University of Bristol

DEPARTMENT OF COMPUTER SCIENCE

Identity Based Encryption from the Tate Pairing to Secure Email Communications

Matthew Baldwin

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering

MAY 2002 | CSMENG-02

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Masters of Engineering in Computer Science (G503)in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Matthew G Baldwin, Monday May 13th, 2002

Executive Summary

This aim of this project is to demonstrate the way in which a new cryptosystem can be used to simplify existing solutions to email security. This new crypto system is known as Identity Based Encryption[1.2].

The project implementation is composed of several parts, client tools[3.1] and a server tool[3.8]. In turn, the client tools suite, has several components, comprising of a COM Library[3.3], an Outlook Add In[3.4], a Windows Shell Extension[3.5] and a stand alone IBE Wizard[3.6].

The IBE functionality was provided by Hewlett Packard, and is based on the Boneh and Franklin IBE scheme proposed in section 1.2.2. This code was used as the back-end processing for an IBE API. The API was developed to enable the rapid application development of front end add-ins to existing applications providing IBE functionality to them[3.3].

One of the key benefits of an IBE system is the ability to distribute the trust among a number of Trusted Authorities[4.3]. By concatenating a date with the identity in a public key it is possible to yet again further the power of key generation, by providing built in key expiry, or other features.

IBE can be used for more than securing email communications. An example of another use would be to simplify the SSL handshake protocol[4.4].

Achievements

- Wrote over 10,000 lines of C++ code utilizing Microsoft COM and ATL. The code comprised of an IBE library, a Windows Shell Extension and a Microsoft Outlook Add-In to provide encryption and decryption support to the operating system and program environments.
- Learned Microsoft COM and ATL in order to develop the IBE library and application extensions.
- Learned the Microsoft Windows Shell Extensions API and the Microsoft Outlook Object Model to enable the development of the required extensions to Windows and Outlook.
- Designed and implemented an API for using the IBE encryption and decryption functions.
- Designed and implemented a full architecture for the IBE Library, allowing for concurrent processing of multiple encryption tasks.

Contents

1 Background

1.1 Project Overview

The basis of this project was to develop a system for securing email communications using a public key cryptosystem that is scalable. By this, the system must be able to work on large networks with thousands of users.

The most popular standard for email security is *Pretty Good Privacy*, or *PGP* as it is more commonly known. In principle PGP is a good idea, however there is a major problem associated with it whereby for every recipient of an encrypted email, you are required to know their public key before encrypting it. To ensure the validity of a public key, a third party is required to distribute and sign the key.

Public Key Infrastructure, or *PKI*, is a standard for Public Key Cryptosystems that is being widely backed by large e-security companies. However, the flaws in the system are similar to that in PGP. Maintaining lists and directories of public keys, both on a central server and users' machines, is a management nightmare.

The problems of key and certificate management, issues in both PGP and PKI, are to be addressed by the IBE system proposed and developed as part of this project.

1.2 Identity Based Encryption

Identity-based cryptosystems resemble ordinary public-key systems, involving a private and a public transformation, however users do not have explicit public-keys. Instead of explicit public-keys, the public-key is constructed from a user's publicly available identity. Any publicly available information that is uniquely associated with the user may serve as identity information. Email addresses are uniquely associated with specific users and are publicly available, making them an obvious and ideal choice for an identity based public key.

1.2.1 A brief history of Identity-Based Systems

It was initially suggested by *Shamir* in 1984 that a public key encryption scheme be developed where a public key can be an arbitrary string. The motivation for such a scheme was to simplify the management of keys and certificates in secure email systems.

Since being proposed in 1984 by Shamir, there have been several proposals for Identity Based Encryption (*IBE*) schemes, however none are fully satisfactory. Conversely, the related identity based signature and authentication schemes, also proposed by Shamir, do have satisfactory solutions. The scheme on which this project is based was fully proposed by *Boneh and Franklin* (see section 1.2.2). Their scheme, however, was based using the *Weil Pairing* (see section 1.4.1) and has been adapted to use the *Tate Pairing* (see section 1.4.1).

Identity based systems differ from other public key cryptosystems in that there is no need to verify the authenticity of the public key. Forgery is protected against by the inherent redundancy in public data together with the authentic public data. Incorrect public data results in the cryptographic transformations failing. Signing and verification will fail, encryption will produce decipherable cipher-text and decryption will produce an invalid plain- text.

Shamir's original motivation behind the idea of an identity based public key system forms the basis of this project. The technical implementation aims to use an already developed IBE implementation (in terms of the cryptographic algorithms) to create a working prototype for an ideal secure mail system. That is a mail system with the following properties:

1. There is no need to exchange symmetric keys or public keys between users;
2. Public lists of public keys need not be maintained;
3. The Trusted Authority is only needed during the setup phase (for authentic public key parameters and generation of private keys).

All three of these are problems associated with traditional secure mail solutions. PGP is the most widely used of these systems and requires users to obtain recipients public keys from a public key directory before encryption. This is a list maintained by a Trusted Authority which distributes these keys, electronically signing them for verification.

1.2.2 Boneh and Franklin IBE Scheme

Boneh and Franklin proposed a fully working solution to the problem of an IBE scheme in a paper in 2001. This paper forms the basis for the IBE scheme used from here in. This section provides a background to this scheme, describing each of the stages involved in sufficient detail so as to provide the reader with a working knowledge of the system.

There are four stages to identity based algorithms such as this: **(1) setup** generates global system parameters and a **master-key**, **(2) extract** uses the **master-key** to generate the private key corresponding to an arbitrary public key string $ID \in \{0,1\}^*$, **(3) encrypt** encrypts messages using the public key ID, and **(4) decrypt** decrypts messages using the corresponding private key.

To understand the Identity-Based algorithms used throughout this paper, there are several background issues that must first be understood. These are presented below, followed by a description of each of the four stages of the IBE system.

1.3 Microsoft Component Object Model

The Microsoft Component Object Model is the base programming technology for the design and implementation of the proposed solution. The following is quoted from [1] documentation and gives a simple introduction into what COM is.

> The Component Object Model (COM) is a platform-independent, distributed, object-oriented, system for creating binary software components that can interact. COM is the foundation technology for Microsoft's OLE (compound documents), ActiveX (internet enabled components), as well as others.

A COM compliant application consists of two parts, a *COM server* and a *COM client*.

The server is any object that provides services to clients. Services are in the form of implementations of COM interfaces that can be called by any client who is able to get a pointer to one of the interfaces on the server object.

A COM client is whatever code or object gets a pointer to a COM server and uses its services by calling the methods of its interfaces.

COM server objects can be either *in-process* or *out-of-process* servers. An *in-process* server is a DLL that is loaded and run in the same address space as the invoking client application. *Out-of-process* servers are executables that are loaded and executed in their own address space, separate to the client application.

In-process servers are faster to use than out-of-process servers as the client application doesn't need to make remote procedure calls across the process boundary to communicate with the server object. However, out-of-process servers can run as a stand alone application, something an in-process server is unable to do.

1.3.1 Microsoft Windows Shell Extensions

As the name suggests, shell extensions allow developers to add functionality to the existing Windows shell. Some examples of shell extensions are Context Menus (menus that change based on what object has focus when you right-click), Property Sheet Handlers (tabbed pages that appear when the Properties menu item is selected from an objects context menu), Icon Overlays (appear as the arrow on top of an icon that points to a shortcut or the hand that appears on shared folders), Folder Customization, and many, many more.

Shell Extensions are implemented as In Process COM servers. Windows Explorer invokes the appropriate extension in response to shell-wide events. Explorer was designed to respond in very specific ways when the user performs various functions within its shell. The first thing Explorer does is check for any modules that have been registered for a specific event and if one exists it attempts to load the module.

To be a valid shell extension, the COM server must implement an interface that defines the specific behavior for the desired extension and it must implement an interface that defines its initialization behavior. Finally, to be a valid shell extension, the COM server must follow the approved method of registering itself with the system.

Table 1 lists all of the currently available shell extensions.

1.3.2 Microsoft Outlook Add-Ins

COM add-ins are an Office feature that was first implemented in Microsoft Office 2000. The architecture is supported by all Office programs, including Outlook. Key benefits of

Type	Apply To	Version	Interface Involved	Description
Context Menu	File class and shells object	Windows 95+	IContextMenu, IContextMenu2, or IContextMenu3	Allows you to add new items to a shell objects context menu.
Right Drag and Drop	File class and shells object	Windows 95+	IContextMenu, IContextMenu2, or IContextMenu3	Allows you to add new items to the context menu that appears after your right drag and drop files.
Drawing Shell Icons	File class and shells object	Windows 95+	IExtractIcon	Lets you decide at runtime which icon should be displayed for a given file within a file class.
Property Sheet	File class and shells object	Windows 95+	IShellPropSheetExt	Lets you insert additional property sheet pages to the file class Properties dialog. It also works for Control Panel applets.
Left Drag and Drop	File class and shells object	Windows 95+	IDropTarget	Lets you decide what to do when an object is being dropped (using the left mouse button) onto another one within the shell.
Clipboard	File class and shells object	Windows 95+	IDataObject	Lets you define how an object is to be copied to and extracted from the clipboard.
File Hook		Windows 95+	ICopyHook	Lets you control any file operation that goes through the shell. While you can permit or deny them, you aren't informed about success or failure.
Program Execution	Explorer	Desktop Update	IShellExecuteHook	Lets you hook any programs execution that passes through the shell.
Infotip	File class and shells object	Desktop Update	IQueryInfo	Lets you display a short text message when the mouse hovers over documents of a certain file type.
Column	Folders	Windows 2000	IColumnProvider	Lets you add a new column to the Details view of Explorer.
Icon Overlay	Explorer	Windows 2000	IShellIconOverlay	Lets you define custom images to be used as icon overlays.
Search	Explorer	Windows 2000	IContextMenu	Lets you add a new entry on the Start menus Search menu.
Cleanup	Cleanup Manager	Windows 2000	IEmptyVolumeCache2	Lets you add a new entry to the Cleanup Manager to recover disk space.

Table 1: This table lists all the available types of shell extensions, the minimum shell version each requires, the involved interfaces, and a brief description.

COM add-ins include:

- COM add-ins run in-process with the host program, so custom code generally runs faster than code implemented in Visual Basic for Applications.
- The basic COM add-in architecture is consistent across all Office programs, whereas in previous versions of Office, each program had its own add-in architecture.
- You can create one COM add-in to use with more than one Office program.

