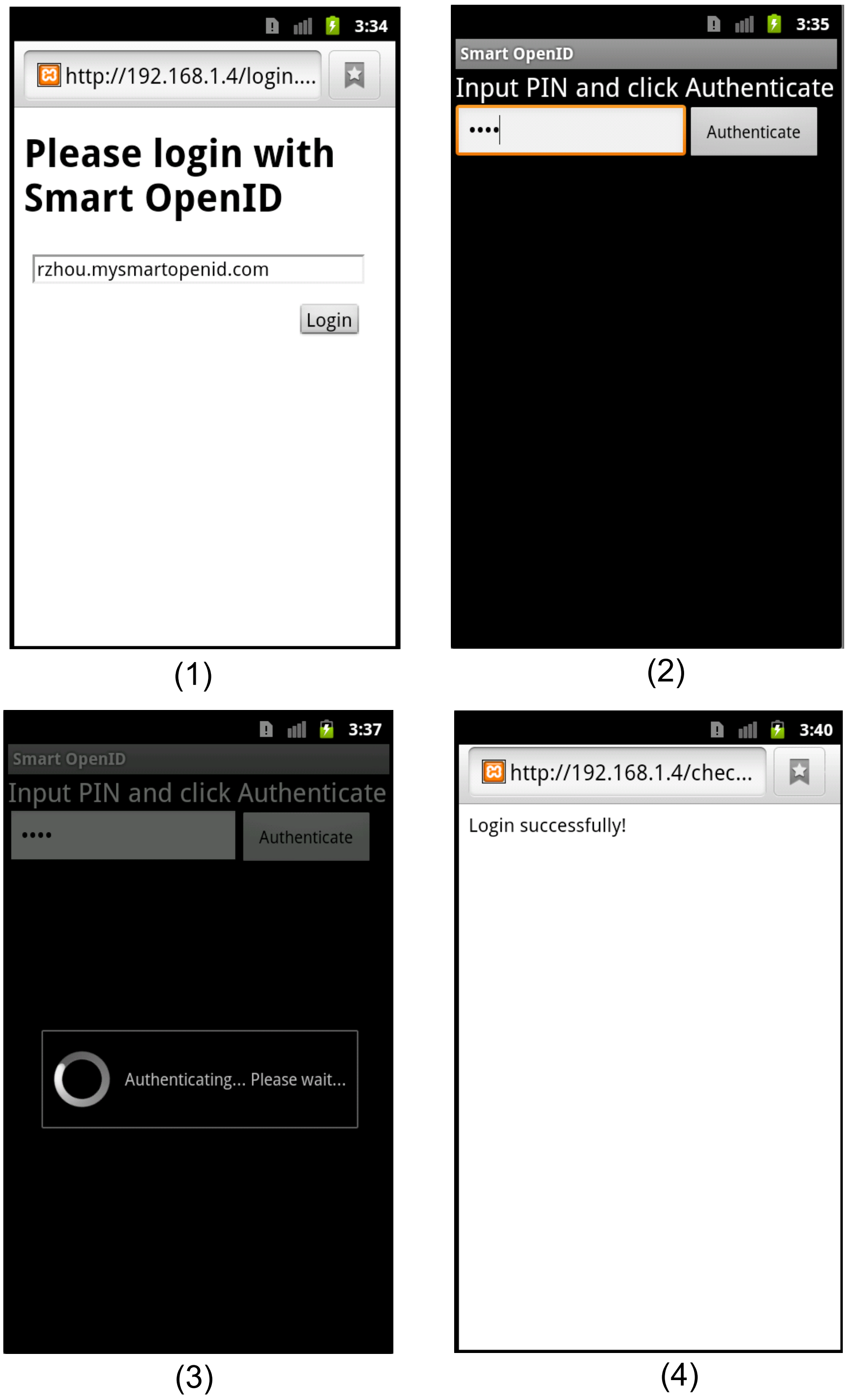
In this test, the *resetPIN* method in the UICC is tested. Using the menu in the Android application, the end user can input the old PIN and the new PIN number to call the method. The testing process is as follows:

1. The end user first clicks the menu button, and then clicks the “Reset PIN” button in the menu list. The *SOID_Activity* generates an Intent object to start the *ResetPIN_Activity*.
2. The end user submits the old PIN and the new PIN number, and then clicks the “Reset” button.
3. The *ResetPIN_Activity* opens a logical channel with the UICC applet, and then uses the *UICCService* object to require the *resetPIN* service.
4. After the UICC applet has finished the operation, the *ResetPIN_Activity* closes the opened channel and informs the user about the result.

