Design and Implementation of a Cryptographic Plugin for E-mail Clients

Steven Arzt

Technische Universität Darmstadt,
Department of Computer Science,
Hochschulstraße 10, D-64289 Darmstadt, Germany
s_arzt@rbg.informatik.tu-darmstadt.de

Abstract. Most applications that use cryptography implement and use only a limited set of cryptographic algorithms. In addition it is not easy to update these applications and extend their provided set of algorithms. In this paper we concentrate on e-mail clients and discuss a cryptographic plugin with focus on Mozilla Thunderbird. We provide a modular framework for exchangeable signature, hash, and encryption algorithms. We create an integration layer between e-mail applications and cipher implementations. With such a layer, both parts can be exchanged freely and independently of each other which is not possible with current solutions because they mostly rely upon a static set of ciphers and hash functions or directly integrate the ciphers into the application code.

Keywords: Certificates, S/MIME, cryptography, smart cards, e-mail clients.

1 Introduction

Signing e-mails is a common task today. S/MIME specified in [Ram04b] and [Ram04a] is mostly used and integrated in common mail clients like Mozilla Thunderbird or Microsoft Outlook. However, almost all application only provide a standard set of algorithms, for example RSA [RSA78], DSA [NT94], and ECDSA [ANS05]. There is no possibility to use different cryptographic functions like Niederreiter (CFS) [CFS01].

Due to the limited lifetime of cryptographic algorithms and hash functions in general as well as the prospect of quantum algorithms breaking most number-theoretic ciphers and signature schemes in particular, we need alternative methods of cryptography. Especially RSA which is one of the most popular ciphers used in e-mail signature is no longer secure with quantum computers. Therefore an easy to implement mechanism to use strong and innovative cryptography while preserving practical usability has to be developed.

Another motivation is to easily update cryptosystems. If one cipher or hash function has reached the end of its lifetime, it has to be replaced either by a stronger variant or a completely new cryptographic primitive. New cipher developments may also be more efficient while providing the same level of security as

existing ones. In either case, the update has to be done as smoothly as possible. Important systems should not be unavailable due to this maintenance which is however necessary necessary with many current systems. This happens because cryptographic primitives are often hard-coded into application programs or operating systems.

The JCA/JCE framework ([JCA]) can be used for providing base operations like signing, encrypting, and hashing as custom providers. Many existing ciphers are available as JCA providers. However, this still requires a lot of effort when integrating novel ciphers into e-mail systems due to the missing integration into their respective APIs or extensions frameworks.

For the use of smart cards, a communication library is necessary. The input data available on the computer has to be transferred to the card reader, the respective signature function on the smartcard has to be invoked, and the result needs to be transferred back to the application on the computer. While PKCS#11 provides a common standard for basic operations of this kind, many functions of modern cards are missing. Implementors thus provide card and platform specific non-standardized solutions for single applications. Whenever new card types or applications are developed, these libraries need to be separately adapted, tested, and distributed for all combinations of user applications and card models.

Therefore we design and implement an extensible platform for coupling applications and exchangeable cryptographic modules for the different ciphers. As an example for the application interface, we integrate our solution into a widely used e-mail client. With our platform, all new developments of cryptographic algorithms or system changes can either be provided as new modules or incorporated into dedicated existing ones. This does not require changes to existing modules. With the support provided by the platform, the necessary development and maintenance effort is minimized.

This paper is organized as follows. We address the problem we are working on in Section 2 and then give a general overview of our solution in Section 3. In Section 4 we show how the solution is designed in detail for signing mails and managing certificates and private keys. In Section 5 we describe how our software is integrated into the Mozilla Thunderbird e-mail client. Section 6 lists and describes the third-party components we currently use. In Section 7 we give an overview over all implemented signature and key management modules. We conclude our paper in Section 8.

2 The Problem

A modular framework providing exchangeable ciphers for data signature is required that overcomes the restriction to the fixed and limited set of signature algorithms and hash functions implemented into many applications. In this paper, we focus on electronic mail which is one of the most widespread and most frequently used applications. Our concrete implementation is integrated into Mozilla Thunderbird which is a famous and commonly used mail client. Still,

our solution is designed to be extended for other mail clients like Microsoft Outlook, too. All clients providing sufficiently sophisticated extension APIs can be integrated.

