Implementation of an Identity Based Encryption sub-system for secure e-Mail and other Applications

Neil Costigan

Bachelor of Science (Computer Applications)

A dissertation submitted in partial fulfilment of the requirements for the award of

MSc. (Research) Computer Science.

to the

Dublin City University

School of Computing

Supervisor: Dr. Michael Scott

July, 2004

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of MSc. (Research) is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed 

Student ID 52165809

Date July, 2004

Acknowledgments

I'd like to thank my supervisor, Mike Scott, who took the brave decision to take this mature, under-qualified candidate into his research group. I hope the trust has been repaid.

My colleague Noel McCullagh who answered my countless trivial questions in a manner befitting his endless patience. A true gent.

Barry, Claire, David, Dru, Gav, Hego, Karl, Niall, Noreen, Rohan, Tunney, & the countless others who graced the post-grad Dilbert zone.

Enterprise Ireland who sponsored the candidates research work under RIF fund.

My sister Carolyn for taking the time to edit many drafts, my brothers Paul and David for constant encouragement.

My former partners and colleagues at Celo Communications & Gemplus who, in a way, paid for this.

Residents and friends of :

Rathnapish :- Carlow. Collins Ave, 123 Swords Rd, Grace Park Heights, Dalcassian Downs, Russell Avenue, 147 Pearse Street, 65 Dame Street & 25 Percy Lane :- Dublin. Finsbury Park, Brixton, Archway :- London. Comm Ave :- Boston. Uppsalla. Södermalm, Birkastan, Stora Essingen, Årsta, Norrviken, Hammerby SjöStad :- Stockholm. Geary&Laguna, Bay&Stockton : - San Francisco. Avenue des Infirmeries :- Aix-en-Provence. Kalviksholm :- Luleå.

Mum & Dad for encouraging us to stick to the books regardless of the cost to themselves. I hope they still think it was worth it.

In Memory of my brother Joe who certainly livened up the first attempt at a degree.

To Vicki.

For suffering me and our many re-locations :- I'll write the next thesis in Swedish. I promise.

Abstract

This thesis describes the requirements for, and design of, a suite of a sub-systems which support the introduction of Identity Based Encryption (IBE) to Internet communications. Current methods for securing Internet transmission are overly complex to users and require expensive and complex supporting infrastructure for distributing credentials such as certificates or public keys. Identity Based Encryption holds a promise of simplifying the process without compromising the security. In this thesis I will outline the theory behind the cryptography required, give a background to e-Mail and messaging protocols, the current security methods, the infrastructure used, the issues with these methods, and the breakthrough that recent innovations in Identity Based Encryption hopes to deliver. I will describe an implementation of a sub-system that secures e-Mail and other protocols in desktop platforms with as little impact on the end user as possible.

Contents

Chapter 1

Introduction

1.1 Background

The original Internet (DARPANET) developed by the US Department of Defence in 1969 was never intended to be the global disparate network it has evolved into. Its design was modest. Its developers envisaged a respectful, honest, user-base where privacy and security was based on trust. The network was build for military purposes, to design a data network that withstood outrages such as lose of nodes from military attack. This rapidly grew into the Internet, based on TCP/IP protocols, as we know it today. The designers never foresaw the evolution into a communications network which is a fundamental part of modern society where academic, commercial, and 'ordinary' users rely on it for much more than can be supported by its insecure basic structures.

Unfortunately this growth has introduced as communities of users intent on abusing the freedoms and trust given to others. The commercial reality is that business and personal use of data networks required security before they will be adopted. Since the internet's inception there has been a rapid rise in the number of attacks by so called 'hackers' against personal and business interests. A recent BBC news article [5] claimed that the total hack attacks for the first eight months of 2002 reaching over 31,000 - more than the total for the whole of 2001 with conservative projections suggesting there could be up to 45,000 hack attacks across the globe in 2002.

The original set of internet protocols were designed to be simple to implement, low on bandwidth requirements, and robust. They included remote terminal services, simple file transfer and a form of messaging. All of the protocols transmit the data payloads in plain text easily readable by anyone who can intercept the traffic. None of the original protocols included security features, later revisions added in simple user password authentication, less for the security benefits, but more for multi-user convenience.

Computing power at that time didn't allow for inclusion of computationally expensive real time encryption and even options to include such security features didn't distract from

the higher priorities of robust, easy to implement protocols.

1.2 Requirements

At project start the desirable design requirements where that the code be portable, the application be lite-weight, to have low impact in implementation, that we secure with IBE (and AES) at least email POP3 & SMTP and we attempt to support all email clients without any requirements on servers.

Chapter 2

Crypto 101

This chapter outlines the background to the mathematics of cryptography, details a number of the standard algorithms, then evolves to a description of Identity Based Encryption (IBE).

2.1 Cryptographic concepts

As a background to cryptography and an introduction to security protocols. I'd like to provide an historic example of an algorithm called the Caesar Cipher attributed to Julius Caesar. The algorithm is a very simple but effective example of the principles involved in the encryption of messages.

In the Caesar cipher the algorithm is a simple symbol swap. All the letters of the alphabet A through W are substituted with the character three places after it in sequence, with X, Y and Z been represented by A, B, C. Hence A is represented by D, N by Q etc. For example the message "ATTACK GAUL" can be encoded to "DWWDFN JDXO". A simple backward step of subtracting 3 places lets one arrive at the original message. This is a crude cipher on which the success of keeping the content secret depends on the casual observer having no knowledge of the algorithm (the simple alphabet switch) and the offset (3). Simple improvements include having a jumbled up alphabet (A=B ,B=Z, C=N etc.) as a look-up table with both sides knowing the new lookup table and possibly having an increment on the offset in some formula also agreed by the participants. The fundamentals of modern cryptography build on these simple ideas of message, algorithm and key.

Messages can be transformed to numbers via simple ASCII (American Standard Code for Information Interchange) representation of the characters making up the message [23]. This Is where each character/letter of the message is represented by a well know number ('a'=97, 'b'=98, 'z'= 122 etc.) and that in computers these numbers are represent by binary (1 or 0) bits.

Operations which transform these ASCII numbers are equivalent to transforming the original message they represent. There are other methods of cryptography but the mathematical methods described below are believed to be the strongest. Modern cryptography relies upon the mathematics of making transformation operations, which are hard to invert, even when one knows the transformation used e.g RSA or the solving of the discrete logarithm problem El Gamal etc. It is the hidden or secret parameters of the operation that provide the security.

2.2 Random numbers on computers.

Computers are by their nature deterministic finite state machines and thus cannot generate truly random numbers. Strong random numbers are crucial to the security of the mathematics used in cryptography. Random numbers are often used as keying material to encryption algorithms. So non-randomness can result in predictable keys and lead to security weaknesses.

To overcome this limitation various techniques are used to substitute for the deterministic issues and the making of secure random number devices has evolved and created a commercial market all of its own. Most dedicated cryptographic hardware will contain a random number generating device.

Common sources of randomness derive from random physical sources such as white noise input on sound cards, samples of raw Ethernet traffic, contents of memory page files, even a dedicated co-processor on the Intel Pentium 4 (now discontinued), all of which are deemed to be relatively hard to guess.

Random devices rated for military grade use are certified by the US Government's standards bureau NIST [44] to FIPS [46] standard and usually contain some Geiger counter inputs which detect radioactivity as an source of randomness.

Randomness is such an important concept inside a security protocol that a deeper discussion is merited 2.3.6 on page 8 and a section 5.11 on page 75 is dedicated to the methods used by our sub-system.

2.3 Some building blocks used in Modern Cryptography.

2.3.1 Hash functions

$$x = H(m)$$

Hash functions are functions which map (*hash*) a message to a collision free value (also called a hash value just to further confuse) and so this new, usually shorter, value can be used as a representation of the original message.

Hash functions without full uniqueness are commonly used in other fields of computer science but for cryptography its a 'must have' property that the hash value is unique.

A useful hash will hash arbitrary sized messages to a usually much-condensed fixed sized hash, typically 160 bits. Commonly used hash functions are the relatively fast MD5 [51] and SHA-1 [47]. Recently MD5 has fallen out of favor because of recently exposed weaknesses in its ability to always product a strong hash [16] and is only now used in legacy systems.

One way hashes are hash functions where given the hash of a message and the algorithm, it is impossible to find the original message.

2.3.2 Exclusive OR (XOR)

The bitwise operation XOR denoted $\bigoplus$ is identical to additional modulo 2 and can be represented as a table

y\x	0	1
0	0	1
1	1	0

The advantage to cryptography is that if you XOR a number with itself it disappears from the equation. e.g.

Message $\bigoplus$ a $\bigoplus$ a = Message

= Message $\bigoplus$ 97 $\bigoplus$ 97 = Message

= Message $\bigoplus$ 1100001 $\bigoplus$ 1100001 = Message

2.3.3 Modular arithmetic

Modular arithmetic deals with a set of integers where if N is positive then the numbers modulo N are the set of numbers $\{\ i \mid 0 \leq i < N\}$. If two numbers have the same remainder when divided by the modulo N then we say they are congruent modulo N .

An everyday example of modular arithmetic is the set of hours on a clock

$0, 2, 3, 4, 6, 7, 8, 9, 10, 11$ *mod* 12. So if it is two o'clock and we add three hours it is five o'clock as it will also be in fifteen plus two hours

$2 + 15 \mod 12 = 5.$

Two of the most popular public key algorithms use modular exponentiation as their underlying mathematical process.

2.3.4 Prime numbers

Webster's New Collegiate Dictionary defines a prime as follows:

```
Prime \'prim\ n [ME, fr.  MF, fem.  of prin first, L primus;
akin to L prior] 1 :  first in time :  ORIGINAL 2 a :
having no factor except itself and one <3 is a ~ number> b :
having no common factor except one <12 and 25 are relatively ~> 3 a :
first in rank, authority or significance :  PRINCIPAL b :
having the highest quality or value <~ television time>
```

Simply put, a prime is a number that has exactly two positive integer factors, 1 and itself.

A great description of how to find primes comes from the 2003 best selling novel "The curious incident of the dog in the night time" [26].

Eratosthenes (275-194 B.C., Greece) devised a 'sieve' to discover prime numbers. Using this method to find all the prime numbers first write down all the positive whole numbers.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	*etc.*

Then you take away all the number that are multiples of 2 , then all the numbers that are multiples of 3, then 4, 5, 6, 7 and so on (numbers known as composite numbers).

The numbers that are left are prime numbers.

	2	3		5		7			
11		13				17		19	
		23						29	
31							37		*etc.*

Prime numbers have fascinated mathematicians for centuries. The problem is no one has yet figured out a computationally efficent method to say if a number is a prime nor how to predict what the next prime number will be. The only known method is brute force (exhaustive search) i.e going through all possibilities. Picking a number large enough brings us to primes that are usable in cryptography.

2.3.5 Groups

A group is a set of numbers with an operator. This is a relatively simple group to illustrate the properties useful for cryptography. The numbers 1 to 4 and an operator being multiply $\times$ *mod* 5.

$[\times, 1, 2, 3, 4 \ mod \ 5]$

The group is closed when the answer is always within the group

$(3 \ \times \ 2) \ = \ 6 \ mod \ 5 \ = \ 1$

The group has an identity which is a number which when applied by the operator does not change the element.

$2 \times 1 = 2$

It is associative

$(2 \times 3) \times 4 \ = \ 2 \times (3 \times 4) \ mod \ 5$

Every element has an inverse

$(3 \times 2) = 6 = 1 \ mod \ 5$

2 is the inverse of 3

A group is cyclic if it has a member that when subjected to repeated applications of the operator will give every member of the set

$2 = 2 \ mod \ 5 = 2$

$2 \times 2 = 4 \ mod \ 5 = 4$

$2 \times 2 \times 2 = 8 \ mod \ 5 = 3$

$2 \times 2 \times 2 \times 2 = 16 \ mod \ 5 = 1$

If you want to find how many times to apply the operator to get 3 you simply keep applying it - a method known as exhaustive search.

$2 \times 2 \times 2 = 3 \ mod \ 5$

This is easy since this group has 4 members. The groups that we use have around

2^{512} elements. So the above approach would take too long. Therefore, the determination of x given 2^x in the group is very difficult.

The point here is that applying the operation N times is easy. Finding out how many times an operation was applied is computationally too expensive. This is a one way function.

$$y = g^x \mod p$$

The formula can be used to generate elements of the group over the field $\mathbb{F}_q$. In this context g can be described as the generator of the group. The number of elements in the group will be a divisor of $p - 1$. In our example 2 is a generator of group of order 4 over $\mathbb{F}_5$.

2.3.6 Random Numbers

As mentioned earlier, a critical function of any crypto related applications is the secure treatment of the keys used in the cryptographic operations and protocols. While most emphasis is on the secure placement / storage or transport of keys there also exists a requirement to stop an attacker gaining access to the keys by either deducing or by guessing what they could be. Even a guess which reduces the time or computation required for a brute force attack is a risk. One of the techniques used to foil such an attack is the use of ephemeral keys for any session and the generation of hard to guess keys by use of good random sources. If the generation of keys is not truly random then even the most advanced key protection methods will fail.

A computer by its very nature is a finite state machine. Any attempt to generate a truly random event is futile as any observer can just replay events until the similar state exists. One then has to rely on pseudo random events that hold statistical properties which overwhelm any attempt to pre-empt or guess their outcome. The best way would be if the computer, especially an important one in the infrastructure, had access to either specialized key generation hardware (AEP [60], nCipher [42] etc.) or can use the same techniques as such hardware by observing truly random external nature events such as room noise in a noisy environment, radioactive decay, Brownian motion in liquids etc. It is not practical for each machine to have such (expensive) hardware. So we make do with the practical inputs available and attempt to get this as close to cryptographically secure by a few tricks available to us.

Most programming languages do provide a random function which can be suitable for trivial applications such as games and statistical input. However for cryptographic purposes this function is weak. For example, consider functions based on some initial value called the seed. When given the same seed, two different instances of a program utilizing rand() will produce the same random values. In many implementations of C, if a seed is not explicitly specified, it is calculated from the current value of the system timer which is not considered genuinely random input as it is very easy to guess if you know when a protocol took place. To overcome this we can mix numerous sources of data into one virtual data source taking care that in doing so we don't weaken the outcome. The most common 'mixer' is use of a good, fast hash function such as MD5 [51] or SHA1 [47] [52].

The measure of the randomness of data is known as entropy. The definition given to us by the Handbook of Applied Cryptography [1] 'the entropy of a data source is the uncertainty about the outcome before an observation of the data source'. What we are trying to do is produce sufficient entropy so that the randomness is random enough for our purposes : to distill 160-bits of genuine randomness. Once you have enough entropy to seed a random number generator, you have enough for all keys used out of that generator. What we want to do is to gather enough entropy (the accumulator function in the language used by Peter Guttmann in his cryptographic secure architecture thesis [25]) to seed a secure

mixer (state/PRNG) which can henceforth supply us with good random data used for ephemeral keys.

There is a classic case of a weak entropy random number source being used to seed key generation. In 1995 it was discovered that Netscape (authors of one of the first widely used web browsers) was using a very small domain (current time in seconds, process number, some bit shifting and an MD5 hash of the inputs). By analyzing the potential inputs it was noticed that the values used as session keys in the critical SSL/TLS protocol [61] could be deduced from only 47-bit entropy for their 128 bit keys, and so could easily found by a brute force attack. Similar weaknesses have been found (and fixed !) in a number of applications including Kerberos [30], and the Java 1.1 SDK [40]. A good source of information on the nature and use of randomness is given in RFC 1750 [11].

So what can be measured on a PC or smart-phone that can be viewed in some way as non-deterministic and sufficient for cryptographicly secure entropy ?

There is programmable access to various time measurements (which we have seen to be weak)

- Raw samples of Ethernet traffic,
- The sound card for background room noise
- Disk for I/O measurements (page-files etc.)
- Network card for MAC identifiers (unique to each machine)
- Location of the mouse pointer as the mouse is moved.

None of the above can be taken on their own, but mixed together they can provide close enough true randomness for most applications. It has to be noted that Linux and most Unix operating systems use the steps outlined to take care of the generation of cryptographic secure entropy at the kernel layer. Linux for example can be configured so that at boot time the `/dev/random` pseudo device driver will only be active (and allow use) if the underlying device has gathered statistically enough random data to allow for its secure use. The Unix implementation uses this driver as a random source, as do common server components as the Apache [24] web server.

2.3.7 Random Oracle proofs.

To prove that public key cryptography systems are as secure at their inventors claim, one of the methods cryptographers enlist is a security proof called the *Random Oracle Model.*

Under the random oracle model it is assumed that all parties including an adversary have access to an infinite amount of idealized random functions. These oracles are like a

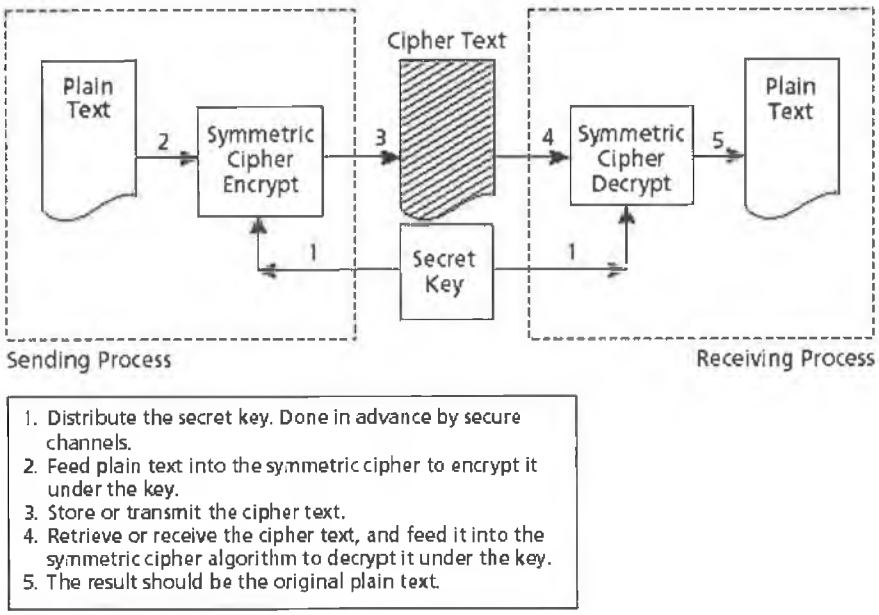

Figure 2.1: Symmetric Key *(source [17])*

Utopian SHA or MD5 functions that are deterministic, efficient, and uniform where the output is actually truly random i.e not just random against a computational attack and is collision-resistant. One then replaces the random oracle with a 'strong cryptographic hash' and assumes the security proof holds.

Weaknesses have been shown in this model and it is out of favor in some circles. While it is still commonly used as a security proof and no 'random oracle proved algorithm' has been shown to be weak unless one employes a selectively weak hash, protocols that only are provable in the random oracle are still viewed by some as open to potential weakness 'in the real world'.