In Figure 6.7, screenshots of the *resetPIN* testing are depicted.

6.4 Summary

The test result shows that the implemented PBKDF2 method works properly as required in [Jos11], while the behavior of the dual application architecture is also as expected. The Intent mechanism in system testing has successfully binded the RP to the LocalOP by user authentication. Besides the functionality, the LocalOP has also archived a reasonable performance. Therefore, the Smart OpenID authentication protocol is proven to be feasible.



http://192.168.1.4/login....

Please login with Smart OpenID

rzhou.mysmartopenid.com

Login

Smart OpenID

Input PIN and click Authenticate

••••

Authenticate

Smart OpenID

Input PIN and click Authenticate

••••

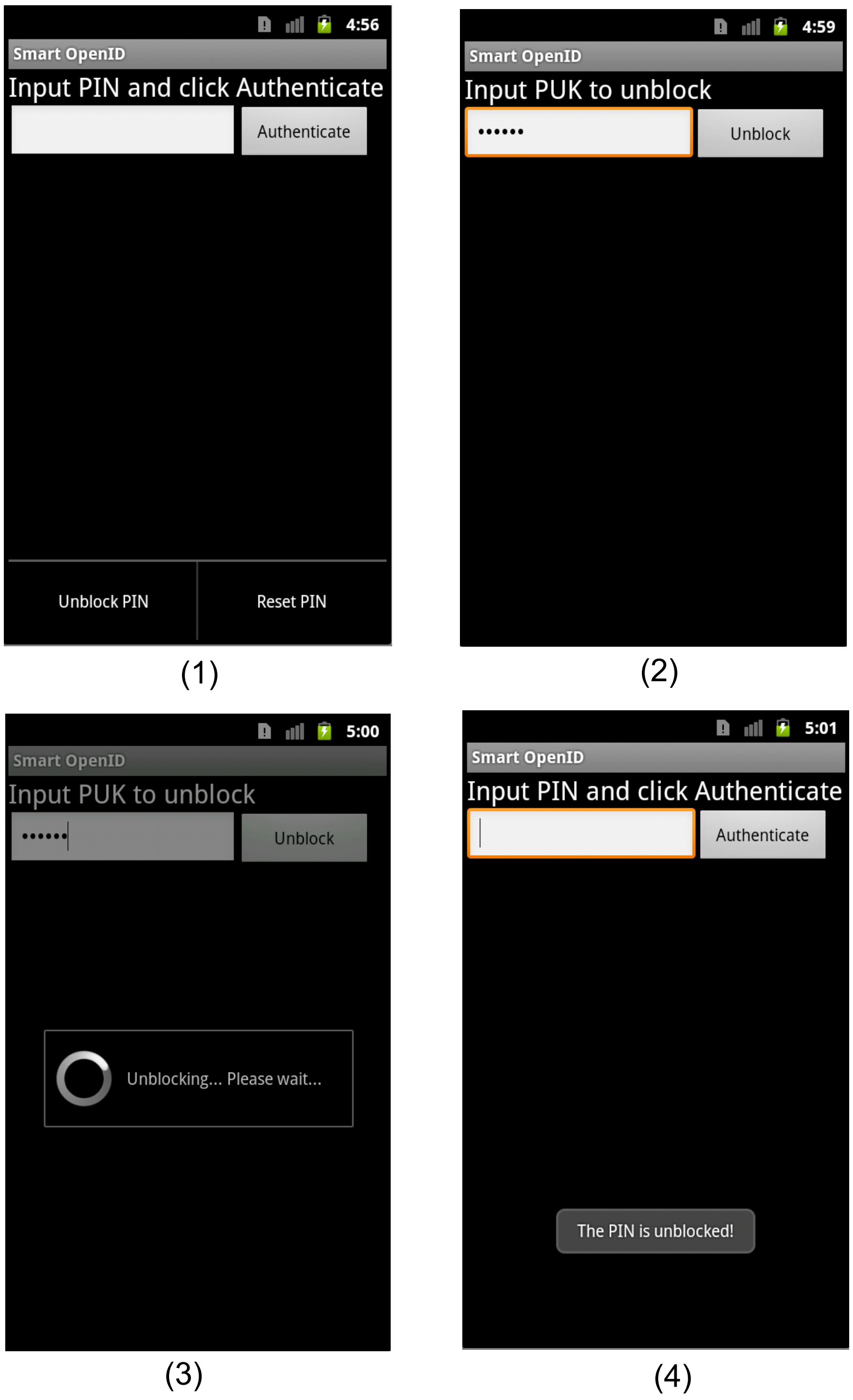
Authenticate

Authenticating... Please wait...

http://192.168.1.4/chec...

Login successfully!