Integrating new cryptographic primitives into existing products requires wide system changes as the built-in cipher set has often been regarded more or less static during development. The extension features a growing number of software systems provides help with the task, but they are often too complex for practically testing newly developed signature functions with. This results in a high amount of source code necessary. Additionally, extension or plugin APIs often are not specially designed for cryptography, so the developer has to manually integrate his plugin into the host's control flow. For Mozilla Thunderbird, for example, this means that every such plugin has build its own infrastructure for obtaining the source mail message, integrating the signature, and then inject the result back to the original chain of processing without breaking it even if arbitrary other plugins are used, too. Besides that, an own S/MIME implementation needs to be provided or an existing one to be manually integrated with the plugin in addition to the control flow integration problem stated above.

Implementing new cryptographic algorithms as JCA/JCE providers is a common method of delivering new developments to a wider community of users as almost all Java applications are based on this architecture and can thus use it. Many mail clients, however, are not written in Java, so the additional integration layer described above is again required.

We introduce such a layer with our solution called Thundercrypt. With Thundercrypt, algorithm implementations only need to focus on direct signing of binary data. Our software performs all communication and control flow handling with the mail client.

In corporate or institute networks, software deployment is a concern which makes it more difficult to quickly exchange ciphers no longer meeting the required security levels. Thundercrypt can work as a network service providing a single point of maintenance and key management which additionally minimizes the risk of key compromise.

Many existing frameworks for mail signature rely on specific means of key storage. Microsoft Outlook uses the Windows key store, Mozilla Thunderbird uses its own key store, other systems store keys in the Java keystore. Thundercrypt flexibly combines the cryptographic modules with a configurable set of key store modules reducing the dependance upon one specific system and facilitating the migration between different signature implementations.

3 Design

In this chapter, we first discuss the general design of the Thundercrypt system. Then we show how the configuration infrastructure as the linking part between infrastructure components and freely exchangeable modules works.

3.1 General

Thundercrypt consists of four main parts: The communication services handle the communication between the Thundercrypt application and the mail client, either via a direct integration of the communicator classes into Thunderbird 3 or newer or via a socket-based server solution. The server can be configured for remote or only local access. The second part contains the key management modules. A key management module provides other Thundercrypt modules with X.509 certificates and (if applicable) corresponding private keys through a common interface. There is one module for every key store type. The third part contains the cryptographic modules which perform the signing and decryption operations. There is one module for every algorithm or external cryptographic library to be used. The segregation of key management and cryptographic operations enables an easy combination of different key stores with different cryptographic providers. For instance, an RSA key pair can be obtained either from a file or the Java Key Store and can both be used with native JRE providers and the FlexiProvider [Gro] RSA implementation. The fourth part are infrastructure services available to all other modules. They implement common features like logging and configuration management. This is especially useful when integrating new ciphers or key management modules as they can be built upon an existing, flexible, and consistent infrastructure simplifying module development and management. All modules and parts are loosely coupled so that they can be exchanged without much effort. Key management and cryptographic modules are instantiated by factory classes loading the modules specified in the configuration file. As this method is applied for both built-in and additional modules, adding a new modules can be done by just providing a compatible class and registering its fully qualified class name in the configuration file. Likewise, only the class name in the configuration file needs to be changed for switching modules. The loose coupling also holds true for the communication services part, so a web service interface or interfaces to other mail clients like Microsoft Outlook can be implemented in the future. After developing a new communication module, Thundercrypt's full functionality is instantly available to the new client.

3.2 The modular configuration by example

As described in the general part, Thundercrypt uses a uniform configuration file centrally processed by infrastructure components that make the respective parts available to the different modules. The configuration example in figure 2 uses the TUD smartcard [BF] for signing mails, but loads the user certificate that is embedded into the mail from a file on a local hard disk for enhanced performance.