2.4 Types of cryptography

There are two main types of mathematical cryptography

- Symmetric or *secret* key
- Asymmetric or *public* key

2.4.1 Symmetric Key

Symmetric key is an easier to understand concept. Essentially one party (the encryptor) uses a secret key to a mathematical function which encrypts the plaintext message to a

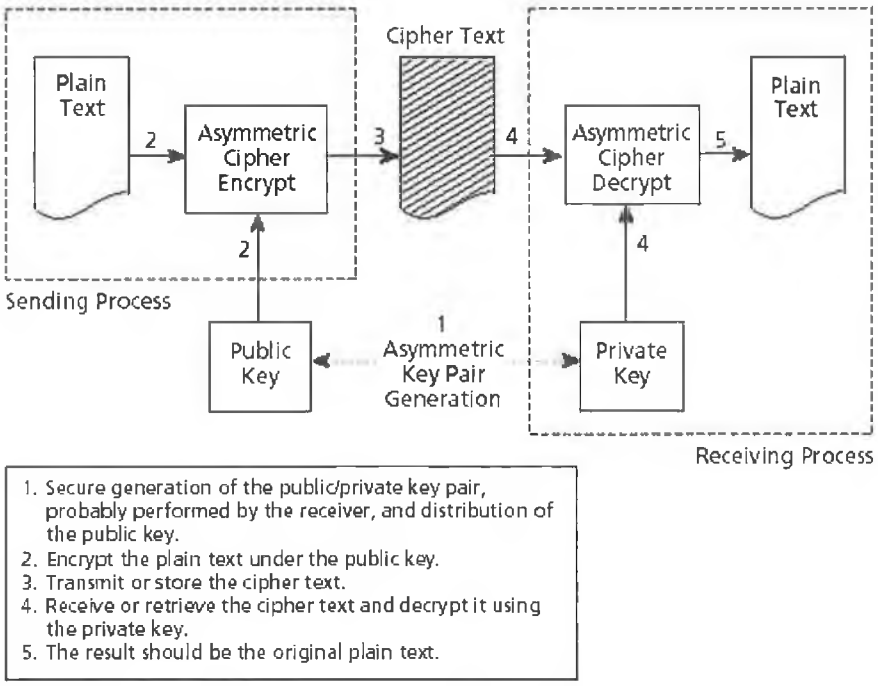

Figure 2.2: Asymmetric Key *(source [17])*

secure form. The message is passed to the decrypter who uses ***the same key*** to an inverse mathematical function which decrypts the message returning it to its ordinal plaintext. The principal issue in symmetric cryptography is the secure transport of the key between the parties. Examples include DES and its variants, IDEA, & AES.

2.4.2 Asymmetric Key

Background

Asymmetric key is a harder to grasp concept but is generally more useful to e-commerce. The key is broken up into two parts known as the *public* key and the *private* key. The public is used to encrypt the message and the private to decrypt. Asymmetric is popularly known as *public key cryptography.*

The original discovery of methods suitable for public key cryptography have traditionally been attributed to Diffie, Hellman & Merke in 1977, more commonly known as the Diffie-Hellman [15] method. However in 1997 history has corrected itself with the declassification of papers from the British secret services which lay claim to the fact the James Ellis probably invented and that Clifford Cocks probably discovered a method for public key cryptography long before Diffie et al. But, suitable to persons working inside the intelligence services, they kept the claim to themselves. GCHQ (now CESG [9]) have since backed up their claims with documentary evidence [18]. The American equivalent (NSA)

have claimed an even earlier idea but evidence is scant. Background to this interesting story is in Wired magazine [35] where Diffie (a colorful character in his own right) visits a retired Ellis and brings him to a pub in England attempting to prise the real story out of him. Ellis is too humble (and clever !) and leaves Diffie none the wiser but doubtful of his own place in history.

Technical

Asymmetric key protocols are commonly based on the one-way function.

$$y = g^x \ mod \ p \text{ where } p \text{ is a prime.}$$

Assume g and p are public. Then given x finding y is "easy". However given y to find x is assumed to be very hard. This is based upon the fact that certain problems are intractable. The issues with the above equation (also written as $x = \log_g(y)$) is known as the discrete logarithm problem (DLP). Based on this we can build asymmetric key encryption. The function looks "simple" and it would seem that with a simple "try every x" or brute force attack would yield y. However, for acceptable levels of protection to make such an attack unfeasible it is required to use a p with at least 1024 bits and the exponent x with at least 160 bits. The size of p is referred to as the *field size*, and x as the *group order size*. The field size is so much larger than the order size as there exists index-calculus methods for solving the discrete logarithm problem, which require a relatively large field size to resist.

As an alternative to the one way function above there also exists a similar 'hard problem' on an Elliptic Curve.

$$y = x^3 + Ax + B \ mod \ p$$

Take a point on the curve $P(x, y)$. Then assume P is used as a 'public' group generator.

$$Y = xP$$

is also a hard one way problem. If we know x then calculating Y is easy. However given Y finding x is hard. This is basically the same Discrete Logarithm problem. The advantage of using Elliptic Curves is that index calculus methods used to attack the modular equation are not known and hence both the field and order sizes can be as low as 160 bits for practical security. This lower bit size reduces computation overhead and allow for efficient use in restricted devices such as mobile phones or smart cards.

The advantage of all this asymmetric cryptography lies with the property that possessing the public key provides no clue to the private key allowing the public key to be freely published to allow anyone to encrypt to the holder of the private key. This unique property

overcomes the limitations of symmetric cryptography which requires prior 'key swapping' before use. However this advantage comes with a performance penalty. Asymmetric algorithms are much more computationally expensive than symmetric and are not suitable to encrypt large amounts of data.

The solution to practical use is using a combination of both methods to provide both practical performance together with the benefits of usability.

2.5 Commonly Used Symmetric Algorithms

2.5.1 DES

Background

The **D**ata **E**ncryption **S**tandard (DES), is the common name of the Federal Information Processing Standard (FIPS) 46-3, which describes the data encryption algorithm (DEA) widely used in American government and banking circles. Its since been adopted for most security protocols but is currently being replace by its successor AES (see 2.5.2) DES is an improvement of the algorithm Lucifer developed by IBM in the 1970s. Differential cryptoanalyis, a form of attack against block ciphers, was 'discovered' in the late 1980's but it has since been acknowledged that it was known to the NSA as far back as the early 1970s. It has emerged the team behind the creation of DES have admitted that defending against differential cryptanalysis was a design goal of DES, and the technique was the reason the design process of DES was kept secret [[64]]

DES has been extensively studied since it was so widely adopted. Its easily the best known and most widely used symmetric algorithm.

Technical

DES used a 64−bit block size (breaks data into 64− bit chunks before encryption) and uses a 56−bit key during execution (8 parity bits are removed from the 64−bit key). Data passes through 16 rounds of byte substitution (S-boxes). Like Lucifier, its predecessor, it was originally designed for implementation in 1970s hardware.

Like most symmetric block ciphers DES can operate in a number of modes of operation, the simplest being **e**lectronic **c**odebook ECB where each block is encrypted with the same key. This has a weaknesses where identical message blocks encrypted with the same key result in the same ciphertext output. A more pragmatic mode is **c**ipher **b**lock **c**haining CBC where the results of the previous block encryption are XORed in as the key to the next plaintext block. This has the effect of making each identical block encrypted not result in the same output, as the input is actually different. Using DES in the wrong mode opens a

weakness which would allow an attacker to compromise the communications by 'injecting' malicious data into the data stream. To overcome this, most security protocols require a check sum (MAC) using one of the hash algorithms mentioned in 2.3.1.

An improvement to the original design is to use a 128−bit key (actually 112−bit because of the parity stripping) and, using a first 56-bit to encrypt, a second 56-bit key to decrypt and finally the first key again to encrypt each block twice (as secure as 3 keys) the resulting 'triple DES' was the preferred mode of operation until AES was accepted.

2.5.2 AES

Background

By 1997 triple DES was viewed by many as flawed purely by the fact that it had been around so long. Conspiracy theorists claimed that it was probably compromised by some unknown methods attributed to large budget organizations with an interest in cracking messages such as secret service agencies like the US's National Security Agency (NSA), the British GCHQ or Israel's Mossad. Others claimed that organisations like IBM had know of weaknesses for years.

The solution to these claims was for NIST [44] to propose an open international competition for a new algorithm (the Advanced Encryption Standard) to replace DES. The governing rule was that it be

- Unclassified,
- Publicly disclosed.
- Symmetric-key.
- Have minimum block sizes of 128, minimum keys sizes of 128−, 192− and 256−bits
- Strength of triple DES
- More efficient than triple DES
- Available royalty free world wide.

Fifteen candidates were chosen for a short list including some by world renown cryptographers like Ron Rivest of RSA [50], Bruce Schneier of Counterpane [56] , and Paul Kocher of Cryptography Research [34]. This list was then further shortlisted to five and the international community debated the various strengths and weaknesses of each. In 1998 NIST, to the surprise of many, selected a non-US winner (Belgium) by Daemen and Rijmen called the Rijndael [13,14,12,48] cipher. The independence of the winner inspired confidence in the choice.

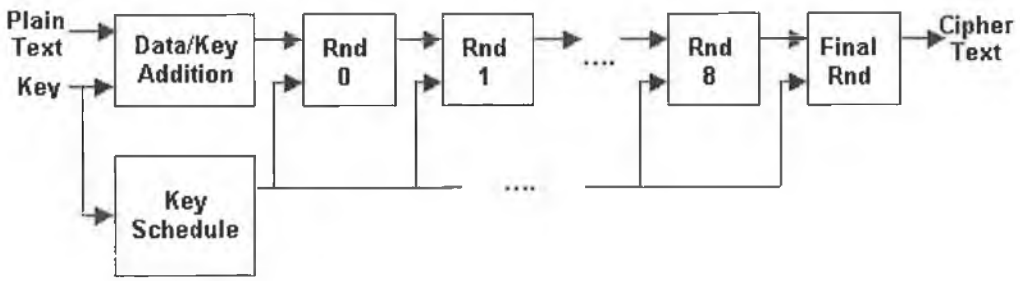

Figure 2.3: AES rounds (*source* [43]).

Increased use of AES as default symmetric algorithm with its minimum key size of 128−bits has all but ended any speculation that there is a weakness known by the NSA or others.

We use AES exclusively as the preferred symmetric algorithm in our IBE subsystem.

Technical Overview.

AES (obviously) follows the minimum specification detailed in the competition. any combination of allowed key size and fixed block size of 128-bits are usable. Its a block cipher not unlike DES in its design with rounds of operations on blocks. Operations used in AES are either byte-shuffling operations, or those defined over the finite field $\mathbb{F}_{2^8}$. Addition in this field corresponds to the bitwise XOR operation outlined in section 2.3.2. The multiplication operation in this field is harder, and often implemented with simple lookup tables. AES does no arithmetic operations and assumes non-endianness. These properties allow for very efficient implementation in hardware.

A simplified description : The basic operation is that of data processed by transformations via 'round' operations on data stored in a four a four matrix. Each round [1] is a set of 4 operations on the matrix. The number of rounds depends on the key size chosen. The 4 operations in each round are ByteSubstitution (like a DES S-box), Mix-Column, Shift-Row, Round-Key Addition.

A concise description suitable for implementors, by the authors of Rijndael, is available in [14].

[1] except the final round which is special.

2.6 Commonly Used Asymmetric Algorithms

2.6.1 RSA

Background

Discovered in 1977 and named after its inventors, Ron **R**ivest [50], Adi **S**hamir and Leonard **A**dleman, RSA [49, 53] encryption is based in factoring and transforms the message "M" into the number "C" with the formula

$C = M^e \mod N$

The numbers e and N are the two public numbers you create and publish. They are your "public key." As before the message M can be simply the digital value of a block of ASCII characters.

The formula says: multiply the Message M by itself e times, then divide the result by the number N and save only the remainder. The remainder that we have called C is the encrypted representation of the message.

Example Application

Alice publishes the public key numbers $e = 29$ and $N = 77$. Bob wants to send Alice the message "I have it". In decimal ASCII the message is

7332104971181013210511 6. Break this number string into smaller blocks less then N like so

73 32 10 49 71 18 10 13 21 05 11 6

To encrypt these blocks, apply the formula

$$C = M^e(\mod n)$$

to each block.

Technical Overview

Here we present an alternative description due to Mao [37]

As before we deal with Bob attempting to send a message to Alice.

Key Set-up.

Alice creates her public and private key pair thus

1. Choose two large random prime numbers p and q such that $\mid p \mid < \mid q \mid$

2. Compute $N = pq$
3. Compute $\phi(N) = (p-1)(q-1)$
4. Choose a random integer $e < \phi(N)$ such that $\gcd(e, \phi(N) = 1)$ and computing the integer d such that $ed \equiv 1(mod\ (\phi(N)))$
5. Publicise (N, e) as her public key, discarding p, q and $\phi(N)$ and keeping d as her private key. e can be small but d must be impossible to guess.

To encrypt to Alice

To send a message $M < N$ to Alice, the sender Bob creates a ciphertext C by

$$C \leftarrow M^e(\mod N)$$

For Alice to decrypt

To read the ciphertext C from Bob, Alice computes

$$M \leftarrow C^d(\mod N)$$

2.6.2 ECC

Background

Another form of 'hard problem' that mathematicians have found useful to the field of cryptography is that of Elliptic Curves.

The discovery of the use of Elliptic Curves for public key cryptography can be attributed independently to Neil Koblitz [33] and Victor Miller [41] who both made discoveries in 1985.

Technical

Note: Curves can be defined in Affine (2 dimensions) or Projective (3 dimensions co-ordinates) - The equations I present are in Affine co-ordinates.

An elliptic curve is a graph (curve) which can be defined by equations of the form.

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

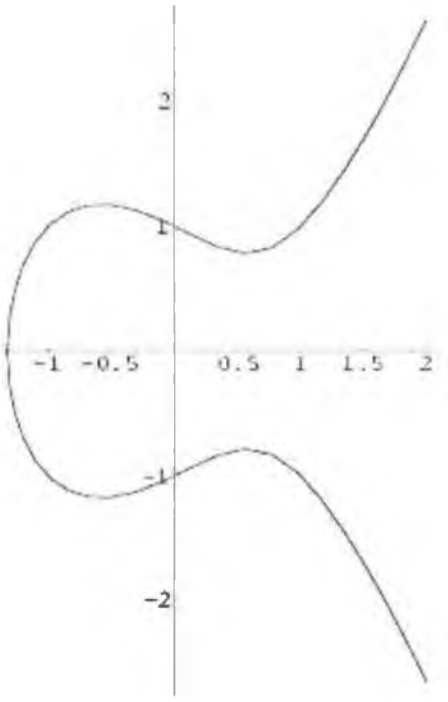

Figure 2.4: Elliptic Curve

Which all the a's are constants. Formally this is described as an *elliptic curve* $E(\mathbb{F}_q)$, and is the set of solutions (x, y) over $\mathbb{F}_q$ to an equation of the form $E : y^2 + a_1xy + a_2y = x^3 + a_2x^2 + a_4x + a_6$, where $a_i \in \mathbb{F}_q$ and there also exists a *point at infinity* denoted O *[4]*

The special properties are that elliptic curves behave 'well' when operations are performed with a prime modulus greater than 3 and any elliptic curve of the form (1) can be converted by the transformation to a *Weierstrass* form 2.6.2.

$y^2 = x^3 + a_4x + a_6$

To be more confusing and to follow general cryptographic convention a_4 is renamed a and $a6$ is renamed b.

This gives us the curve

$y^2 = x^3 + a * x + b \mod p$

This means we are only allowed to use the integers from zero to $p - 1$ as input.

For example let us take an equation with $p = 11$, $a = 4$ and $b = 7$.

$y^2 = x^3 + 4x + 7 \mod 11$

Plug in all values for x and we get the associated values for y plus the point at infinity (denoted by O). Now any pair of points that satisfy the equation can be used.

To add two points on a curve we can't simply add the coordinates to find a point which still satisfies the curve.

However there are a set of rules which one can apply for curves of this type.

The rules are (see Smart [59])

- Rule 1: $O + O = O$
- Rule 2: $(x_1, y_1) + O = (x_1, y_1)$
- Rule 3: $(x_1, y_1) + (x_1, -y_1) = O$

Now it becomes more complicated.

- Rule 4: if $x_1 \neq x_2, (x_1, y_1) + (x_2, y_2) = (x_3, y_3)$

  where

  $x_3 = (\beta_2 - x_1 - x_2) \mod p,$

  $y_3 = (\beta(x_1 - x_3) - y_1) \mod p,$

  and

  $\beta = ((y_2 - y_1)/(x_2 - x_1)) \mod p.$
- Rule 5: if $y_1 \neq 0, (x_1, y_1) + (x_1, y_1) = 2(x_1, y_1) = (x_3, y_3)$

  where

  $x_3 = (\beta_2 - 2x_1) \mod p,$

  $y_3 = (\beta(x_1 - x_3) - y_1) \mod p,$

  and

  $\beta = ((3x_1^2 + a)/(2y_1)) \mod p.$

The 'why' behind this is beyond the scope of this thesis. I recommend Nigel Smart's 'Introduction to Cryptography' [59], Wenbo Mao's 'Modern Cryptography' [37] Meneze's 'Elliptic Curve Cryptography' [39].

Using these rules two points on the curve may be added to yield a third point also on the curve. For example in the figure above point $P + Q = (P + Q)$ on the curve.

Note 1: If we ever, in our calculations, divide by zero we can stop and say the result is the point at infinity (O).

Note 2: the point at infinity (O) acts like zero in regular addition : Add a point to the point at infinity we get the original point. This is the additive identity for the group.