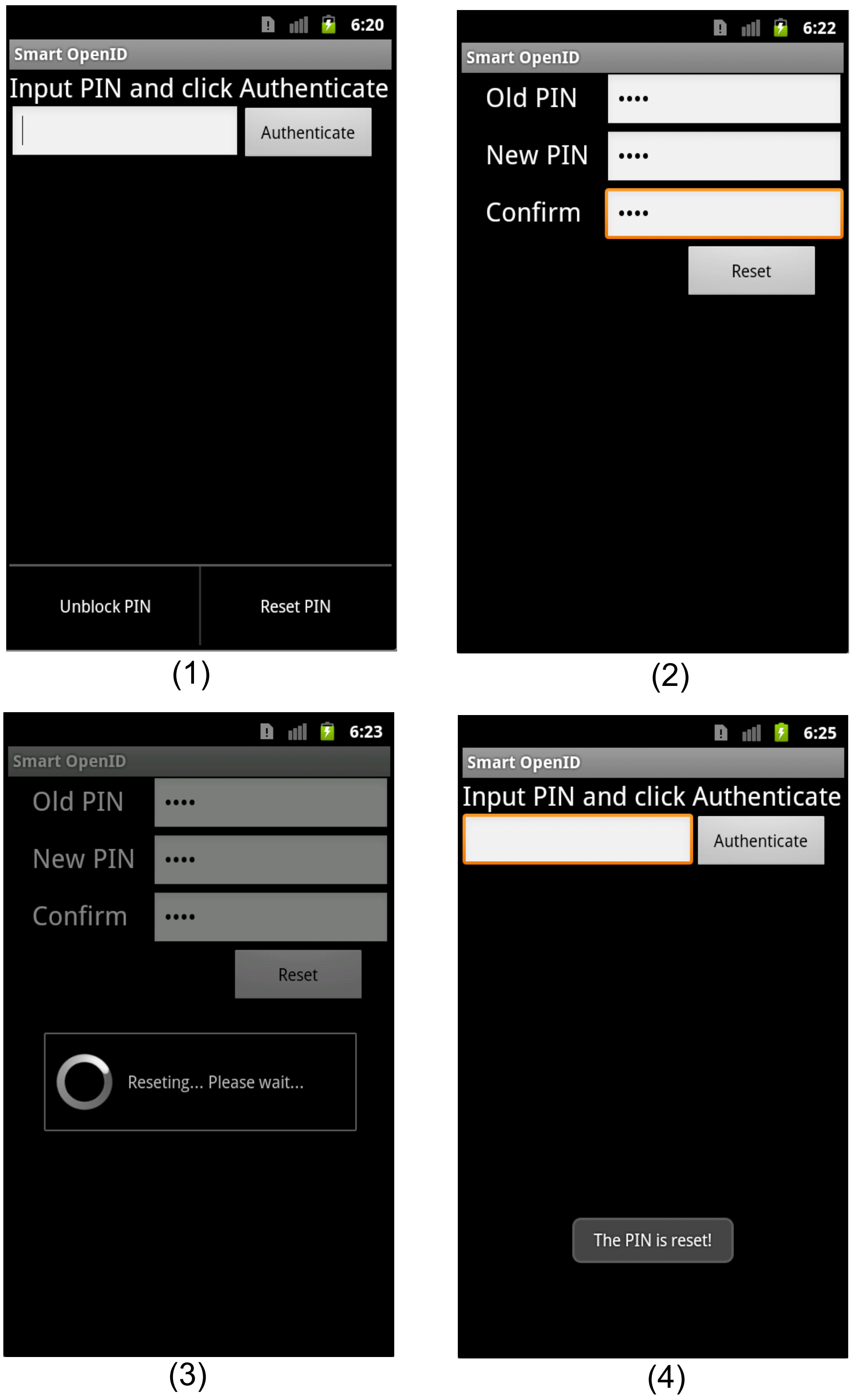
Figure 6.5: Authentication Testing



Unblocking... Please wait...

The PIN is unblocked!

Figure 6.6: Unblock PIN Testing



Reseting... Please wait...

The PIN is reset!

Figure 6.7: Reset PIN Testing

7 Summary and Future Work

Security features of the UICC makes it a suitable secure element on the smartphone. Using Open Mobile API, a dual application architecture can be deployed on the smartphone for security sensitive application and services. Smart OpenID protocol, which requires to setup a local authentication server on the smartphone, provides a proper argument of using the UICC to enhance the security of the mobile application. The implementation of the Smart OpenID local server with the dual application architecture is an example of leveraging the UICC for secure applications and services. The security analysis proves that the UICC is able to enable a more secure mobile environment, while the functionality and the performance test show that the use of Open Mobile API with the UICC is practical and feasible.

The implemented local authentication server contains only the HMAC-SHA-1 method for signing the authentication assertion. In the future implementation, the HMAC-SHA-256 method should also be included, since this is also supported by Smart OpenID.

The access control mechanism required by Open Mobile API is not included in the implementation, since the Android emulator does not support it. If the application is ported to other platforms or real phones, which support the access control module, then the UICC applet should also be implemented to support it.