COM add-ins aren't the only way to extend the functionality of Outlook or Office applications. The simplest way is with *Visual Basic for Applications*, or *VBA*, a trimmed down version of Visual Basic which a programmer can develop simple macros for the application.

Using COM to extend outlook combines the benefits of ease of development from VBA with those of developing a precompiled DLL, such as speed of execution, enhanced programming power, and the ability to develop a single add-in that will work with several applications.

1.4 Mathematical Notation and Definitions

This section describes the mathematical principles and notation behind the algorithms that form the IBE scheme which is used throughout this document.

Group: A mathematical *group* is a set with the following properties

- *identity*: for every element a there is an operation I such that $I(a) \mapsto a$;
- *associativity*: $(A \cap B) \cap C = A \cap (B \cap C)$;
- every element must have an inverse.

Groups are represented by the following notation: $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$, etc.

Galois Field: A mathematical group of integers modulus a prime integer.

Bilinear Map: A map $f(x, y)$ that is linear in each of its two variables; i.e., a mapping $f(x, y)$ from the Cartesian product $E \times F$ of two R-modules into R, such that for each $x \in E$, the function f_x that takes y to $f(x, y)$ is linear in y , and for each $y \in F$, the function f_y which takes x to $f(x, y)$ is linear in x.

Fiestel Cipher A Fiestel Cipher is a special kind of iterative block cipher. Regardless of the choice of the function f, which applies a transformation to the key and data, the function is invertible. Figure 1 diagrams a Fiestel cipher as a black box.

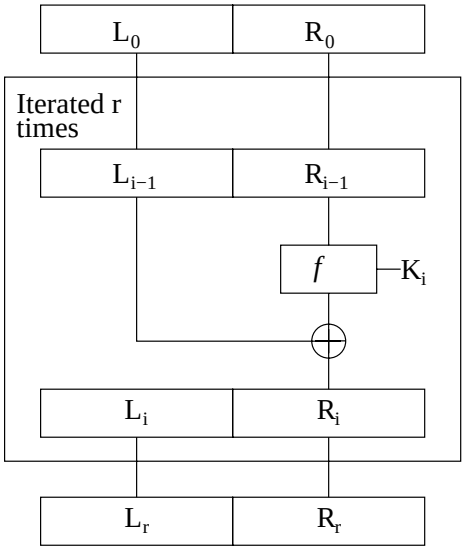


Figure 1: For each round, i, there are several stages. The first stage is the splitting of the input block into two equally sized halves. One half is then transformed under some function f using the round key, K_i. The result is then XOR'd (exclusively OR'd) with the other half, forming the new Right hand side. This is continued for r rounds.

1.4.1 Tate and Weil Pairing

The Weil pairing was first used in cryptography by Menzes, Okamoto and Vanstone [11] to reduce the complexity of the discrete logarithm problem on certain elliptic curves. Frey and Rück [6] introduced the Tate pairing to cryptography as an extension of the work carried out by Menzes, Okamoto and Vanstone.

The Tate and Weil Pairings are bilinear maps:

$$\widehat{e} : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$$

$$P \times Q \mapsto \langle P, Q \rangle$$

where $\mathbb{G}_1$ and $\mathbb{G}_2$ are groups.

Tate and Weil pairings are useful as they have various nice properties for cryptographic purposes such as:

$$\langle [\lambda]P, Q\rangle \equiv \langle P, Q\rangle^{\lambda}$$

The Tate pairing is given by the notation $\widehat{t}\langle P, Q\rangle$.

The Weil pairing is given by the notation $\widehat{e}\langle P, Q\rangle$.

Tate Pairing versus Weil Pairing

Advantages:

- It applies to higher genus curves as well as elliptic curves;
- It requires less computation time;
- It works over smaller fields in some cases;
- It is simpler.

Disadvantages:

- To get a unique value you have to exponentiate by $(q^k - 1)/l$;
- Understanding non-degeneracy is more difficult.

1.4.2 The Bilinear Diffie-Hellman Assumption (BDH)

The Bilinear Diffie-Hellman Assumption (BDH) is a variation of the Computational Diffie-Hellman assumption and is what the security of the IBE system is based upon. This is because the Decision Diffie-Hellman problem (DDH) in $\mathbb{G}_1$ is easy, DDH cannot be used to build cryptosystems in the group $\mathbb{G}_1$.

Decision Diffie-Hellman problem (DDH) The Decision Diffie-Hellman problem (DDH) in $\mathbb{G}_1$ is to distinguish between the distributions $\langle P, aP, bP, abP\rangle$ and $\langle P, aP, bP, cP\rangle$ where a, b, c are random in $\mathbb{Z}_q$ and P is random in $\mathbb{G}_1$. Joux and Nguyen [9] point out that DDH in $\mathbb{G}_1$ is easy. To see this, observe that given P, aP, bP, $cP \in \mathbb{G}_1^*$:

$$c = ab \bmod q \iff \widehat{e}(P, cP) = \widehat{e}(aP, bP)$$

The Computational Diffie-Hellman problem (CDH) in $\mathbb{G}_1$ can still be hard (CDH in $\mathbb{G}_1$ is to find abP given random $\langle P, aP, bP\rangle$). Joux and Nguyen [9] give examples of mappings $\widehat{e} : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$ where CDH in $\mathbb{G}_1$ is believed to be hard even though DDH in $\mathbb{G}_1$ is easy.

Bilinear Diffie-Hellman Problem: Let $\mathbb{G}_1$, $\mathbb{G}_2$ be two groups of prime order q. Let $\widehat{e} : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$ be a bilinear map and let p be a generator of $\mathbb{G}_1$. The BDH problem in $\langle \mathbb{G}_1, \mathbb{G}_2 \widehat{e}\rangle$ is as follows: Given $\langle P, aP, bP, cP\rangle$ for some $a, b, c \in Z_q^*$ compute $W = \widehat{e}(P, P)^{abc} \in \mathbb{G}_2$. An algorithm A has advantage ϵ in solving BDH in $\langle \mathbb{G}_1, \mathbb{G}_2 \widehat{e}\rangle$ if

$$\Pr[\mathcal{A}(P, aP, bP, cP) = \widehat{e}(P, P)^{abc}] \geq \epsilon$$

where the probability is over the random choice of $\langle a, b, c\rangle$ in Z_q^*, the random choice of $P \in \mathbb{G}_1^*$, and the random bits of $\mathcal{A}$.

BDH Parameter Generator: A randomized algorithm $\mathcal{IG}$ is a *BDH parameter generator* if (1) $\mathcal{IG}$ takes a security parameter $0 < k \in \mathbb{Z}$, (2) $\mathcal{IG}$ runs in polynomial time in k, and (3) $\mathcal{IG}$ outputs the description of two groups $\mathbb{G}_1$, $\mathbb{G}_2$ and the description of a bilinear map $\widehat{e} : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$. It is required that the groups have the same prime order $q = |\mathbb{G}_1| = |\mathbb{G}_2|$. The output of $\mathcal{IG}$ is denoted by $\mathcal{IG}(1^k)$.

Bilinear Diffie-Hellman Assumption: Let $\mathcal{IG}$ be a BDH parameter generator. An algorithm $\mathcal{A}$ has advantage $\epsilon(k)$ in solving the BDH problem for $\mathcal{IG}$ if for sufficiently large k:

$$\mathrm{Adv}_{\mathcal{IG}}(\mathcal{A}) = \Pr\left[\mathcal{A}(\mathbb{G}_1, \mathbb{G}_2, \widehat{e}, P, aP, bP, cP) = \widehat{e}(P, P)^{abc} \;\middle|\; \begin{array}{l} \langle \mathbb{G}_1, \mathbb{G}_2, \widehat{e}\rangle \leftarrow \mathcal{IG}(1^k), \\ P \leftarrow \mathbb{G}_1^*, a, b, c \leftarrow \mathbb{Z}_q^* \end{array}\right] > \epsilon(k)$$

It is said that $\mathcal{IG}$ satisfies the BDH assumption if for a random $\langle , \mathbb{G}_1, \mathbb{G}_2, \widehat{e}\rangle$ generated by $\mathcal{IG}$, no efficient algorithm can solve BDH in polynomial time. It is occasionally said that BDH is hard in groups generated by $\mathcal{IG}$.

2 Technical Basis

There are several main technical problems which this project set out to address, listed below. For each problem, this section explains what it is and how it was solved by the project implementation.

Developing an IBE library Developing an IBE library that could be interfaced to easily was core to the success of this project (section 2.1). The advantage of developing such a library is that developing client applications to use it would be very simple.

Developing an Outlook Add-In This was the major problem to be addressed in terms of developing a solution, as the project is based around email security. Once Outlook had been identified as the email client to be developed for, the main project goal was set. This is described in detail in section 2.4.

Setting up a Trusted Authority The Trust Authority is responsible for the setup and extraction of users' private keys. To demonstrate the proposed email system, a Trust Authority was required to be set up to allow for this. Details of this can be found in section 2.6.

2.1 Developing an IBE Library

Development of the IBE Library composed of two sections:

1. Writing/Obtaining IBE code;
2. Wrapping the code in ATL/COM.

Hewlett Packard (HP) supplied the IBE code, which includes encryption and decryption routines, as well as the private key generation routines. The first stage in developing an API was to learn how the code worked, followed by designing a COM library.

Developing a COM library can be broken down into two stages:

1. Defining the Interface(s);
2. Implementing the Interface(s).

There is a fundamental distinction between an *interface* and it's implementation that is made by COM.

Interface An interface is a contract that consists of a group of related function prototypes whose usage is defined but whose implementation is not. There is no implementation associated with an interface.

Interface Implementation An implementation of an interface is the code written by a programmer to perform the actions specified by the interface methods. The implementation is associated with an object when an instance is created, and provides the services offered by the objects interfaces.

COM interfaces differ from C++ interfaces, as only a predefined subset of the methods offered by a class are included in a COM interface, not all the methods, such is the case in C++ interfaces. In this way, COM interfaces are equivalent to Java interfaces.

2.1.1 Interface Development

The design of the interface and implementation details for all COM objects associated with this project can be found in section 3.

The COM interface is defined in *IDL* (Interface Definition Language), which is pre-processed by an IDL compiler to produce *proxy* and *stub* code for the interface. The proxy is an interface specific object which provides the parameters required for a client application to communicate with a COM server. The proxy would be linked into the client and communicates with the stub which is part of the COM server object.