Note 3. Prime Modulus. The group is cyclic, it has a generator function such that when this function is applied to any member of the group, it will only result in another member

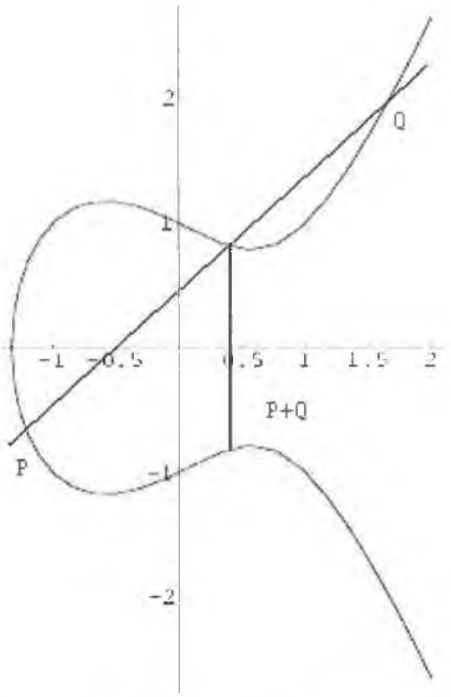

Figure 2.5: Point Addition . *(source [38])*

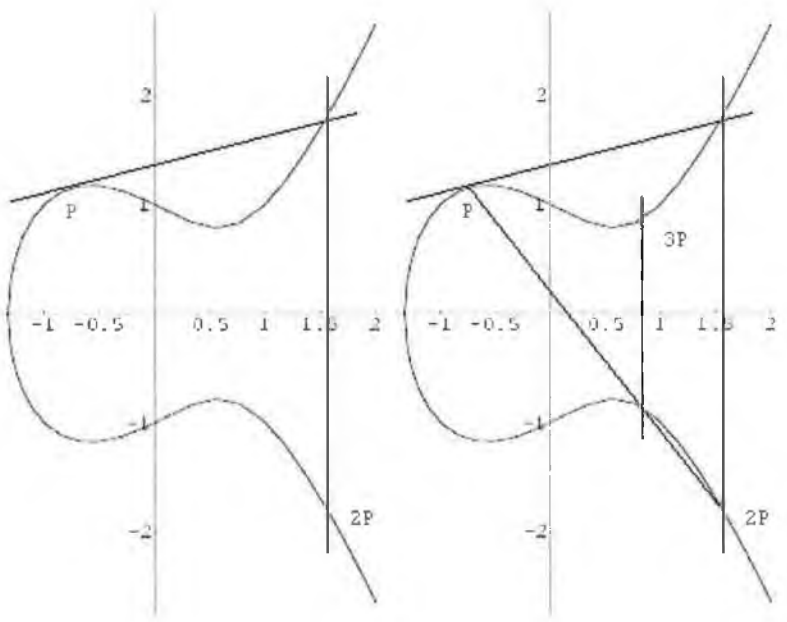

Figure 2.6: $2 * P$,$3 * P$*source [38])*

of the group. If applied to all members of the group then it will produce another set containing all members of the original group. Interestingly the output sequence of the results of the application of the generator holds a random distribution see [63]

Note 4: Multiplication

Take a point P an (x, y) point and multiply it by an integer d. dP is $d * P$ and we can break down $d * P$ to $(P + P + ...P)$ i.e. P added to itself d times. Then apply the rules of addition. in the illustrated figure $3P = P + P + P$.

This integer d is called a *scalar* as opposed to the coordinate (point) P.

What does all this mean to cryptography ?

We'll take an elliptic curve (i.e. modulus p and parameters a and b) and a point on this curve P.

Then take a scalar d and find $d * P$ to get another point on the curve Q.

We keep d secret. We can use the curve (p, a, b) and points P, Q as the public key. The challenge for any attacker is to find d. No-one has found a sub-exponential algorithm to be able to compute d. This is another manifestation of the discrete logarithm problem.

If the modulus p is large enough (200 bits or so. a big number .....) then it would take todays supercomputers many thousands of years. This is the basis for the application to cryptography

Scalar multiplication on an elliptic curve is relatively easy, but the inverse, which is a discrete logarithm problem is extremely hard.

To illustrate we will consider Key agreement using Elliptic Curve Diffie-Hellman.

Diffie-Hellman is a technique to allow unauthenticated key agreement using exponentiation. It's security rests on the intractability of the Computational Diffie-Hellman problem and the Discrete Logarithm Problem.

Alice calculates her curve and makes p, a, b and a point P public.

She generates some random d_a and keeps this secret.

Alice sends Bob Q_a which is equal to $d_a P$.

Bob gets Alice's public components and generates his own random d_b.

He calculates Q_b by computing $d_b P$.

Bob then computes a secret value

$S = d_b Q_a$

Since Q_a is just $d_a P$ what Bob has computed is

$S = d_b d_a P$

He sends Alice Q_b and Alice uses this to compute her secret value

$S = d_a Q_b$

since Q_b is $d_b P$, what Alice has done is to compute

$S = d_a d_b P$

this is the same as Bob computed !

So Alice and Bob can 'secretly' get to the same point on the curve S, by using this protocol. A simple method to extract a key is just to ignore the y coordinate and take the x coordinate as a number. This derived number can be used as a key. Bob can use this secret value to make an AES encryption key. Alice can use the method outlined above to get same encryption key. So what Bob encrypts, Alice can decrypt. The Attacker Eve intercepting this exchange just knows p, a, b, P, Q_a, and Q_b. The only way for Eve to figure out S is to get either d_a or d_b which Alice and Bob are holding secret.

Apparently the only way she can get a d is to calculate one of the d's is by using the fact that she knows either

$Q_a = d_a P$ and she knows Q_a and P

OR

$Q_b = d_b P$ and she knows Q_b and P.

This is exactly the discrete logarithm problem of ECC as outlined above and this is computationally unfeasible with todays technology and for the foreseeable future.

2.6.3 Pairings

Joux [32] and Sakai, Ohgishi & Kasahara [54] independently proposed using properties of pairing-mapping functions applied to cryptography to establish ID-based PKI. Initially the area of pairings had been discounted by Menezes as holding no promise for cryptographic applications. It wasn't until the Joux publication in English that countered this claim that this whole area was opened to researchers culminating with Boneh & Franklin's much publicised paper in 2001.

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ denote two groups of prime order q, where $\mathbb{G}_1$, with an additive notation, denotes the group of points on an elliptic curve; and $\mathbb{G}_2$, with a multiplicative notation, denotes a subgroup of the multiplicative group of a finite field.

Multiplicative groups will be represented here as $\mathbb{Z}_n^*$, which is the set of positive integers less than n and relatively prime to n, under multiplication modulo n. An integer is relatively prime to another if their only common positive divisor is 1. For example, 8 and 15 , even though they are not prime numbers, are relatively prime.

A pairing is a computable bilinear map between these two groups. Two pairings have been studied for cryptographic use. They are the **Weil**[2] **pairing** and the **Tate pairing** . For the purposes of discussion, we let $\hat{e}$ denote a general bilinear map, i.e. $\hat{e}$:$\mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$, which can be either a modified Weil pairing or a Tate pairing.

In this notation the Diffie-Hellman (DH) solution described above is a tuple in $\mathbb{G}_1$ as $(P, xP, yP, zP) \rightarrow \mathbb{G}_1$ for some x, y, z (chosen at random) $\rightarrow \mathbb{Z}_q*$

satisfying $z = xy \mod q$.

Properties of Pairings

Bilinear: If $P, P_1, P_2, Q, Q_1, Q_2 \in \mathbb{G}_1$ and $a \in \mathbb{Z}_q^*$, then $\hat{e}(P_1+P_2, Q) = \hat{e}(P_1, Q).\hat{e}(P_2, Q)$, and $\hat{e}(P, Q_1 + Q_2) = \hat{e}(P, Q_1).\hat{e}(P, Q_2)$.

Non-degenerate: There exists a $P \in G_1$ such that $\hat{e}(P, P) \neq 1$.

Computable: If $P, Q \in G_1$, one can compute $\hat{e}(P, Q)$ in polynomial time.

2.6.4 Other 'hard problems'

As mentioned above the **Diffie-Hellman (DH)** tuple in $\mathbb{G}_1$ is a tuple $(P, xP, yP, zP) \in \mathbb{G}_1^4$ for some x, y, z chosen at random from $\mathbb{Z}_q$ satisfying $z = xy \ mod \ q$.

Computational Diffie-Hellman (CDH) problem: Given the first three elements in a DH tuple, compute the remaining element. The CDH assumption: there exists no algorithm running in expected polynomial time, which can solve the CDH problem with non-negligible probability.

Decision Diffie-Hellman (DDH) problem: Given a tuple $(P, xP, yP, zP) \in \mathbb{G}_1^4$ for some x, y, z chosen at random from Zq, decide if it is a valid DH tuple. This can be solved in polynomial time by verifying the equation $\hat{e}(xP, yP) = \hat{e}(P, zP)$.. Note that this is contrast to the suitation in the simple finite field where the DDH problem is also hard.

Bilinear Diffie-Hellman (BDH) problem: Let P be a generator of G_1. The BDH problem in $\mathbb{G}_1, \mathbb{G}_2, \hat{e}$ is given $(P, xP, yP, zP) \in \mathbb{G}_1^4$ for some x, y, z chosen at random from Zq, compute $W = \hat{e}(P, P)^{xyz} \in \mathbb{G}_2$.

[2] Pronounced "Vay". Andre Weil in the 1940's.

2.7 Computation Expense.

To perform either RSA or elliptic curve cryptography on a large set of data is very computationally expensive and would take too long a time for real-time data communications such as a VPN or an email system.

The solution is to use some other more efficient algorithm like a symmetric algorithm such as AES or DES and perform the bulk data encryption and use the public key methods (ECC or RSA) to encrypt the key(s) used by this efficient algorithm and transport them with (usually by simply attaching them to) the encrypted data. This hybrid solution is used by most common protocols such as SSL, SSH, IpSec and S/Mime.

2.8 Identity Based Encryption (IBE)

In 1984, Shamir [58] (the 'S' in RSA) proposed the first identity based signature scheme and outlined a solution to this key distribution / certificate management problem calling it Identity Based Encryption (IBE). However he didn't have an implementation.

The idea was that if any string can be a public key, then one could use an identifier of the recipient to be the public key in which case one doesn't need to locate the public key associated with an identity. The identity IS the key. Furthermore one can use the 'identifier string' combined with something like a date/time and encrypt messages to be read *into the future.*

In this system our friends Alice and Bob are to communicate. Bob can simply use Alice's email address (alice@wonderland.com) and the public parameters of a trusted third party as the public key to encrypt the message. When Alice receives the message, she contacts the trusted third party (KDC key distribution center), validates herself and receives the private key associated with her identity.

Shamir's proposed IBE remained an elusive 'holy grail' for cryptographers until Cocks [8] working at the British secret service GCHQ discovered a method relying on quadratic residues. Unfortunately this method is impractical for widespread use because of the bandwidth overheads. It has recently come to light that two Japanese researchers Sakai and Kasahara [55] also made a significant discovery relevant to IBE using pairings but due to language issues their implementation wasn't widely known.

In 2001, Boneh and Franklin [7,6] announced a more viable method using the Weil pairing 2.6.3. This method, while demonstrable, still lacked a pragmatic implementation which could be considered widely usable. Optimizations of the pairing mathematics were proposed by Barreto-Kim-Lynn-**Scott** [3] [4] in 2002 which moved the processing overheads close to that of the widely used RSA algorithm. These optimizations involved

[3] Michael Scott is the MSc. supervisor of the author

- Point tripling for super-singular elliptic curves over $\mathbb{F}_{3^m}$.
- Removal of irrelevant operations from conventional algorithms.

Background

An IBE system involves, using the language of Boneh & Franklin's seminal paper [4] a set of four algorithms.

Setup: A key generator (KDC) which runs a 'setup' algorithm to generate global system parameters and a *master-key* which the KDC keep safe. The whole security of the system relies on the safe keeping of this master-key

Extract: The KDC runs an extract algorithm inputting the users identity (or any bit string) and using the master-key from the setup. The output is the users *private-key* associated with the users identity. Its important that the private-key is transported to the user in a safe manner and that the KDC has made a full examination of the user credentials before issuing a key corresponding to those credentials.

Encrypt: A probabilistic algorithm. Any user encrypts using the global system parameters and public key *ID*. The output is the ciphertext.

Decrypt: This process takes the ciphertext from the encrypt function, global system parameters and the private key issued by the KDC. The output is the corresponding plaintext.

The Boneh & Franklin's paper provides a random oracle security proof for their IBE method. That means it is secure against an adaptive chosen ciphertext attack assuming the hardness of the so-called Bilinear Diffie Hellman problem.

Technical

(from Mao [37])

Set-up

1. Generate two groups $\mathbb{G}_1, \mathbb{G}_2$ of prime order q and a mapping-in-pair $e : {\mathbb{G}_1}^2 \to \mathbb{G}_2$. choose a generator P $\mathbb{G}_1$.

2. Pick $s \in_\cup \mathbb{Z}_q$ and set $P_{pub} \leftarrow [s]P$; s is the *master key.*

3. Using a strong hash algorithm $F : \{0,1\}^* \to \mathbb{G}_1$. To map the identity string ID to an element in $\mathbb{G}_1$.

[4] Extended abstract in [6]

4. Specify another suitable hash algorithm $H : \mathbb{G}_2 \rightarrow \{0,1\}^n$.

The KDC keeps s as the system master-key and publishes the parameters

$$(\mathbb{G}_1, \mathbb{G}_2, e, n, P, P_{pub}, F, H)$$

Private Key Generation (= Extract)

Let ID denote an authenticated and validated user's identity (it can be any string)

1. Compute $Q_{ID} \leftarrow F(ID)$ this is an element in $\mathbb{G}_1$ and is the users public key.
2. Set the users private key d_{ID} as $[s]Q_{ID}$.

Encryption

To send encrypted message get the system parameters $(\mathbb{G}_1, \mathbb{G}_2, e, n, P, P_{pub}, F, H)$. Using them compute $Q_{ID} = F(ID)$. To encrypt $M \in \{0,1\}^n$ pick $r \in_U \mathbb{Z}_q$ and compute

$g_{ID} \leftarrow e(Q_{ID}, [r]P_{pub}) \in \mathbb{G}_2$,

$C \leftarrow ([r]P, M \bigoplus H(g_{ID}))$.

The ciphertext is $C \leftarrow ([r]P, M \bigoplus H(g_{ID}))$.

Decryption

To decrypt C using ID's private key d_{ID} as $[s]Q_{ID}$, compute $M = V \bigoplus H(e(d_{ID}, U))$.

Chapter 3

e-Mail Communication Protocols

In this chapter we introduce the background to the Internet protocols, outline the various email standards, detailing document standards and the security of existing solutions.

Email messaging is one of the oldest Internet services with Ray Tomlinson credited with sending the first email in 1971, and it is still among the most used, second only to a relative newcomer *hypertext transfer protocol* HTTP.

Its fundamental workings, governing sending and receiving, remain unchanged since 1982. It has been 'on top of' these communication protocols that user features such as attachments, read receipts, etc. have been added. The benefits of this are an infrastructure then hasn't fundamentally changed in over 20 years allowing for easy deployment, a large number of people versed in its inner workings and an ability to exploit the reliability and connectivity of disparate networks.

3.1 Message transmission

Message transmission is defined by 3 mail protocols, one for sending

- SMTP the **S**imple **M**ail **T**ransport **P**rotocol (RFC 821 [45])

And two for receiving

- POP3 the **P**ost **O**ffice **P**rotocol version **3** (RFC 1939 [31])
- IMAP the **I**nternet **M**essage **A**ccess **P**rotocol (RFC 3501 [10])

3.1.1 SMTP

The Simple Mail Transport Protocol (RFC 821 [45]) usually resides on socket port 25. It follows a simple line based ASCII/text command based communication over a very limited

command set. Each command is identified with mnemonic and ID number. The sequence is very similar to transaction communications between two people

- Simple pleasantries : hello

- Introductions: who is who, and a decision whether or not to continue or not

- What the client wants to do: send messages etc. and again a decision to continue or not.

- Transaction: the message content

and finally

-OK and goodbye.

To illustrate:

By convention the server waits for connections on port 25.

When it receives a connection it replies with a short string which by convention identifies the host, the server operating system, mail server software & version and the local time.

e.g `250 neil.costigan.com homegrown mail-server version`

`0.1 Oct 29OCT 1967 IMT`

The client responds with an "HELO" to this and the server will reply 220 command "Hello" and again the server id (fully qualified host-name, possibly IP address)

e.g `220 Hello neil.costigan.com 10.0.0.1`

Lately it is common for system administrators to remove the server software name and version to try to thwart any attacks on known vulnerabilities of this software or operating system. This technique is known as server fingerprinting. The client then issues a "MAIL" command in which it identifies itself

e.g. `MAIL From neil@nospam.costigan.com`

This allows the SMTP server to decide if it agrees to accept to process mail from the client address. The reason for a failure here would be spam control or authentication etc.

It is possible in SMTP extensions to demand further authentication at this point.

The server replies with another "250 OK" indicating it accepts the address

The client then tells the server the recipient(s) of the message. It can issue a number of these if there are multiple recipients

e.g `RCPT To:  santa@northpole.com`

Again this allows the server to determine if it will accept mail to this user or domain. The main reason here for refusal is if it determines the user is unknown, spam management, or, in the commercial world, if it can't find a way to charge for the service. On success the server issues its familiar "250 OK". The client then tells the server it will pass the mail itself

by issueing a DATA command. The server will reply with a description on how it wants to receive the data. This is usually text (no binary), ending with <CR><LF>.<CR><LF> i.e a (full stop) between the newlines.

e.g `354 Start mail input; end with <CRLF>.<CRLF>`

The client then proceeds to deliver the message itself ending with the "carriage return linefeed ." .

e.g.

```
"Date:  Sat, 01 Dec 02 13:31:12 GMT
From:  neil@nospam.costigan.com
To:  santa@northpole.com
Subject:  Christmas at my place
I've been good and I want a Porsche this year.
.
"
```

Again the server replies, on success, with a simple "250 OK" or it may add a queue message "250 OK message queued with id 42a2ffe" this can help with any problems tracking lost messages. Finally the client, issues a "QUIT" to break the session.

If the mail contains binary data (like a graphic or file attachment) the client can encode the message to a Base 64 form. It acheives this by taking blocks of 6 bits and converting them to a printable ASCII byte. The mail itself can contain multiple parts like a simple body, rich text representation, HTML representation, and attachments. There is a conversion format (MIME) which can transform all components of an email to a representation another client can decode. However for the the server and for the communications between the client and server, the contents of the 'DATA' blob are ignored as long as they are in printable ASCII.

3.1.2 POP3

The Post Office Protocol (RFC 1939 [31]) is similar to SMTP and usually resides on port 110. It also follows a simple line based ASCII/text command based communication over a very limited command set. This command set is more suitable for a response / request as the client is deciding what it wants. All commands are four letters long and all keywords are separated by spaces. Like most of the early protocols each reply command is identified with status indicator mnemonic + for success or - for failure followed by an OK or ERR with a description depending on the previous command. The POP3 command set follows a concept of 'states' which indicate what commands can legally follow a previous command.

Some POP3 responses are multi-line as indicated by a CRLF pair.

The sequence is also very similar to transaction communications between two people.

- Simple pleasantries : hello

- Introductions: who is who. decision to continue or not

- The server tells the client what is has for it

- The client decides (based on its message store) what it wants to do: get list of messages, get a specific message etc. and again a decision to continue or not.

- Transaction: get the message content

finally

-OK and goodbye:

Next we illustrate a simple POP3 email session.

The first thing is to open a TCP connection from your computer to your mail server usually on port 110.

When the POP3 server receives a connection it replies with a short string which by convention identifies the host, the server operating system, mail server type & version.

The session is now in the authorization state.

e.g..

```
+OK POP3 server ready (7.0.016)
<F0CD2FB3F2445493E9CF51878AF7F40451A89525@hawk.dcu.ie>
```

The client then responds to this by providing the USER command with user-name.

```
USER user-Name
```

This should give you:

```
+OK Password required for user-Name.
```

Or some other request for authentication

The client responds with the corresponding password

```
PASS passW0rd
```

Any failure with the previous commands the server responds with an -ERR message like

```
-ERR [AUTH] Password supplied for
"user-Name" is incorrect.
```

The password supplied was not the one expected by the server; retype the password, failing that. Find out if you've got the correct password.

```
-ERR [AUTH] PAM authentication failed for user "user-Name": Authenticatio
failure (7)
```

On success this should yield an `OK` with

```
+OK user-Name has ? visible messages (? hidden) in ????? octets.
```

At this point the session passes to the transaction state.