In this example configuration, the PIN for decrypting the private key is specified in the configuration file for simplified testing. If the respective element is omitted, the user is asked for his PIN or password every time he sends a signed e-mail.
The certificate can also be loaded from the card that is used for signing by just

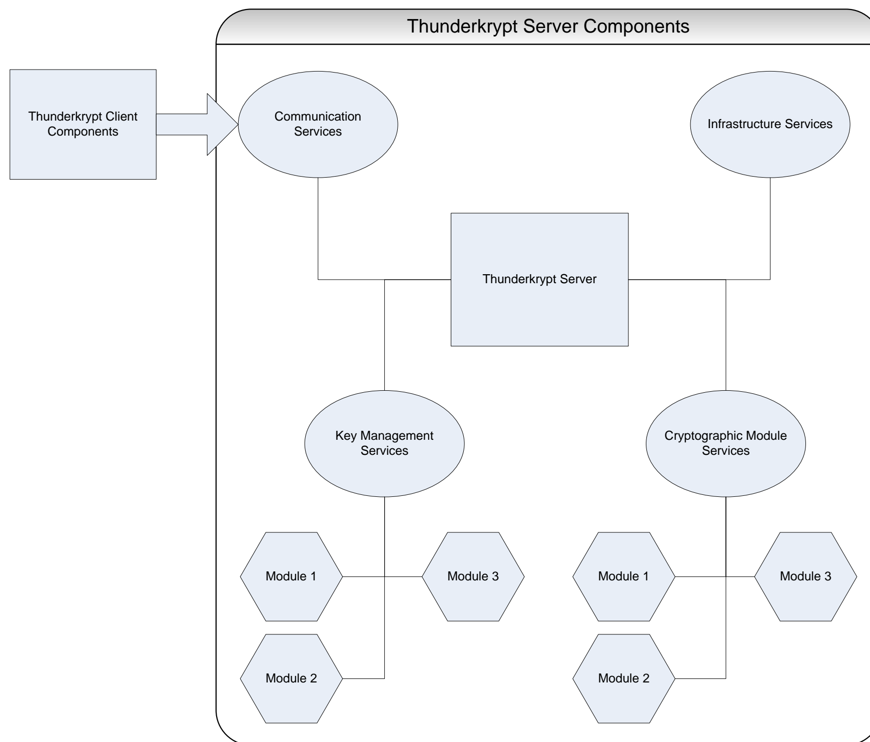

Fig. 1. General software design

exchanging the configured certificate manager. With such a configuration, only the physical card needs to be exchanged in the case of a key compromise for example and the system continues to function without any further changes. On the other hand, reading the certificate from the card is significantly slower than using a file on a hard disk or RAM disk. As a compromise, Thundercrypt also implements a cache for private keys and certificates. With smart cards, the private key cache is deactivated.
The infrastructure components parse the <Configuration> elements and associate them with the correct classes as defined by the "target" attribute. If a class needs access to its configuration data, it can directly access the respective DOM node.

