Institutionen för datavetenskap
Department of Computer and Information Science

Master’s Thesis

Design and Implementation of the Ephemerizer System

Shangjin XU

Reg Nr: LITH-IDA-EX--07/032--SE
Linköping 2007
Master’s Thesis

Design and Implementation of the Ephemerizer System

Shangjin XU

Reg Nr: LITH-IDA-EX--07/032--SE
Linköping 2007

Handledare: Supervisor
PROF.DR.CHristoph Schuba, Linköpings universitet

Examinator: Examiner
PROF.DR.CHristoph Schuba, Linköpings universitet
This thesis describes the system design and implementation of the secure Ephemerizer System that was first introduced by Radia Perlman in 2005. The system is designed to enable users to keep data for a finite period of time before making the data unrecoverable by destroying the keys with which the data was encrypted. The task of the Ephemerizer System service is to create, advertise, and destroy keys required for the Ephemerizer System’s functionalities.

We designed the Ephemerizer System Service’s security by placing the sensitive key management modules into a Trusted Computing Base (TCB). Our compartmentalized approach distributes security requirements at different sensitivity levels into different protection domains. In our approach, we implement the trusted protection domain (our TCB) on a tamper-resistant JavaCard.

We placed the key storage database into the partly trusted protection domain to improve scalability and availability of the Ephemerizer System. The partly trusted protection domain requires memory isolation and other security mechanisms provided by the underlying operating system. We implemented several mechanisms on the TCB, such as the signature engine, cryptographic modules, the on-card expiration validator, and on-card time verification. We make the Ephemerizer System available to users as a web service and expose it through a uniform API. This approach enables the seamless integration of the Ephemerizer System into business processes on heterogeneous platforms.
Abstract

This thesis describes the system design and implementation of the secure Ephemerizer System that was first introduced by Radia Perlman in 2005. The system is designed to enable users to keep data for a finite period of time before making the data unrecoverable by destroying the keys with which the data was encrypted. The task of the Ephemerizer System service is to create, advertise, and destroy keys required for the Ephemerizer System’s functionalities.

We designed the Ephemerizer System Service’s security by placing the sensitive key management modules into a Trusted Computing Base (TCB). Our compartmentalized approach distributes security requirements at different sensitivity levels into different protection domains. In our approach, we implement the trusted protection domain (our TCB) on a tamper-resistant JavaCard.

We placed the key storage database into the partly trusted protection domain to improve scalability and availability of the Ephemerizer System. The partly trusted protection domain requires memory isolation and other security mechanisms provided by the underlying operating system. We implemented several mechanisms on the TCB, such as the signature engine, cryptographic modules, the on-card expiration validator, and on-card time verification. We make the Ephemerizer System available to users as a web service and expose it though a uniform API. This approach enables the seamless integration of the Ephemerizer System into business processes on heterogeneous platforms.
Acknowledgments

There are many people I would like to thank for a huge variety of reasons.

Firstly, I would thank my professor Christoph Schuba for offering this chance to study with him. He gave me advice in my research and in life. When I was at crossroad of my life, he gave me his experiences and suggestions. I would also like to thank all the rest of the academic and support staff at IDA.

I would like to say a big 'thank-you' to my parents. They gave me much support from every aspect.

I would like to thank many of my friends for helping with this thesis and my life in Linköping. Thanks for Yang’s food, Shuang’s parties, Qi’s jokes, and all the other guys. It is difficult to list all of you guys, however, you know what I am thinking.
# Contents

List of Figures iv

List of Tables v

1 Introduction 1
   1.1 Notation ........................................ 3
   1.2 Usage Scenarios ................................ 3
   1.3 Technologies Background ....................... 5
      1.3.1 Javacard Technology ....................... 5
      1.3.1.1 Smart cards ................................ 5
      1.3.1.2 The reader side application ............. 7
      1.3.1.3 The card side application ............... 8
      1.3.2 Trusted Computing Base .................... 9
      1.3.3 Web Services ................................ 9
   1.4 Typographic Conventions ....................... 11

2 Problem Statement 13

3 Ephemizer System and Its Protocols 15
   3.1 Previous Work .................................. 15
      3.1.1 Ephemeral Key Management ................ 19
   3.2 Secure Channel Between Service Consumers and EPHEMERIZER SYSTEM .................. 20

4 System View 23
# List of Figures

1.1 Scenario ................................................. 4  
1.2 Architecture of a Javacard Application ............... 8  
3.1 Secure Channel ......................................... 20  
4.1 The Ephemerizer Service Infrastructure ............... 25  
4.2 Database Administration ............................... 27  
4.3 Javacard Administration ............................... 28  
4.4 The Request of an ephemeral key pair ............... 29  
4.5 The Retrieval of the Ephemeral Key Pair ............ 30  
4.6 The Encryption Procedure ........................... 31  
4.7 The Decryption Procedure ........................... 31  
5.1 The Request of the Ephemeral Key .................... 34  
5.2 Decryption Procedure ............................... 35  
5.3 Security Level of Ephemeral Key Management Modules ... 37  
5.4 The encryption ......................................... 39  
5.5 Security level of AES Engine ....................... 40  
5.6 Security level of Expiration Validator ............... 41  
5.7 The bits of the private ephemeral ................... 43  
5.8 Mutual Authentication ............................... 46  
6.1 Javacard Applets Installation Sequence .............. 55
List of Tables

1.1 Typographic Conventions ........................................... 11
6.1 Data Mapping ........................................................... 50
8.1 Speed Test of Javacard ............................................... 78
Chapter 1

Introduction

Keeping data accessible for at most a finite period of time is an important and difficult problem to solve. Let us consider the scenario where a company is electing its president. According to their rules, the ballots should be readable for at most three months after the election. We are dealing with the problem to ensure that the documents will indeed be destroyed, because even after deleting them, copies may remain accessible on media of some election group members.

The straightforward solution is to store data encrypted and deactivate the key after expiration. Radia Perlman presents her approach in the paper "The Ephemerizer: Making data Disappear" [5] published in February 2005. The idea is to concentrate all key management expense and expertise in one place, a service which is called the "ephemerizer". A client uses this service to read an ephemerized message.

While this approach presents a basic solution to the problem of making data disappear, the problem remains that there is no protection for the
ephemerizer itself. Perlman’s paper assumes the Ephmerizer is a trusted entity. However, there are at least two ways in which an ephemerizer can fail\cite{5}. One was that a ephemeral key is forgotten prematurely or is unavailable when needed for decryption. The other way is that the key is still active beyond its expiration time. The absence of security measures in key generation, storage, and expiration validation results in many drawbacks and vulnerabilities. On the one hand, for example, without proper encryption, the key database cannot be backed up, because otherwise its contents might leak to the wrong party. On the other hand, without a backup, the entire key database could get lost in an Ephemerizer system crash, causing all documents encrypted with the lost keys to remain encrypted forever.

During the ephemeral key generation, security measures should be included to protect the key generation service in a potentially hostile environment. One measure accomplishes the attack on the Ephemerizer by uploading binary code to be executed. The running state of the key generation process may be monitored and tracked. A snapshot of the memory can be produced and thus any ephemeral keys present in the memory would be compromised. Therefore, the availability of the ephemerizer is threatened.

The key storage should be placed in a database to improve scalability and availability. Ephemeral key privacy solutions should be installed in multiple places. Encryption performed on the ephemeral key distributes secure storage responsibility to the underlying database. There should be limited access to the means to decrypt ephemeral keys in the database, and this access should be locked down and monitored, with suspicious activity logged\cite{3}. In this thesis we present as our contribution a security design
and implementation for ephemeral key management, expiration validation, and storage.

1.1 Notation

In this thesis, we use the following expression to represent certain terminology.

$Key^+$ represents a public key, while as $key^-$ means a private key.

$\{M\}key^+$ means the message $M$ is encrypted with the public key $key^+$.

$\{C\}key^-$ means applying the decryption with the private key $key^-$ to the cipher $C$.

$HMAC(T, X|Y)$ means applying $HMAC(T, X|Y)$ with key $T$ to the quantity $X$ concatenated with $Y$.

1.2 Usage Scenarios

Figure 1.1. illustrates our approach:

1. **Encryption Service Consumer** generates a session secret key $S$.

2. **Encryption Service Consumer** encrypts message $M$ with the secret key $S$, which produces $\{M\}S$.

3. Secret key $S$ is encrypted by **Decryption Service Consumer’s private key** $K_{bob}$, this encrypted argument $\{S\}K_{bob}$ is again encrypted by $K_{eph}$, resulting in $\{{\{S\}K_{bob}}\}K_{eph}$, further on $\{{\{S\}K_{bob}}\}K_{eph}$ is encrypted by a session secret key $T$ resulting in $\{{\{{S}\}K_{bob}}\}K_{eph}\}T$. 

3 / 86
4. The session secret key $T$ is encrypted with $K_{bob}$ which produces $\{T\}K_{bob}$.

5. **Encryption Service Consumer** computes HMAC value by using session secret key $T$ and arguments from step 2, step 3, and $K_{eph}$.

6. **Encryption Service Consumer** sends the $K_{eph}$, $\{M\}S$ (from step 2), $\{\{S\}K_{bob}\}K_{eph}T$ (from step 3), HMAC (from step 5), and Key ID to the **Decryption Service Consumer**.

7. **Decryption Service Consumer** decrypts $\{T\}K_{bob}$ with his private key.

8. **Decryption Service Consumer** verifies the HMAC value (generated in step 5) by using decrypted $T$ from step 7.

9. **Decryption Service Consumer** decrypts $\{\{S\}K_{bob}\}K_{eph}T$ by $T$ (from step 7), and generates a session key $J$ and encrypts $\{\{S\}K_{bob}\}K_{eph}$ with $J$, which produces $\{\{S\}K_{bob}\}K_{eph}J$.

10. **Decryption Service Consumer** encrypts $J$ with $K_{eph}$.

---

**Figure 1.1.** Scenario
11. **Decryption Service Consumer** sends Key Id, $\{{{S}K_{bob}\}K_{eph}\}J$ (step 9) and $\{J\}K_{eph}$ (step 10) to the ephemizer.