The server has indicated the number of messages in the mail drop and assigns them an order number from 1 to N with the size of the message in octets.

The client decides what messages it would like to request or if it needs to see a list of the messages with an associated size to see if it has them already. This usually happens if the amount of messages on the server doesn't match the amount the client believes it has.

POP3 can be seen as the pull technology to match SMTPs push.

3.1.3 IMAP

The Internet Message Access Protocol was originally developed in 1986 at Stanford University. It allows a client to access and manage mail messages on a server. IMAP permits management of remote message folders, called "mailboxes", in a way that is equivalent to having the mailboxes stored locally like in traditional POP clients. IMAP also provides the capability for an offline client to resynchronize with the server and includes operations for creating, deleting, and renaming mailboxes, checking for new messages, removing messages, searching, and selective fetching of message attributes, texts, even portions of text.

Messages in IMAP are accessed by the use of numbers. These numbers are either message sequence numbers or unique identifiers.

IMAP does not specify a means of posting mail. It is entirely read-only. Posting is handled by a mail transfer protocol such as SMTP. The major benefit of IMAP is that the entire body (the bulky part) does not need to be transported unless specifically requested by the client.

3.1.4 MIME

Multi-part/Signed and Multi-part/Encrypted or MIME (RFC 1847 [20]) is not a transport protocol per se. We mentioned above that the original simple protocols allow for basic ASCII text. To transport binary data just as images, propriety format documents, files etc. we need to encode them to an ASCII form in a method that is easily decoded.

MIME defines the format of the contents of Internet mail messages and provides for multi-part textual and non-textual message bodies.

3.2 Secure e-Mail

Securing the email message communications is more difficult than regular TCP Internet client / server protocols because the sender and receiver of the message don't usually communicate directly together making it difficult for the parties to swap security credentials like keys. Instead the message is passed through a number of third parties who route, store and forward the message as dictated by a complicated set of rules allowing for the possibility that the receiver may be off line and/or not able to store the message because of location, bandwidth, or storage restrictions. This is as a letter is transported through the regular printed matter postage system.

The usual channel security mechanisms (SSL, VPN, SSH etc.) fail to be truly 'end to end' as the message is still essentially in the clear at the control points en route.

POP3 / SMTP / IMAP ignore the security issues assuming the security is solved inside a higher or lower layer. Common approaches range from using simple, proprietary 'out of band' secret-sharing to decode symmetric cipher based encrypted messages, to complicated public key infrastructures to allow for more large scale, widely distributed, user base. Unfortunately there are bandwidth and scalability overheads, which have made deployment of such systems rare.

Inside these messaging protocols there are a number of different approaches to secure the message being transported.

3.2.1 S/MIME

S/MIME (RFC 1847 [20]) or Secure / MIME expands on MIME (3.1.4) to add secure services to email messages using PKCS7. Currently S/MIME is at its third version. It was originally proposed by RSA and is based on RSA's public key cryptography. It has evolved to include the Cryptographic Message Syntax (CMS) (RFC 3369 [22]) a cryptographic algorithm independent format. The S/MIME message comprises of a Base64 encoded PKCS7 of a MIME body which can itself include other MIME bodies. It is notoriously difficult to code, leading to many interoperability problems between vendors.

3.2.2 PGP mail

Pretty Good Privacy (PGP) was written by Phil Zimmermann in 1991 in a time when crypto implementations were strictly controlled by the US government and RSA's patent lawyers. Zimmermann quickly got himself into trouble but not before his creation had been uploaded to hundreds of bulletin boards and sites across the Internet. The implementation was open sourced and portable, quickly becoming the de facto email security system for non-US academics. Zimmermann found himself in legal tangles with both the US government and RSA and became an Internet cult figure. It was five years before an

agreement was reached. During this time multiple illegal and non approved US and non-US versions were created causing multiple interoperability problems which limited its usefulness. In 1996 PGP was commercialised by Zimmermann and a free version was published by MIT.

The main issue with PGP mail is its overly technical nature making it an unlikely candidate for ordinary use.

As mentioned PGP exists in a number of forms, one is the commercial implementation from Network Associates and another is the platform independent, freely available tool-set from the GNU group called GnuPG. Most modern implementations of Linux have GnuPG in their distribution. A number of email clients carry GUI front ends for integration of PGP services.

Here we walk through a typical set up which allows two people to communicate using. What we are trying to achieve is to pass a message encoded with a secret key to the other party allowing them, and only them to decode it.

First both sides generate a public/private key pair. Here for simplicity and illustrative purposes we use the command line tools for GnuPG.

```
gpg --gen-key
```

Next follows is a small question and answer session. The first is to do with the encryption algorithm for the public key pair. Choices are RSA and ElGamal. The second is about the key length. One is given choices from 512−bit to 2048−bit, 1024−bit being the most common. Then one is asked to supply the name, email address, and some comments about the user. Then (finally !) a pass-phrase is required to secure the file that stores the private key component on the local hard disk. This is very important as disclosure of the private key effectively reduces the whole security to zero.

After the key generation is complete the user is left with a number of files on his or her disk - gpg.conf, pubring.gpg, random_seed, secring.gpg, trustdb.gpg.

The file secring.gpg contains the private key in a binary format and is particularly important. The pubring.gpg contains the public keys of people with whom you have swapped credentials, and the file trustdb.gpg contains information about the levels of trust you place on the different private keys. This would typically depend on the mechanism by which one had collected them.

Confused ? Read on !

To pass your public key to the person you want you need to extract (*export*) it is a suitable format for transport, for example ASCII.

```
gpg -a --export <password> > bartkey.asc
```

This result is a simple text file which can be attached to any outgoing email. One then emails it to the other party. They then extract the attachment and *import* it

```
gpg --import bart.asc
```

This will place the public key in the other users pubring and trustdb files as outlined above. It is possible to view the contents of this files using

```
gpg --list-keys
```

After the above steps, one would get something like

```
pub 1024D/1234ABC 2004-04-01 Bart Simpson the
coolest kid on the block bart@simpson.com
```

corresponding to the key length, key id, date, full name and comments.

Now before one can send messages securely to this person one needs to *sign* their public key to verify its level of security. This is associated with how sure you are that this key actually did come from the person that sent it. The issue here is trust. To do this out of band (say by telephone), first one gets the key's fingerprint

```
gpg --fingerprint bartpub 1024D/1234ABC 2004-04-01 Bart Simpson
the coolest kid on the block bart@simpson.com
```

The output would be something like

```
key fingerprint = A123 B456 ...  C7890
```

Then one asks the other party to do the same thing on their side over the phone or by secure courier. If the fingerprints match then one finally signs

```
gpg --edit-key Bart
```

which prompts for the pass-phrase for your secret key. One can then proceed with secret message swapping by using the gpg encrypt and decrypt functions.

note: The GnuPG package also allows for digitally signed messages after such a set up.

note: Some of the email packages recognize the various attachment formats and will automatically do the import step via GUI 'wizard' steps.

As you can imagine, attempting to bring this type of set-up to a larger groups of users (enterprise or Internet domain) in a scalable way requires that one introduces some infrastructural components, so that each user is not required to email everyone and then required to verify the validity of the key set. There are a number of approaches usually combining the use of a 'key server' or directories of public keys on a publicly accessible server. Then one needs to trust the signature of each downloaded public key (to avoid a potential 'bad-guy' masquerading as someone else). This signature trusting is achieved by checking that the key was signed by someone you already trust (web of trust) or by a trusted authority like a bank or government who one already trusts. As you can imagine this opens a can of worms with regard to a simple chicken & egg problem. How do you get the first trusted signature and how does every party interconnect with out overloading the key servers ?

As you can gather from the above 'set-up'. The root of the problem of the lack of use of security in email, messaging and other network traffic is the technical complexity. The main issue here is that the number of complex questions the typical end user is required to answer gegarding key length, algorithms, pass-phrases, key servers, trust etc. etc. are not widely understood. Worst still is the cost associated with using such a system, from training to installation and configuration.

3.3 Approaches to adding security to an e-Mail system.

There are a number of approaches to introducing encryption to email systems.

1. Develop a new email client.
2. Develop add-ons to existing mail clients.
3. Catch the message en route (a proxy)

3.3.1 New Email client

While this approach seems to be the least complicated, we have to recognize that to develop an email client which can satisfy expectations for all the features in a standard off-the-shelf client, including robustness, address book management, junk mail filtering, multi-format display, multi-server support, multi-protocol, support etc. etc. is beyond the scope of this project. In practice we will not manage to 'move' customers from an existing installed base 'just' to add an easy-to-use security feature.

Nevertheless developing a prototype email client allowed us to test and fine tune formats and usage parameters etc. I will not go into detail on the client as it is very much a test tool. But it suffices to say the client was developed in Java using the rich availability of the language's packages for SMTP and POP3 support.

3.3.2 Develop add-ons to existing mail clients

Modern email clients have matured to the level that they allow for some level of customization which allow third parties to develop 'add-ons' which manipulate, among other things, how messages appear, are transported, or filter what they contain.

This offered us some possibility to add in the IBE fuctionality by transforming the messages before they leave and as they are received. The upside is that this overcomes the problems

- Of converting users to 'our' email system - instead they can use familiar interface, training etc.

- Of continually updating our client to whatever feature is currently in favor.
- Of supporting exotic or propriety email protocols like Microsoft's Exchange, and instead allows us to focus on the issue at hand (IBE).

The downside of this approach is that the 'add-ons' are

- Propriety to applications: the API of an Outlook add-on is very different from that, say, a Netscape add-on.
- Limited in some respects, making it hard to add on the crypto.
- May not be portable: Outlook's SDK is unavailable on Outlook Mac X.
- Have stability issues: many plugin mechinisms may be poorly documented and prone to crashing the client.
- May not even be possible on some popular clients (e.g. Eudora)

Nevertheless the approach is possibly the most commercially interesting and, given the commercial nature of our funding, was pursued for the most popular client (Outlook) and platform (Windows 2000 and higher). We developed such an IBE plug-in as a prototype. I won't go into the specifics of development it suffices to say that it was developed with a combination of Visual Basic scripting and C++ using Microsoft COM technology. We found that this environment was immature, and the effort in development did not reap benefits which could be applied to other applications. A lot of the coding APIs were undocumented and we found we were reverse engineering Visual Basic scripts to catch event hooks. This avenue was abandoned in favor of the method outline in the next section 3.3.3.

3.3.3 Catch the message en route

This approach involves either

- Adding software to the SMTP and POP3 servers, the advantages being central control and ease of deployment, while the main disadvantage is that it is not securing the traffic end-to-end.
- Developing an 'agent' which resides on the client machine and can filter Internet traffic in some way which lets it grab emails as they are transmitted and intelligently transform the mails to embed them in IBE wrapped messages. It overcomes the problems associated with (3.3.1) and (3.3.2) and so allows for greater application and platform support, freeing us from the problems of maintenance of 'other' features. The downside is that such an agent can difficult to program and may be frowned upon by IT managers as 'yet another' point of failure in the chain of message delivery.

The bulk of the work for this project was to attempt to write a generic solution which could be deployed on both a client and a server, yet is possible to be used by non email protocols like a VPN.

3.4 Message format.

3.4.1 XML

MIME and S/MIME were revolutionary in their abilities to encode and secure complex data sets. Unfortunately S/MIME (which the reader will recall was developed by RSA) is too explicit in its algorithm choices and while we could modify a version of it to a non-standard implementation for our IBE it would be of little value as existing S/MIME aware clients are likley to fail, possibly crash, attempting to decipher our encodings. A better choice for a new format is the e**X**tended **M**arkup **L**anguage : XML.

There is a lot of hype associated with XML, many people claiming it is the universal cure for computing interoperability problems. While this may be an overstatement XML and its associated technologies do appear to hold the answers for many of the common data processing demands which most Internet applications require. Our IBE application is no exception.

We do not require all of the features and we could have opted for a simpler XML-like subset. But this would have excluded us from many of the rich parsing and transport facilities available - Technologies which we would have had to develop anyhow.

The XML 'buzz words' or technologies we use in the IBE application are XML, XML Namespaces, DTD and/or XML schema.

I'll try evolve my description of XML to briefly encompass these, then expand the IBE requirements to show the decisions I've taken.

The simplest definition of XML is to say it is 'self describing data'. So instead of an arbitrary data blob whose contents, format, offsets etc. is know only to proprietary applications XML data describes itself in some way

for example instead of `AB123CDF` we can describe this as

`<productcode>AB123CDF</productcode>` allowing for some context.

This would be XML in its simplest form. But who is to say what this product code is for ? What if we were to add context aswell?

Taking the previous example `<productcode>AB123CDF</productcode>` we can expand this to say

```
<product>
<drill>
```

```
<code>AB123CDF</code>
</drill>
</product>
```

This allows for an easy evolution and adding more 'fields' to the data is easy

```
<product>
<drill>
<description>3 inch drill bit</description>
<code>AB123CDF</code>
</drill>
</product>
```

This is a similar namespace problem which recent revisions to the C++ standard have solved.

To give this some more power one can add a few wrappers like versioning, descriptions on the data expected in a document (or namespace) and some limits on the data types to allow for integrity checking.

```
<XML version 1.0>
<catalog xmlschemaurl="http://site.org/catalog.xsd=>
<product>
<drill>
<description>3 inch drill bit</description>
<code>AB123CDF</code>
</drill>
</product>
```

In addition a developer creates another document called an XSD document describing, in XML, the fields and data types expected. Although XSD is falling out of favour, being replaced by scheamas

3.4.2 Why all this for IBE ?

There are a number of 'touch points' for external data processing in the IBE mail application

- Configuration : relatively static data like the SMTP relay servers name or IP address, proxy servers etc.
- Public parameters like the server's elliptic curve descriptions. This can be local or served via the net

```
<?xml version="1.0" encoding="UTF-8" ?>
<IBEMessage>
<IBEEncryptedSessionKey>
<ds:KeyInfo>
<ds:KeyName>neil@ibe.dcu.ie</ds:KeyName>
</ds:KeyInfo>
<ibe_enc_sk>
<x>439da011d8b7b8e6c4f154c9602
83be9b397a89c4eefd7ecea0d4931d
e4961b47bc3f8f71c57d89446a946b
c15676b2a3c2eb40933beb141f11b7
21850d33fd3</x>
<y>77a53e9a126330d00916f2deaee
a0d5d471c3d3e81f26105425838e05
b939baa93ec7da1810ec03084c126e
6c324222e03519aa731d4a7d87feb4
cc3a3cdbe68</y>
<v>0bce4270b1653825f4bcbbd61b91d62e</v>
<w>82aec6fa590eb104df284b7981671773</w>
</ibe_enc_sk>
</IBEEncryptedSessionKey>
<EncryptedData
xmlns="http://www.w3.org/2001/04/xmlenc#"
xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3.org/2001/04/xmlenc#xenc-schema.xsd">
<EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#kw-aes128" />
<CipherData>
<Transforms><ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#base64"
/></Transforms>
<CipherValue>REEFCtTuOaqj2PbOPyQFUd69fFU
36DLRlAn1KPVbceCEprvEdPC5lvNfhBKmd6dYJOK
Guno2z+i2BLHSa+qCfS4FKNaMJwjjGpzPiMrD0e+
JryROxo+EpvhqEiTK7KSQtFfw3foLzavE9jRBm3ol/w==
</CipherValue>
</CipherData>
</EncryptedData>
</IBEMessage>
```

Figure 3.1: Sample IBE message

- The eMail messages passing through the application.

The specifications of each of the XML files are too detailed to expand on so to demonstrate I'll illustrate using just the actual email message on the wire.

Fortunately there has been a lot of work done in this area of adding encryption to XML by the DigSig interest group [62]. We are essentially adding on our XML tags on top of this. Preference is given to using any existing tags that are relevant.

An actual email message in the IBE mail system looks like :

The interesting sections are

- `<KeyName>` which allows us to specify multiple IDs (public keys) that the message is encoded to. Multiple recipients can have one body of symmetric encrypted data (in AES) and only the ephemeral keys need to be IBEed to each ID key.

- `<ibe_enc_sk>` is an actual session key encrypted in IBE, `<X>` and `<Y>` being points on the elliptic curve.
- `<EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#` etc. >

  describes the XML name space for XML encryption.
- `<EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#kw-aes128" />`

  describes the algorithm used for the symmetric encryption. In this case 128 bit AES.
- `<CipherData><Transforms><ds:Transform Algorithm=`

  `"http://www.w3.org/2000/09/xmldsig#base64" />`

describes the data as been encoded (transformed) to Base64.

Finally

- `<CipherValue>REEFCtTuOaqj2PbOP...>` is the actual encrypted data.

Chapter 4

PKI & IBE in practice

4.1 PKI issues

While there are many advantages to public key cryptography some of the downsides are the problems of

- Locating the public key in a communications setting like email where neither side 'speaks' directly to each other. The solution is either the overhead and often impractical idea of each side issuing the public key in some out-of-band authenticated channel, or for large public directories holding up to date public keys.
- Trusting the other side even if you don't know them. This introduces the concept of a Trusted Third Party (TTP) where each side has a TTP sign their keys to allow the other to trust, not each other, but a third party (like a government agency or bank) who makes sure (validates) the identity of each party. This solution is kind of 'chicken and egg' because one also has to find a common TTP and make sure their keys haven't been compromised.
- Knowing that the public keys you have aren't associated with a compromised private key. What if, for example, your recipient has 'lost' his/her private key and you use its associated public key to encrypt a message ? It is possible that an adversary has managed to find it and can therefore decrypt any messages with this stolen private key.

The common solution to all these problems is the idea of a certificate. Each public key has some additional information associated with it such as key validity dates, identification data, information on where status information on the certificate can be located, and the signature of a trusted third party who has validated that the certificate owner is, in fact, who they say they are.

Then there acan be secure hardware holding the private keys. These range from smart cards for client keys to expensive tamper resistant hardware for server or 'important' keys.

Next there are more directories, directory entries, or signed lists of revoked certificates and Certificate Revocation Lists (CRLs) holding information about certificates which are no longer valid.

Then there are servers who can answer queries about certificates via OCSP (On-line Certificate Status Protocol) [36] which is a query to an authoritative server about the current status of the certificate as viewed by another, usually a TTP.

This then introduces small insecurities around time shifting (can I really trust the revocation list I just got ?), and makes the whole usage extremely complicated for the non specialist user of security. As you can see this 'simple' concept gets rapidly complex and expensive.

For more information on PKI see [3] and for a closer look at PKI standards see [28].

4.1.1 PKI repudiation problem

A recent example of the scalability and centralisation problems of PKI was the recent failure of Verisign's Certificate Revocation structure as it apparently was not designed to handle the load of large numbers of systems accessing crl.verisign.net. When the date passed over from 2003 to 2004 a number of old code-signing certificates went out of date. Verisign were forced to shed 50 percent of the load on their servers. Verisign redirected traffic to several RFC1918 [21] (black) addresses, which are not routable on the Internet but are frequently used in enterprise networks. It is claimed that Verisign had thus created a Denial-of-Service issue on enterprise services using the same RFC1918 addresses as internal systems checking for crl.versign.net. The consolidation of 'security network power' in a single company creates its own threat to the critical infrastructure when a single certificate expires instead of being randomly distributed among several different organizations.