4 Modules by Example

For a better overview over the interface for building cryptographic provider and certificate management modules, this chapter explains one cryptographic provider and one certificate manager as examples.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Thunderkrypt>
  <Configuration target="tud.thunderkrypt.server.ServerConfiguration">
    <MailCryptoProvider>
      tud.thunderkrypt.mailprovider.tudcard.TUDCardMailSignatureProvider
    </MailCryptoProvider>
    <CertificateManager>
      tud.thunderkrypt.certmanager.filebased.FileBasedCertManager
    </CertificateManager>
  </Configuration>
  <Configuration target="tud.thunderkrypt.certmanager.filebased.CertManagerConfiguration">
    <Certificates>
      <Certificate ownCertificate="true" defaultOptions="Signing">
        <CertificateLocation>C:\Certificates\Personal.crt</CertificateLocation>
        <PrivateKeyLocation>C:\Certificates\Personal.p12</PrivateKeyLocation>
        <PrivateKeyPin>123456</PrivateKeyPin>
      </Certificate>
    </Certificates>
  </Configuration>
</Thunderkrypt>
```

Fig. 2. Configuration example

4.1 The Java native cryptographic provider

As an example for a cryptographic module, we take the Java native cryptographic provider. The Java native provider uses the basic JCA functions for creating RSA signatures and is thus relatively simple.
Modules are implemented as classes extending the *AbstractMailProvider* class. Signature and management operations are implemented as abstract method overwrites which enables the Thundercrypt infrastructure to directly integrate the new module.
As this basic class directly works on JavaMail's MimeMesssage objects, implementors have to provide their own S/MIME implementation. While this option offers great flexibility when required, it leads to unnecessary complexity in most cases. Therefore, Thundercrypt provides the *AbstractMailProviderSPI* class implementing the S/MIME features and redirecting all signature requests to new abstract methods working on byte arrays instead of mail messages. Modules thus only need to be capable of signing arbitrary byte data for the use with Thundercrypt. The UML diagram in Figure 3 shows the abstract methods defined by the *AbstractMailProvider* abstract base class and implemented by the Java native provider in class *JavaMailSignatureProvider*. As most of these methods just return provider-specific, but fixed management information (like algorithm names), a simple provider like this one is implemented with minimal development effort. Certificate and private key handling is completely used as provided by the certificate manager infrastructure. As the modules are intended to be interface modules for independently developed cipher implementations, most steps in the signature creation process can just be delegated to this implementation. The provider's main responsibility is converting input and output data between Thundercrypt's module interface and the arbitrary interface of the cipher implementation.

4.2 The file based certificate manager

The file based certificate manager provides certificates and private keys loaded from DER-encoded certificates and PKCS#12 files.