2.1.2 Interface Implementation Development

As mentioned above, the interface implementation is the code written by a programmer to perform the methods associated with the interface. This is the core of the object, and can be done in any language that supports COM, for example: C++, Java (using Microsoft J++) or Visual Basic.

To implement an interface the programmer must create a COM object and select the interfaces he wishes to implement. Using Microsoft Visual Studio, this is very simple as it

will supply a list of known interfaces that are registered and once selected will generate the skeleton implementation, which is the function prototypes in the class definition and the basic function body. The programmers responsibility is to replace the basic function bodies with the code necessary to perform the methods exposed by the interface.

The IBE Library architecture is very important. Encryption and Decryption of large files can be very time and resource consuming processes, so the architecture must be designed in such a way that any user interface features remain responsive throughout the procedure. This is discussed in detail in section 3.2.

2.2 IBE Implementation

The IBE code implementation supplied by HP was completely implemented and tested, which meant that the Library development would consist of developing the COM wrapper. This section describes in more detail the IBE scheme used.

2.2.1 Admissible Encoding Function

MapToPoint is an admissible encoding function[1] used in the IBE system proposed in section 2.2.2. The proof of this function and the explanation as to why it is needed is covered in Boneh and Franklin's paper [4]. This section is intended to provide an overview of its operation.

Let p be a prime satisfying $p = 2\text{mod}3$ and $p = 6q - 1$ for some prime $q > 3$. Let E be the elliptic curve $y^2 = x^3 + 1$ over $\mathbb{F}_p$. Let $\mathbb{G}_1$ be the subgroup of points on E of order q. $H_1 : \{0,1\}^* \rightarrow \mathbb{F}_p$ is a hash function.

Algorithm **MapToPoint** works as follows on input $y_0 \in \mathbb{F}_p$:
1. Compute $x_0 = (y_0^2 - 1)^{1/3} = (y_0^2 - 1)^{(2p-1)/3} \in \mathbb{F}_p$.
2. Let $Q = (x_0, y_0) \in E/\mathbb{F}_p$ and set $Q_{ID} = 6Q \in \mathbb{G}_1$.
3. Output **MapToPoint**$(y_0) = Q_{ID}$.

[1] Admissible Function: A description of a search algorithm that is guaranteed to find a minimal solution path before any other solution paths, if a solution exists.

2.2.2 IBE based on the Weil Pairing

The following IBE system is proposed by Boneh and Franklin in [4]. This system was modified for use in this project by substituting the Weil pairing for the Tate pairing.

Setup: The algorithm works as follows:

Step 1: Chose a large k-bit prime p such that $p = 2\text{mod}3$ and $p = 6q - 1$ for some prime $q > 3$. Let E be the elliptic curve defined by $y^2 = x^3 + 1$ over $\mathbb{F}_p$. Choose an arbitrary $P \in E/\mathbb{F}_p$ of order q.

Step 2: Pick a random $s \in \mathbb{Z}_q^*$ and set $P_{pub} = sP$.

Step 3: Pick four hash functions: $H_1 : \{0,1\}^* \rightarrow \mathbb{F}_p$; $H_2 : \mathbb{F}_{p^2} \rightarrow \{0,1\}^n$ for some n ; $H_3 : \{0,1\}^n \times \{0,1\}^n \rightarrow \mathbb{Z}_q^*$; and $H_4 : \{0,1\}^n \rightarrow \{0,1\}^n$.

The message space is $\mathcal{M} = \{0,1\}^n$ The cipher-text space is $\mathcal{C} = E/\mathbb{F}_p \times \{0,1\}^n$. The system parameters are **params**$= \langle p, n, P, P_{pub}, H_1, \ldots, H_4 \rangle$. The **master key** is $s \in \mathbb{Z}_q^*$.

Extract: For a given string $ID \in \{0,1\}^*$ the algorithm builds a private key d as follows:

Step 1: Compute **MapToPoint**$(H_1(ID)) = Q_{ID} \in E/\mathbb{F}_p$ of order q.

Step 2: Set the private key d_{ID} to be $d_{ID} = sQ_{ID}$ where s is the master key.

Encrypt: To encrypt $\mathcal{M} \in \{0,1\}^n$ under the public key ID do the following:

Step 1: Compute **MapToPoint**$(H_1(ID)) = Q_{ID} \in E/\mathbb{F}_p$ of order q.

Step 2: Choose a random $\sigma \in \{0,1\}^n$.

Step 3: Set $r = H_3(\sigma, M)$.

Step 4: Set the cipher-text to be

$$C = \langle rP, \sigma \oplus H_2(g_{ID}^r), M \oplus H_4(\sigma) \rangle \text{ where } g_{ID} = \widehat{e}(Q_{ID}, P_{pub}) \in \mathbb{F}_{p^2}$$

Decrypt: Let $C = \langle U, V, W \rangle \in \mathcal{C}$ be a cipher-text encrypted using the public key ID. If $U \in E/\mathbb{F}_p$ is not a point of order q reject the cipher-text. To decrypt C using the private key d_{ID} do:

Step 1: Compute $V \oplus H_2(\widehat{E}(d_{ID}, U)) = \sigma$.

Step 2: Compute $W \oplus H_4(\sigma) = M$.

Step 3: Set $r = H_3(\sigma, M)$. Test that $U = rP$. If not, reject the cipher-text.
Step 4: Output M as the decryption of C.

2.3 Improving Efficiency

The scheme outlined above is the basis for the HP IBE implementation, however there are several issues of efficiency that needed to be addressed.

1. Tate pairing efficiency;

2. Overall efficiency of the cryptosystem.

Hewlett Packard developed the IBE code, and in doing so overcame the problem of efficiently implementing the Tate Pairing. Although this is not something that was carried out as part of the project, it is critical to the viability of the technical solution.

Galbraith, Harrison and Soldera in [7] provide methods to achieve fast computation of the Tate pairing, and are mentioned briefly here.

Three observations are made in [7]:

1. Subfields: It is possible to work on subfields to reduce the number of operations;

2. Divisions and Multiplications: Divisions are more expensive than multiplications, something that it is possible to exploit in the calculation of the Tate pairing, by combining three divisions into a single division.

3. Window methods (see [3] and [8]): window methods provide significant improvement for elliptic curve point exponentiation algorithms.

The efficiency problems associated with the technical implementation that forms part of this project come from the actual encryption and decryption process. Using IBE to encrypt an entire file is not fast as it is an asymmetric system that requires calculations of exponents in elliptic curves. This was proved by early testing, although figures were not recorded, it can be said that the encryption process took a substantial amount of time on a 2kb file and the decryption process took longer still.

Time is not the only efficiency issue associated with this problem - size is also a factor. As the IBE scheme used in works over small parts of plain-text at a time, producing the cipher-text $< U, V, W >$ for each block of 64 bytes. For a large file, this could result in a cipher-text of considerable length, far greater than that of the original plain-text.

These problems were overcome by using a symmetric cryptosystem that works much faster than an asymmetric one to perform the bulk encryption and decryption, whilst using IBE to encrypt and decrypt the key for use as input into that system. More details on how this was designed and implemented can be found in section 3.3.3.

Another problem is introduced by this solution to the efficiency problem:

- Generating a cryptographically random key of sufficient length for the symmetric cryptosystem.

As explained in section 3.3.3 this is done using the Microsoft Crypto API that forms one of the base services in the Windows operating system.

2.3.1 DES and Triple DES

The Data Encryption Standard (DES) algorithm is a Fiestel cipher which processes plain-text blocks of 64 bits, producing 64 bit cipher-text blocks. A detailed description of the DES algorithm can be found in [10]. Figure 2 gives details of the DES algorithm.

Triple DES is triple encipherment of the plain-text to produce the cipher-text. Figure 3 shows the process of encryption under Triple DES in EDE[2] mode.

2.4 Developing a Solution for Outlook

The developing of a Outlook Add-In is much the same as the implementing of the IBE Library. Outlook Add-In's are COM objects that implement the correct interfaces.

An Outlook Add-In is easiest to develop using Visual Basic, however, this language doesn't provide the same level of power as developing it in C++, for example. C++,

[2]EDE: Encrypt-Decrypt-Encrypt

Encryption:
INPUT: plain-text $m_1 \ldots m_6 4$; 64-bit key $K = k_1 \ldots k_6 4$ (includes 8 parity bits)
OUTPUT: 64-bit cipher-text block $C = c_1 \ldots c_6 4$.

1. (key schedule) Compute sixteen 48-bit round keys K_i from K
2. $(L_0, R_0) \leftarrow$ IP$(m_1, m_2 \ldots m_6 4)$. Split the result into left and right 32-bit halves L_0 and R_0. $IP(x)$ is the Initial Permutation.
3. (16 rounds) for i from 1 to 16, compute L_i and R_i:
   $L_i = R_{i-1}$
   $R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$, where $f(R_{i-1}, K_i) = P(S(E(R_{i-1}) \oplus K_i))$
   computing $f(R_{i-1}, K_i) = P(S(E(R_{i-1}) \oplus K_i))$ as follows
   (a) Expand $R_{i-1} = r_1 \ldots r_3 2$ from 32 bits to 48 using the Expansion Permutation $T \leftarrow E(R_{i-1})$
   (b) $T' \leftarrow T \oplus K_i$. Represent T' as eight 6-bit character strings: $(B_1, \ldots, B_8) = T'$
   (c) $T'' \leftarrow (S_1(B_1), S_2(B_2), \ldots, S_8(B_8))$. $S_1, \ldots, S_8$ represent the Substitution boxes.
   (d) $T''' \leftarrow P(T'')$. $P(x)$ is the Permutation function.
4. $b_1 \ldots b_6 4 \leftarrow (R_1 6, L_1 6)$.
5. $C \leftarrow$ IP$^{-1}(b_1 \ldots b_6 4)$. Transpose Initial Permutation, $IP(x)$, to IP^{-1}.

Decryption:
DES Decryption uses the same algorithm as Encryption, however the Key Schedule is reversed and L and R are swapped around.

Figure 2: For a full description of the algorithm and details of the permutations and expansions, see [10].

being an object orientated language, has all the features required to subclass components etc, something that is missing in Visual Basic. Another advantage of C++ over VB is the speed of execution.