4.2 IBE issues

4.2.1 IBE Public Key construction.

The models outlined previously hint at a Utopian world where there is a master elliptic curve and key distribution center (KDC) that all parties worldwide subscribe to. In reality its much more likely that there will be a multiple of KDCs as each enterprise and Government is more likely to be able to verify users IDs locally and that they will be concerned with escrow issues as a master system could be compromised or susceptible to a remote countries data protection rights which may not be compatible with their own.

This then leads to some interoperability problems as users try to acquire the system parameters for a domain from which the recipients keys are likely to be issued and the policy that the specific KDC operates for the formatting of public key ID strings.

In practice the curve parameter distribution is simpler than the theoretical model. The generation of unique KDC curve public parameters $(\mathbb{G}_1, \mathbb{G}_2, e, n, P, P_{pub}, F, H)$ is not required for the security of the system to be maintained.

While keeping their individual security intact each KDC can associate with a 'master curve' and can reuse most of the parameters. The KDC reuses the public parameters $(\mathbb{G}_1, \mathbb{G}_2, e, n, P, P_{pub}, F, H)$ and only need to vary the master secret s (kept private) and issue a new $P_pub = sP$.

To facilitate this concept of 'fixed parameters' rather than a multiplicity, the project team picked and named a small set of standard curves (IBE0, IBE1, etc.) and the software and file formats are optimised to take advantage of these fixed curves. A message therefore embeds the curve *name* rather than a list of all parameters.

This practice has been adopted before in other forms of ECC where a number of curves have them ratified, numbered, and recommended by NIST.

There are numerous advantages to adopting such a scheme

- The curves can be placed under more rigorous security analysis.
- Interoperability testing can be restricted to a know set of curves.
- Bandwidth is saved and each KDC connection doesn't need to download all the curve parameters

4.2.2 Future Encrypting.

The scheme outlined in Boneh & Franklin refers to an ID being used as the public key for IBE and implies that this ID is an email address. In fact the system works for any string and this opens up some interesting properties not currently available in existing systems.

If the ID string is simply concatenated with a date string it is possible to build encrypting systems which encrypt *into the future.*

For example, consider a scenario in which the KDC issues identity-based private keys by setting ID to be an ID concatenated with a date string, and Alice encrypts to Bob by applying an IBE scheme to (Bob's ID || ID = date), e.g. "bob@wonderland.com || 25th December 2004". This scheme is discussed in Boneh & Franklin's paper.

Why is this arrangement useful?

Imagine Alice is going away for Christmas holidays. Alice can encrypt Bob's Christmas present and safely give Bob the cipher text knowing the Trusted party (KDC) will not issue

Bob a corresponding private key for the encrypted message until the message is presented on the date encoded inside the (human readable) public key string.

Another scenario is a case that the KDC issues public keys like in the above example but with a short window such as

"(Bob || week number || year)" e.g. (bob@wonderland.com || 42 || 2004)

In this case its possible for the KDC to issue Bob (on approving his credentials) a set of keys for the future and allow Bob to remain offline from the KDC and yet still receive messages.

The benefits here are multiple.

- The KDC doesn't need to be a highly available system,

- If Bob loses a private key then it is only messages for the short window (like a week in the example) that can be compromised.

- It is much more suitable for an environment where bandwidth is scarce as it limits the need for the system to be connected to acquire decryption keys.

The problem is that Alice needs to understand how to construct a public key / ID string for Bob for which she can be confident Bob will have a decryption key. Therefore its important that in addition to the system parameters, the KDC issues a machine readable policy on how to construct public keys for that particular curve.

- This concept of variable ID string is possible to expand upon infinitely and there have been proposals that the ID could hold more data to perform full policy and role based ID action strings. In an environment where IBE technology is used to authenticate users as well as encrypt data (for example VPN etc.) it is possible that the ID could contain information to the users intentions and/or roles and the KDC have, in real time, an approval engine to allow for such keys to be issued.

As an example consider IBE used in a remote terminal client-server model. The encrypting ID string could hold information about the connecting user, the reason why login is needed, a time window for the policy to be allowed and the list of authorizing users that need to sign off on access rights.

Another advantage is that the idea of keys generated to cover a short time window helps in the area of revocation. One of the biggest problems with traditional PKI systems if that of revocation. For a number of reasons, mainly commercial but with some technical inputs, X.509 certificates are generally issued for a year. One needs to knowing that a public key is still a valid token to use and that it hasn't been reported as compromised since it was acquired. IBE changes this model somewhat, as its switches the revocation model from one dependent on the encryptor to one where its shared between the receiver and the KDC. This switch puts much less demands on infrastructure and bandwidth requirements.

Now that the issue of worrying about compromised keys lies with the KDC we can see that if the keys are only issued with a short time window (like a week or a day) then the risks of compromise are much reduced.

4.2.3 KDC system parameters and policy

Sections 4.2.1 and 4.2.2 outlined how the simple model of encrypting to an identifier in IBE quickly gets more complex than originally envisioned. One of the major issues for client software to deal with is the issue of how exactly does it figure out how to construct an ID suitable to use as a public string, knowing that it will be properly received and dealt with by the receiving entity.

As mentioned in 4.2.1 it is unlikely that one unified method for public-key-ID formats will be adopted, and even if such a policy existed one would still need to acquire (securely) the KDCs curve parameters.

I believe that the following will be the model of choice for most KDCs

- Like most CAs in traditional PKI, the KDC will manage a number of curves, each with separate policy for authentication and key issuance.

- The KDC will publish each curve's identifier, unique parameters and a 'policy' at a well known URL.

- The Policy will be machine readable, and most likely encoded in XML.

- The Policy will describe the type of strings expected as 'IDs' for a given curves e.g.

(id || week number || year) or (ID || begin-date-format || end-date-format)

- The Policy will describe measures take by the KDC to verifier a users ID credentials. This is to satisfy any concerns an end entity may have about securing confidential information using these parameters where the TTP may have given out keys in error.

- The Policy is likely to make a statement about its escrow policy.

- The Policy is likely to make a statement about the intent of the KDC to keep user's data, for example, to keep any authentication data for a period of time.

4.2.4 Locating KDC parameters

To find a suitable KDC for an end user it is likely that the encrypting party will use a KDC location dependent on the *domain* associated with an ID.

Most electronic identity contain some form of domain information

- An email identity is of the form *user@domain.*

- An IP address is from a block assigned to a domain by an ISP
- A phone number is of the form +(country code)-(provider code)-number.

We contend that an end entity in IBE will make a lookup to a well known location based on the domain part of an ID

Examples:

- For a user *alice@wonderland.com* the end entity could do a HTTP 'GET' to a URL based on the domain wonderland.com for example

  http://kdc.*wonderland.com*/kdc_policy.XML

- For a phone number +(353)-(87)-42222. The End entity could make a call to +(353.87.12345) where 12345 would return an encoded form of the 87 domain's policy.
- For a user *alice@wonderland.com* the end entity could do a DNS lookup to an new mail record (MX) type. based on the domain wonderland.com e.g.

```
> dnsdig type=IBE_kdc wonderland.com
```

This method of associating the domain with the KDC was viewed by the projects sponsors (Enterprise Ireland [29]) project team as a significant step in the commercial realization of IBE technology and has since been put forward for intellectual property protection (patent) under Enterprise Ireland's Commercilisation of Research & Development (CORD) scheme. The actual claim to both the European and US patent offices is reproduced in Appendix A.

4.3 IBE Vs. PKI

4.3.1 Comparison Points

In a traditional PKI the main problem Alice faces when attempting to encrypt a message to Bob is how to reliably obtain Bob's public key and be certain that at the time of use that Bob's public key is still valid. Typically this involves a Certificate Authority (CA) who validates Bob's identity and if satisfied signs Bob's key. Then the CA usually maintains an infrastructure to publish this key and an on-line service to distribute up to the minute information about a public key's status.

An IBE scheme overcomes this by replacing the CA by a private key generator (PKG) which publishes its public parameters and then operates like a CA in that it will verify Bob's identity before issuing him with a private key.

An advantage of IBE over PKI is that the PKG does *not* need to manage infrastructure hosting a public key database nor the parallel information server keeping real-time data on public key status. This is because the public key of Bob is *implicit* in the identity : Alice can generate Bob's public key without needing to go on-line. The PKG solves the revocation issue by embedding time period windows (e.g a week number) inside the ID it assigns to Bob. Bob needs to renew his keys at the beginning of this next time window.

4.3.2 IBE Advantages

The main advantage to IBE is this ability of one end entity to generate a public key of another's without having to find it on the CA's infrastructure or from the other entity in a prior transaction. This saves on bandwidth requirements on the client and saves the cost of always-on, highly-available, securely hosted servers available in a many-to-one model. In comparison all the IBE PKG needs to do is distribution of private keys in a one-to-one setting.

This also has business advantages over the PKI model where the services have to be available to all, often third-party non-paying clients. In IBE the server can choose only to work with paying registered clients and has no reduction on the security of the model.

Another advantage is the fact that, unlike PKI, the keys are generated centrally, and that keys can be recreated on demand by re-entering the original parameters. When there are legal demands for retaining keys, like those imposed by US regulatory bodies on public companies, IBE makes it extremely simple to mange the potential large key set an organization generates over time. Only one secret needs to be secured.

4.3.3 IBE Disadvantages

One disadvantage of IBE is the requirement that the channel over which an identity's private key is transported from the PKG to the identity needs to be secure. In a PKI system there is no such equivalent transaction but there are demands that other channels (OCSP) do need security.

Another potential problem is that of revocation. If, for example, Bob loses his private key, does Bob need to change identity ? The solution is to modify the ID public key string to add time window based keys. The management of the increased number of keys adds some overheads to the end user.

The main perceived disadvantage of IBE over other security schemes is the issue of privacy. The IBE scheme is inheirently, *escrowed,* in that the PKG has the ability do recreate the keys if it so wishes and it therefore decrypt any IBE secured data. It is important that the PKG is trusted and has some public policy on its powers of escrow. Many view this as prohibitive to mass deployment in general - that there is some kind of big brother who

decides what can be read. The counter argument to this is that one often uses encryption in a *role* rather than as an individual and that ownership of the encrypting keys belongs to the PKG and is tied to that role.

PKI suffers from this escrow problem with respect to digital signing : if the CA wants to sign data on behalf of an entity all it has to do is generate a fake key pair and sign the public key saying it has been supplied by the entity.

This ability of the PKG to decrypt all data makes it a vulnerable point of attack and subsequently it has to be managed in a highly secure manner.

These escrow problems for both the IBE and PKI systems can be overcome by methods like splitting up the master secret or CA keys among a number of parties whereby they can only be used in collusion. An alternative is a *hierarchical* IBE where the actual keys are created not by the root PKG but by sub-PKGs.

Chapter 5

Implementation

The software required to integrate IBE in both desktop and server applications with low impact, with the highest level of code reuse with portability, requires analyzing a number of software development technologies and trading off between ease of development and functional requirements. The following chapter outlines just a few of the coding technologies used in the actual delivery. We now describe the software developed to integrate IBE into both desktop and server applications. The aim was to deliver industrial strength software with low hardware requirements. Design decisions allowed for the highest levels of code reuse, whilst allowing maximal portability across platforms. This required analyzing a number of software development technologies and making various trade offs between ease of development and functional requirements. The following chapter outlines just a few of the coding methodologies used in the software, with arguments justifying the decisions made.

5.1 Thread Overview

The inner workings of the proxy applications depend heavily on threading techniques. But what exactly is a thread? The term thread is shorthand for thread of control. A thread of control is, at its simplest, a section of code executed independently of other threads of control within a single program.

5.1.1 Thread of Control

Thread of control sounds like a complicated technical term, but it's really a simple concept: it is the path taken by a program during execution. This determines what code will be executed: does the 'if block' get executed ?, or does the else block? How many times does the while loop execute? If we were executing tasks from a "to do" list, much as a computer executes an application, what steps we perform and the order in which we perform them is our path of execution, the result of our thread of control.

5.2 Overview of Multitasking

It is assumed that the reader is familiar with the use of multitasking operating systems to run multiple programs simultaneously. Each of these programs has at least one thread within it, so at some level, we're already comfortable with the notion of a thread in a single process. The single-threaded process has the following properties, which, as it turns out, are shared by all threads in a program with multiple threads :

- The process begins execution at a well-known point. In programming languages like C/C++ and Java, the thread begins execution at the first statement of the function `main()`.
- Execution of the statements follows in a completely ordered, predefined sequence for a given set of inputs. An individual process is single-minded in this regard: it simply executes the next statement in the program.
- While executing, the process has access to certain data : local variables are accessed from the thread's stack, instance variables are accessed through object references, and static variables are accessed through class or object references.

5.3 Overview of Multi-threading

All of this leads us to a common analogy: we can think of a thread just as we think of a process. We can consider a program with multiple threads running within a single instance just as we consider multiple processes within an operating system.

Within a program, multiple threads have these properties:

- Each thread begins execution at a predefined, well-known location. For one of the threads in the program, that location is the `main()` method; for the rest of the threads, it is a particular location the programmer decides on when the code is written.

- Each thread executes code from its starting location in an ordered, predefined (for a given set of inputs) sequence. Threads are single-minded in their purpose, always simply executing the next statement in the sequence.

- Each thread executes its code independently of the other threads in the program. If the threads choose to cooperate with each other, there are a variety of mechanisms (locks, critical sections etc.) that allow that cooperation.

- The threads appear to have a certain degree of simultaneous execution. The degree of simultaneity depends on several factors : programming decisions about the relative importance of various threads as well as operating system support for various features.

- The threads have access to various types of data. Each thread is separate, local variables in the functions that the thread is executing are separate for different threads. These local

variables are completely private; there is no way for one thread to access the local variables of another thread. If two threads happen to execute the same method, each thread gets a separate copy of the local variables of that function.

5.4 Why Threads?

Historically, threading was first exploited to make certain programs easier to write: if a program can be split into separate tasks, it's often easier to program the algorithm as separate tasks or threads. Programs that fall into this category are typically specialized and deal with multiple independent tasks. Often, these programs were written as separate processes using operating-system-dependent communication tools such as signals and shared memory spaces to communicate between processes. This approach increased system complexity.

The popularity of threading increased when graphical interfaces became the standard for desktop computers because the threading system allowed the user to perceive better program reaction. The introduction of threads into these platforms didn't make the programs any faster, one would need multiple or faster CPUs to achieve that, but it did create an illusion of faster performance for the user, who now had a dedicated thread to service input or display output.

Recently, there's been a flurry of activity regarding a new use of threaded programs: to exploit the growing number of computers that have multiple processors. Programs that require a lot of CPU processing, like our IBE proxy, are natural candidates for this category, since a calculation that requires one hour on a single-processor machine could (at least theoretically) run in half an hour on a two-processor machine, or 15 minutes on a four-processor machine. All that is required is that the program be written to use multiple threads to perform the calculation.

While computers with multiple processors have been around for a long time, we're now seeing these machines become cheap enough to be very widely available. The advent of less expensive machines with multiple processors, and of operating systems that provide programmers with thread libraries to exploit those processors, has made threaded programming a hot topic, as developers move to extract every benefit from these new machines.

However, threading in C/C++ (or Java) often has nothing at all to do with multiprocessor machines and their capabilities; in fact, the first Java virtual machines were unable to take advantage of multiple processors on a machine, and many implementations of the virtual machine still follow that model. A correctly written program running in one of those virtual machines on a computer with two processors may indeed take roughly half the time to execute that it would take on a computer with a single processor. But it also simplifies the external environment the software developer has to consider. One can write for an ideal world of one external interface, and provided the code is constructed right,

the software benefits from the operating system multi-tasking all events and demands on that interface through the threaded sub-system.

Threads are used in the IBE proxy to allow for a simple code architecture for both the receiving and sending entities. The proxy can take and process any number of client connections without any client-side connection noticing that it is sharing the resource with another. The client in this case could be an advanced email client which can download multiple mail messages in parallel from multiple accounts. Not designing the proxy with threading would either involved a very complicated multiplex design, or exposing the user to a perceived network failure by multi-connecting clients. It also allows us to have a constantly updating GUI during the computationally heavy IBE processing. This is achieved by placing the GUI code in one thread and each of the encrypting and decrypting processing in another. The operating system will allow each thread some small time slot on the CPU and to the user it appears as if they are working together.

5.5 Sockets

In the TCP/IP implementation of the OSI layered model, there are two main types of connections, connection orientated or connectionless. The main difference being the guarantee of data arriving in some order or the acceptance that some data can be 'lost' in transit. TCP is connection oriented and UDP is connectionless. Most of the applications we are familiar with sit on top of TCP and UDP.

As a programmer the connection oriented connection is logically defined by a 'socket'. Its like an electrical analogy where there is a connection between two types of equipment where a plug and socket exist at either end. TCP programming has this same jargon where a conceptual socket exists. The socket is a combination of an IP address. eg. 10.45.34.1 and a port number eg. 23 of the host machine. Programming this socket is about building and binding the socket and reading and writing data into this logical pipe. A library providing a special Socket access library is now nearly universally delivered by modern operating systems and commonly called the TCP/IP stack. The interface, or API, to this stack is almost standard but subtle difference exist between platforms. All higher level applications (web browsers, email, consoles etc.) use this stack. This common approach allows us to manipulate connections to our purpose. While it would be possible to replace the stack with a custom, security aware stack, this approach has failed in the past as replacing operating system components is generally frowned upon by IT administrators.