12. **Ephemizer Service** decrypts $\{J\}K_{eph}$.

13. **Ephemizer Service** decrypts $\{{{S}K_{bob}\}K_{eph}\}J$ with J. J is retrieved in step 12.

14. **Ephemizer Service** decrypts $\{{{S}K_{bob}\}K_{eph}\}$.

15. **Ephemizer Service** encrypts $\{S\}K_{bob}$ with J, which produces $\{{{S}K_{bob}\}J\}$.

16. **Ephemizer Service** sends $\{{{S}K_{bob}\}J$ back to the **Decryption Service Consumer** for recovering message M.

### 1.3 Technologies Background

This section describes basic background concepts and some key knowledge related to the research.

#### 1.3.1 Javacard Technology

##### 1.3.1.1 Smart cards

A smart card is a plastic pocket-sized card with embedded integrated circuits. It is portable and tamper-resistant. Smart cards are much more difficult to duplicate than magnetic strip cards. Cryptographic functions can be implemented inside smart cards. It is capable for holding sensitive data. Smart cards have a widespread of applications. Major applications include financial services such as to provide account information for credit, debit, and cash; In IT, it provides services like user authentication and cryp-
1. Introduction

tography. Although there is a diverse range of applications, there are two broad categories of smart cards—memory cards and microprocessor cards.

- **Memory cards** contain only memory storage components and some specific security logic.

- **Microprocessor cards** include memory and microprocessor components. Some smart cards have separate cryptographic coprocessors that support algorithms such as RSA, AES, and (3)DES.[23]

Smart cards are very useful in the areas of security and they provide secure access to stored data. Smart cards are designed so that information stored in them is portable and secure. They can be used for secure access to information systems that require a high level of security. Smart cards are highly secure by design. Tampering it results in the destroying the information it contains.

Smart cards can be either contact or contactless. Contact smart cards work by physical contact to a card reader device that can read data from the card and write information back. Contactless smart card communicates with the card reader through radio frequency signal (at data rates of 106 to 848 kbit/s). The radio communication technology of contactless smart cards is similar to Radio Frequency ID (RFID) used in warehouse to track stocks.

Javacard technology adapts the Java platform for use on smart cards and other devices whose environments are highly specialized, and whose memory and processing constraints are more severe than those of J2ME.
1.3.0. Technologies Background

devices[23]. Javacard applications (applets) are written and debugged with standard Java language environment, and can be installed and executed on smart cards. Javacard applications are based on the Javacard Platform specifications developed by Sun Microsystems. Because Javacard applications (applets) are small, it is easy for multiple applications to run on a single card with limited memory on the card.

A typical Javacard device has an 8-bit or 16-bit CPU running at 3.7MHz, with 1K of RAM and more than 16K of non-volatile memory (EEPROM or flash). High-performance smart cards come with a separate processor, cryptographic chip, and memory for encryption[23]. Java applets are located in EEPROM and separated from card operating system, and thus can be loaded at any time. A typical Javacard system places Javacard Runtime Environment code in the ROM. RAM is used for temporary storage, for instance, execution state, local variables and transient objects are placed in the RAM by virtual machine. Due to the limited memory and computing abilities of the smart cards, only a small set of the Java language features are supported[10].

1.3.1.2 The reader side application

The host application provides general communication mechanism to the Javacard. It works as an interface between Javacard applications (Applets) and the outside world. The host application can reside on desktop or the terminal such as a mobile phone SIM card, a POS terminal, or a PC. As shown in the Figure 1.2, the card reader or card acceptance device functions as a physical interface between the host application and applets lo-
1. Introduction

In order to complete communications among the host application, the card acceptance device and the Javacard, a logical data packet is involved. The packet is called an Application Data Unit (APDU). The card acceptance device receives and forwards all incoming APDUs to the Javacard. The applet residing on the card processes APDUs in sequence and returns responses.

1.3.1.3 The card side application

As illustrated in the Figure 1.2, one or more Javacard applets can run on the card. These Java applets reside on top of the card’s operating system and the Javacard Runtime Environment (JCRE) which consists of the Javacard Virtual Machine and the Javacard framework. The Javacard Virtual Machine hides different manufacturer’s specific technologies with a common language and system interface, and provides a common Java-based plat-
1.3.0. Technologies Background

form. The Javacard framework defines a set of Application Programming Interface (API) classes for building applets and providing system services to those applications. An industry can supply interfaces through additional API built on top of the Javacard framework to provide an enhanced service.

1.3.2 Trusted Computing Base

Two critical components of the computer security problem are access control and data dissemination control[9]. The access control mechanisms prevent unauthorized users from accessing confidential data. The data dissemination control mechanisms protect sensitive or confidential data from being revealed to unauthorized users. These two components make the system more trustworthy, such trustworthy computing requires a Trusted Computing Base.

The US TCESC (“Orange Book”) defines “Trusted Computing Base” as “The totality of protection mechanisms within a computer system—including hardware, firmware, and software - the combination of which in responsible for enforcing security policy”[7]. The software and hardware, which are a set of mechanisms to provide the security of the system, are a part of TCB. A TCB consists of one or more modules to enforce the security policies.

1.3.3 Web Services

Web services are self-contained, self-describing and modular applications that can be published, located and dynamically invoked across the
web (ISO 2001). Web services infrastructure includes different layers. It is based on a set of XML standards such as Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Universal Description, Discovery, and Integration (UDDI). SOAP defines a basic communication protocol for consumers to exchange XML messages[28]. It is a lightweight XML based protocol for exchanging data in a decentralized and distributed environment. Web Service Definition Language (WSDL) supports the implementation of web services by providing a standard XML format of describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information[30]. Universal Description, Discovery, and Integration is a standard for web services to register and make web services available to consumers[30].

Web services provide a mechanism to carry out business process through the exchange of data and the remote invocation of logic. It uses common transport protocols (HTTP) to move data through firewalls and between heterogeneous systems. The programs written in any language using any component, and running on any operating system can access XML Web services[11]. By exposing components of applications as web service and enabling business to invoke these components, applications can be integrated in a loosely coupled way to achieve business goals. Complexity of application connection can thus be decreased in order to reduce the cost of maintenance and updating.
1.4 Typographic Conventions

<table>
<thead>
<tr>
<th>Representation</th>
<th>meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc</td>
<td>The formula of cryptographic content</td>
<td>{M}C</td>
</tr>
<tr>
<td>ABC</td>
<td>The name of the system module</td>
<td>EPHEMERIZER SERVICE</td>
</tr>
<tr>
<td>AaBbCc</td>
<td>Codings</td>
<td>for(int i=0; i&lt;20; i++)</td>
</tr>
</tbody>
</table>

**Table 1.1.** Typographic Conventions

In a printed copy of this thesis, the figures and graphs are likely to appear black and white, which makes some of the figures and graphs difficult to interpret correctly. An electronic copy of this thesis, which contains these figures and graphs in color and high resolution, can be found at http://www.ep.liu.se
Chapter 2

Problem Statement

The Key Ephemerizer System consists of the Ephemerizer Service, an Encryption Service Consumer and a Decryption Service Consumer. A user accesses the Epphemeral Service though the Encryption Service Consumer or a Decryption Service Consumer. The Epphemeral Service’s task is to create, advertise and destroy keys. It concentrates all the expense and expertise of the key management in one service. The Encryption Service Consumer and the Decryption Service Consumer act as service requestors. An Encryption Service Consumer must be capable of requesting an ephemeral public key from the Epphemeral Service to encrypt with it. A Decryption Service Consumer must be capable of communicating with the Epphemeral Service to access its services.

One approach to make data disappear reliably on a pre-determined date is to store the data encrypted and to deactivate the keys on that date. Another essential feature of the Epphemeral System is that it should be centralized to manage its keys. The fact that the Epphemeral System
design is centralized makes it easier to manage the ephemeral key storage database.

We designed the **EPHEMERIZER SERVICE** implementation to consist of three protection domains based on how much trust needs to be placed into them: the partly trusted, the trusted, and the untrusted protection domain. The partly trusted domain is responsible for ephemeral key storage and one-time used key functionality (message decryption). One-time means the key will not be kept in the memory after the decryption process completes. The security of the partly trusted domain is assured by the underlying operation system.

In the trusted protection domain, the ephemeral keys that are stored in the database are enciphered by the ephemeral key encryption functionality. The trusted protection domain is designed and implemented such that it must be trusted for proper operation analogous to a trusted computing base (TCB). We choose to implement the TCB in a Javacard. All execution of code in the TCB is subject to access control. The access control mechanism prevents unauthorized hosts from accessing the ephemeral key pair generation, encryption, and decryption functionalities. The ephemeral private key decryption is monitored by the expiration validator. The expiration validator guarantees that expired ephemeral keys can never be used. All other functionality, such as the request dispatcher, resides in the untrusted domain.
Chapter 3

Ephemerizer System and Its Protocols

3.1 Previous Work

Let us assume that the message $M$ is encrypted with the ephemeral key $K_{eph}$. In this way, an Encryption Service Consumer encrypts message $M$ with the ephemeral key $K_{eph}$ and sends the encrypted message $\{M\}K_{eph}$ to the Decryption Service Consumer. When the Decryption Service Consumer wants to decrypt the encrypted message $\{M\}K_{eph}$, it initiates the requests to the Ephemerizer. The Ephemerizer responds by sending $K_{eph}$ to the Decryption Service Consumer. Once the ephemeral key $K_{eph}$ is destroyed after its expiration time, message $M$ that is still encrypted with key $K_{eph}$ can no longer be recovered.

Some problems arise in above approach.

1. Confidentiality of the encrypted message is not achieved. Confidentiality is the assurance that only an authorized Decryption Service
3. Ephemerizer System and Its Protocols

**Consumer** will be able to read the encrypted message \( \{M\} K_{eph} \). The privacy of the encrypted message \( \{M\} K_{eph} \) can be disclosed by a man-in-the-middle attack. If the attacker can successfully trick either the **Encryption Service Consumer** or the **Decryption Service Consumer** into using an attacker's public ephemeral key, he can obtain an encrypted message \( \{M\} K_{eph} \) and thus recover the message \( M \).