2.4.1 Interfaces

The **IDTExtensibility2** interface is the only interface that is required to be implemented by a Microsoft Office Add-In and is defined in the MSAddIn Designer Type Library. It is required that all COM add-ins for Microsoft Office implement the five methods defined in the interface.

```
interface _IDTExtensibility2 : IDispatch {
```

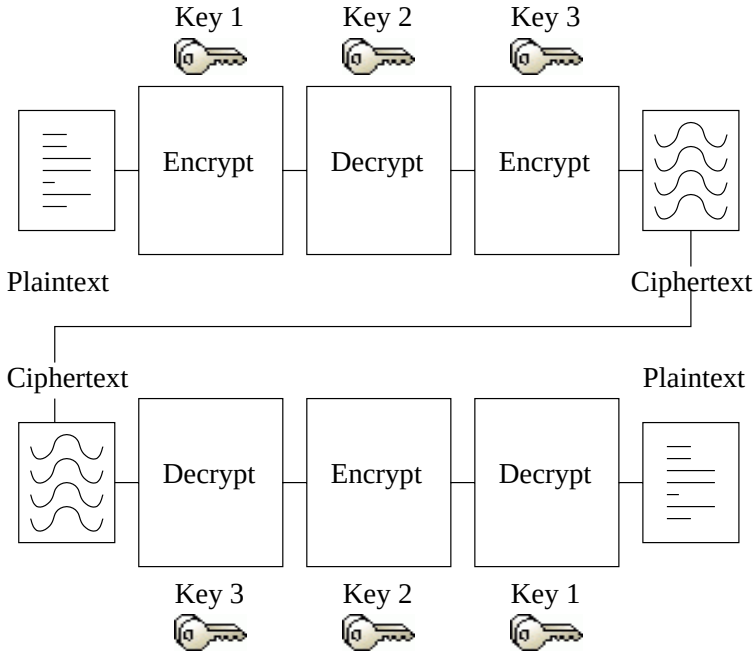


Figure 3: This is DES encryption where an encrypt operation is encrypt-decrypt-encrypt, and decrypt is decrypt-encrypt- decrypt. An implementation of Triple DES would simply be a wrapper around the three DES rounds.

```
  [id(0x0000001)]
  HRESULT OnConnection{
                    [in] IDispatch* Application,
                    [in] ext_ConnectMode ConnectMode,
                    [in] IDispatch* AddInInst,
                    [in] SAFEARRAY(VARIANT)* custom};
  [id(0x0000002)]
  HRESULT OnDisconnection{
                    [in] ext_DisconnectMode RemoveMode,
                    [in] SAFEARRAY(VARIANT)* custom};
  [id(0x0000003)]
  HRESULT OnAddInsUpdate{
                    [in] SAFEARRAY(VARIANT)* custom};
  [id(0x0000004)]
  HRESULT OnStartupComplete{
                    [in] SAFEARRAY(VARIANT)* custom};
  [id(0x0000005)]
  HRESULT OnBeginShutdown{
                    [in] SAFEARRAY(VARIANT)* custom};
};
```

The methods **OnConnection** and **OnDisconnection** are called when the add-in is loaded and unloaded from memory, respectively. The add-in can be loaded at application startup, by the user or through automation, and this is specified by the enumerator **ext_Connection**. **OnAddInsUpdate** is called when the set of loaded add-ins is changed. **OnStartupCom-**

plete is called if the add-in was loaded during application startup up and once the application has fully loaded, and finally **OnBeginShutdown** is called if the add-in was disconnected on the host application shutting down.

Once the COM add-in is implemented, it must be registered for use with the application, and is done using the Windows registry. The values are stored under the key:

```
HKEY_CURRENT_USER\Software\Microsoft\Office\
<TheOfficeApp>\Addins\<ProgID>\
```

This key holds information about the add-in such as the friendly name (as displayed in the host application), a description of the add-in, and the load behavior which specifies how the add-in loads, either at startup, by the user, etc.

2.4.2 Event Handling

For the user to be able to interact with the add-in, it is necessary to handle events generated by user actions. To enable this, the add-in object must inherit from the Event Dispatcher interface for the appropriate event types. An entry must be made in the Event Sink Map for each event the object wishes to catch. The code fragment below shows an outline for handling the **ItemSend** event that is dispatched by Microsoft Outlook.

```
class CAddIn;
typedef IDispEventImpl<OUTLOOK_OBJ, \
                       CAddIn, \
                       &DIID_ApplicationEvents, \
                       &LIBID_Outlook, 9, 0> AppOutlookEvents;
...
class ATL_NO_VTABLE CAddIn :
  ...
  public AppOutlookEvents
{
  ...
BEGIN_SINK_MAP(CAddIn)
  SINK_ENTRY_INFO(OUTLOOK_OBJ, DIID_ApplicationEvents,
                  0x0000f002, OnItemSend, &OnSendItemInfo)
END_SINK_MAP()
  void _stdcall OnItemSend(IDispatch* Item, VARIANT_BOOL* Cancel);
};
```

Before the event handlers could be written for the Outlook Add-In, the events that will enable the encryption of outgoing email and decryption of encrypted email had to be identified. The first and easiest event to intercept is the **SendItem** event. It is generated

when a user sends any item, be it a mail note or an appointment. Interception of this event provides the way to encrypt the email.

Identifying when an encrypted mail item is opened can be done by handling the **OnNewInspector** event that. This is generated whenever an item is opened. A simple check for the item type related to the inspector would reveal if the item being opened was a mail item, with further checks on that item to determine whether it is encrypted. This event is also used to set the initial flag values when composing a new email.

A custom event was required to handle the actual setting of the encryption status, depending on whether the user had selected or deselected the encryption button.

2.5 Developing Shell Extensions

Developing shell extensions is almost identical to developing an Outlook Add In. The only difference being the interfaces to implement. For the shell extension developed as part of this project, the interface was **IContextMenu**. However, shell extensions based on the **IContextMenu** and **IShellPropSheetExt** interfaces must also implement the **IShellExtInit** interface, which is responsible for initializing the shell extension.

```
interface IShellExtInit : IDispatch {
  HRESULT Initialize(
                    [in] LPCITEMIDLIST pidlFolder,
                    [in] LPDATAOBJECT pDataObj,
                    [in] HKEY hkProgID);
};

interface IContextMenu : IDispatch {
  HRESULT GetCommandString(
                    [in] UINT uCmdID, [in] UINT uFlags,
                    [in] UINT* puReserved,
                    [in, out] LPSTR szName,
                    [in] UINT cchMax);
  HRESULT InvokeCommand(
                    [in] LPCMINVOKECOMMANDINFO pCmdInfo);
  HRESULT QueryContextMenu(
                    [in] HMENU hMenu, [in] UINT uMenuIndex,
                    [in] UINT uidFirstCmd,
                    [in] UINT uidLastCmd,
                    [in] UINT uFlags);
};
```

The **Initialize** function on the **IShellExtInit** interface is called to initialize a property sheet extension, shortcut menu extension, or drag-and-drop handler.

The **GetCommandString** function retrieves information about a shortcut menu command, including the Help string and the language-independent, or canonical, name for the command. This information is then displayed in the status bar of the window when the mouse hovers over the selection. The **InvokeCommand** method, as the name suggests, is called when a user selects an item from the context menu and carries out the associated command. The third method, **QueryContextMenu**, actually adds the command to the menu, along with any icons associated with it.

2.6 Setting up a Trusted Authority

The Trusted Authority, or *TA*, has a very important role in the system, yet performs relatively simple tasks. By defining the role, a solution presented itself very easily.

> The role of the TA is to generate the system parameters, extract users' private keys and securely transmit them to the user.

The simple solution is to do this via a web site, where a user would enter their email address and a pass phrase with which to encrypt the key under. A script on the server could then generate the key, encrypt it and then email it to the specified email address.

This solution has several problems in itself.

Pass-phrase is susceptible to attack To get around this problem, the site would have to operate over a secure connection using SSL.

Persisting the storage of the TA private details This problem is somewhat more complicated as the TA's private key would have to be persisted to local storage and some how encrypted to prevent a malicious user from intercepting it, therefore enabling them to generate users' keys themselves. More detail on this problem and solution can be found in section 3.8.

Email Spoofing It is entirely possible that a malicious user could access the web service that provides the TA fucntions to enter the email address of a different user of the

system, along with a pass phrase of their choosing, and intercept the email. This would then give the attacker access to the private key of the user. No solution to this problem has been provided or proposed in this report, however a more secure way of managing user accounts would be required in any commercial version.

This solution, although very simple, provides the necessary functionality and level of security to demonstrate the system.

3 Design and Implementation

This section briefly introduces each of the tools featured in the IBE suite, followed by a description of the architecture used and a more detailed explanation of the design and programming concepts used in the development of each tool.

As well as the client side tools, there is also a simple server side application that is related to the setup and extract (section 2.2.2) parts of the IBE system. For documentation, refer to section 3.8.

3.1 Overview of IBE Tools Suite

The IBE Tools Suite, as it is referred to, is a collection of client side tools for performing the basic tasks of encryption and decryption under IBE. The tools are:

IBE Library The IBE library is a Microsoft COM library that provides an interface to the IBE encryption functions, through higher level function calls.

Microsoft Outlook Add-In A simple COM object that interfaces between Microsoft Outlook and the IBE library, enabling users to encrypt and decrypt email messages.

Windows Shell Extension Provides built in encryption and decryption to the windows shell.

IBE Wizard An application, developed to test the library, that provides a simple user interface for encrypting and decrypting files.

The basic premise behind the IBE tools is to provide a simple, intuitive, user interface, abstracting away from the underlying implementation details. This allows the encryption tools to be built into any application requiring them, through an add-in component.

To achieve the flexibility required to allow the incorporation of the tools suite into any application, the architecture relies extensively upon Microsoft COM (Component Object Model, see section 1.3). This allows any application that supports OLE Automation to have a simple interface to the Tools Suite, therefore providing built in encryption.

The system architecture was designed in three tiers, each operating through an interface to the next. The tiers are explained in section 3.2.

3.2 Three Tier Architecture

As briefly described in section 3.1 the architecture on which the encryption tools are based is in multiple tiers.

User interface feedback is an important characteristic of all applications. It should always be responsive and should never suspend without painting for extended periods. With this in mind, the encryption process is controlled from a dedicated thread. This was the primary reason for implementing a multiple tier architecture. An additional benefit of this feature, cancellation of the process is possible, with all rollback and cleanup handled by the internals of the IBE library.

The main encryption runs in the bottom tier, hidden away from the user in a separate thread.

The middle tier contains the control logic, sometimes referred to as the *business logic*. This tier allows for the scheduling and de-scheduling of encryption tasks, known as contexts, and the control of the actual encryption process. The lower two tiers are contained in a COM library exposing the necessary middle tier functions allowing for external applications to interface with it.