Bibliography

[3GP12a] 3GPP. 3GPP TS 27.007 AT command set for User Equipment (UE). `http://www.3gpp.org/ftp/Specs/html-info/27007.htm`, 3 2012.

[3GP12b] 3GPP. 3GPP TS 33.102: 3G security; Security architecture. `http://www.3gpp.org/ftp/Specs/html-info/33102.htm`, 03 2012.

[ANN05] N. Asokan, Valtteri Niemi, and Kaisa Nyberg. Man-in-the-middle in tunnelled authentication protocols. In *Proceedings of the 11th international conference on Security Protocols*, pages 28–41, Berlin, Heidelberg, 2005. Springer-Verlag.

[Ass11] GSM Association. NFC Handset APIs & Requirements Version 2.0. `http://www.gsma.com/mobilenfc/wp-content/uploads/2012/03/gsmanfcuiccrequirementsspecificationversion20.pdf`, 11 2011.

[ASS12] Ammar Alkassar, Steffen Schulz, and Christian Stble. Sicherheitskern(e) fr smartphones: Anstze und lsungen. *Datenschutz und Datensicherheit*, page 175, 2012.

[BBB10] B. Ballad, T. Ballad, and E. Banks. *Access Control, Authentication, and Public Key Infrastructure.* Information Systems Security & Assurance. Jones & Bartlett Learning, 2010.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 1–15, London, UK, UK, 1996. Springer-Verlag.

[Ben06] M. Benantar. *Access Control Systems: Security, Identity Management and Trust Models.* Springer Science+Business Media, 2006.

[BGH$^+$12] Bastian Braun, Patrick Gemein, Benedikt Hfling, Michael Maisch, and Alexander Seidl. Angriffe auf openid und ihre strafrechtliche bewertung. *Datenschutz und Datensicherheit - DuD*, 36:502–509, 2012. 10.1007/s11623-012-0168-5.