2. Message integrity is not provided in this approach. There is no guarantee that the encrypted message \( \{M\} K_{eph} \) has not been tampered with during transmission.

To prevent these problems, we should offer both message integrity and confidentiality in the protocol. However, an attacker can try various ways to attack a protocol. If someone who is not involved in the protocol attempts to eavesdrop the message transmitted, the message integrity should not be affected. This kind of passive attack does not affect the protocol, because it is difficult to detect. The protocol should prevent it rather than detect it. Alternatively, an attacker can try to perform active attacks which allow an attacker to pretend to be someone else, introduce new messages in the protocol, delete existing messages, substitute one message for another, relay old messages, interrupt a communications channel, or alter stored information in a computer[1]. It is possible that an attacker is a legitimate participant involved in the protocol. For example, in the above scenario, the **Decryption Service Consumer** can keep the ephemeral key \( K_{eph} \) and the message \( \{M\} K_{eph} \) after obtaining them from the Ephemerizer. Hence the protocol should also prevent this kind of active attacks.
We need a self-enforcing mechanism in which no trusted third party is involved to complete the protocol. In our approach, we assume the Decryption Service Consumer has a long term public key pair $K_{bob}^+$ and $K_{bob}^-$. We also assume the Decryption Service Consumer does not store the decrypted message $M$ in a stable storage. "It is possible in many operating systems to have a process that will not be swapped out to stable storage, so the ephemerization software at the client should see that feature to ensure that the decrypted message never appear on stable storage."[5]

If an Encryption Service Consumer wants to encrypt a confidential message $M$ that can be obtained only by a Decryption Service Consumer designated by the Encryption Service Consumer, a randomly generated key $S$ will be used to encrypt the $M$ with the triple-DES algorithm, and then the triple-DES secret key $S$ is encrypted with the public key $K_{bob}^+$ of the designated Decryption Service Consumer. When the Decryption Service Consumer wants to retrieve the secret key $S$, the corresponding Decryption Service Consumer’s private key $K_{bob}^-$ is needed.

The privacy of message $M$ can also be achieved by a public algorithm. The disclosure of ephemeral keys will not reveal the message $M$ associated with it. Thus the encrypted triple-DES secret key $\{S\}K_{bob}^+$ is further encrypted with the public ephemeral key $K_{eph}^+$. We use RSA algorithm as our ephemeral key algorithm.

Another consideration is that "RSA is much slower to compute than a popular secret key algorithm like DES. As a result, RSA does not tend to get used for encrypting long messages. Mostly, it is used to encrypt a se-
cret key, and then a secret key cryptography is used to actually encrypt the message."[2] Symmetric cryptography such as triple-DES is best for encrypting data[1]. It is faster and not susceptible to chosen-ciphertext attacks[1]. Public-key cryptography such as RSA algorithm is best for key management[1]. Therefore, the encryption of the message M is performed with the triple-DES algorithm, and the triple-DES secret key S is secured through the public key encryption algorithm. The secret key S is first enciphered with $K^{+}_{bob}$ resulting in $\{S\}K^{+}_{bob}$, and then this encrypted data is further encrypted by $K^{+}_{eph}$, which produces $\{S\}K^{+}_{eph}$.

The integrity of multiple encrypted triple-DES secret key S is achieved by using a keyed-hash message authentication code, HMAC. It is used to verify both the data integrity and the authenticity of a message. An attacker cannot compute the proper digest without the help of a corresponding key. After the HMAC value is sent to the Decryption Service Consumer, the recipient of these messages should compute the HMAC value with the session secret key S. If the newly computed value matches with the received HMAC, it proves that data has not been tampered with during transmission.

The advantage of the design is the existence of the triple-DES secret key S during the message encryption procedures which are executed by the Encryption Service Consumer. By taking the advantage of the triple-DES algorithm, the availability of the Ephemeralizer System and the security of the message M can be achieved. In addition, the decryption computing load of message M is migrated to the Decryption Service Consumer rather than the Ephemeralizer Service.
As depicted in the Figure 1.1, triple-DES algorithm guarantees that the message M will not be decrypted and stored in the Ephemerizer Service. It is achieved by enciphering message M with triple-DES secret key S, and enciphering S with multiple keys.

It should be noticed that the inner encryption with $K_{bob}^+$ of $\{\{S\}K_{bob}^+K_{eph}^+\}T$ protects triple-DES secret key S from being revealed in case that the Ephemerizer is compromised. The approach can guarantee that the secret key S never appears on the Ephemerizer. If the Ephemerizer is really dishonest, it can obtain S and recover message M by eavesdropping $\{M\}_S$. An attacker gaining access to the dishonest Ephemerizer will not be able to infer the message M.

Because the KeyID and the ephemeral key $K_{eph}^+$ are transmitted without encryption, an eavesdropper is able to obtain them. However, modifying them can only cause a Denial of Service attack which prevents the Decryption Service Consumer from decrypting the message. It will not lead the attacker to discover the message M. We strengthen the integrity of $K_{eph}^+$ by computing the HMAC value.

### 3.1.1 Ephemeral Key Management

Radia Perlman proposes that the Ephemerizer creates ephemeral key pairs and advertises the triples consisting of public ephemeral key, KeyID and the expiration time. This triple should be signed by the Ephemerizer such that whoever obtains the ephemeral key cannot back-date the associated expiration time. The Ephemerizer can hold a long-term key to sign
the triples. It is possible to use a smartcard to protect the long-term signature key. In our approach, we consider expiration time as high sensitive data. We want to keep the security and privacy of this sensitive date. By encrypting the date along with the key, we can restrict access to it during the execution of the Ephemerizer.

### 3.2 Secure Channel Between Service Consumers and Ephemerizer System

The transmission channel between the Decryption Service Consumer and the Ephemerizer System is protected by the triple-DES algorithm. As indicated in the Figure 3.1, the Encryption Service Consumer randomly
generates session keys (S and T), and uses them as the encryption key. The plain message M is further enciphered by S. The session key T is encrypted with the Decryption Service Consumer’s public key $key_{bob}^+$. 

Upon receiving a set of messages, the Decryption Service Consumer recomputes the HMAC value of the received messages. Then, the receiver compares these two values, if they match, the Decryption Service Consumer is assured that the messages did come from the claimed Encryption Service Consumer, and is intact during transmission. The Decryption Service Consumer decrypts the session key T with his private key. The Decryption Service Consumer will generate the session key J to triple-DES encipher the message. The session key J is further enciphered using the Decryption Service Consumer’s public key $key_{bob}^+$. 

The Ephemeral System retrieves the session key J with the corresponding private ephemeral key $key_{eph}^-$. 

The communication channel between the Encryption Service Consumer and the Decryption Service Consumer is secured by the triple-DES session key T. The channel between the Encryption Service Consumer and the Ephemeral System is protected by the Triple-DES session key J. In this topology, the approach adopts the asymmetric cryptographic solution for the transmission channel. If someone obtains the $key_{eph}^-$, the message $\{M\}S$ sent from the Encryption Service Consumer to the Decryption Service Consumer cannot be recovered because the session key S is ciphered by $key_{bob}^+$. 
Chapter 4

System View

Our **Ephemizer System** is comprised of the **Ephemizer Service**, an **Encryption Service Consumer** and a **Decryption Service Consumer**. The **Ephemizer System**, which is responsible for key management, is a general PC platform equipped with a Javacard reader.

The RSA and DES cryptographic modules are implemented inside the card to provide security functions for the ephemeral key management. The communication channel among the **Ephemizer Service**, the **Encryption Service Consumer** and the **Encryption Service Consumer** is secured by triple-DES algorithm.

The Javacard itself is subjected to access control by PIN mechanism. The initial PIN code is issued by the card programmer. The authorized administrator of the **Ephemizer Service** is able to change the PIN code at a later time. In addition, the card monitors the incoming time clock. Once the clock appears faked, the card locks itself. Only the administrator with proper rights can unlock the card in case of denial of service attacks.
4. System View

4.1 EpHEMERIZER Service Infrastructure

Figure 4.1 shows the infrastructure of our EpHEMERIZER Service. The EpHEMERIZER Service is comprised of the Webservice Handler and the Javacard security module. The role of the Javacard is to store the DES key, which is used to encipher/decipher ephemeral private keys, responsible for the ephemeral key management. The Webservice Handler includes administration modules, key management modules and decryption module. It runs on a general PC platform to which the Javacard is attached.

4.2 EpHEMERIZER Service Initialization

We give the following initialization steps of EpHEMERIZER Service:

Firstly, authorized administrators boot up the EpHEMERIZER Service PC platform and prepare the key storage database. Secondly, the authorized administrator inserts the Javacard into the card reader. Finally, the authorized administrator modifies the configuration file of the Webservice Handler to provide the identity to the card, and starts the Tomcat server.

4.3 Administrator View

Administration operations performed on the EpHEMERIZER Service are subject to role-based access control. Role-based access control enforces security policies in a centralized way. The required procedures are performed by the Webservice Handler. The central idea behind role-based access

24 / 86
Figure 4.1. The Ephemerizer Service Infrastructure
control is that users are assigned to roles, permissions are assigned to roles and users acquire permissions by being members of roles[4]. In our approach, identity of administrator is represented by administrator’s private key with his key ID. The membership assignment is done by inserting key ID in the database. Fast membership revocation can be easily accomplished by deleting corresponding key ID in the role table.

Authentication is an essential element for the security of the Ephemizer System which is the procedure of confirming the identity claimed by administrator and the Ephemizer Service. Without supports from the authentication, the identity claimed by administrator may be faked. There is a finite period of time associated with each identity. The identity expires after this pre-specified time. The validation process is accomplished by taking advantage of the Javacard module. After the identity is validated, authorization procedures which are performed on the Webservice Handler assure that the operation conforms to the security policy.

4.3.1 Database Administration

The database operations include the database initialization, data backup and data wipe.

The database system usually suffers from various attacks performed by malicious uses and attackers. As the stored ephemeral keys are encrypted in the database, keys can be prevented from being revealed by privileged database administrators. Without deciphering operations which are performed on the Javacard, only the encrypted keys can be obtained. The
other security mechanisms, including access control, intrusion detection, audit, etc, are supported by the underlying Operating System, some special softwares and hardware. The administration operation requests are subject to authentication and administrator authorization.