The third, or topmost tier, is where the user interacts with the library. This is the responsibility of the client application, and each of the three main tools in the IBE Tools Suite offer different ways in which the user can interact with the library.

Figure 4 shows this architecture.

3.2.1 Middle Tier Logic

The abstraction of the control process to a middle tier allows for an integrated solution to a third party application.

The reuse of the encryption engine is enabled by the separating of the application logic from the user interface and placing it in a publicly accessible central tier. This conceals

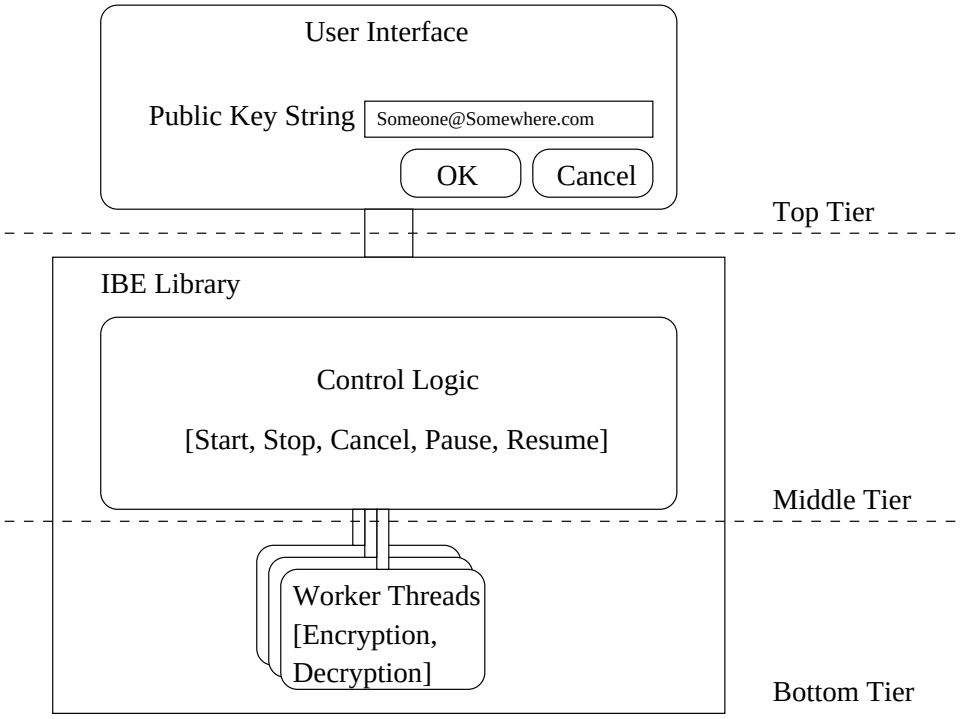


Figure 4: This diagram shows the multiple tier architecture that forms the IBE Tools. Each user tool (Shell Extension, Outlook Add-In, etc.) implements the third tier, or User Interface tier. The IBE Library forms the lower two tiers, the control logic and the worker threads. This architecture enables developers to concentrate solely on the tier in which they are developing, maintaining the interfaces to the other tiers.

the implementation details, presenting a clear set of interfaces and the ability to easily integrate it with other environments.

3.2.2 Architectural Features

Distributed Environment Because the IBE class neatly encapsulates the encryption process, there is only ever the need to create a single instance of it on any host. Because the architecture relies heavily on COM, a client can transparently connect to any host running the IBE service, providing a truly distributed environment.

For this feature to be implemented properly, some changes would be required, notably dedication of a single thread to each individual encryption or decryption task. Another change in the architecture would require communications between the top and middle tiers to somehow be secured, so that an attacker could not intercept remote procedure communications in a distributed environment, giving them access to pass phrase information and unencrypted data.

Concurrent Execution This architecture allows for the simultaneous processing of multiple files, as it would be very simple to handle each job in a separate thread. The current implementation only supports concurrent encryption and decryption, no testing has been performed with multiple encryption or decryption threads running simultaneously.

3.3 IBE Library

The IBE library is a Dynamic Link Library (DLL) for the Microsoft Windows operating system that provides an interface to the necessary methods required to encrypt and decrypt files, text or streams. Utilizing Microsoft COM (section 1.3), the library is a server for the IBE routines, allowing any application supporting OLE automation to interface with them, providing encryption as an application extension.

A job is scheduled by the passing of the context information to the control thread. Once all jobs are scheduled, the process is initiated by a call to the library, and the encryption thread is spawned. Once a job is complete the resulting data is immediately available to the client by simply retrieving the job data from the library. This architecture model allows for multiple jobs to be scheduled and then batch processed by the encryption engine.

Figure 5 shows a UML[3] diagram of the library structure.

[3]UML: Unified Modeling Language

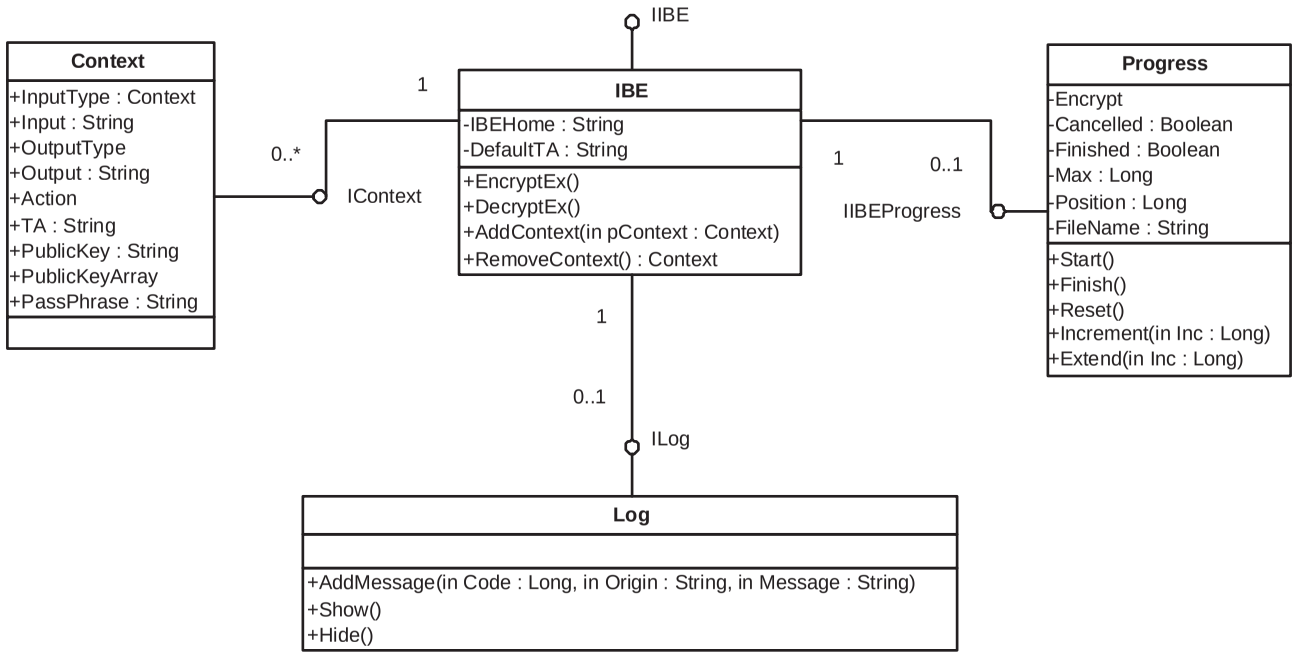


Figure 5: The core object, *IBE*, controls the transfer process and encapsulates the actual IBE implementation. The three other classes, *Log*, *Progress* and *Context* (see section 3.3.2) are objects that are created by the client application. Contexts are encryption and decryption tasks that have been scheduled, and a list is maintained by the IBE object. The Progress and Log objects are user interface features to be implemented by the client and assigned to the IBE object through the appropriate properties.

3.3.1 IBE Library Interfaces

The IBE Library provides three interfaces, **IIBE**, **IIBEProgress** and **ILog**, referred to as the IBE, Progress and Log interfaces respectively.

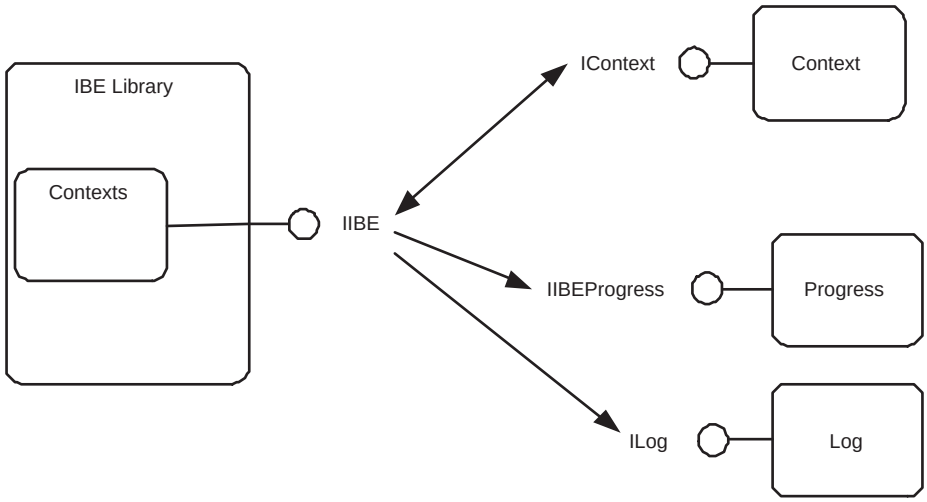


Figure 6: The interfaces exposed by the COM library are shown in the above figure. The arrows represent interaction from one object to the other, through the interfaces. There are 4 interfaces, although 2 are objects that are implemented by the library and 2 are abstract, in that they have no implementation as that is the responsibility of the client application.

IBE Interface This is the core interface of the library, providing access to the encryption and decryption algorithms of the IBE library. It is a very simple interface with few methods, however it forms the core of the middle tier control logic, as well as the bottom tier worker threads.

The IBE library works by scheduling encryption and decryption jobs, or *Contexts* (see section 3.3.2), and then initiating the encryption or decryption process by a call to the appropriate routine.

Progress Interface This interface is primarily for the client application to implement. The purpose is to provide feedback to the user informing them of encryption progress. A default implementation is provided by the IBE Library, as a simple dialog box displaying a progress bar that is incremented by the central tier, as well as the option of cancelling the operation through a button on the dialog.