The UML diagram in Figure 4 shows the abstract methods defined by the *AbstractCertificateManager* abstract base class and implemented by the *FileBasedCertManager* concrete manager class. As with the signature provider, there are some functions that return static information data about the manager class. The function *getMailSigningCertificate* loads the default certificate to be used for signing e-mails. The function *getPrivateKeyForCertificate* returns the private key associated with a given certificate if available. With *getPinForCertificate*, one can get the password or PIN for private key decipherment if one has been configured or is otherwise available in the certificate manager. With *isPinRequiredForPrivateKey*, you can check whether you need a PIN or password to decipher the private key or whether it is freely usable (unprotected).

5 Integration into Thunderbird

Integrating external signature processes into Thunderbird's mail sending process is complex. If a user signs a message, other application features like saving mails as drafts or using a spellchecker that warns if a mail with spelling mistakes is to be sent must still be working. This means that the plugin cannot make any assumptions about additional steps in the mail sending process. Thus, just extracting the mail for external processing an then sending a changed (e.g. signed) version is not feasible. In addition, the plugin should work as comfortably and end-user friendly as Thunderbird's native S/MIME functions, especially without requiring the user to manually transfer signature files to or from an external application. In order create a client component accomplishing these challenges, we had to implement a full integration into the mail client's mail sending process.

The mail processing starts when the user clicks on Thunderbird's normal "Send message" button in the compose window. This call is redirected into Thundercrypt's own code. If the user has not chosen to sign the message, the call is forwarded to the base Thunderbird code as is. If a signature is to be made, the message is extracted and passed on to the server code for calculating the signature which is integrated into the mail as an attachment. Next, the mail headers are changed to mark the message as signed and point MIME multipart element containing the attached signature. Finally, the message is sent via Thunderbird's normal features for not breaking other plugins or organizer functions (like the "Sent messages" folder). The general process of sending a signed mail message is shown in Figure 5.

We will now address the steps in detail. The "Extract Message" step is necessary because Thunderbird generates the mail message directly before sending or saving it. Thus, the message stream is not directly accessible from the compose window in which the plugin's send handler is loaded. As a workaround, Thundercrypt saves the message as a draft and then loads the stored message

either from the local mail box folder's storage file when using SMTP with local folders or as a stream when using remote IMAP folders. The temporary draft is retained for further steps.
The message stream is then transferred to the Thundercrypt server which performs the signature and sends an encoded signature back to the Javascript client. As Thunderbird cannot attach files to already stored drafts without opening the composer which would have meant a more complex handling for the developer and an irritating window that opens up for a second and then automatically closes again for the user, we had to change the message outside Thunderbird. In order to accomplish this task, the first step was extended to attach a dummy file before saving the draft. This dummy attachment can then be replaced by the actual signature in the serialized mail data.