[BJM08] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 75–88, New York, NY, USA, 2008. ACM.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996.

[Boo11] Mickey Boodaei. Mobile users three times more vulnerable to phishing attacks. `http://www.trusteer.com/blog/mobile-users-three-times-more-vulnerable-phishing-attacks`, 1 2011.

[CGRS09] Jean-Louis Carrara, Herv Ganem, Jean-Franois Rubon, and Jacques Seif. The role of the uicc in long term evolution all ip networks. `http://www.gemalto.com/telecom/download/lte_gemalto_whitepaper.pdf`, 1 2009.

[Che00] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. The Java Series. Addison-Wesley, 2000.

[Cia11] M. Ciampa. *Security+ Guide to Network Security Fundamentals*. Course Technology, 2011.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[eDa] Morpho e Documents. SIMply ON Java UICC for GSM and 3G networks. `http://www.morpho.com/IMG/pdf/morpho_telco_simply_on_2p_gb-3.pdf`.

[eDb] Morpho e Documents. Telecoms smart card tools for all phases of the (u)sim lifecycle. `http://www.morpho.com/IMG/pdf/morpho_telecoms_toolsoverview_8p_gb.pdf`.

[fA] SEEK for Android. Secure element evaluation kit for android platform. `http://code.google.com/p/seek-for-android/`.

[FFC$^+$11] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[FGM$^+$99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[FHBH$^+$97] J. Franks, P. Hallam-Baker, J. Hostetler, P. Leach, A. Luotonen, E. Sink, and L. Stewart. An Extension to HTTP : Digest Access

Authentication. RFC 2069 (Proposed Standard), January 1997. Obsoleted by RFC 2617.

[FHBH+99] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999.

[Fou11] The Apache Software Foundation. Open Source Java SE. `http://harmony.apache.org/`, 2011.

[Gem08] Gemalto. Security and trust in mobile applications. `http://www.gemalto.com/telecom/download/security_trust_in_mobile_applications.pdf`, 10 2008.

[Glo11] GlobalPlatform. Globalplatform card specification version 2.2.1. `http://www.globalplatform.org/specificationscard.asp`, 1 2011.

[Gol09] Ivan Golubev. IGHASHGPU. `http://www.golubev.com/hashgpu.htm`, 11 2009.

[Gui] Android Platform Developer's Guide. Radio layer interface. `http://www.kandroid.org/online-pdk/guide/telephony.html`.

[Gui12] Android Dev Guide. Intents and intent filters. `http://developer.android.com/guide/topics/intents/intents-filters.html`, 2012.

[Hyp06] Mikko Hypponen. Malware goes mobile. *Scientific American*, pages 70 – 77, 2006.

[IA06] M. Ilyas and S.A. Ahson. *Smartphones: Research Report*. Research Report Series. International Engineering Consortium, 2006.

[Inf] Android Tec Info. Android security overview. `http://source.android.com/tech/security/index.html`.

[IT02] ITU-T. Information technology ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), 7 2002.

[JFH+05] Audun Jøsang, John Fabre, Brian Hay, James Dalziel, and Simon Pope. Trust requirements in identity management. In *Proceedings of the 2005 Australasian workshop on Grid computing and e-research - Volume 44*, ACSW Frontiers '05, pages 99–108, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[JKLW11] Woongryul Jeon, Jeeyeon Kim, Youngsook Lee, and Dongho Won. A practical analysis of smartphone security. In Michael Smith and Gavriel Salvendy, editors, *Human Interface and the Management of Information. Interacting with Information*, volume 6771 of *Lecture Notes in Computer Science*, pages 311–320. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-21793-7_35.

[JOG07] GOPAL RAO JOGINIPALLY. Assune all input is malicious until proven otherwise. `http://joginipally.blogspot.de/2007/09/assume-all-input-is-malicious-until.html`, 9 2007.

[Jos11] S. Josefsson. PKCS #5: Password-Based Key Derivation Function 2 (PBKDF2) Test Vectors. RFC 6070 (Informational), January 2011.