Name	Type	Required for	Description
InputType	ItemType[4]	Both[5]	Represents the type of input the process is to work on, either a MIME encoded string or a filename.
Input	BSTR[6]	Both	Either the MIME encoded source text or a filename, including full path, for the source data.
OutputType	ItemType	Both	Represents the type of output the process is to work on, either a string or a filename.
Output	BSTR	Both	Either the filename, including full path, for the target data, or an empty string (if the target is a string output).
Action	Method[7]	Both	Describes what kind of action to be performed, eitehr Encryption or Decryption.
TA	BSTR	Both	The Trusted Authority to use for encrypting or decrypting this context. If left empty, the default TA is used.
PublicKey	BSTR	Encryption	The public key string ID the source should be encrypted to.
PublicKeyArray	Variant[8]	Encryption	A SafeArray[9] of all public keys (as BSTRs) the source should be encrypted to.
PassPhrase	BSTR	Decryption	The pass phrase that protects the encrypted private key file.

Table 2: This table lists all the properties on the context object.

Log Interface Again, this interface is the responsibility of the client application developer to implement, however a simple default implementation is available through instantiation of the object in the IBE Library.

3.3.2 Encryption/Decryption Contexts

Contexts are simple objects that contain all the information required about a job to execute it. A context is made of several properties ranging from the source text or filename of the document to work on through to the pass-phrase required for decryption. A complete list of the properties can be found in table 2.

[4]Enumeration ItemType: Values can be ibeFile or ibeString

[5]Both: Required for Encryption and Decryption

[6]BSTR is a COM datatype for strings, that supports multibye characters and Unicode

[7]Enumeration Method: valid values are ibeEncrypt or ibeDecrypt for encryption or decryption respectively

[8]Variant is a generic data type

3.3.3 Encryption Process

The encryption process for a single document takes place in multiple stages, and is described in figure 7.

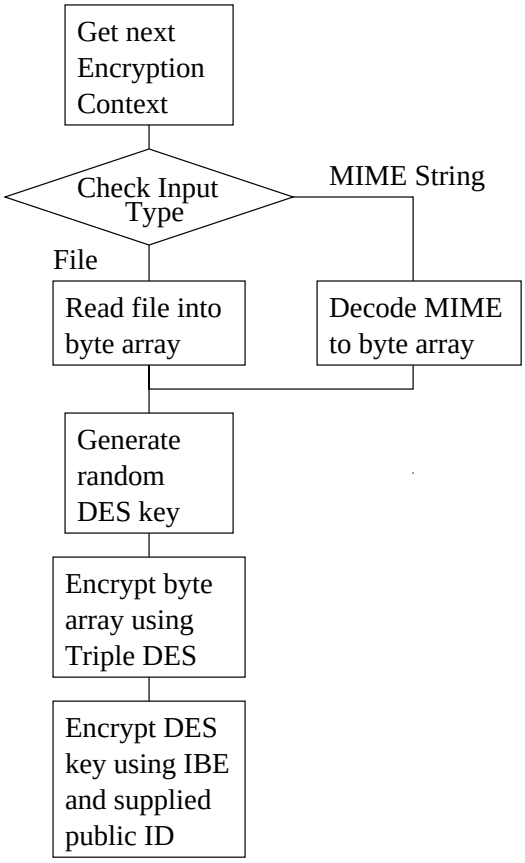


Figure 7: This diagram shows a flow of execution when encrypting a string. Only the main stages are shown each is described in more detail below.

Setup: The setup phase for the Encryption process consists of loading the source text, either from a MIME encoded string[10] or the specified file and generating a DES (Digital Encryption Standard) key.

The DES algorithm used was taken from the Crypto Plus Plus library [11], which has been used throughout this project implementation.

[9]SafeArray: A COM array that can be used across all COM languages

[10]The reason for MIME encoding the string is because it is otherwise impossible to pass binary data between COM objects easily.

[11]Crypto Plus Plus is a very robust, professional standard cryptographic library written in C++, available for free use in commercial applications.

The DES key is generated using the Microsoft Crypto API functions that form a part of the Microsoft Windows operating system. The randomness generator that forms part of this API is cryptographically random, and generates random data from hardware interrupts, jitters and CPU flag states, to name some of the sources. 64 bits are read from this source and used as the DES key.

Encryption: The encryption process is very simple. The randomly generated DES key is used to encrypt the input data using the Tripe DES algorithm (section 2.3.1), and then is itself encrypted using the IBE algorithm under the supplied public key string.

Output: The output is either written to a file, if specified by the context parameters, or MIME encoded to a string, and returned through the context object. The IBE file format is explained in figure 3.3.5. When MIME encoded, the file format is converted from a raw data array to a MIME encoded BSTR.

3.3.4 Decryption Process

The decryption process is the reverse of the encryption process, and the flow of execution is shown in figure 8.

3.3.5 IBE File Format

IBE encrypted data files are identified from the signature at the top of the file, figure 9 shows the byte sequence that is used as the signature. This signature both identifies the file as an IBE file and provides for immediate detection of common file-transfer problems. The first two bytes distinguish IBE files on systems that expect the first two bytes to identify the file type uniquely. The first byte is chosen as a non-ASCII value to reduce the probability that a text file may be misrecognized as an IBE file; also, it catches bad file transfers that clear bit 7. Bytes two through four name the format. The CR-LF sequence catches bad file transfers that alter new line sequences. Byte 7 identifies the file as a data file, key files are represented by a 'K' in the place of byte 7. The control-Z character stops file display under MS-DOS. The final line feed checks for the inverse of the CR-LF translation problem.

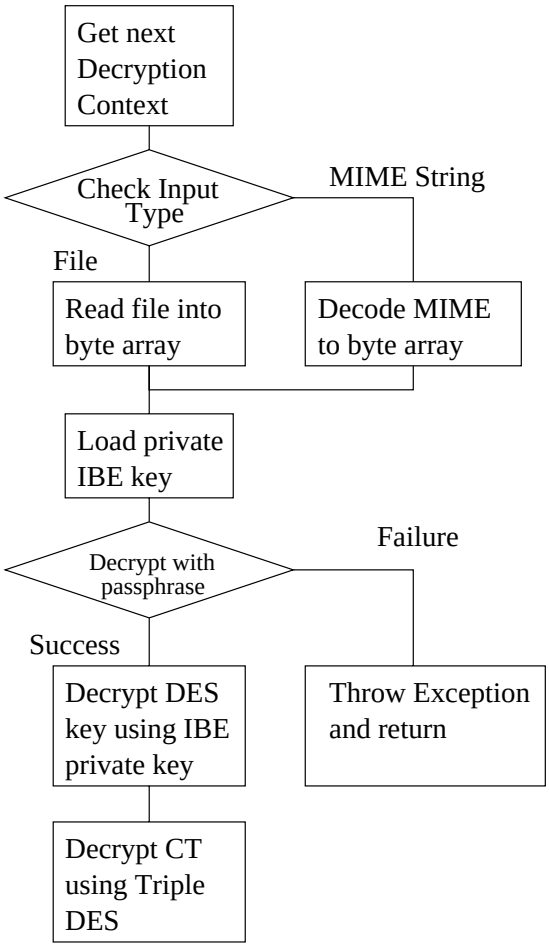


Figure 8: This diagram shows a flow of execution when encrypting a string. Only the main stages are shown each is described in more detail below.

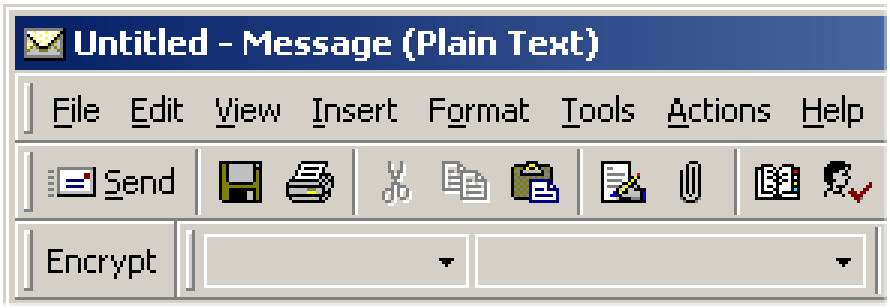
3.4 IBE Outlook Add-In

The Outlook Add-In developed as part of this project is very simply an interface to the IBE COM Library. Figure 10 shows the addition of the Encryption option to the Compose Message Window.

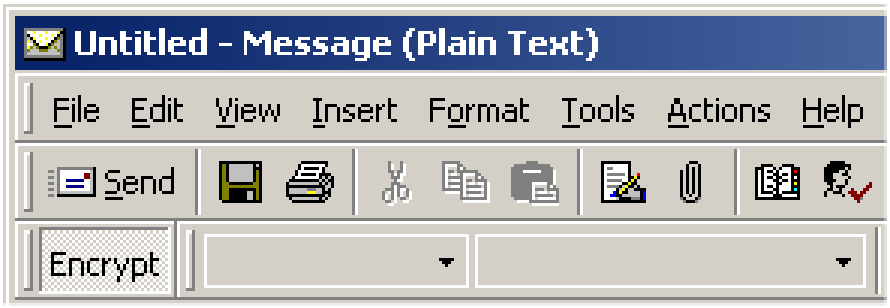
To encrypt a message, the sender simply has to select the button, this will flag the message for encryption. When the send event is raised, by the user hitting the send button, the IBE Add-In intercepts this and performs a single encryption of the data, using a random DES key, and encrypts the DES key, using IBE, to each of the recipients and the sender. A single copy of the encrypted message is saved in the sender's *Sent-Items* folder. This way, only one email is sent and all desired recipients, and the sender, can read it. On receipt of an encrypted message, the recipient can only view it by opening the message. Viewing in the preview pane will only display the MIME encoded message.

(byte)	1	2	3	4	5	6	7	8	9
(decimal)	137	73	66	69	13	10	70	26	10
(hexadecimal)	89	49	42	45	0d	0a	46	1a	0a
(ASCII C notation)	\211	I	B	E	\r	\n	F	\032	\n

Figure 9: This format is modified from the PNG format to suit IBE files and keys.



(a) Encryption Not Selected



(b) Encryption Selected

Figure 10: Modifications to the Outlook Compose Message Window. When a message has been selected for encryption the “Encrypt” button state changes to depressed, as shown in subfugre 10(b)

.