Up to the current step, the message is a normal unsigned e-mail with an arbitrary attachment. The next step changes the mail headers marking the message as signed with the attachment as the signature data.
External message processing is now complete. If local message folders are used, the respective data file is read from the profile folder and the original temporary draft is replaced before reloading the folder data in Thunderbird. For IMAP folders, the new message is streamed to the IMAP server.
For sending the changed mail, Thunderbird's function for sending messages later (a part of its offline working features) is used. The message is copied into the "Unsent messages" folder that is afterwards scheduled for sending. From now on, the message is a normal (but delayed) message again.
The whole process described takes a few seconds to complete. Given the internal complexity created by the external processing steps, the performance is still better than expected.

5.1 Alternative Integration Methods

We chose the approach of implementing a standalone plugin and redirecting some of Thunderbird's internal chrome functions to own code. The own code then performs the necessary preprocessing (generating the signature) and then sends the manipulated message with Thunderbird's original functions. This provided the greatest flexibility and guaranteed our independence of other external projects. We also considered and decided against the following alternatives:

- Use an existing Thunderbird plugin like Enigmail [Pro] as a basis upon which to build the own solution. This approach was however infeasible as we would have had to manage a large code basis split from a project under constant development with whose development we would have had to keep up as to constantly integrate their changes into our split or vice versa.
- Integrate a custom S/MIME implementation into Thunderbird. This approach has shown to be infeasible since the mail signature features of Thunderbird are only available as a monolithic black box that does not allow enough interference for custom ciphers without changing the Thunderbird

source code. Since Thunderbird is open source, this would have been possible, but would have created the same problems that made us decide against extending an existing plugin.

6 Used Components

For handling S/MIME messages, we use an implementation taken from the free BouncyCastle library (`www.bouncycastle.de`) . The BouncyCastle provider is used "as is" while there have been minor changes to the S/MIME classes. For writing out signatures, we employ internal classes that are normally not part of the external interface, for instance. As this still does not provide us with the required flexibility, we plan to provide our own S/MIME implementation in the future, especially with regard to integrating post-quantum ciphers at some later point.

The built-in FlexiProvider cryptographic module uses FlexiProvider (`www.flexiprovider.de`) [Gro].

For handling mail messages, we use the Java mail API "javax.mail". Base64 encoding and decoding is done with the CoDec package (`www.sourceforge.net/projects/codec/`)).

The TUD card cryptographic provider uses the PC/SC API of Java 6.0.

7 Implemented Modules