[Kal00] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), September 2000.

[Kas11] Kaspersky. Mobile security software what it must do. `http://newsroom.kaspersky.eu/en/texts/detail/article/mobile-security-software-what-it-must-do`, 6 2011.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. Updated by RFC 6151.

[KR95] B. Kaliski and M. Robshaw. Message authentication with MD5. *CryptoBytes (RSA Labs Technical Newsletter)*, 1(1), 1995.

[Lea11] Neal Leavitt. Mobile security: Finally a serious problem? *Computer*, 44:11–14, 2011.

[LSS12] Andreas Leicher, Dr. Andreas U. Schmidt, and Dr. Yogendra Shah. Smart openid: A smart card based openid protocol. In *SEC*, pages 75–86, 2012.

[mar] marcoslot.net. Beginner's guide to openid phishing. `http://www.marcoslot.net/apps/openid/`.

[MC07] Tyler Moore and Richard Clayton. An empirical analysis of the current state of phishing attack and defence. In *In Proceedings of the 2007 Workshop on the Economics of Information Security (WEIS)*, 2007.

[McA12] McAfee. Mcafee threats report: First quarter 2012. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2012.pdf`, 2012.

[MK09] H. Mosemann and M. Kose. *Android.* Hanser, 2009.

[Net11] Juniper Networks. 2011 mobile threats report. Technical report, Juniper Networks Mobile Threat Center, 2011.

[Ope07] OpenID. OpenID Authentication 2.0 - Final, 12 2007.

[PLPS03] Jose Puthenkulam, Victor Lortz, Ashwin Palekar, and Dan Simon. The compound authentication binding problem. `http://tools.ietf.org/pdf/draft-puthenkulam-eap-binding-02.pdf`, 3 2003.

[PP03] C.P. Pfleeger and S.L. Pfleeger. *Security in Computing*. Prentice Hall PTR, 2003.

[PUB12] FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION. Secure hash standard (shs). `http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf`, 3 2012.

[RE10] W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley & Sons, 2010.

[Res99] E. Rescorla. Diffie-Hellman Key Agreement Method. RFC 2631 (Proposed Standard), June 1999.

[SIM11a] SIMalliance. Open Mobile API specification V2.02, 11 2011.

[SIM11b] SIMalliance. Secure authentication for mobile internet services: Critical considerations. `http://www.simalliance.org/en?t=/documentManager/sfdoc.file.supply&fileID=1322759212556`, 12 2011.

[SIM11c] SIMalliance. Simalliance open mobile api an introduction. `http://www.simalliance.org/en?t=/documentManager/sfdoc.file.supply&fileID=1302013515265`, 4 2011.

[SM06] Inc. Sun Microsystems. Runtime Environment Specification Java Card Platform, Version 2.2.2. `http://download.oracle.com/otndocs/jcp/java_card_kit-2.2.2-fr-oth-JSpec/`, 3 2006.

[Sma09] SmartTrust. The role of sim ota and the mobile operator in the nfc environment, 4 2009.

[TAN05] V. Torvinen, J. Arkko, and M. Naslund. Hypertext Transfer Protocol (HTTP) Digest Authentication Using Authentication and Key Agreement (AKA) Version-2. RFC 4169 (Informational), November 2005.

[Top12] TopTenREVIEWS. Smartphones review. `http://cell-phones.toptenreviews.com/smartphones`, 2012.

[TT07] Eugene Tsyrklevich and Vlad Tsyrklevich. Single Sign-On for the Internet: A Security Story. https://www.blackhat.com/presentations/bh-usa-07/Tsyrklevich/Whitepaper/bh-usa-07-tsyrklevich-WP.pdf, 2007.

[VLP11] S. Verclas and C. Linnhoff-Popien. *Smart Mobile Apps: Mit Business-Apps ins Zeitalter mobiler Geschäftsprozesse*. Springer, 2011.

[Whi11] Martin Whitehead. GSMA Europe response to European Commission consultation on eSignatures and eIdentification. Technical report, GSMA, 2011.