The mutual authentication builds a mutual trust relationship between administrators and the EPHEMERIZER SERVICE. The administrator can trust the EPHEMERIZER SERVICE resource to manipulate without interference. The EPHEMERIZER SERVICE can also trust the administrator not to perform malicious operations on it. In our scenario, when the administrator performs the backup of the database, mutual authentication assures database contents, such as keys, roles, and access permissions granted by EPHEMERIZER SERVICE, are trusted. The running state of the mutual authentication is kept in the database by the WEBSERVICE HANDLER. Furthermore, the WEBSERVICE HANDLER keeps the freshness of the state information by up-
4. System View

4.3 Javacard Administration

dating the database. The scalability and manageability of the database are enhanced by remote admin operations.

4.3.2 Javacard Administration

The operations on the Javacard, which are performed by administrators, are to unlock the time validator and to renew the PIN code.

Because the Javacard contains highly sensitive modules of the Ephemizer Service, additional authentication mechanism is required. Administrator should prove he knows the PIN code of the Javacard so that his operations are permitted. If the PIN code does not match, the operation is prohibited.
4.4 User View

4.4.1 Ephemeral Key Pair Request

Here we give an example to show how an ephemeral key is generated and transferred to the encryption service consumer. As indicated in the Figure 4.4, the encryption client requests public ephemeral key from the W EBSERVICE H ANDLER of the E PHEMERIZER S ERVICE. Upon receiving the request, the W EBSERVICE H ANDLER, who in turn forwards it to the Javacard, revokes the key generator residing on the Javacard to get the ephemeral key pairs. The ephemeral key pairs consist of one RSA public key and the corresponding encrypted private key with the expiration date. At the final step, the W EBSERVICE H ANDLER stores the ephemeral key pair in the database along with a randomly generated key ID, and sends the key ID back to the E NCRYPTION S ERVICE C ONSUMER.
4. System View

4.4.2 Ephemeral Key Pair Retrieval

As illustrated in the Figure 4.5, at step I, the Encryption Service Consumer requests the public ephemeral key from the Webservice Handler. It sends the request along with the key ID. The Webservice Handler module retrieves the ephemeral public key from key storage database based on the key ID.

4.4.3 Encryption Procedure

As depicted in the Figure 4.6, the Encryption Service Consumer encrypts the message $M$ by a triple-DES key $S$ to obtain $\{M\}S$. Then it generates a secret session key $T$, and computes a HMAC value. The HMAC value guarantee the message is intact during transmission and it really comes from the Encryption Service Consumer. Then the Encryption Service Consumer sends encrypted $T K^+_{bob}$. 
4.4.4 Decryption Procedure

As depicted in the Figure 4.7, the EPHMERIZER SERVICE selects $K_{eph}^-$ from the database, and decrypts the $\{J\}K_{eph}^+$ to acquire the session secret J. With the help of $K_{eph}^-$ and the session secret J, $\{S\}K_{bob}^+$ can be obtained through $\{\{\{S\}K_{bob}^+\}K_{eph}^+\}J$. Then the EPHMERIZER SERVICE encrypts $\{S\}K_{bob}^+$ with J and returns it to DECRYPTION SERVICE CONSUMER.
Chapter 5

System Design

5.1 Detail Scenarios

5.1.1 Request of Ephemeral Key

Figure 5.1 illustrates the process. Firstly, the service consumer requests services from the WEBSERVICE HANDLER. The ENCRYPTION SERVICE CONSUMER needs to provide the expiration date required for the generation of the ephemeral key pairs, which is indicated as step 2. Secondly, the ENCRYPTION SERVICE CONSUMER requests the ephemeral key through the WEBSERVICE HANDLER. Thirdly, the WEBSERVICE HANDLER performs several operations to invoke the key generation service on the Javacard. The key generator creates a pair of 512-bit RSA keys as ephemeral keys, which is depicted as step 3. The private key is encrypted with the expiration date by the AES engine, which is depicted as step 4. The signature of the public key is generated through the signature engine, which is depicted as step 5. Finally, the public key and the encrypted private key are stored in the database referenced by a random generated Key ID, and the WEBSERVICE
Figure 5.1. The Request of the Ephemeral Key
5.2.0. Determination of Security Level

\textbf{HANDLER} responds to the \textsc{Encryption Service Consumer} with the corresponding Key ID.

5.1.2 Decryption Procedures

Figure 5.2 illustrates the process. Firstly, the \textsc{Decryption Service Consumer} requests the decryption service through the \textsc{Webservice Handler}. To invoke the decryption service, the \textsc{Decryption Service Consumer} should provide his key ID. Secondly, the \textsc{Webservice Handler} processes the request by retrieving the encrypted ephemeral private key in the database, which is indicated as step 2 in the Figure 5.2. Thirdly, this encrypted key is forwarded to the Javacard for further verification and decryption. The encrypted key is deciphered in the AES engine, which is indicated as step 3. The decrypted key with the expiration date is validated by the expiration validator, which is described as step 4. If the key is not expired, the decryption process continues. Once the key is expired, a NULL value is returned. Finally, the \textsc{Webservice Handler} responds to the \textsc{Decryption Service Consumer} based on the result returned by the Javacard.

5.2 Determination of Security Level

Performance problems are caused by TCB mediations of all accesses between subjects and objects in the untrusted software[14]. The efficiency is degraded while the TCB performs its activities. The solution is to minimize TCB mediation as much as possible without sacrificing the security integrity of the system[14]. In addition, with limited resources on the Javacard, the
5. System Design

Figure 5.2. Decryption Procedure
5.2.0. Determination of Security Level

The implementation of the TCB is hindered. The necessary modules on the Javacard should be kept as minimal as possible.

Current Ephemeralizer Service is designed to offer different levels of protection domains, which is referred to as our compartmentalized approach. We began by separating protection domains into three levels: the trusted domain (high-level security), the partly trusted domain (medium-level security) and the untrusted domain (low-level security). The trusted protection domain is supported by TCB. The operating system with property configured security policies provides basis of the medium security domain.

5.2.1 Ephemeral Key Management Modules

Our design strives to place the storage of the ephemeral key pairs in a distributed environment. Only authorized administrator with proper rights can do operations on the database, such as back up the database. It means that the ephemeral key-pair database resides on the medium level security.
domain. With the limited computation power offered by the Javacard environment, any large block of data decryption will degrade the system’s performance. Furthermore, it may be time consuming because of data transfer between the Javacard and the Ephemizer Service. In addition, due to the fact that the card is a single-threaded platform that can serve only one request at a time, the Webservice Handler needs to queue other requests while an APDU command is being processed on the card. Hence, the message decrypter should run on the medium security domain.

The ephemeral keypair generator should not be known to any user process. If someone exploits the potential vulnerability and successfully controls the generator, the service consumer can be tricked into using the attacker’s ephemeral key, the attacker can thus compromise the Decryption Service Consumer’s long term key $K_{bob}$ and decrypt the message. Therefore, the key generator should be protected by the TCB.

Upon determining what modules must be trusted, as shown in the Figure 5.3, we need consider which portions of the infrastructure should also reside on the TCB.

### 5.2.2 Bell-LaPadula Model

The Bell-LaPadula model focuses on data confidentiality. It mainly enforces the following policies.

* A subject at a given security level cannot read an object at a higher security level (no read-up).
5.2.0 Determination of Security Level

Figure 5.4. The encryption

* A subject at a given security level must not write to any object at a lower security level (no write-down).

The security related software which does not conform to the Bell-LaPadula model must be trusted [14].

5.2.3 AES Engine

The **EPHEMERIZER SERVICE** should grant one form of access which is prohibited under the Bell-LaPadula Model with the mandatory policy. As shown in Figure 5.4, it is the encryption module that accepts data from high level protection domain as inputs and outputs cipher text to medium level protection domain. This form of access can be granted because the high level data is released in the form of encryption. It must be in knowledge that the value of the key is not known by any processes in a lower protection domain.[15]

The cryptographic module is essential for lower level protection domains so that accesses of contents in the database running on the partly trusted protection domain will not violate the Bell-LaPadula model. Because keys must not be exposed to the lower level protection domain, we use the
5. System Design

Figure 5.5. Security level of AES Engine

Javacard to fit our needs. The AES key and signature key are written into the EEPROM of the card so that they can be permanently stored in the card. As keys cannot be derived outside the card, the security can be guaranteed. Thus the cryptographic module(AES) is implemented as an integrated part of the TCB.

5.2.4 Expiration Validator

Let us consider the case of a date validation example. When the DECRYPTION SERVICE CONSUMER requests the decryption operation from the WEBSERVICE HANDLER, the WEBSERVICE HANDLER retrieves the encrypted private key from the database, and in turn forwards it to the AES decryption module which is protected by TCB. After the key is decrypted, it is passed to the expiration validator. If the key is not expired, the decryption process continues. At the first glance, the expiration validator should be placed on the medium level security domain because the validator need
check the clock and the Javacard itself does not have the clock. However, as the analysis continues, the assumption is wrong, this process does not conform to the Bell-LaPadula model, as indicated in Figure 5.6, because the expiration validator reads the ephemeral key from the higher level protection domain and the ephemeral key is not encrypted. Therefore we identify the requirement of the expiration validator to be trusted in TCB.

5.3 Javacard Module

For the Ephemerizer Service, as shown in Figure-4.1, several function modules, such as the ephemeral key pair generator, the ephemeral key decryption/encryption engine (AES engine), the expiration validator and the signature module exist on it. The cryptographic modules must be protected in order to transfer data among protection domains. For instance, to provide security in encryption/decryption module, keys must be protected. Hence, trusted hardware and platform are needed in our scenario. Trusted
5. System Design

hardware comes from different flavors. It becomes more difficult to provide tamper-proof hardware the larger the hardware gets[12].