When opening an encrypted email, the recipient is prompted with a dialog box, shown in figure 11, where the pass phrase under which their private key is encrypted is requested. Once successfully entered, the message is displayed in the original state.

Design of the add-in required the identification of events to handle that would enable encryption of outgoing emails and decryption of incoming emails. These were identified in section 2.4.2.

3.4.1 Composing and Reading Mail Messages

The compose window was required to be modified so that there was some way for a user to select the message for encryption, by toggling a flag associated with the message. This

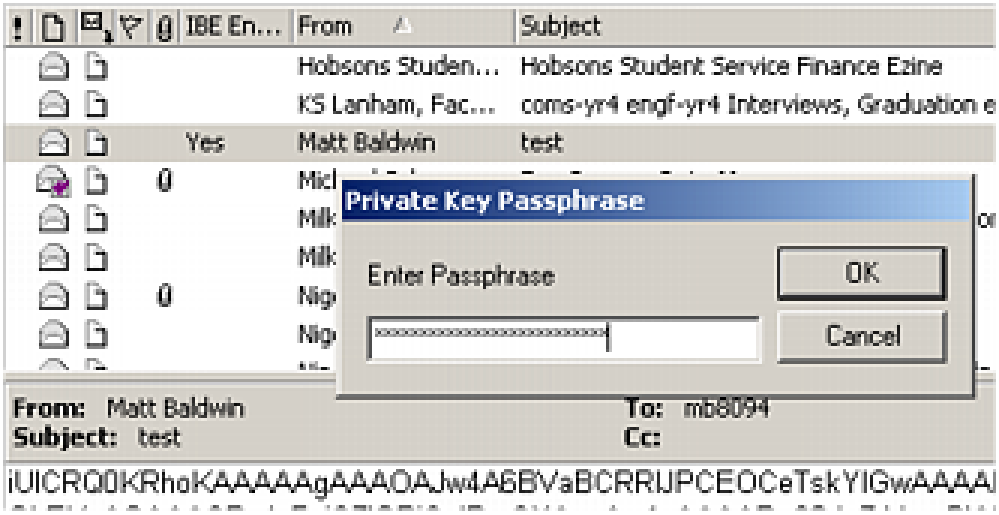


Figure 11: The user is prompted to enter his or her secret pass phrase that was chosen to encrypt the message. Unsuccessful entry of the pass phrase will result in the opening of the message failing.

was achieved by identifying when a message was being composed and adding a tool bar button that enables encryption (see figure 10).

To enable this tool bar, the add-in intercepts the **OnNewInspector**[12] event issued when an item of any kind is opened, either for composition or reading. The process is outlined in figure 12.

3.4.2 Encryption Tool bar

The Encrypt button that is added to the tool bar has an event handler which toggles the encrypt flag on and off depending on the user toggling the button itself.

3.4.3 Sending Mail

When a mail item is sent, the **OnItemSend** is raised by Outlook and intercepted by the add-in. The event is triggered for all items that are sent, so the same check on the inspector for a mail item is carried out as before. If the item is a mail item then the encrypt flag is checked, and if set the message is encrypted to all recipients and the sender. This is so the

[12] An inspector object represents the window in which an Outlook item is displayed.

- If the inspector is for a mail item:
  - If the mail item is not encrypted:
    - Create tool bar
    - Add Encrypt button
    - Set Encrypt flag to FALSE
  - Else
    - Prompt user for pass phrase
    - Attempt to decrypt message
    - If Decryption succeeds
      - Display Plain-Text message
    - Else
      - Display Cipher-Text message
    - EndIf
  - EndIf
- Else Ignore
- EndIf

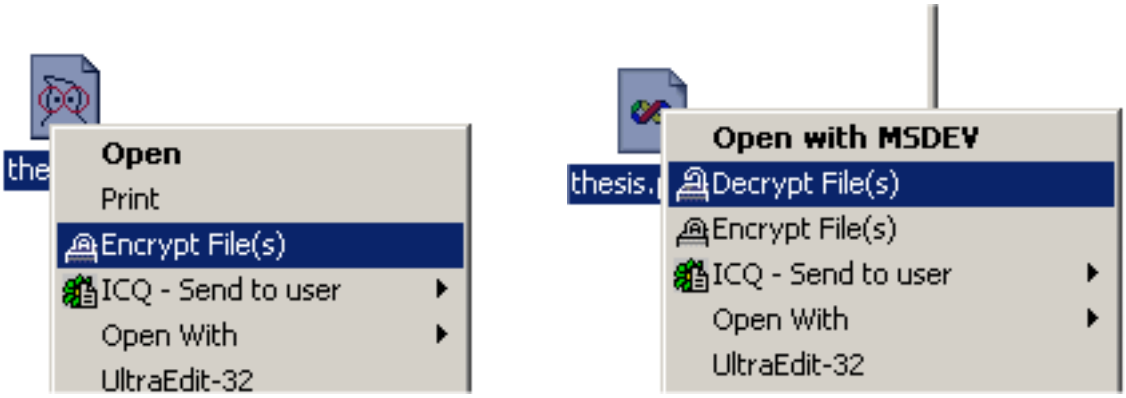
Figure 12: This pseudo code shows the process involved when a new inspector object is created by Outlook. This handles the setting up of the encryption flags, the tool bar and the decryption of encrypted messages.

sender can read it again if necessary.

3.5 IBE Shell Extensions

As explained section 1.3.1 there are many types of shell extension that are supported by Microsoft Windows. This particular extension is a Context Menu extension, which allows the user to right-click on any file and select *Encrypt File(s)* from the menu.

Figure 13 shows the right click menu on a file with the shell extension installed. As well as working on single files, the application will also encrypt multiple files. The shell extension also supports decrypting encrypted files, and the user interface change is similar to that for encrypting files, and can also be seen in figure 13.



(a) Encryption (b) Decryption

Figure 13: The new context menu with encryption and decryption support supplied from the IBE Shell Extension.

3.6 IBE Wizard

The IBE Wizard was designed for the purpose of providing an easy way to test the IBE Library. Developed in Visual Basic, for speed and simplicity of development, it is comprised of a simple user interface that communicates using COM with the IBE Library. This application fully demonstrates the three tiered architecture, as it implemented the entire upper tier. Figure 14 shows several user interface features, the progress window and the pass phrase dialog.

3.7 Setup/Configuration Program

A simple installation program has been developed to allow a user to install the IBE Tools Suite on a Microsoft Windows 95 or greater operating system. The package ships with the following components:

- IBE Library;
- IBE Shell Extension;
- IBE Outlook 2000 Add-In;
- IBE Outlook XP Add-In;
- IBE Wizard;
- IBE Setup program.

Although not discussed previously, the IBE Setup program allows a user to specify the Home directory for storing IBE keys, and will install any key opened with it to that location. The package is available from *http://www.cs.bris.ac.uk/ mb8094/IBE/*.

3.8 Server Tools

The Server software, also known as the Trust Authority, runs as a simple CGI script on a web server. The script requires 3 parameters to it, passed through an HTTP PUT request. The parameters are:

- Email Address - The email address for which a private key has been requested;
- Pass phrase - a pass phrase of any length with which the private key is to be encrypted under;
- Pass phrase Confirmation - to ensure the user remembers the pass phrase, the CGI script requires it to be confirmed.

The first time the script is run, the TA generates the parameters for the system and a random private key of it's own. This private key needs to be persisted in some form so that it can be re-loaded every time the script is run.

The key is currently persisted in a flat file format in the current working directory - something that is not very secure, despite it being encrypted, as it is simple to recover the file and start a brute force attack on it.

With the system parameters, the TA's private key and the email address of the user, the system can then generate a private key for that email address. Once generated, the key is encrypted using Triple DES under the pass phrase supplied. The output from the DES encryption is encoded in MIME format so that it can be transmitted by email.

Sendmail is invoked to send an email to the email address supplied. The email has two attachments:

1. The private key;
2. System parameters and TA public key.

The email body informs the user that these be placed in their *IBE Home* directory, which would have been configured when the IBE tools were installed.

The System Parameters file, also known as the TA's public key, is not encrypted, as it is public knowledge. The name of the file also specifies the Trust Authority and follows the following format:

TA-<trust authority name>.ibek

So, for the University of Bristol Computer Science Department, the TA file would be named:

TA-cs.bris.ac.uk.ibek

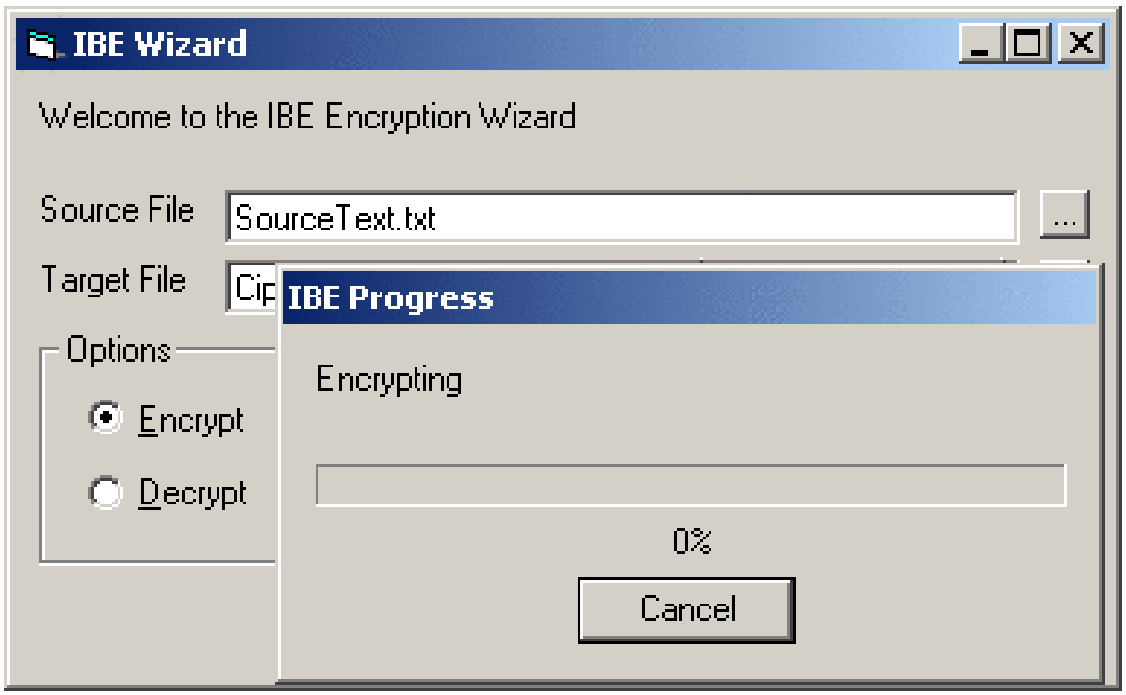
Private key files are always called *private.ibek*, however this assumes the user only has one private key. As users will also belong to mailing lists, for example for each course they study, it would be necessary to provide private keys for those lists so that they can decrypt email distributed to the group. To enable this, either multiple key files would be required, or a way to store multiple keys in one file.