5.6 IBE for all TCP Protocols

The best way to implement an application which does more than secure email is to catch all protocols at a choke point. From the diagram in 5.5 notice how all TCP (and there-

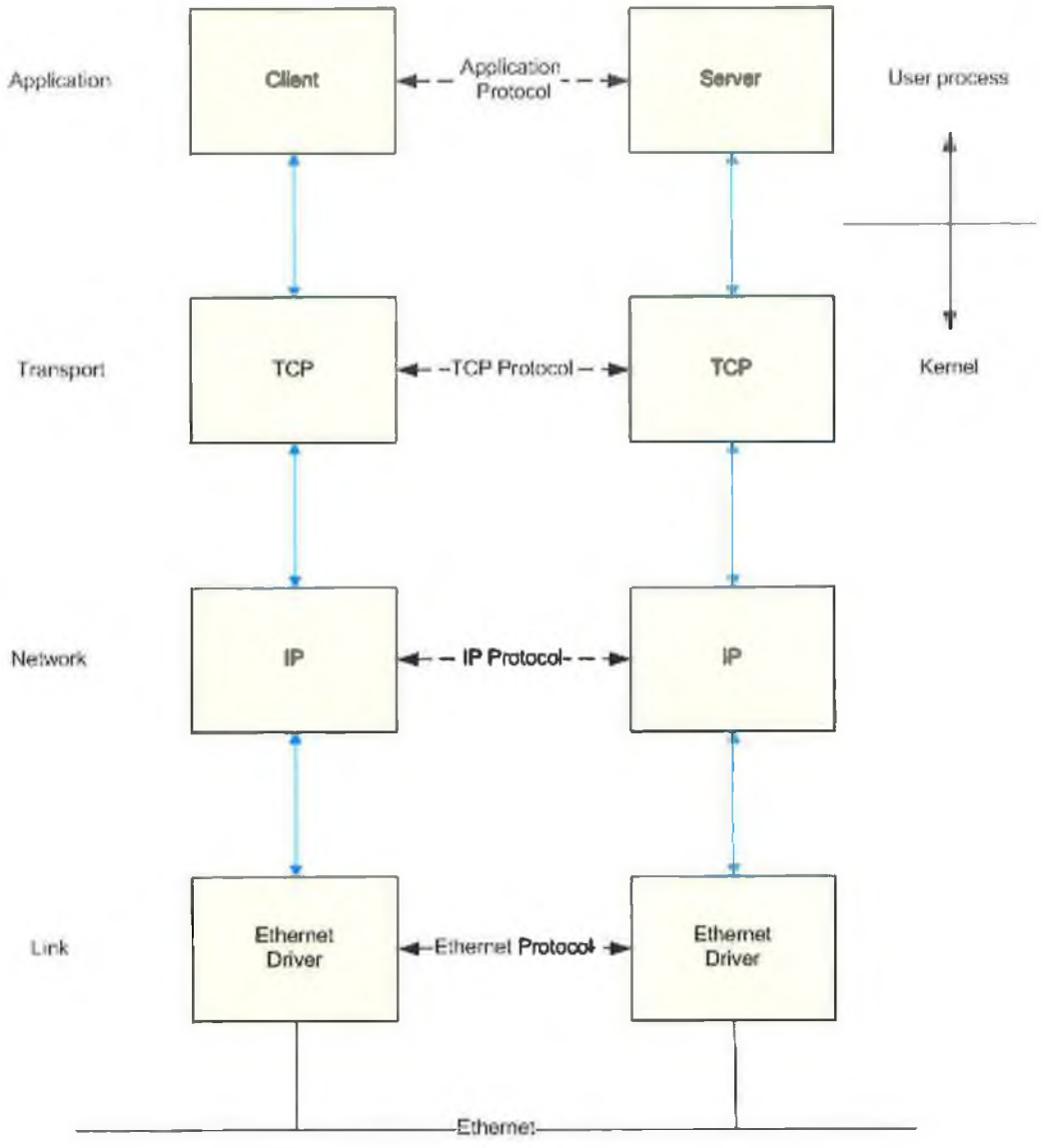

Figure 5.1: TCP/IP Stack

```
[HKEY_CURRENT_USER\Software\DCU\Socket Relay\Rule4]
"disable"=dword:00000000
"peer.addr.str"="localhost"
"peer.port.str"="5001"
"peer.mask.str"="255.255.255.255"
"dest.addr.str"="ibe.dcu.ie"
"dest.port.str"="25"
"secure.in"=dword:00000001
"secure.out"=dword:00000000
"ibe.public.parameters"="C:\\\\ibe\\neil@ibe.dcu.ie.xml"
"ibe.private.parameters"="C:\\\\ibe\\ibe.dcu.ie.xml"
"ibe.to.id"="neil@ibe.dcu.ie"
```

Figure 5.2: Configuration 'Rule'

fore socket based) applications require a server address either implicitly via DNS name or explicitly by direct address and then pass into the TCP layer. To intercept the traffic our solution is non-intrusive (from our design requirements), where the client/server application itself isn't modified but the client *configuration* is set to connect the client to a 127.0.0.1 (locahost) address on a non-standard port where our 'proxy' application is waiting to examine the connection and its data payload.

What we do is *redirect* the proxy client to a new server socket via a 'rule' which is just a set of parameters in the application's configuration. Listening on this new server socket is our proxy, acting as a server, waiting to re-interpret (parse) the data pipe and redirect this data stream to its original intended target.

An example of a Rule in the configuration illustrates this process.

Peer settings are where the data comes from, *dest* settings are where data is redirected to, *secure* indicate if the payload contains a secure data header contains key data, *ibe* settings are global curve data.

Our IBE client and server proxies (which can be the same application) essentially sit as re-directors in the data stream at the choke points indicated. Inside this re-directed stream, between the IBE client and IBE server we encrypt the data. The IBE proxy still resides in the application layer.

In our generic case of any TCP stream we use the IP address concatenated with the port number as the unique identity. This setup is possible to override by configuration.

Illustrated in steps

1. The client application (e.g. Telnet) opens connection to server
2. The client proxy catches it and inserts ibe session header with random AES session key.
3. The client proxy connects to the server specified in this port's rule.

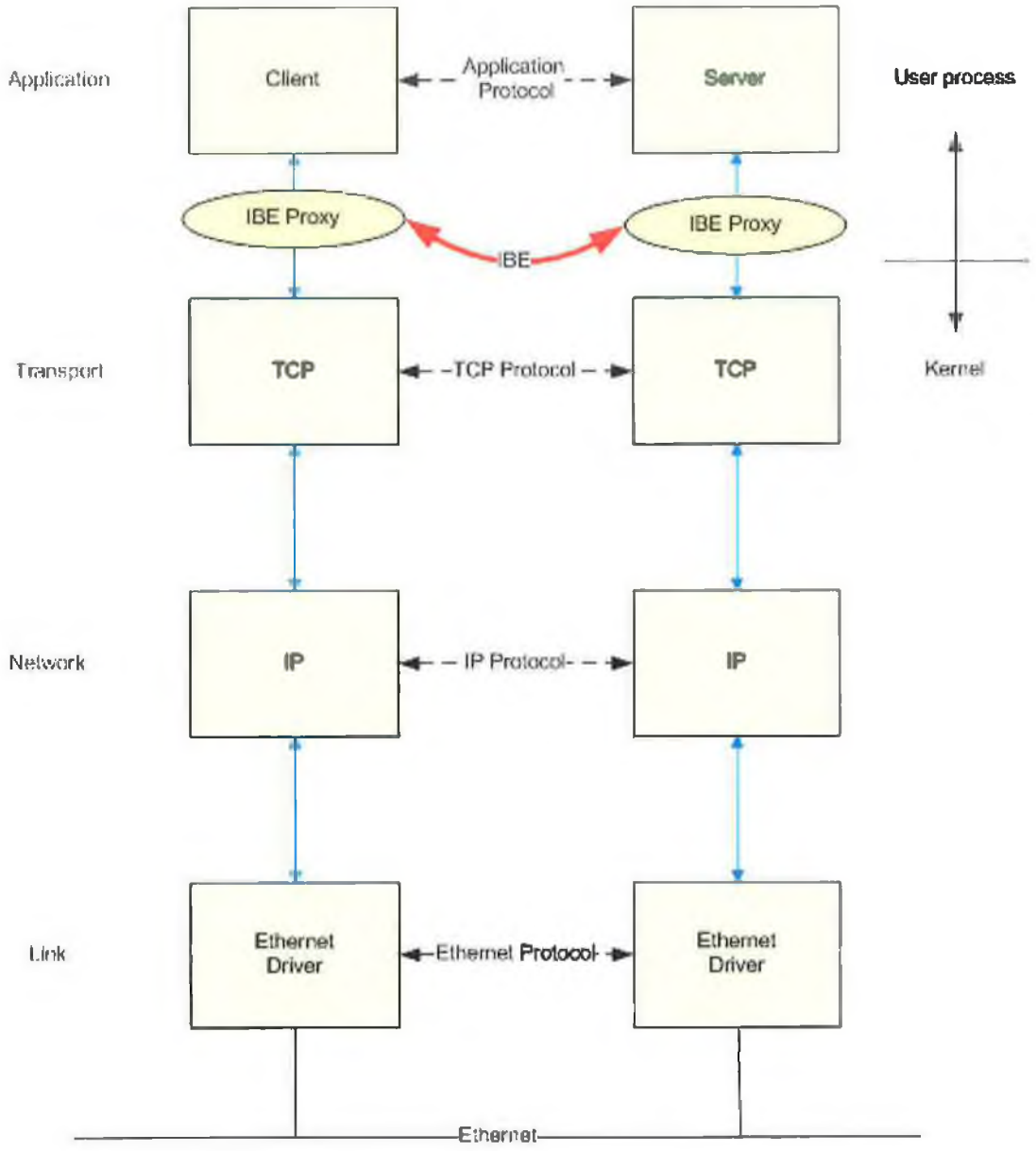

Figure 5.3: Socket Redirect

```
HELO bartspcMAIL FROM: bart@simpson.comRCPT TO: lisa@simpson.comDATAFrom:
"Bart Simpson" < bart@simpson.com >To:  lisa@simpson.com Subject:  itchy &
scratchy  love each other..QUIT
```

Figure 5.4: Example SMTP stream

```
HELO bartspcMAIL FROM: bart@simpson.comRCPT TO: lisa@simpson.comDATA<ibesess
>x,yc,v etc </ibesess><body cipher=AES>abcdef123456abcdef...</ibesess>.QUIT
```

Figure 5.5: Encrypted SMTP stream

4. The server proxy 'catches' the connection and removes IBE session header to get the bulk (AES) encryption session key.

5. The server proxy decrypts and passes data onto the intended server (also got from its rule).

6. The client and server applications require no modifications besides updates to their settings or configurations.

5.6.1 Email is a special case of generic network proxy.

In Chapter 3 we pointed out that securing email poses problems as the client and server don't directly connect allowing for each side to swap and verify security parameters like public keys. We designed our proxy to make a special case when it is configured as an email agent (i.e. listening to traffic on ports 25 or 110). The setting is explicit as the well known ports may not be actually used by email clients and the overhead for auto-detect by traffic analysis isn't worth the benefit.

We do this by observing that if one *flattens* each side of each protocol we see it really is just a deterministic token stream which we can parse

eg. SMTP

What we do is on sending (SMTP) is watch the data stream for the DATA token and splice in the secure parameters (Example in figure 5.5. Parameters in red) and all subsequent data is AES encrypted in blue.

What we then do in the reciprocal POP3 session is :- have the listening agent catch the message by watching for first the POP header data, then catch the XML tag for <ibesess>. Then strip out all the subsequent payload out and decrypting the data using the AES session key which was encrypted in the 'IBE session block' in the header.

5.7 Non-Blocking I/O

In most programming languages, when you try to get input from the user, you execute a read() method specifying the user's terminal or some kind of GUI dialog. When the

program executes the read() method, the program will typically wait until the user types at least one character before it continues and executes the next statement. This type of I/O is called blocking I/O : the program blocks until some data is available to satisfy the read() method.

This type of behavior is often undesirable. If you're reading data from a network socket, that data is often not available when you want to read it: the data may have been delayed in transit over the network, or you may be reading from a network server that sends data only periodically. If the program blocks when it tries to read from the socket, then it's unable to do anything else until the data is actually available. If the program has a user interface that contains a button and the user presses the button while the program is executing the read() method, nothing will happen: the program will be unable to process the mouse events and execute the event-processing method associated with the button. This can be very frustrating for the user, who thinks the program has hung.

Traditionally, there are three techniques to cope with this situation:

5.7.1 I/O multiplexing.

Developers often take all input sources and use a system call like select() to notify them when data is available from a particular source. This allows input to be handled much like an event from the user (in fact, many graphical toolkits use this method transparently to the user, who simply registers a callback function that is called whenever data is available from a particular source).

5.7.2 Polling.

Polling allows a developer to test if data is available from a particular source. If data is available, the data can be read and processed; if it is not, the program can perform another task. Polling can be done either explicitly–with a system call like poll()–or, in some systems, by making the read() function return an indication that no data is immediately available.

5.7.3 Signals.

A file descriptor representing an input source can often be set so that an asynchronous signal is delivered to the program when data is available on that input source. This signal interrupts the program, which processes the data and then returns to whatever task it had been doing.

While this issue of blocking I/O can conceivably occur with any data source, it occurs most frequently with network sockets. If you're programming sockets over the Internet,

the chances of this backlog happening are greatly increased; hence the chance of blocking while attempting to write data onto the network is also increased.

5.7.4 Approach

Code steps inside the proxy application are

- Start.
- Thread random.
- Thread reading Config (GUI or Service is main()).
- Read Configuration as global data and then a set of Rules.

  A Rule is :

  What port to listen on (e.g 110)

  What server IP / Host + Port to connect to, Socket Filters Etc.
- For each rule set up listening thread
- On connect:

  Check rule /filters and set up tight read/write proxy I/O. (low memory overhead)
- In Input/Output loop
- Parse based on rule and *inject* or *remove* IBE session details
- Generate or extract random AES key encrypted with IBE and after which do AES bulk {en|de}cryption using IBE session details

note: All logging data, GUI interaction etc. go through calling functions stubs for specific operating systems or environments ensuring portable code construction.

5.7.5 The Input/Output loop

The following code snippet is the thread handler for an incoming socket this can be either a client or server. The 'rule' is found for connections on that socket (and incoming IP) to determine where to connect too. This rule (held in the data structure `dst`) also contains the setting (`dst->secure_out`) which determine to encrypt (adding an IB Encrypted AES key in front of the data stream handled in `secureParametersSend()`) or to decrypt, stripping our this header to get the key (in `secureParametersRecv()`). The code has minor editing changes from the source to make it more readable.

```
////////////////////////////
// This is the actual I/O thread. One of these per connection.
// Uses blocking I/O with select.
DWORD WINAPI streamIoThread(PCONN conn)
{
SOCKADDR_IN dest;
SOCKET dest_socket;
#ifdef WIN32
   FD_SET rfd;
#else
   fd_set rfd;
#endif
PRULE dst;
char sesskey[IBE_HEADER_BUF_LEN];
char ntoa_buf[128];
int rc,src_pending,dest_pending;
STATE state;
aes_state aes_s;
memset(&dest,0,sizeof(dest));
initialiseProxyState(&state); // initialise STATE
// FindRule adds to the reference count,
// so we will have a valid rule handle even if the rule is
// later deleted by the user !
// And FindRule locks out FreeRule while it's searching,
// so we should be thread safe...
dst=FindRule(conn->src_addr,conn->port);
if((rc=ioThreadSetUp( conn, dst,dest, &dest_socket )))
   return rc;
if(dst->secure_out={
    rc=secureParametersSend(
             dest_socket,dst->IBEPublicParameters,
             dst->IBEToID, sesskey,
             IBE_HEADER_BUF_LEN, &aes_s );
}
if(dst->secure_in){
    rc=secureParametersRecv(
              conn->src,dst->IBEPublicParameters,
              dst->IBEPrivateParameters , sesskey,
              IBE_HEADER_BUF_LEN, &aes_s );
}
if(rc==0){
    rc=CAESInit(sesskey,&aes_s);
}
```

Then the function enters a loop, constantly polling the source and destination sockets to see if any data is pending. If data is pending it determines which socket and then passes on the connection into the 'tightReadWrite()' (see below). Notice how the socket parameters to this function are switched depending on the socket polled allowing for reuse no matter the function (encrypt or decrypt) required.

```
for(;;){
    FD_ZERO(&rfd);
    FD_SET(conn->src,&rfd);
    FD_SET(dest_socket,&rfd);
    if((rc=select(FD_SETSIZE,&rfd,0,0,0))<0) { break; }
```

```
        src_pending=FD_ISSET(conn->src,&rfd);
        dest_pending=FD_ISSET(dest_socket,&rfd);
        if(src_pending){
            if(tightReadWrite(
                    conn->src, dest_socket, sesskey,
                    dst->secure_in,dst->secure_out ,&aes_s)
                            <=0)break;
        }
        if(dest_pending) {
            if(tightReadWrite(
                    dest_socket,conn->src, sesskey,
                    dst->secure_out,dst->secure_in,&aes_s)
                            <=0){break;}
        }
    }
```

If either socket is closed (select() reports this) the loop is broken and the function cleans up. The global statistic data is surrounded by a critical section to make sure it is thread safe. This data is passed into the performance monitor data collection in windows or the proc mon subsystem in Unix. The logging function `WriteLog` is not a native event log or syslog call. This thunking allows us to substitute an operating system dependent logging function in a separate code module which is complied in depending on the OS environment parameters.

```
/// and clean up....
if(settings.stats_enable){
   EnterCriticalSection(&stats_cs);
        stats.n_active--;
    LeaveCriticalSection(&stats_cs);
}
rc=WSAGetLastError();
strncpy(ntoa_buf,inet_ntoa(conn->src_addr),sizeof(ntoa_buf));
if(settings.verbose){
    WriteLog(ERR_INFO,cc_disconnect_msgID,src_filename,__LINE__,
                "Disconnect %s.%d  from %s.%d: %s",
                 ntoa_buf,ntohs(conn->src_port),inet_ntoa(dest.sin_addr),
                 ntohs(dest.sin_port),WSAErrStr(rc));
}
closesocket(dest_socket);
closesocket(conn->src);
FreeRule(dst);
cc_free(conn);
_heapmin();
return 0;
}
```

The design guide for the proxy specifies a low impact. This IO thread simple reads and writes the socket data inline, reading a maximum 1024 bytes, typically the size of a TCP frame inside the TCP/IP stack, adding little extra buffering overheard.

```
static int tightReadWrite(SOCKET in, SOCKET out, const char *sesskey,
                          const int recvSecure, const int sendSecure,
```

Figure 5.6: Read/Write Loop

```
                                aes_state *Paes_s)
{
    int rc, wc;
    char buf[IO_BUF];

    memset(buf,0,IO_BUF);
    if((rc=recv(in,buf,IO_BUF,0))<=0) // EOF on recv {
        return rc;
    }
    if(recvSecure){ //decrypt
        char aesBuf[TMP_AES_BUF];
        int lenAES=0;
        memset(aesBuf,0,TMP_AES_BUF);
        lenAES=CAESdecryptFinish(buf, aesBuf,Paes_s);
        if(lenAES>=0)memcpy(buf,aesBuf,lenAES);
        rc=lenAES;
    }

    if(sendSecure) {
        char aesBuf[TMP_AES_BUF];
        int lenAES=0;
        memset(aesBuf,0,TMP_AES_BUF); //encrypt
        lenAES=CAESencryptFinish(buf, aesBuf,Paes_s);
        memcpy(buf,aesBuf,lenAES);
        rc=lenAES;
    }

    if((wc=send(out,buf,rc,0))!=rc) {
        return wc;
    }

    if(settings.stats_enable){
       EnterCriticalSection(&stats_cs);
           stats.n_incoming+=wc;
       LeaveCriticalSection(&stats_cs);
    }
    return 1;
}
```

5.7.6 Security issues with socket listeners

Our generic solution can act in either client of server mode, meaning it can be expect to be on the receiving end, or sending end of a pipe. Some security issues arise with this scenario.

For example given our client side sockets are in listening mode intended for the client on the local machine, there is an issue that any machine can connect to the socket, which, at API level doesn't have any special characteristics limiting the connections.

What we do is, at applications level above the socket, to examine all connections *after*

they occur and quickly compare to our own Rule (see figure 5.2) comparing the incoming source address with our Rule setting.