Commodity operating systems are complex programs that contain millions lines of the code, thus they provide low assurance[13]. Building a simple and the most trusted security domain on top of these operating system is difficult because the security domain ultimately depends on the functionalities offered by the low-level operating system as its trusted computing base. Javacard, which is designed as taper-resistant devices, is a good candidate to deploy trusted computing base for cryptography modules. This feature-riched multifunctional chip card offers a new platform for realizing trusted hardware inexpensively and easily. It best suits the security needs.

5.3.1 Invocation Model of Javacard

Sun provides applets with a new communication model: Javacard Remote Method Invocation (JCRMI). It relies on a subset of the J2SE RMI distributed-object model. In JCRMI, the Javacard applet is the server, and the host application is the client. From a practical point of view, a server application residing on the card creates remote objects and makes them accessible to host applications. A host application gets access to the object by selecting the applet on the Javacard.

JCRMI provides a distributed-object model mechanism which is built on top of the APDU-based messaging model[23]. The message-passing model is the basis for all Javacard communications. The JCRMI messages are
transferred by Application Protocol Data Units (APDU), which is the communication protocol specified in ISO7816. To invoke a method on such object, the host application encapsulates the JCRMI message within the APDU commands and sends them to the on-card server application.

### 5.3.2 On-card Time Validator

The Javacard platform should assure the input clock is not faked. To achieve this goal, we designed our on-card clock validator. To thwart an attempt to update the on-card clock with fraudulent time, the on-card clock validator checks that no outdated clock is redeemed. The on-card clock validator works as the gateway to the legacy card acceptance. It extracts the incoming time from the input and compares it with the on-card clock. When the card is inactive, the clock is stored in the EEPROM on the Javacard.

The Javacard module also implements a lock mechanism. If the incoming time appears faked, the time validator locks all components located on the card. The unlock mechanism can be performed by an authorized administrator.
5.3.3 On-card AES module for Ephemeal Key Pairs

Another role of the Javacard is to store the AES key, encrypt, and decrypt the private ephemeral key. For the sake of the persistent protection, the AES key is written into the EEPROM of the Javacard. In contrast to RAM, EEPROM does not lose its value in case of a power loss. As keys cannot be derived out of the card, the security of the keys can be guaranteed. The AES encryption and decryption operations are carried out in the Javacard which avoids the exposure of the AES key in the memory. The AES engine is involved to encrypt/decrypt the sensitive private ephemeral key along with its expiration date. As depicted in the Figure 5.7, the bit block comprises of the private ephemeral key and the corresponding expiration date.

5.3.4 RSA Generator

Usually, a Javacard integrated with a special encryption algorithm processor which supports a variety of cryptography algorithms. By utilizing the cryptographic supports of the Javacard, the complexity of the RSA key generation is reduced. The RSA keys are represented as its modulus and exponent.

5.4 Card Accessor

The card accessor module acts as an access interface, which allows the Webservice Handler to communicate with the Javacard. The problem arises when a malicious card accessor accesses the Javacard. More gener-
ally, an unauthorized host may use the Javacard. Because of this, we have to take this threat into account. In such scenario, we adopt PIN mechanisms to protect the Javacard module.

5.5 Webservice Handler

The Webservice Handler works as the intermediator of the Ephemizer Service and service consumers. It presents a uniform interface to consumers, and it is responsible for regulating functionalities between Javacard and the key storage database.

The service consumers should be capable of dynamically and asynchronously entering the Ephemizer Service. We do not want to develop new interfaces solely for the new coming service consumers. The webservice interfaces to the Ephemizer Service with standard internet protocols. For the information to be transmitted through the network, it must be packaged in the format conforming to the Simple Object Access Protocol (SOAP). SOAP supports information exchanges by specifying a way to structure XML messages.

5.6 Mutual Authenticaiton Between Ephemizer Service and Administrator

Identification and access control are critical for the security of the administrations. Typically, identifications are following 3 categories: password based (What you know), token based (What you have) and biometric
5. System Design

![Diagram of Mutual Authentication]

**Figure 5.8. Mutual Authentication**

based (Who you are). In our approach, we adopt double factor identification, which is the combination of both the password based and the token based factors. Password is taken as the factor what you know. The token is represented as the private key of the administrator; it is taken as the factor what you possess. The administrator’s private key can be stored on a smartcard for security reasons. The **WEBSERVICE HANDLER** takes care about the security of the **EPHEMERIZER SERVICE** by intercepting each administrator and by checking his conformance to the security policy, taking into account the access rights granted to administrator. Both the administrator of the **EPHEMERIZER SERVICE** and the **WEBSERVICE HANDLER** perform a mutual authentication process. The scheme is depicted with reference to Figure 5.8. This shows a secure session, which indicates the agreement of the cryptographic algorithm and shared secret keys. The authentication of the administrator is performed with the help of the Javacard that veri-
fies his associated private key. Unlike the one-to-one relationship between the \texttt{EPHEMERIZER SERVICE} and the administrator, our approach assumes that the \texttt{EPHEMERIZER SERVICE} has a relationship with multiple administrators. In this approach, we avoid the use of a long time shared secrets. We thus adopt the asymmetric cryptographic solution and make full use of our \texttt{EPHEMERIZER SYSTEM} key management proposal.

The \texttt{EPHEMERIZER SERVICE} generates a pair of keys, $K_{\text{admin}}^+$ and $K_{\text{admin}}^-$, for the authentication of the administrator. The generation and deposition of the key pair are the same as ephemeral keys. The administrator obtains certificates for his respective private key $K_{\text{admin}}^+$ and the public key of \texttt{EPHEMERIZER SERVICE} $K_{\text{eph}}^+$. The associated trust issues are outside the scope of this thesis. At that point, the administrator and the \texttt{EPHEMERIZER SERVICE} know each other's public key.

The \texttt{EPHEMERIZER SERVICE} retains the state information of authorized administrator to complete the authentication protocol. If the request is not proved to be from the administrator, the authentication procedure terminates at the side of the \texttt{EPHEMERIZER SERVICE}. This state information is two random session variants which are encrypted with the participants' public keys respectively. Both the administrator and the \texttt{EPHEMERIZER SERVICE} assure the freshness of the state information through these two random session variants which are denoted in the Figure 5.8 as $R_1$ and $R_2$. During the authentication process, the \texttt{EPHEMERIZER SERVICE} maintains and refreshes the state information stored in the database. The \texttt{EPHEMERIZER SERVICE} and the administrator decrypt the session variants with their own private key, and send them back for verification. When the $K_{\text{admin}}^+$ is in
need of admin's authentication, the key $K^{+}_{\text{admin}}$ is retrieved from key storage database, then it is expiration verified and decrypted on the Javacard module.
Chapter 6

Implementation

6.1 Webservice Handler

Web Service is a service-orientated architecture incorporating the entities of Service Provider, Service Consumer, and Service Register. We use this simple communication model to encapsulate our service messages in SOAP over http. The Webservice Handler is developed as Web Service in Axis (Apache Extensible Interaction System). Axis is an Apache open source project and a framework for constructing SOAP processors. Apache Axis is developed on top of the Jakarta Tomcat application server[18]. Those services are implemented in the J2EE environment.

The Webservice Handler is packaged as a variety of class files and deployed on the Tomcat Server. We use the JAX-RPC model together with tools from Apache Axis and Ants to create service WSDL, and deploy it on the Tomcat Server.
6. Implementation

<table>
<thead>
<tr>
<th>SOAP</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:base64Binary</td>
<td>byte[]</td>
</tr>
<tr>
<td>xsd:byte</td>
<td>byte</td>
</tr>
<tr>
<td>xsd:string</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

Table 6.1. Data Mapping

6.1.1 Data Mapping

As shown in the Table 6.1, the basic mapping between Java types and SOAP in Axis is determined by the JAX-RPC specification.

Here we give an example to show how the mapping works.

```java
public byte[] getPubExp(String id)
{
    byte[] result = null;
    try
    {
        Call call = (Call)service.createCall();
        call.setTargetEndpointAddress(new URL(endpoint));

        //set the name of the function that will be invoked
        call.setOperationName(new QName("getPublicExponent"));
        call.addParameter("id",
              javax.xml.rpc.encoding.XMLType.XSD_STRING,
              ParameterMode.IN);
        call.setReturnType(XMLType.XSD_BASE64);
        Object o = call.invoke(new Object[]{id});
        result = (byte[])o;
    }
    catch(Exception ex)
    {
        ex.printStackTrace();
    }
    return result;
}
```

Axis puts explicit typing information in the method parameter to make the invocation self-describing. We also need a description of the service that tells us what to expect as the return type. Axis will deserialize XML
into Java objects which can be fed into our service, and will serialize the service object back to XML.

6.1.2 Deployment

Axis Web Service Deployment Descriptor file provides a quick way to publish our classes as Web Service. It contains the information that we want to deploy into Axis.

Listing 6.1. deploy.wsdd

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/
xmlns:Java="http://xml.apache.org/axis/wsdd/providers/Java">
<service name="CardServiceRequestor" provider="Java:RPC">
  <parameter name="scope" value="Application"/>
  <parameter name="className" value="server.webservices.
                       CardServiceRequestor"/>
  <parameter name="allowedMethods" value="genRSAKeys
                                         getPublicExponent
                                         getPublicModulus
                                         decrypt
                                         adminRequest
                                         adminOperation"/>
</service>
</deployment>
```

Axis supports "scoping" the service object in different ways. "Application" scope, which is shown in the 4th line of Listing 6.1, will create a singleton shared object to all requests. It fits our needs, because one card can be selected once at the same time. This is due to the fact that the Javacard is a mono-threaded platform that can only serve one request during the execution of an applet. While an APDU command is being processed and before its APDU response is sent back, the card cannot queue any other APDU requests.
6. Implementation

6.1.3 Ant for **WEBSERVICE HANDLER**

Apache Ant is a Java-based build tool. It extends Java classes. Instead of writing shell commands, the configuration files are XML-based, they call out a target tree where various tasks get executed. Each task is run by an object that implements a particular task interface.

The following code automatically builds the "deploy" process.