3.9 Maintenance of Code

The code that forms the technical implementation of this project is very well documented using comments. A design and implementation document that would be of use to anyone wishing to maintain or adapt the code is available for the IBE Library[2]. The client applications which make use of this library are not documented in such a manner as the comments in the code do a sufficient job. The IBE Library documentation covers many of the aspects discussed in this report, and is effectively a tailored version of this section.

Anyone wishing to maintain the code would learn a lot from reading this section, perhaps more so than from reading the IBE Library documentation, as this report covers the client applications as well as the IBE Library.

Due to the intellectual property rights, the source code to the IBE methods, provided by HP, are not available for public download, only precompiled binaries (stripped of debugging info) are available.



(a) IBE Wizard User Interface - Encryption

(b) IBE Wizard User Interface - Pass phrase Dialog

Figure 14: The UI is very simple, requesting input and output filenames, the type of operation to perform (Encryption or Decryption) and the public key string which to encrypt to (for encryption only). The modal pass phrase dialog, shown when decrypting a file, prompts the user for their pass phrase. If incorrectly entered the process will simply fail, with an error message.

4 Current Status and Future Plans

The technical implementation that forms a part of this project is more than just a simple proof of concept, as the architecture has been fully designed, and to a large extent fully implemented. However, the code that drives the encryption, the IBE algorithm, that is used is supplied by Hewlett Packard purely to prove the system works. The University of Bristol Department of Computer Science is currently developing an improved algorithm that is to be used in a commercial implementation of the email security.

4.1 Client Tools

The software developed that forms the client suite of tools is largely complete. Some user interface features and improved functionality are all that is missing to make this a fully working solution.

4.1.1 Functionality Enhancements

Email Attachments: As the Microsoft Outlook Add-In currently stands, there is no support for encrypting email attachments. This is a major piece of functionality which needs to be implemented for any commercial versions of the software.

Digital Signatures: The main piece of functionality that is missing the ability to digitally sign outgoing emails. This is feature is core to the application, as digital signatures provide an extra level of security by authenticating the message sender. The algorithm to be used for digitally signing emails using IBE is outlined in figure 15.

4.2 Server Tools

The server side tools developed for this project are by no means the final version, and were developed as they are simply for the purpose of distributing keys in a quick and easy solution.

A full solution may require a separate application that connects over a secure connection

Signing

$$\begin{aligned} r &= \widehat{t}(P,P)^k \\ v &= H(m \oplus r) \\ u &= vS_{ID} + kP \end{aligned}$$

The signature on message m is (u, v).
Verification:

$$\begin{aligned} s &= \widehat{t}(U,P)\widehat{t}(Q_{ID}, -Q_{TA})^v \\ & \quad \text{Accept the signature if and only if:} \\ v &= H(m \oplus s) \end{aligned}$$

Proof:

$$\begin{aligned} s &= \widehat{t}(u,P) \times \widehat{t}(Q_{ID}, -Q_{TA})^v \\ &= \widehat{t}(vS_{ID} + kP, P) \times \widehat{t}(Q_{ID}, -Q_{TA})^v \\ &= \widehat{t}(vS_{ID}, P) \times \widehat{t}(kP, P) \times \widehat{t}(Q_{ID}, -rP)^v \\ &= \widehat{t}(S_{ID}, P)^v \times \widehat{t}(P,P)^k \times \widehat{t}(rQ_{ID}, P)^{-v} \\ &= \widehat{t}(S_{ID}, P)^v \times \widehat{t}(P,P)^k \times \widehat{t}(S_{ID}, P)^{-v} \\ &= \widehat{t}(P,P)^k \\ &= r \end{aligned}$$

Figure 15: IBE Digital Signature Scheme proposed by Florian Hess

to a server, as opposed to using a web server with CGI scripts. As this project concentrated mainly on the client applications of IBE very little thought has gone into how such an application would work.

4.3 Identities and Keys

IBE allows any string to be a valid public key, something that is a very powerful feature as it allows for more control over the key. An example is the concatenation of a date to the key. This date could represent the date after which anything encrypted using the key could be viewed. Or, conversely, some kind of control could be built in that specifies the date when the key expires, making key revocation much easier, as it is automatic.

The current Private Key Generation mechanism relies on a single Trust Authority, however the IBE scheme used allows for a distribution of this trust amongst many authorities. This key escrow provides more security as all parts of the key, from each TA, are required

to form the entire private key.

To extend the power of the key beyond a simple email address, a program that enables a user to construct the key from a set of parameters, such as place, date, time, user, and selecting the TA's to use, would be required. The user program would communicate with a server application which resides on public servers for each of the TA's.

4.4 Other Applications of IBE

Identity Based Systems are very powerful and have many potential applications, from email to securing wireless networks. Email was the original motivation behind the development of such a system, and is the obvious application of the technology.

Another use for an IBE system would be the development of a new Secure Socket Protocol, that uses machines public identities when establishing the connection and performing the key session key exchange. Figure 16 shows the current Handshake Protocol as defined by the SSL standard.

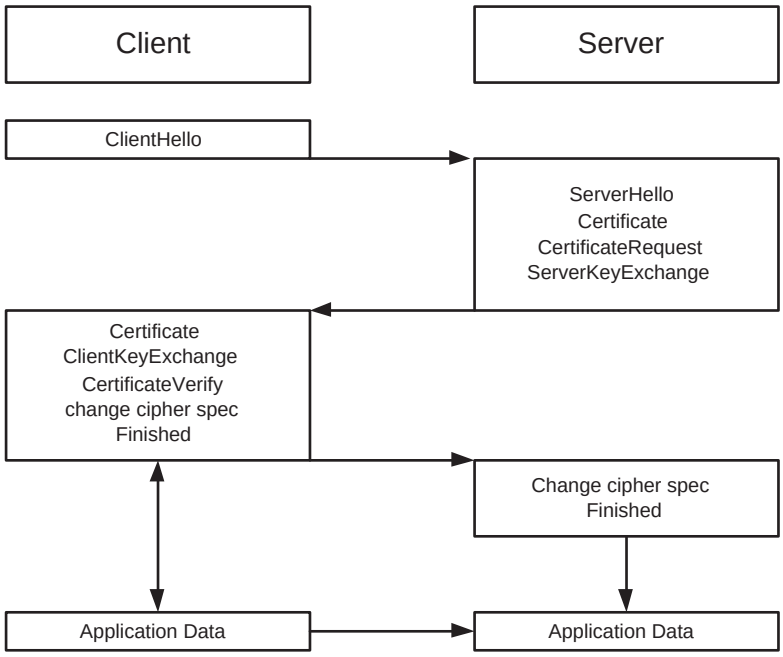


Figure 16: The client initializes communication with the server by sending the ClientHello message, which negotiates the security parameters of the connection. Once the public keys and certificates are exchanged the session key is exchanged by one of several key exchange protocols (see [5]). The session key is used for encryption of the application data.

A revised version of this protocol might work as shown in figure 17. In this protocol,

the handshake requires that only the system parameters (which would be signed using the servers private key for verification using the public identity) be transferred. The key exchange could be carried out by encrypting the session key using the public identities, over some key agreement protocol as described in [5].

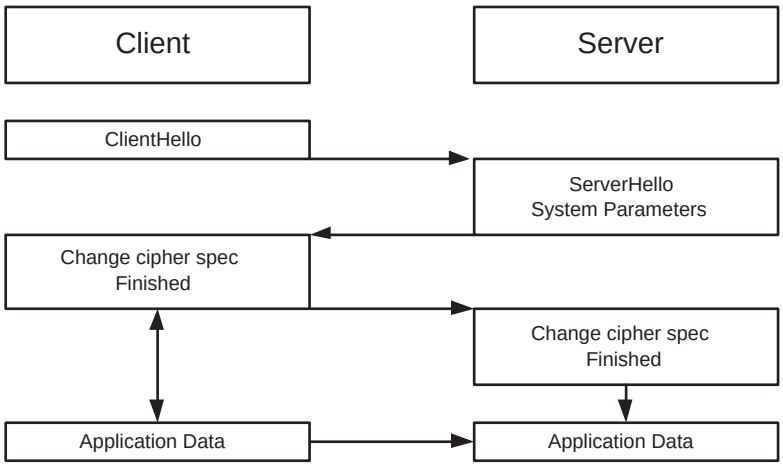


Figure 17: This revised handshake protocol adds the benefit of not having to verify the public key and certificate of the server. As the current protocol requires that either A. the certificate be produced from a trust authority, or B that the user explicitly accepts the certificate, there is a cost of time involved. By removing this, the process becomes much quicker and much simpler.

A Lists of Tables, Figures and References

List of Figures

List of Tables

References

[1] *Microsoft Developer Network*. Microsoft Press, 6.0a edition, 1997.

[2] M. G. Baldwin. “ibe tools suite: Ibe library”, 2002.

[3] I. Blake, G. Seroussi, and N. Smart. *“Eliptic Curves in Cryptography”*. Cambridge University Press, 1999.

[4] D. Boneh and M. Franklin. “identity-based encryption from the weil pairing”, 2001.

[5] A. O. Freier, P. Karlton, and P. C. Kocher. “the ssl protocol”. Technical report, 1996.

[6] G. Frey and H.-G. Rück. “a remark concerning m-divisiblity and the discrete logarithm problem in the divisor class group of curves”, 1994.

[7] S. D. Gallbraith, K. Harrison, and D. Soldera. “implementing the tate pairing”, 2002.

[8] D. Hankerson, J. Hernandez, and A. Menzes. “software implementations of elliptic curve cryptography over binary fields”. 1999.

[9] A. Joux and K. Nguyen. “separating decision diffie-hellman from diffie-hellman in cryptographic groups”.

[10] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, revised edition, 1996.

[11] A. Menzes, T. Okamoto, and V. S.A. “reducing elliptic curve logarithms to logarithms in a finite field”. *IEEE Trans. Inf. Theory*, 1993.