5.8 MIRACL

MIRACL [57]or **Multi-precision** Integer and **R**ational **A**rithmetic **C**/C++ **L**ibrary is a Big Number Library developed by Michael Scott which implements all of the primitives necessary to design Big Number Cryptography. MIRACL is compact, fast & efficient, portable with inline assembly for per processor optimizations. MIRACL does not merely provide an interface set of Cryptographic methods, but can be seen as a set of tools that enable any new number-theoretic technique to be implemented quickly. The primitives for the math behind IBE were built using examples taken from MIRACL's supporting code documentation.

5.9 Implementation language

5.9.1 C Vs C++ Vs Java Vs...

At the project start it was decided to use C++ wherever possible but to always consider options. As the proxy is to effectively 'add-on' to existing user set-ups an important design decision was to optimise performance wherever possible so as not to restrict distribution possibilities because of perceived or demonstrable overheads.

At each corner (GUI images, executable size, 3rd party DLLs requirements, crypto processing speed etc.) all options were considered and ultimately performance considerations won out.

Java was considered and during the project life cycle various prototypes were developed in Java including a standalone Mail client, a chat tool, and configuration management GUI.

The most optimal underlying crypto code was C++ based. Inside this library is processor tuned assembly code that is rebuilt on a per platform basis. It was considered to implement the proxy, GUI etc. in Java, but for both performance reasons (see below) and GUI quality it was decided the C/C++ hybrid was best with Visual C++ for Windows development and GCC/make for UNIX. Code was designed to be portable ANSI C.

It was my opinion before the project started that Java was nothing special and essentially C++ for people who didn't understand pointers. That the overheads of running virtual machines and garbage collection made it useful for nothing else than having more engineers available with the skills required to do basic software engineering. Its popularity came from a marketing campaign by Sun leveraging a popular anti-Microsoft bias prevalent among developers. "Some have called Java a cleaned-up version of C++ that eliminates the

confusing features of C++ rarely used and poorly understood" [2]. During the course of this project my opinion changed as I used it for rapid development of cross platform GUI prototypes like the chat client/server. Java is more like an evolution of C++ with C# being a not so distant evolution of Java (for similar commercial reasons as Java was marketed). The Java VM's ability to garbage collect in a relatively effective manner and the huge amount of Java supported add-on packages for file I/O, crypto etc. has made rapid coding and program maintenance much more streamlined. The fact that threading and networking can be considered 'native language features' is a huge benefit to a modern coding and system environment. Java's library having all the tools to effectively work with TCP/IP protocols like HTTP making accessing network resources as simple as working with the local file system was an attractive proposition for this network centric project. That the GUI is no longer platform specific is a huge bonus. However (at least currently) GUI style, and GUI development, is still pretty primitive in Java, it being both ugly and not up to commercial quality on most platforms. It seems like the lowest common dominator of the supported systems came to prevail. For our project Java's main downside came in the poor native support of the elliptic curve processing primitives over the demonstrated high performing MIRACL. It is also hard to use close to the bone memory management techniques to tweak every ounce of performance from each native platform. The machine abstraction is the price paid for turning memory management over to Java.

My colleague Noel McCullagh coded both a 100% Java IBE library and one that used the JNI (java native interface) to thunk into the optimised MIRACL code. Both have been shown to be usable with the only real obvious performance hit being on the PDA and smart phone platforms (Pocket PC and SymbianOS on ARM processors) where IBE primitive operations took over 20 seconds. In GUI / human interaction on standard desktop and server machines the overhead, while still present, was negligible.

5.9.2 Coding/ Coding guidelines.

The proxy code uses C++ syntax but little of the advanced C++ features like overloading and multiple inheritance. It mimics these features like Objective-C or the windows *winsys* code by using Object Oriented design techniques. An example of this is the two main data structures Rule and Config. Both are defined as C structs in a C++ 'syntax-ed' wrapped header. Both are complex 'deep C' structs/structures requiring element level memory management. Although any code with scope to see a variable of either type also has the ability to 'see' into what would be 'private' data types because of the C. All access to the data is via a read-only accessor or a modifier. Both are optimised with const qualifiers so as to allow for compile time optimisation.

Example of header file coding guidelines

```
#ifndef _ibeRule_H_
#define _ibeRule_H_     // <<---------- protects namespace using
```

```
                          //pre-processor rather then c++ namespace
#ifdef  __cplusplus       // <<---------- allows for module to be called
                          //from c without non portable C++ name mangling
extern "C" {
#endif
Struct {
  int numDestinations  // <<---------- proper variable naming convention
                       //              allows for maintenance
  char * ...
  ...
  } Rule, *pRule:
const int getNumDestinations(){ return numDestinations; }  //
                 //<<---------- use of const makes functions calling over heads
                 //             negligible via complier optimisation
setNumDestinations(const _in){ return numDestinations =_in; }
#ifdef __cplusplus
}
#endif
#endif
```

The benefit is coding discipline allowing for better maintenance and proper de-coupling of code. The requirement that the code should as easy to port to a restricted device like a PDA, smart-phone, and also to high-end systems like Unix, or mainframes, limited the choices. It was perceived that this decision (basic C++) would lead itself to portability. In hindsight it would have been possible to use inheritance and STL features as a time saver. There is a lot of simple data structure maintenance code which is error prone to develop and maintain.

One limiting factor was the Microsoft Window GUI. Deciding not to use a Microsoft framework like MFC/ATL or a third party like Qt, as it ties one too close to a particular variant of an operating system, made porting easier but each platform is then much more painstaking. I choose to do native 'raw' Windows GUI and system coding for the GUI, registry management, threading and shell interaction, even though there exist C++ wrapper classes. The C++ classes are heavy, have many dependences, have performance overheads and don't always support older (98, NT 3.x) systems. However there were huge amounts of code management to keep the discipline of thread safe GUI and registry calls coupled with robust memory management.

The code architecture decided the GUI implementation. In the advanced frameworks it's hard to truly separate out the portions, which are traditionally non-portable like GUI, logging, and configuration.

5.9.3 Code design for reuse.

True to the design guidelines the application is delivered in many different forms depending on the requirements with the core 'engine' being the same and written in a portable C++

.

The goal was to

- Deliver just one .exe executable module for whatever platform/format.
- To have little or no dependencies on specific versions of static or dynamic libraries being pre-installed.
- To have no installation location dependencies other than those required or suggested by specific operating system guidelines.
- Run with user level (non-administration) privileges for the run-time feature set.

This is not to say that any installer will not default to common practice when setting up or configuring defaults.

All the above features allow for painless upgrade for delivering maintenance releases.

The proxy application is packaged in the following formats :-

- Windows standalone with advanced GUI allowing for dynamic reconfiguration
- Windows lite-weight GUI for installations that have pre-configured settings which won't subsequently be updated. This allows for GUI to show 'something; is happening without over loading the user with too much technical information.
- NT service. To allow for background and always on installations with control panel applet for NT service remote configuration.
- Unix variants

Specific Platform/form dependencies tend to be

- GUI
- Configuration
- Installation
- Logging

In the following sections detailing each package I expand on the specifics of each. To understand the mechanics of the code I detail the main windows package and then describe the variations required for other package.

5.9.4 Versioning

Each application .exe or .dll containing versioning information. The 'About' button on any of the GUI main screens allows one to see the currently installed components and the relevant version information (see figure 5.7). In addition the 'About dialog' also gives a data of current build of the running application.

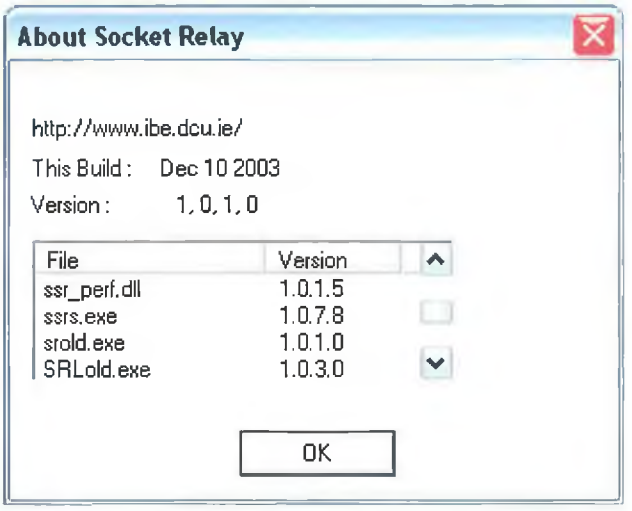

Figure 5.7: GUI 'About' Dialog

Figure 5.8: GUI Tray Dialog

5.9.5 Invocation

All applications are installed to open from the standard program start menu. Once running, any GUI based application can be opened from the system tray by clicking on the gray/black circle icon see in figure 5.8. When there any network traffic passing through the proxy this icon animates activity by spinning clockwise.

5.10 Windows Package Versions

5.10.1 Windows Advanced GUI

This package is a one .exe executable binary with no dependencies and can be run from any installation point by just double clicking. It required no parameters but you can change the configuration registry load point and display a 'sniffer' window to show what is passing through the proxy.

Command line parameters

- sniffer

the recommend installation point is

`\Program Files\Dublin City University\Secure Relay\SR.exe`

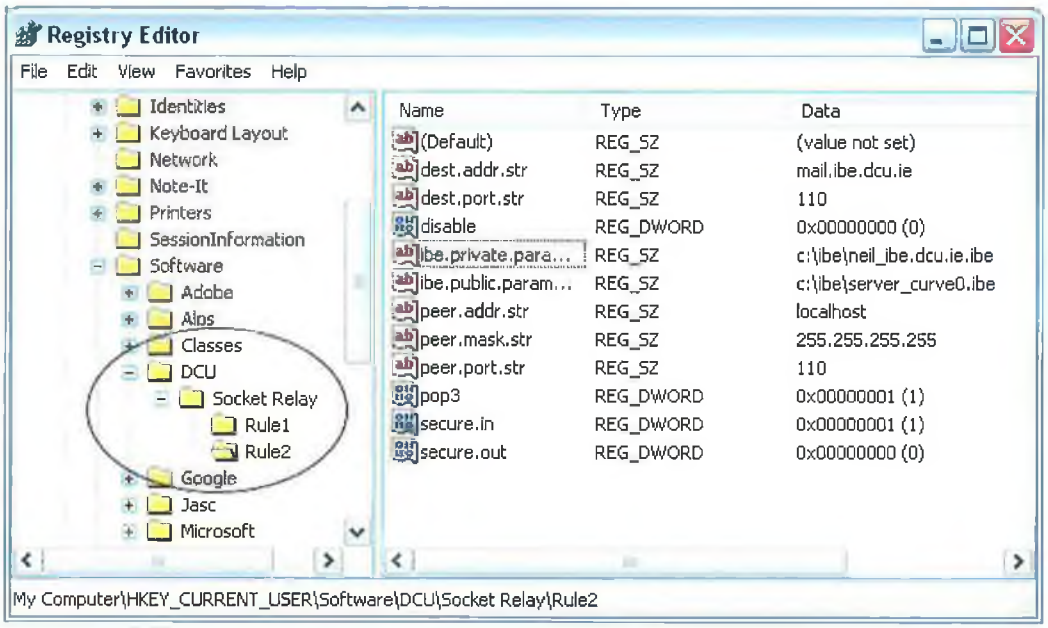

Figure 5.9: Registry Settings

With no command line options it reads its configuration from

`CURRENT_USER\Software\Dublin City University\Secure Relay\SR.exe` as per Microsoft guidelines. See 5.9

On start-up the application reads the current configuration into a set of data structures. The configuration can be system wide like log level or specific to a 'Rule'. All system wide parameters are stored at the root of the configuration and the rules are a variable list of sub keys under the root called `Rule1, Rule2, ... Rulen.` These rules can be seen in the screen shot of the Microsoft Registry editing tool in figure 5.9.

A full annotated description of the system settings is outlined in 5.5 on page 52. Each of the system parameters are loaded into the data structure 'Config' in the function `Configure (HKey root)` and then there is an array of Rules data structs allocated by the number of sub-keys under the registry root.

The two data structures are Config and Rules

`<config.h>`

`<rule.h>`

The data structure code is 'C' but with C++ class like accessors and modifiers compiled as const to allow the compiler optimise out any function call. The benefits of not accessing directly into the data structures allow us to carefully managed the memory allocations for optimum use of resources and to allow for various methods of GUI modification including distributed monitoring.

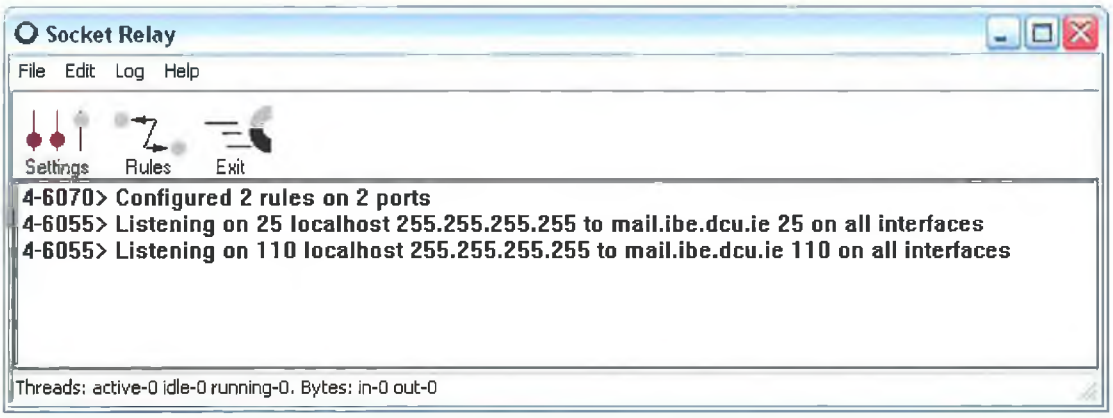

Figure 5.10: GUI Main Screen Dialog

After reading the configuration from the Registry the application displays the main command dialog as shown in figure 5.10. This example (5.10) shows a system with two rules one for POP3 and one for SMTP. Both have filters to accept connections from the local machine only, and connect to the relevant servers on mail.ibe.dcu.ie. It is in full expanded mode displaying each message ID for an administrator to trace any issues.

From this main start dialog it is possible to open either global settings or the rule editor by using either the tool-bar or drop down menus.

The 'Settings' button opens a tabbed dialog like the example in figure 5.11. This example reflects a setup that wants all types of logs to be recorded in expanded form but not to the level of debug which would add source code locations (module/line) of any errors or messages.

Selecting 'Rules' displays a dialog like the example in figure 5.12 and lets one create a new rule, delete existing rules, change the priority of a rule (only relevant to rules on the same ports) and edit any of the existing rules.

Selecting 'edit' or 'new' opens a dialog like the example figure 5.13. Note that any of the items added in any 'drop down combo box' are also saved to allow for reuse as 'history' for any list that appears when one choose to 'drop down' any combo box. The hosts in the host box are pre-filled to include the hostname and localhost. The mask list contains A, B and C net masks. The advanced button expands the dialog to include a few more settings including rule enable/disable and binding to a named interface for machines with a number of different interface cards.

Figure 5.14 outlines the interaction between each of the components outlined above. All GUI applications that allow for settings changes reuse the dialogs and their handling code which are implemented via callbacks.

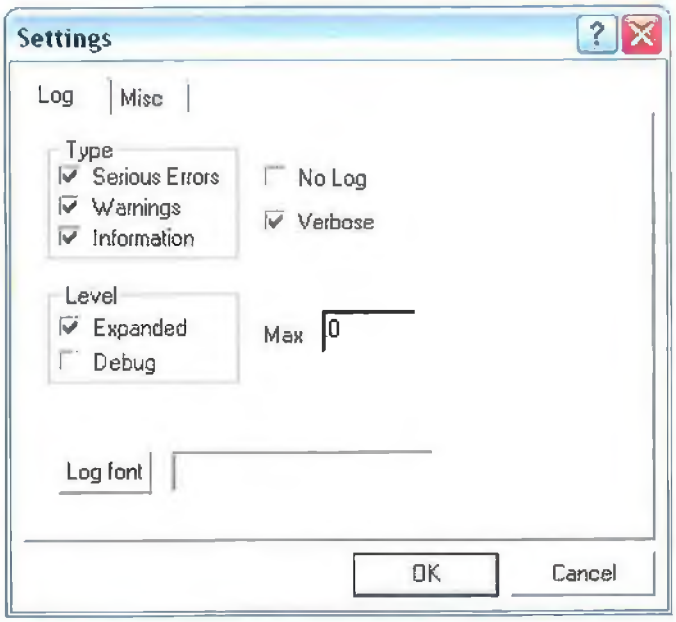

Figure 5.11: GUI Settings Screen Dialog

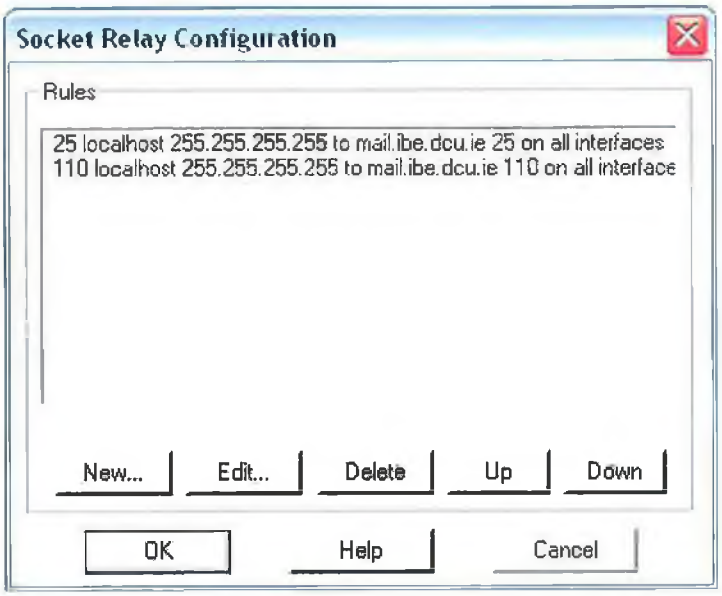

Figure 5.12: GUI Rule Select Dialog

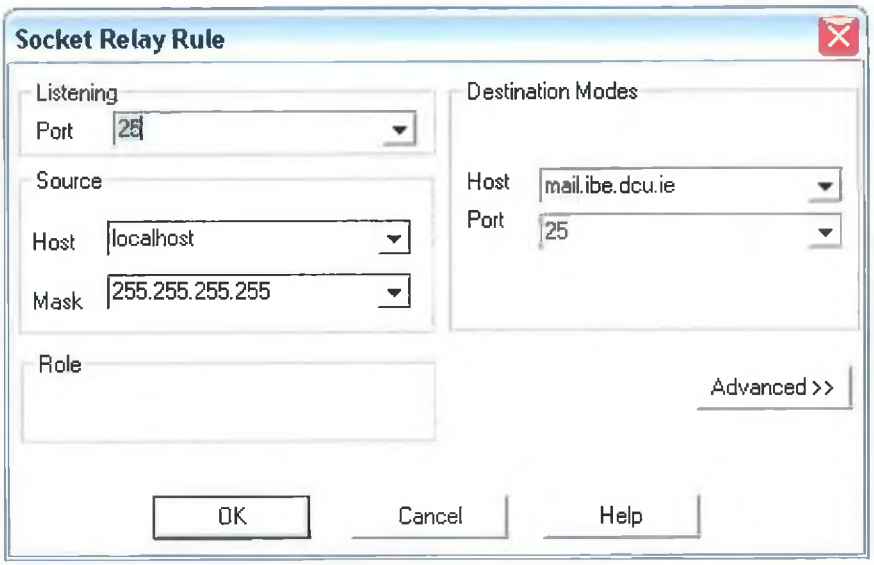

Figure 5.13: GUI Rule Dialog

5.10.2 Windows light GUI.

Is similar to the Advanced GUI version, includes all code and functionality but the GUI components are substituted with a smaller dialog which doesn't allow on to modify any of the Rule or global settings. It reads data from the same registry location.