```xml
<target name="deploy" depends="init,compile" description="deploy web services">
  <echo message="deploy......."/>
  <exec dir="${build.dir}/classes/server/webservices" executable="Java">
    <arg line="org.apache.axis.client.AdminClient deploy.wsdd"/>
  </exec>
</target>
```

6.1.4 Database Layer

The database layer is responsible for communication with the key storage database. This database layer implementation is developed as a database component. This design helps in adapting the EPHEMERIZER SYSTEM to various Database Management System by adopting different drivers.

We choose SQLserver 2000 as our database demonstration. List 6.2 gives an example.

**Listing 6.2.** DatabaseUtil.Java

```java
private void initialConnectionParam()
{
  strDriver = "com.microsoft.jdbc.sqlserver.SQLServerDriver";
  strCon = "jdbc:microsoft:sqlserver://127.0.0.1;user=sa;password=sasa;DatabaseName=eph";
}
private void connect()
```
6.1.5 Card Accessor

We use a subclass of CardAccessor which is a existing class in Javacard API to perform the WEBSERVICE HANDLER authentication and to sign every message for integrity purpose. It overrides the exchangeAPDU() method declared in a super class to sign each message before it is sent out.

Parts of the code are from Sun’s Javacard development kits examples.

Listing 6.3. CardAccessor

```java
public boolean authenticateUser( short ID )
{
    byte[] externalAuthCommand = new byte[8];
    externalAuthCommand[0] = CLA_AUTH;
    externalAuthCommand[1] = INS_AUTH;
    externalAuthCommand[4] = 2; // Lc
    externalAuthCommand[5] = (byte)(ID>>8);
    externalAuthCommand[6] = (byte)ID;
    externalAuthCommand[7] = 0x7F;

    try
    {
        byte[] response = this.exchangeAPDU( externalAuthCommand );
        if(response[response.length-2] != (byte)0x90
            || response[response.length-1] != 0x00)
            return false;
        else return true;
    }
    catch (Exception e)
    {
        return false;
    }
}
```
6.2 Javacard

The Javacard environment shares the basic architecture with standard Java environment. The Javacard does not support all primitive types, for instance, integer cannot be coded into the card runtime environment. Because Javacard runtime environment does not allow dynamic download of Java class, a converter is used to package classes into one image executable file, and to reduce its size by prelinking it for the execution on the card as far as possible[17]. Sun’s Javacard Development Kits offer tools to compile and convert Javacard applets into .cap files. Cap file is sent to the card reader that establishes a secure session with Javacard, and downloads the Javacard application in a series of APDU commands. Once the applet is downloaded, appropriate APDU commands are transmitted for applet instantiation and installation.

6.2.1 Javacard Applets Installation

Figure 6.1 depicts the processes of the Javacard applet installation, and how different components of the installer interact with other parts of Javacard technology. The dotted line encloses the installer components.

The off-card installer is called “scriptgen”. The on-card installer is simply called “installer”.

The Converter processes Java class files, depending on the command line options, it outputs a CAP file, a Javacard Assembly file, and an export file.

The CAP file is a JAR-format file and represents the executable binary for-
mat of the classes in a Java package. The CAP file also contains a manifest file that provides human-readable information regarding the CAP file.

The scriptgen tool converts a CAP file into an APDU script file. The script file contains a sequence of APDUs in ASCII format suitable for another tool, such as apdutool, to send to the CAD.

The apdutool reads an APDU script file and sends the APDU commands to the C-language Javacard Runtime Environment (cref). Each APDU is processed and returned to the apdutool, which displays both the command and response APDUs on the console.

### 6.2.1.1 APDU script modification

Because the CAP file is downloaded and applet creation is postponed until a later time, these steps should be followed to perform the installation:
6. Implementation

Use scriptgen to convert a CAP file to an APDU script file. Prepend following snippets to the APDU script file:

Listing 6.4. header.scr

```plaintext
powerup;
```

Append these commands to the APDU script file:

Listing 6.5. rooter.scr

```plaintext
// Select the installer applet
0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00
0x00 0x62 0x03 0x01 0x08 0x01 0x7F;

// create Eph
0x80 0xB8 0x00 0x00 0x0c 0x0a 0xa0 0x00 0x00
0x00 0x62 0x03 0x01 0x0c 0x8 0x01 0x00 0x7F;

powerdown;
```

6.2.1.2 Ant for Javacard

We provide the following Ant configuration snippets for automatic applets installation.

Listing 6.6. applets installation

```xml
<target name="deploy" depends="init,compile" description="deploy web services">
  <!-- Definitions for tasks of javacard tools -->
  <taskdef name="convert" classname="com.sun.javacard.ant.tasks.ConverterTask"
    classpath="${ant_lib}/jctasks.jar"/>
  <target name="convert_eph" depends="init,compile"
    description="Build export files and CAP files">
    <convert
      CAP="true" JCA="true"
      packagename="uni.liu.ida.adit"
      packageaid="0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x08"
      majorminversion="1.0"
      classdir="${build.dir}\classes"
      outputdirectory="${card_relevant}"
```
<AppletNameAID
    appletname="uni.liu.ida.adit.EphApplet"
    aid="0xa0:0x0:0x0:0x62:0x3:0x1:0xc:0x8:0x1" />
    <exportpath refid="export"/>
    <classpath refid="classpath"/>
</convert>
</target>
<taskdef name="scriptgen"
    classname="com.sun.{\jc}.ant.tasks.ScriptgenTask"
    classpath="${ant_lib}/jctasks.jar"/>
<target name="scriptgen_eph" depends="convert_eph" >
    <scriptgen
        noBeginEnd="false"
        noBanner="true"
        CapFile="${card_relevant}\uni\liu\ida\adit{\jc}\adit.cap"
        outFile="${card_relevant}\scr\adit.scr" >
        <classpath refid="classpath" />
    </scriptgen >
    <echo message="backup adit.scr..">
    <concat destfile="${card_relevant}/scr/adit_backup.scr">
    <fileset file="${card_relevant}/scr/adit.scr"
    <concat>
    <echo message="concate...">
    <concat destfile="${card_relevant}/scr/adit.scr">
    <fileset file="${card_relevant}/header.scr"
    <fileset file="${card_relevant}/scr/adit_backup.scr">
    <fileset file="${card_relevant}/rooter.scr"/>
    </concat>
</target>
<!--only generate rom mask!-->
<target name="GenRom" depends="scriptgen_eph">
    <echo message="cref...">
    <exec dir="${card_relevant}/scr/" executable="cref.exe" spawn="true">
    <arg line="-o demoee"/>
    </exec>
    <echo message="apdu...">
    <exec dir="${card_relevant}/scr/" executable="apdutool.bat">
    <arg line="-nobanner -noatr adit.scr > adit.scr.cref.out"/>
    </exec>
    <echo message="tskill...">
    <exec executable="tskill">
    <arg line="cref"/>
    </exec>
</target>
6.2.2 Javacard Applets

In the Javacard technology each Javacard application is identified and selected by an Application Identifier (AID). Multiple applet cabs reside on one card. An AID is a sequence of bytes between 5 and 16 bytes in length. A Javacard application’s lifetime begins at the point when it is selected by corresponding APDU commands.

The applet must register itself to JCRE to be selectable. The Javacard Runtime Environment initiates the remote object on the card via the applet’s public install method, where the applet constructor is invoked and registered. The main task of the install method is to create and register an instance of the Applet subclass at the runtime environment. All other objects that the applet will need during its lifetime can be created as feasible.

Listing 6.7. EphApplet.Java

```
public class EphApplet extends javacard.framework.Applet {

    ...

    public EphApplet()
    {
        ...
        register();
    }

    public static void install(byte[] aid, short s, byte b)
    {
        new EphApplet();
    }

    ...

}
```

A typical Javacard RMI remote object interface extends the Java.rmi.Remote interface. The interface defines a remotely accessible object.

Listing 6.8. Eph.Java
6.2.2.1 Applets dispatcher

In a typical message passing model, all APDU's are received by the JCRE, which passes a instance of the APDU class to the process method of current selected applet. The runtime environment initiates the applet installation by calling the install() method. The select command will be forwarded to the applet's select() method by the runtime environment, each following command will be forwarded to its process() method.

In our JCRMI model, a dispatcher, which is indicated in the 8th line of Listing 6.9, must be initialized. It simply dispatches simple Javacard RMI requests and security related requests. The dispatcher can delegate commands to corresponding services.

Listing 6.9. EphApplet.Java

```java
public interface Eph extends Remote {
}
```

```java
public class EphApplet extends javacard.framework.Applet {
    public EphApplet() {
        sec = new EphSecurityService();
        eph = new EphImpl(sec);
        disp = new Dispatcher((short) 4);
        serv = new RMIService(eph);
        disp.addService(serv, Dispatcher.PROCESS_COMMAND);
        disp.addService(serv, Dispatcher.PROCESS_INPUT_DATA);
        disp.addService(serv, Dispatcher.PROCESS_COMMAND);
        disp.addService(serv, Dispatcher.PROCESS_OUTPUT_DATA);
        register();
    }
    public void process(APDU apdu) throws ISOException {
```
6. Implementation

6.2.2.2 Applets stub-skeleton

To simplify the calls, we have built a Stub/Skeleton mechanism. In our JCRMI client-side scenario, we use RMIC-generated stubs for remote objects. RMIC is the Java RMI stub compiler.

Listing 6.10. RMIC invocation

```xml
<target name="run-rmic" depends="init,compile" description="Run RMIC.">
  <echo>rmic "uni.liu.ida.adit.EphImpl" ....</echo>
  <rmic base="E:\My Documents\Java\Java projects\Javacard\Eph\build\classes"
    classname="uni.liu.ida.adit.EphImpl" stubversion="1.2">
    <classpath refid="classpath"/>
  </rmic>
</target>
```

List 6.10 gives an ANT script solution to build the stub.

6.2.2.3 Object deletion mechanism

EEPROM provides persistence storage. There is a risk of failure at any time with Javacard. Failure can easily happen due to a removal of the card by error. Because a Javacard does not contain a integrated power, writing operations are guaranteed to succeed in case of sudden power loss. Persistence data such as the AES key is located in EEPROM, because these data still have to be kept in EEPROM after power is removed. EEPROM is extremely limited, typically ranging up to 16KBytes, in addition, Javacards...
often have no garbage collection implemented, the object deletion mechanism is involved.

The object deletion mechanism reclaims memory that is being used by "unreachable" objects. It is performed only when an applet requests.

**Listing 6.11. Object Deletion Mechanism**