This section addresses the key management and cryptographic modules already implemented in Thundercrypt. At the moment, there are modules for reading X.509 certificates from DER encoded files, the Java Key Store, the TUD Card (student/employee smart card at Technische Universität Darmstadt) and the native operating system store on Windows systems. Private keys can be read from PKCS#12 files, the Java Key store, and the Windows native store. For signing e-mails with the RSA cipher, the BouncyCastle provider, the FlexiProvider[1] implementation, the standard Java implementation, the native Windows crypto API functions, and the TUD card can be used. When using the native Windows module, signing is also possible with non-exportable keys from the native store. For decrypting e-mails, there is currently only a legacy module that has been migrated to Thundercrypt as a proof-of-concept.

8 Conclusions

We have created a flexible, easily extendable and yet versatile platform providing new cryptographic e-mail functions to the Mozilla Thunderbird e-mail client. With our platform, new ciphers and key management modules can be integrated

[1] FlexiProvider's DSA functions are also available via the respective Thundercrypt module.

as modules without any changes to the platform itself, closing the gap between cryptographic research and practical application.

For the future, we plan to provide our own S/MIME implementation and integrate the process of signature verification into Thundercrypt which by now remains with the mail client. For verifying signatures inside Thundercrypt, we need a more flexible S/MIME handler that allows access to the parsed data without making any assumptions about the used algorithms. Another point of extension are the mail encryption and decryption modules that by now only exist as one proof-of-concept implementation. Like with the signature modules, we plan to provide multiple encryption methods. The high level of architectural flexibility also allows us to implement the plugin for other e-mail clients allowing extensions.

The author thanks Eduardo Lidanski, Zakaria Drissi Maniani and Dirk Volland for starting the project and designing a first prototype of the application.

References

[ANS05] American National Standards Institute ANSI. X9.62: Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), November 2005.

[BF] J. Becker and M. Frohna. Tudcard. `http://www.hrz.tu-darmstadt.de/dienste/id/tudcard`.

[CFS01] N. Courtois, M. Finiasz, and N. Sendrier. How to achieve a maceliece-based digital signature scheme. *Asiacrypt 2001*, 248, 2001.

[Gro] Theoretical Computer Science Research Group. Flexiprovider software and api documentation. `http://www.flexiprovider.de`.

[JCA] Java cryptography architecture (jca) reference guide. `http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`.

[NT94] National Institute of Standards NIST and Technology. FIPS 186 – Digital Signature Standard (DSS), May 1994. `http://www.itl.nist.gov/fipspubs/fip186.htm`.

[Pro] The Engimail Project. Openpgp email security for mozilla applications. `http://enigmail.mozdev.org`.

[Ram04a] B. Ramsdell. Secure / Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Certificate Handling. *IETF Request For Comments*, 3850, July 2004.

[Ram04b] B. Ramsdell. Secure / Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification. *IETF Request For Comments*, 3851, July 2004.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

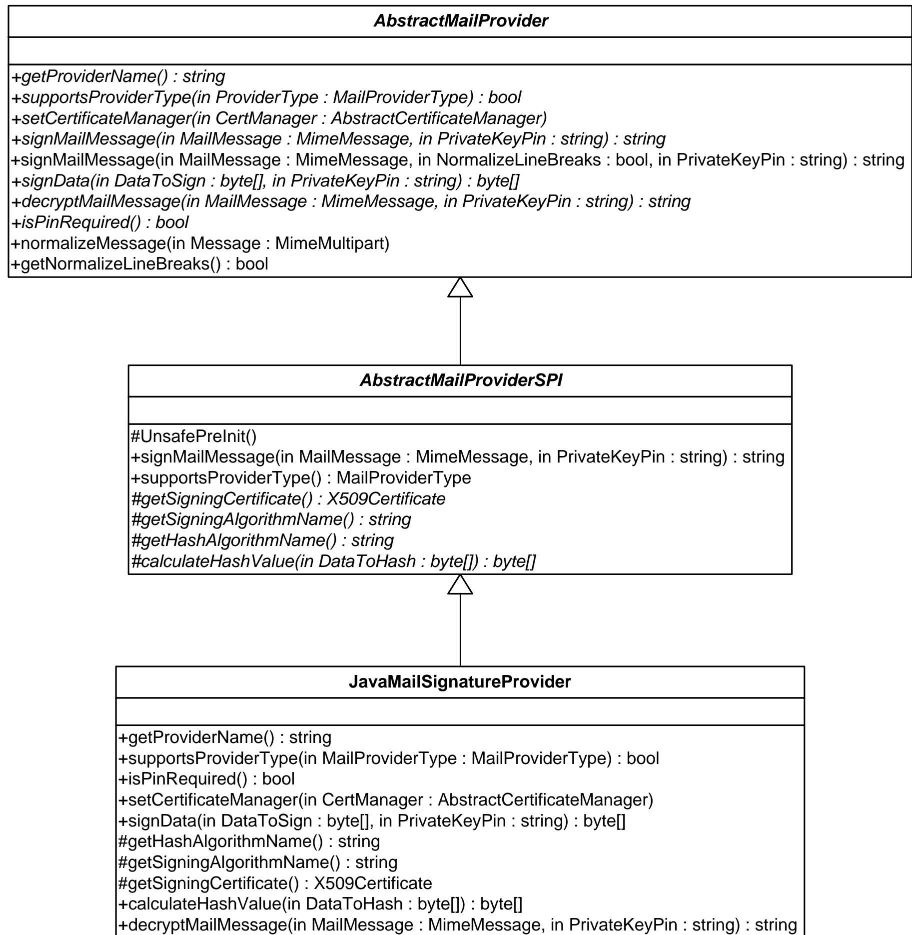

Fig. 3. Mail provider classes

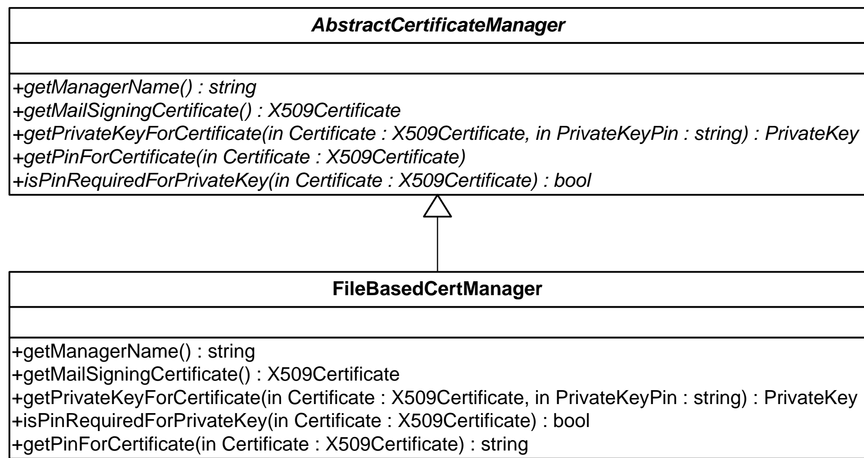

Fig. 4. Certificate manager classes

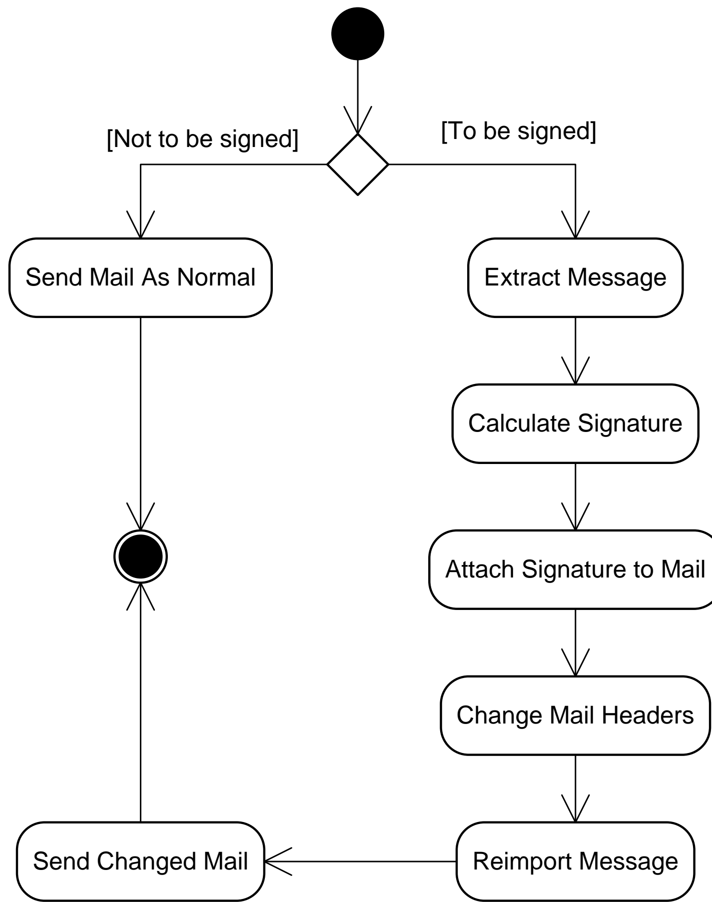

Fig. 5. Processing steps for sending signed mails