```java
1. this.outputBuffer = null;
2. JCSystem.requestObjectDeletion();
```

RAM is used for temporary data and transient objects. It is the most expensive memory and requires the most space per bit. We tried to avoid creating objects in method or block scope. In addition, when a large object, such as large arrays or keys, must be replaced with a new one, the object deletion mechanism is used.

### 6.2.2.4 Transmission of data segments

Because each APDU is limited in size, data and objects, such as RSA ephemeral keys, are transferred to and from the card with a series of commands. In addition, in Javacard RMI, the arguments to and return values from remote methods are restricted. Javacard platform does not support object serialization. It means we need build data segmentation mechanism.

**Listing 6.12. Data segmentation**

```java
1. int length = eph.getOutputBufferLength();
2. result = new byte[length];
3. int times = length / this.MAX_BUFFER;
4. times += 1;
5. for(int i=0; i<times; i++)
6. {
7.    byte[] seg = eph.getOutputBuffer((short)((short)i*(short)this.MAX_BUFFER),
8.      (short)this.MAX_BUFFER, Util.getCurrentTime());
```
6. Implementation

```java
for (int j = 0; j < seg.length; j++)
{
    result[i * this.MAX_BUFFER + j] = seg[j];
}
```
Chapter 7

Experience

We designed the Ephemizer Service with security as one of its building blocks. The project was to develop a robust and scalable design, and implementation of the Ephemizer Service. We use the concept of the Trusted Computing Base to protect important functionality in the Ephemizer Service. We chose the Javacard platform as our candidate for its implementation.

7.1 Database

The choice which system components should be excluded from the TCB depends on design strategies and the limitations of the implementation device, in our case the Javacard platform. If everything must be inside it, then a TCB is no more useful a concept for organizing the design of a secure system than the universe is a useful concept for organizing matter[7]. In the first version of our design, we tried to place key storage and management modules into the TCB. However, this approach was impractical...
because of the limited computing power and the limited memory capacity of the Javacard. In addition, we found that the key storage should be implemented by a database management system to improve the scalability and the availability of the Ephemizer System. In the absence of the backup of key storage, the system suffers from a possible Denial of Service attack by destroying all valid ephemeral keys. Without a backup, they could never be recovered in this situation, although this issue can be partly solved by a thresholding solution, which is introduced by Radia Perlman, the scalability of the system is degraded. The thresholding solution is to break the secret into n pieces, such that any k pieces can recover the secret, and to give each of the n pieces to each of n ephemizers. Thresholding does not solve the problem completely; yet it reduces the risk of Denial of Service Attack.

However, our solution introduces new challenges. The database system plays an important role in our design. In our model, sensitive ephemeral key storage can be stored on a remote server. Because most attacks are targeted at applications and servers, the privacy and integrity of ephemeral keys may be compromised. Encryption is a proper technique for protecting sensitive key storage. Data encryption can be done in either the database-layer or the application-layer. Database-layer encrypts data inside a DBMS. We explored the other alternative. In application-layer encryption, data is encrypted and decrypted at client-side instead of the database server. Consequently, we encrypt the sensitive key on the Javacard and store the already encrypted key on a database server.
7.2 Compartmentalized Approach

One primary goal of database technologies is to provide shared access to its contents, while the primary objective of encryption is privacy[20]. This shared privacy should also be protected by the underlying system to make ephemeral keys more trustworthy. This would require some additional layer of software or even hardware on the Ephemeral Service. While this additional layer raises the resilience to attacks, access control should restrict its configuration even from its administrator. Thus we propose a compartmentalized approach by delegating the security task of ephemeral key to the underlying OS.

The partly trusted protection domain requires the database to run in conformance to authorized operations which are identified by access-control decisions. The authorized operation provides much stronger protection of personal and private data on personal computers[16]. There should be mechanisms that protect the database against other softwares running on the Ephemeral Service. The mechanisms should guarantee that viruses, and other unknown applications cannot access the database and send results to remote recipients. Malicious operations performed by administrators must be prevented in the partly trusted protection domain. A trusted communication channel between administrators and the database is a prerequisite for the secure communication. For instance, the key stroke of a database administrator to the database should not be overheard.

The partly trusted protection domain requires isolations between applications. Otherwise, the compromising of any module on the Webservice
7. Experience

Handle compromises the entire Ephemizer Service, because the platform’s security level is reduced to its most vulnerable application[13]. During the running period of the message decrypter, the isolation of its memory prevents the ephemeral keys from being revealed to other malicious applications such as Trojan virus and memory dumping applications. The partly trusted protection domain may require a special hardware to provide tamper-resistant features for data in memory. It makes sure that data in memory cannot be modified by malicious attacks to cause misbehaviours of the Ephemizer Service. With the help of such protections, the running state of the message decrypter can be intact.

In the partly trusted protection domain, it is desirable to deploy an Intrusion Detection/Prevention System to detect failures of the Ephemizer Service. Intrusion Detection/Prevention System should not prevent administrators’ operations with false hit warnings and ignore the real threats.

Another issue is about ephemeral key expiration validator. In our second design, the validator is excluded from the TCB, because the Javacard has no clock itself. Compromising the expiration validator creates the opportunities for an attack on the Ephemizer System. Thus we place the validator inside TCB.

7.3 Javacard

Building a simple Trusted Computing Base (TCB) on top of an operating systems that contain millions of lines of code is difficult because the TCB ultimately depends on the functionality offered by the underlying operating
Cryptographic keys contain the most sensitive information. Traditionally keys are generated and managed in applications running in the Operating System. The weakness in this approach is that keys can be recovered by dumping the memory of an application and inspecting its contents. Instead, we generate the keys within the Javacard. This Javacard-based EPHEMERIZER SYSTEM is to enhance the security of the ephemeral key management. In the original design, a compromised EPHEMERIZER SERVICE can expose the client’s ephemeral key to the risk that an attacker can make use of it and recover the message.

To ensure that the ephemeral key generator is not exposed to an untrusted module, the ephemeral keys will be generated on the Javacard. The Javacard is preloaded with an AES secret key and an RSA private key. The Javacard has an embedded, non-volatile memory and a random access memory. They make it difficult to physically extract these preloaded keys from the Javacard. The AES secret key is used to encrypt and decrypt the ephemeral keys generated on the Javacard. The digital signature of a public ephemeral key is generated by the EPHEMERIZER SERVICE’s RSA private key on the Javacard using the SHA message digest as a "digital fingerprint" that verifies that the public ephemeral key comes from the EPHEMERIZER SERVICE and has not been tampered with during transmission. By making use of RSA signature engine on the Javacard, a malicious ephemeralizer cannot generate an ephemeral key pair in software under the premise that the key pair was generated on the Javacard, because all public ephemeral keys
need to be signed on the card. All encryption and decryption of ephemeral keys using the secret AES module are done within the Javacard. The Javacard employs the secure Javacard Runtime Environment that is capable of prohibiting the exposure of the preloaded keys while permitting the keys to encrypt and sign ephemeral keys. When the Decryption Service Consumer requests operations of deciphering data, the Webservice Handler must decrypt the private ephemeral key and verify the key's expiration date.

Having no trusted source of time is a basic limitation of the Javacard technology[21]. In addition, the Javacard does not provide modules to track and handle time. We adopt behavior-based methods to detect time-based attacks. Behavior-based security is a group of techniques used to monitor the activity of system to identify abnormal behavior and possible attacks in progress[21]. Because of the expiration verification module on the Javacard, time must be provided to the Javacard from an external source, the Webservice Handler. To thwart an attempt of introducing a fraudulent time, implementation of an on-card behavior-based clock verification mechanism is necessary. The countermeasure is to track time and withhold the Javacard services if an activity violation is detected. Our approach is software-based and tracks the last known time. Once the latest incoming time violates the last known timestamp, the Javacard locks itself and refuses any service. Only the authorized administrator can unlock the card.

The Javacard offers a storage protected by a personal identification number (PIN). By providing a PIN, an ephemeralizer can prove what you know. Javacard incorporates the PIN mechanism to guard against unauthorized
access to the secret AES key and to key management modules. To impersonate an ephemerizer, an adversary must steal both Javacard, and its corresponding PIN.

Conventional link speed between a Javacard and its card reader is 10kbps. The link may be a burden for the entire system. The storage capacity and computing power of a Javacard are limited, but the Javacard provides higher security than an ordinary PC platform. Javacard contains a cryptographic coprocessor to execute cryptographic computations in an efficient way. Through time, more computing power and larger memory storage capacity can be integrated into the card. The Javacard is the core processing unit of the whole system.

There are several methods to physically tamper with devices that are labeled 'tamper-proof', which show that breaking into a 'tamper-proof' device, such as a Javacard, is only a matter of knowledge, technical equipment, and time[12].

7.4 Webservices

In our original design, we adopted socket network connections as a communication substrate. However, it requires the Ephemizer Service to listen on certain bound sockets. In addition, this model complicates the communication among the Encryption Service Consumer, the Decryption Service Consumer, and the Ephemizer Service. Scalability is one of the major design considerations. In the Ephemizer System, participants interact by exchanging messages. The message or request sender
7. Experience

opens a connection to a remote port, posts messages, and receives responses. As the user base grows, connections to the Ephemizer Service will continue growing at an accelerating speed. However, the available connection ports are limited.

Web services provide an easy way to wrap the Ephemizer Service service and expose it through a uniform and widely accessible interface over the Internet. It replaces traditional socket-to-socket interfaces with a universal API which provides access to service clients. Moreover, due to intelligent adaptability of the integrated system, addition of a new service or upgrading of an existing service can expected to be seamless and effortless [22]. By incorporating web services, we can easily add new interfaces and publish them outside.

Web services also present the chance to integrate a Ephemizer Service to different business processes. We even don’t have to convert our protocol and data formats so that our system can adapt to other heterogeneous platform. We can also publish our policies to let service consumers or client applications conform to the Ephemizer System.

7.5 Message Decrypter

The message decrypter contains the main vulnerability of the Ephemizer System. Although there are underlying software and hardware protections, it is not secure enough. If someone makes use of this potential vulnerability and uploads a code to the Ephemizer System so that it is executed. He may dump the memory to work out what the Webservice
HANDLER is currently executing or even track running instructions. Hence an attacker would be able to uncover the private ephemeral key.
Chapter 8

Conclusion and Future Research

This thesis describes the system design and implementation of the Ephemerizer System that was first introduced by Radia Perlman in 2005. The system is designed to enable users to keep data accessible for a finite period of time before making the data unrecoverable by destroying the keys with which the data was encrypted. The task of the Ephemerizer System is to create, advertise, and destroy keys required for the Ephemerizer System’s functionality.

Radia Perlman proposed the original design of the Ephemerizer. However, her paper assumes the Ephemerizer is trusted. Contrary to Radia Perlman’s approach[5] where the Ephemerizer is built on a general PC and limited protection is provided, we designed our Ephemerizer Service’s security by placing sensitive key management modules into a Trusted Computing Base on a special hardware (Javacard) to enforce security. Moreover, we implemented a compartmentalized approach to distribute security
requirements to different domains. The ephemeral key database and message decrypter module reside on the partly trusted domain, which resorts to the underlying software and special hardware to provide secure mechanisms. The partly trusted domain requires memory isolation and other security mechanisms such that sensitive information will not be disclosed and tampered with.

Our approach solves the problem of the ephemeral key storage, generation and expiration validation, and makes the key management more secure. We designed the EPHEMERIZER SYSTEM security by placing the sensitive key management module on the Javacard by which it could resist possible attacks. Inside the card, data is secure but it is not the case outside the card. The message decryption module, residing on the partly trusted protection domain, is protected by the underlying operating system. If the partly trusted protection domain is not configured correctly, it is a potential vulnerability. We do not place it into the TCB because it would degrade the performance of EPHEMERIZER SERVICE.

We make the EPHEMERIZER SYSTEM available to users as a web service and expose it though a uniform API. This approach enables the seamless integration of the EPHEMERIZER SYSTEM into business processes on heterogeneous platforms.

### 8.1 Limitations

The performance of our platform cannot satisfy large scale business usage due to limitations of the current Javacard technology. The memory
capacity, computing power and transmission rate are limited. These limitations become a burden for the entire system.

An interaction with a Javacard is comprised of a basic pair of Application Protocol Data Units (APDUs), one request APDU followed by another. It is not possible to provide transparent scheduling of access to a Javacard, because it is unknown what state was established by one card client, and how to reestablish that state, after another client has been using the card in between\cite{24}. This characteristic results in Denial of Service attacks. A card client can block the Javacard if the client does not release the connection to the Javacard.

The Javacard is a single-threaded platform. While an APDU is processed on the card, no more commands can be accepted. This design creates the problem that while the Javacard is generating ephemeral keys or deciphering ephemeral keys, the \textsc{Ephemizer Service} cannot initiate further requests from the card until current on-card task is finished.

A Javacard is a passive device to read or write data. A storage device has this passive characteristic. It does not initiate requests to other devices. The card always waits for a command APDU from the card reader, and it never initiate actions with the outside.

The Javacard has no source of time and no dedicated modules to handle the time within the card. When the \textsc{Ephemizer Service} decrypts the ephemeral key, an additional time clock is required by the Javacard. Thus modification of the time can produce different results.

The Javacard itself has no power. It must depend on the power sup-
plied by the card reader. It induces several power-based attacks such as a side-channel attack in which characteristic power consumption is analyzed. Power analysis attacks exploit the fact that the instantaneous power consumption of the circuit depends on the data being processed by the circuit[26].

### 8.1.1 Enhanced TCB Platform

We can design or choose a more powerful and secure hardware to implement our TCB. The performance and security of the **EPHEMERIZER SERVICE** can thus be greatly improved. The hardware should be a tamper-resistant platform to prevent from the leakage of running statement and stored data. By involving more powerful cryptographic processor, fast transmission channel, and multi-threading scheme, the performance of the **EPHEMERIZER SERVICE** can be greatly enhanced. A powerful cryptographic processor can shorten the key generation time, and increase the speed of encryption and decryption. A multi-threading capable platform enables the system processes requests concurrently. This improved performance enables the embedded integration of the message decrypter into this hardware. The clock should also be integrated into this specifically designed hardware and thus the **EPHEMERIZER SERVICE** can be more secure.

With a larger integrated storage, we can even place the ephemeral key storage into this platform. This design improves the security of the ephemeral key storage. As the ephemeral key cannot be derived out of the hardware without access permissions, the attack risk is reduced.
8.1.2 Reasons for Choice of the Javacard as TCB

A Javacard is a tamper-resistant hardware, it provides protections of data at both the hardware and software level.

At the hardware level, the Javacard has a temper-resistant property which makes it difficult for an attacker to physically extract data from the card. Tamper-resistant refers to a device’s ability to successfully overcome any attempt to influence the device in such a way that data which should be kept inside the device becomes visible outside[12]. Our key idea is to place our sensitive modules and data on a Javacard, and to invoke functions on that card. In this way, the sensitive modules such as the ephemeral key generator are secured even if the server, where the Webservice Handler resides, is compromised. Compared with a Javacard, a computer hard disk or an external storage media are considered less secure for abilities of the protection. It is easier for hackers and viruses to compromise such devices, which results in possible data leakages especially damaging of ephemeral keys. When an applet is loaded into a Javacard, it can neither damage the card nor access any protected areas that it contains, nor can it be reverse engineered by the possessor of the card[19].

At the software level, a Javacard uses an applet firewall to prevent individual applet from accessing contents of objects owned by other applets. The mechanism creates a secure environment. The communication to the applet is under control of a host Operating System. One can download multiple applets into the card. The Javacard uses the Java sandbox mechanism that acts as firewalls between different on-card applications. The fire-
8. Conclusion and Future Research

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Key</td>
<td>0.01135s</td>
</tr>
<tr>
<td>Read Key</td>
<td>0.01074s</td>
</tr>
<tr>
<td>Generate RSA Keys</td>
<td>1.91205s</td>
</tr>
<tr>
<td>AES-128 Encryption</td>
<td>85us</td>
</tr>
<tr>
<td>RSA-1024 Signature</td>
<td>79ms</td>
</tr>
<tr>
<td>DES</td>
<td>8us</td>
</tr>
</tbody>
</table>

**Table 8.1. Speed Test of Javacard**

wall mechanism prevents one card applet from tampering with the data or code of another applet. In addition, because security modules and critical data are contained in the Javacard, it is important that the Javacard itself should be able to recognize its associated Webservice Handler. We use a PIN code verification process which binds the Javacard to its designated Webservice Handler.

The trend in building a secure platform is to design and build secure hardware and then to add a secure operating system on top of it. Ideally, such a design is based on a theoretical framework[25]. It becomes more difficult to provide tamper-proof hardware the larger the hardware gets[12]. Building a simple and secure platform on top of a commodity operating system is difficult, because the platform ultimately depends on the functionality offered by the low-level operating system. Moreover, such specially designed hardware may not be readily usable for rapid prototyping. As shown in Table 8.1, the performance of the current Javacard technology satisfies requirements of small businesses. In the small business usage case, there are only few concurrent invocations and the speed of cryptographic functions is tolerable.
Following the methodology curve in the microelectronics industry, more powerful processors, faster memory access speeds and larger memory capacities are expected to be integrated into the Javacard[27]. In the near future more secure cryptographic functions and more robust handshaking mechanisms can be implemented by the embedded software on the Javacard. According to the RESET Roadmap, which is established by the European industry and academic stakeholders on smartcard-related technologies, with faster memory such as floating gate memory, FeRAM and MRAM will be integrated into the Javacard platform in the near future[27]. The fastest speed can achieve an access time as short as 10ns[27]. In order to execute cryptographic functions in an efficient way, a Javacard usually incorporates a cryptographic coprocessor. The coprocessor can greatly speed up cryptographic computations. In fact, most Javacards in the market already contain both RSA and DES crypto-processors to perform the cryptographic computation quickly. With these improvements in the Javacard technologies, the EPHEMERIZER SERVICE speed can increase greatly.

With the Java Virtual Machine implemented on the Javacard, the Javacard platform can provide platform-independent services. Thus we do not have to face hardware compatibility problems. The Javacard platform provides a simplification of software development. It is achieved by incorporating a high-level, standard programming language to the Javacard. The Javacard platform also provides card issuers with the ability of updating on-card functionality dynamically. With the Javacard platform, we can employ well-known object-oriented design methodologies to develop on-card services.
8. Conclusion and Future Research

8.2 Future Research

A component-based design of Javacard functionalities can be applied in the future work. All the services should be built on the Abstract Factory Design Pattern. Component-based design enables the functionalities to be reusable and to be easily configurable. The extensibility of the EPHemerizer Service can be achieved by adding or substituting new Javacard service component to EPHemerizer Service as long as the component adheres to specified interface.

To make the Javacard active to initiate a request to the outside environment solves the on-card time clock problem. For instance, the Javacard could connect via its network interface to request the current time. The Javacard itself would need to be able to initiate requests to the outside environment.
Bibliography


Bibliography


[23] C. E. Ortiz *An Introduction to Java Card Technology*


### Appendix A

**Glossary**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>APDU</td>
<td>Application Protocol Data Unit</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DoS</td>
<td>Denial-of-service</td>
</tr>
<tr>
<td>EEPROM</td>
<td>Electrically Erasable Programmable Read-Only Memory</td>
</tr>
<tr>
<td>HMAC</td>
<td>keyed-Hash Message Authentication Code</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>JCRE</td>
<td>Java Card Runtime Environment</td>
</tr>
<tr>
<td>JCRMI</td>
<td>Java Card Remote Method Invocation</td>
</tr>
<tr>
<td>MAC</td>
<td>Message Authentication Code</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio Frequency Identification</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>TCB</td>
<td>Trusted Computing Base</td>
</tr>
</tbody>
</table>
A. Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDDI</td>
<td>Universal Description Discovery and Integration</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Services Description Language</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>