5.10.3 Windows NT Service

On modern Microsoft operating systems (NT 3.5 and higher, Windows 2000, XP, media center and their future derivatives) there is a subsystem to allow for applications not suitable for end user GUI and that may run when the user is not logged in.

This suits us for a number of usage scenarios

- Use by users not allowed by security policy to modify settings.
- Home user set up where possible intimating security GUI would be a put off.
- Used as endpoints for network VPN tunnels.
- High secure systems where secure logging is paramount.
- Systems requiring remote management, ease of upgrade, and performance monitoring
- Providing security resources to other machines on a network like a firewall application.

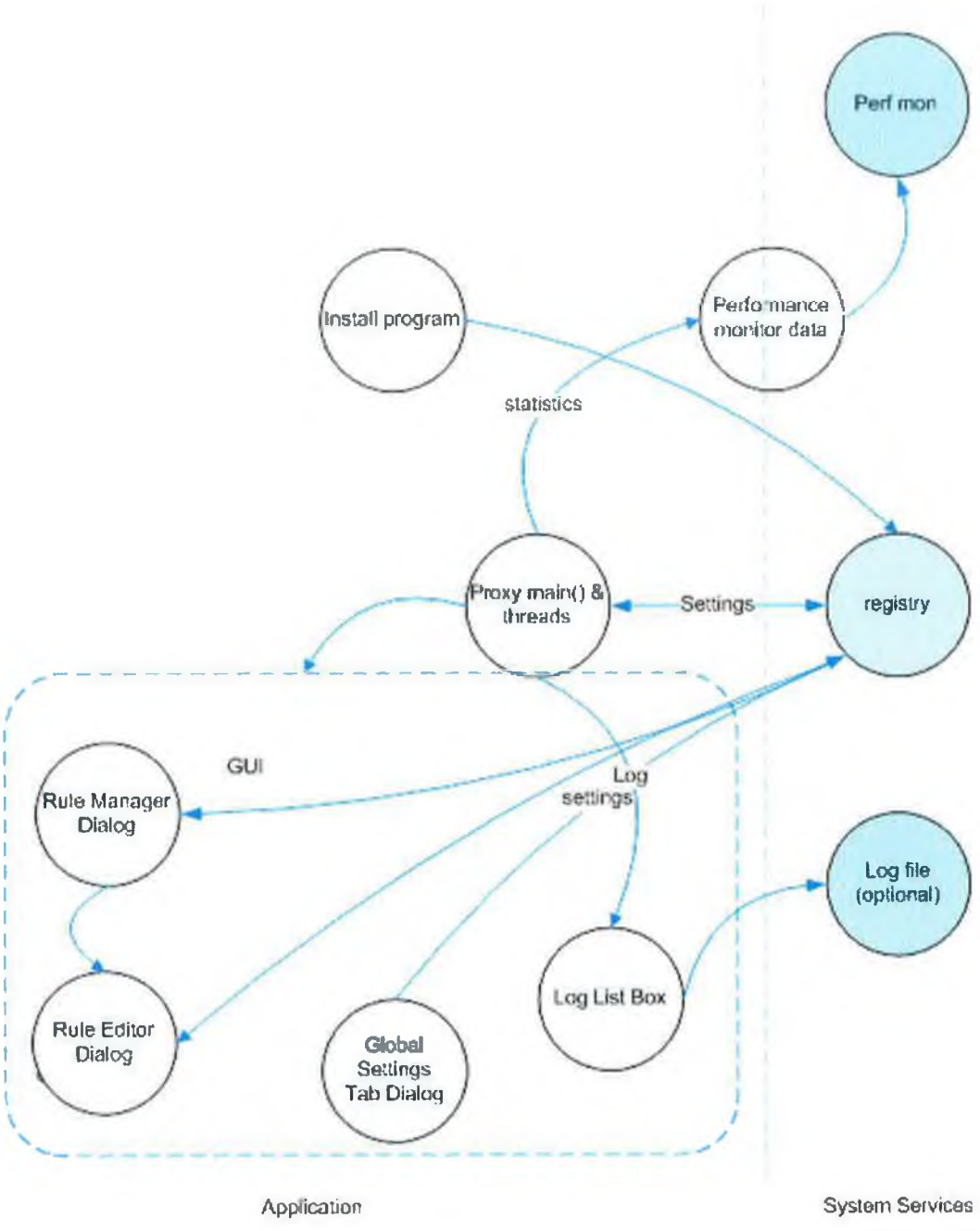

Figure 5.14: Proxy Advanced GUI component interaction.

It also fits our design goal of high performance. A service is a critical system component and puts demands on application design allowing for huge usage without over overwhelming the limited resources. An NT service is an executable program (.exe) that runs as a background task and whose lifetime is controlled by the service control management (SCM) sub-system. Services maybe run at system start-up, or may be started (via the SCM) by the interactive user, a process that a user is running, or by remote distributed system management tools such as Microsoft own, IBM's Tivoli or HP's open-view.

Figure 5.15 outlines the interactions between the components outline below.

Based on the instructions given when the service is installed the SCM decides how and when to load the service application when needed and involves no subsequent user interaction. This model is different from the way normal program executables are loaded and run. i.e. when a user double clicks on an exe via an explorer GUI or programmatically via another process. This ability to live independently of the individual user or depend on specific call requests provides us with its useful properties.

Services can, but shouldn't, have console or GUI feedback to the user. All information should be passed via the system event log (see Logging)

Types of start-up state set on installation

- Manual. Required user interaction
- Automatic (most common) When the NT/2000/XP splash screen is being started and the disk is still whizzing on system start-up.
- Disabled Start-up: Installed but required *administrator* rights to start-up

One sees and can modify the start-up options via the services panel on the administrator portion of the control panel. We also provide access via a control panel applet, command line or our own service and 'rule' management GUI.

We provide a number of command line settings to allow the user to tell the SCM how to install the service on the common line.

To be part of a really effective client/server managed architecture an NT service should be capable of remote management. We achieve this by having an 'install relative' registry entries controlling all logic configuration set as per Microsoft service deployment guidelines and then its up to the machine system administrator to provide rights to allow remote access to qualified/approved remote admin.

All required entries for the IBE proxy are below a known registry root

"<known root>\Software\DCU/IBE Secure Relay"

following the convention of

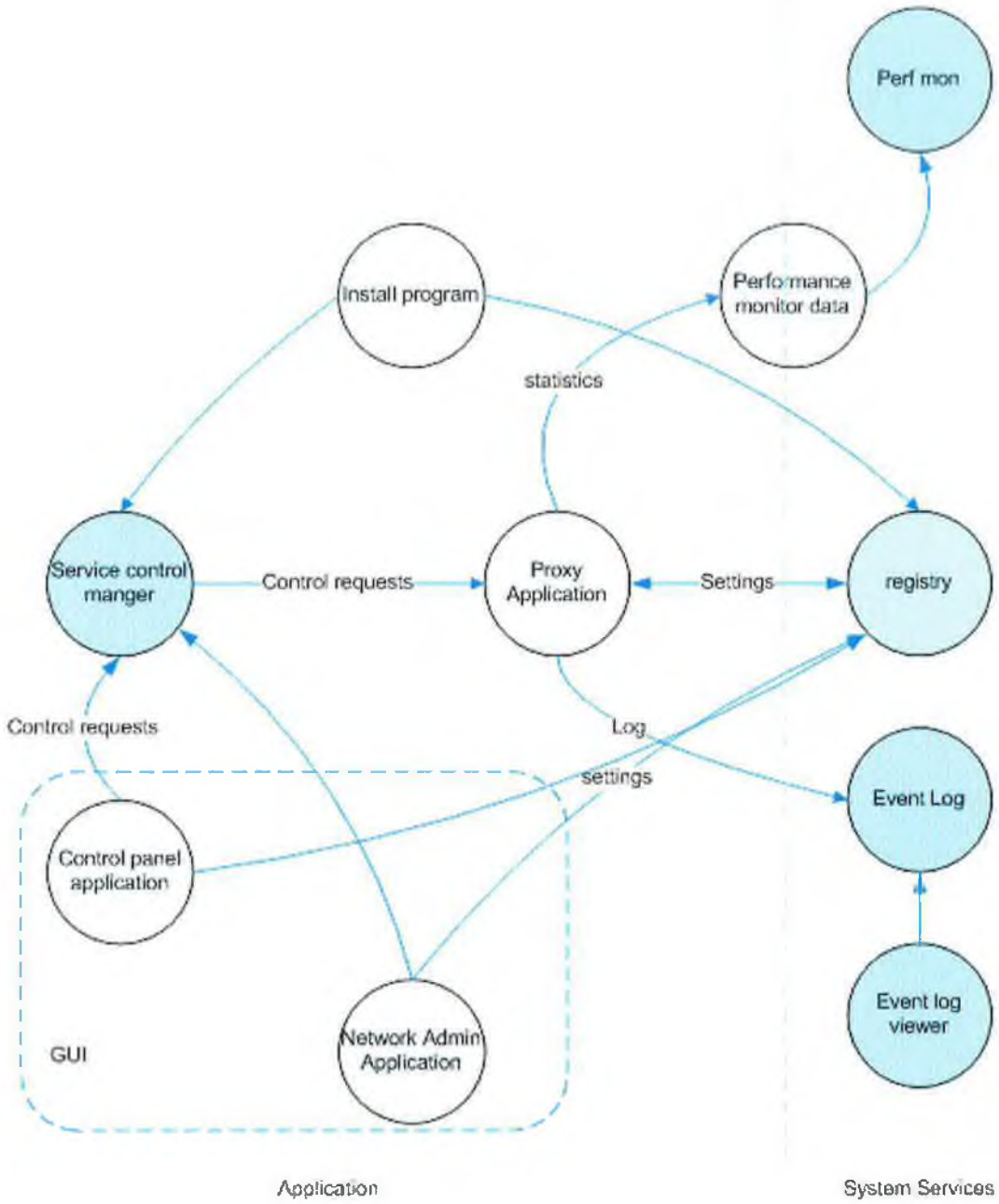

Figure 5.15: System Interactions of NT service IBE proxy

"<user dependent>\Software\company\application name" which is dependent on the context of the main (). If its compiled as an NT service then the 'user dependent' settings have to be under LOCAL_MACHINE for permissions. This is because at start-up the system may not know what user rights the application will be run as. Straight inside the code block which implements the NT service entry point we call a Configure (LOCAL_MACHINE) call to set the relative configuration access point. All other configuration is exactly the same as any GUI based proxy.

To follow Microsoft guidelines a service should provide a management GUI independent of the service management which modifies the set of registry entries. Ideally this is a control panel application but we also provide a stand alone GUI capable of remote configuration and management by utilizing the underlying SCM sub system if NT privileges allow so. Not making this application stops any system carrying the 'approved for Microsoft' logo.

A service and service management apps are slightly different from a standard C/C++ program code in that immediately after the main () process 'thread of execution' it has to make a call to the SCM to register itself for control calls via a `StartserviceCtrlDispatcher ()` call. The main() does nothing else and the `StartServiceCtrlDispatcher ()` is then considered the application entry point.

This dispatcher starts a thread and holds a pointer to the real main () called the

`ServiceMain()` then it starts a new thread which registers itself to the SCM and is called the control handler which at a minimum handles control operations stop, pause or continue. The service also has to provide updates back to the SCM about its state (running, paused etc.) via a status information. The dispatcher thread of control just sits in a while () loop passing control requests from the SCM to the appropriate handling functions which each which have the responsible of provided real time data on its state. At a minimum these are stopped, starting, running, stopping, start pending (reading Config etc.) stop pending (waiting for threads to finish I/O for example), pausing.

This allows the real main () (ServiceMain ()) to implement the logic of the service, dispatching worker threads doing I/O etc. it continues to do this until it receives a stop request from the SCM where it cleans up its resources and terminates.

All this code is in the service.cpp source code module.

5.10.4 Event Log

On NT (including 2000 and XP), rather than each application writing to a separate file with no standard format for logging errors, exceptions, etc., the creators developed a central location via an operating system 'sub-system' called the EventLog. It is intended to provide a central reporting facility for various operating system components. For applications with no GUI like drivers and service applications like our proxy in VPN tunnel mode, they can report error conditions and detailed information to administrators and

users. This event log has a standard unified programming interface. The calling application first registers itself with the subsystem as a log generator. There are three types of logs.

- Error, Information, and Debug.

and four types of severity

- Success, Information, Warning, and Error.

Typically the event log viewer can filter or trigger on these types plus the event ID. The application effectively sends the log type & severity, plus an application specific numeric identifier for the log entry, data for the entry. Its interesting to note that the event log sub system doesn't actually copy the full entry but keeps the time/date, the calling application binary locations and then uses the ID to locate a resource string back in the original .exe file. It only does this when the log is being read/viewed or backed up. This mechanisms required discipline on the application programmer to understand the type of log being generated. To tag it appropriately and to provide a unique ID for the message. It's also optimal to keep the actual data short.

The Microsoft complier (Visual C++) provides a message complier which can take a text file with a list of the log ids and strings and generates a c++ header file and a '.bin' file for inclusion into the linked binary. The header .h file then can be used by the program to refer to messages ids by a mnemonic rather than by a 'magic' number. The strings are short macros with special substitution characters (%A, %B etc) on linking the .exe these strings are complied into the application for lookup by the event log subsystem when required. The strings also include a mechanism for language independence, which can be useful to have one exe, but in the end users (viewers) native language.

To take full advantage of this system all log entries in the code are thunked into the stub WriteLog () call located in the ccWritelog.c source module.

Because the event log is tightly linked to the operating system, and had unified API. It is possible for remote monitoring of the application by an suitable privileged administrator on a remote machine, and to place data mining (or filtering) tools.

5.11 Random Number generation.

As outline in the earlier section 2.3.6 described the importance of locating good randomness in any application using cryptography, In our Microsoft Windows 32 based implementations we start our entropy accumulator by threading out a random thread to run in parallel as soon as the application starts attempting to emulate the operations of the

UNIX `/dev/random` inside the applications themselves. Inside this thread we poll a number of system resources, mixing them with MD5 (for speed), and settle into a loop which constantly gathers variable system data in an attempt to add to the non deterministic data.

```
// Random seeding modeled after Netscape SEC package
// but keeps looping, sampling the high resolution timer
// and the cursor position periodically. This should be
// random if there is any load at all on the machine.
unsigned long WINAPI RandomThread(PVOID dummy)
{
UUID uuid;
POINT pt;
LARGE_INTEGER ci;
MEMORYSTATUS mem;
DWORD dw1,dw2,dw3,dw4;
char vol[128],fs[128];
void *p;
B  dw1=GetTickCount();
A  RAND_seed((unsigned char *)&dw1,sizeof(dw1));
C  p=GetCurrentProcess();
RAND_seed((unsigned char *)&p,sizeof(p));
C  dw1=GetCurrentProcessId();
RAND_seed((unsigned char *)&dw1,sizeof(dw1));
D  p=GetCurrentThread();
RAND_seed((unsigned char *)&p,sizeof(p));
D  dw1=GetCurrentThreadId();
RAND_seed((unsigned char *)&dw1,sizeof(dw1));
E  dw1=GetLogicalDrives();
RAND_seed((unsigned char *)&dw1,sizeof(dw1));
F  GetVolumeInformation(0,vol,sizeof(vol),&dw1,&dw2,&dw3,fs,sizeof(fs));
RAND_seed(vol,strlen(vol));
RAND_seed(fs,strlen(fs));
RAND_seed((unsigned char *)&dw1,sizeof(dw1));
RAND_seed((unsigned char *)&dw2,sizeof(dw2));
RAND_seed((unsigned char *)&dw3,sizeof(dw3));
G  GetDiskFreeSpace(0,&dw1,&dw2,&dw3,&dw4);
RAND_seed((unsigned char *)&dw1,sizeof(dw1));
RAND_seed((unsigned char *)&dw2,sizeof(dw2));
RAND_seed((unsigned char *)&dw3,sizeof(dw3));
RAND_seed((unsigned char *)&dw4,sizeof(dw4)); mem.dwLength=sizeof(mem);
H  GlobalMemoryStatus(&mem);
RAND_seed((unsigned char *)&mem,sizeof(mem)); dw1=sizeof(vol);
I  GetComputerName(vol,&dw1);
RAND_seed(vol,dw1);
memset(&uuid,0,sizeof(uuid));
J  UuidCreate(&uuid);
RAND_seed((unsigned char *)&uuid,sizeof(uuid));
  for(;;)
  {
K    QueryPerformanceCounter(&ci);
     RAND_seed((unsigned char *)&ci,sizeof(ci));
L    GetCursorPos(&pt); RAND_seed((unsigned char *)&pt,sizeof(pt));
M    Sleep(1000);
B    rand_cb(gCtx);
  }
return 0; // unreachable. but the function must have a return value !
```

```
}
```

- A `RAND_seed((unsigned char *)& dw1,sizeof(dw1)) & rand_cb(gCtx);`

  Allow for all gather data gathered is constantly feed into the mixer which is based on the SSLEAY/OpenSSL [19] generator (see description in Peter Guttmann's book [25]) which hashes via SHA1 an initial 20-byte all zero value then successive 20 byte block blocks of 1 KByte randomness pool along with the randomness pool which we supply from our updating random thread.Which then become the next 20-byte hash value and XORed back into the pool. This is then feed into the PNRG generator which mixes again and generates the output. This outline comes from notes made in a conversation with Dr Stephen Henson [27] of the OpenSSL [19]team. The _seed() is the constant background update, the _cb() binds it to the PRNG.

- B `GetTickCount();`

  Retrieves the number of milliseconds that have elapsed since the system was started.

- C `GetCurrentProcess()` & `GetCurrentProcessId();`

- D `GetCurrentThread()` & `GetCurrentThreadId();`

  Retrieves a pseudo handle and ID and for the current process or thread.

- E `GetLogicalDrives();`

  Retrieves a bitmask representing the currently available disk drives. We use this for seed and as input to F and G.

- F `GetVolumeInformation(0,vol,sizeof(vol),&dw1,&dw2,&dw3,fs,sizeof(fs));`

  Retrieves information about a file system and volume from the directory specified in E.

- G `GetDiskFreeSpace(0,&dw1,&dw2,&dw3,&dw4);`

  Retrieves information about the disk from E, including the amount of free space on the disk.

- H `GlobalMemoryStatus(&mem);`

  Obtains information about the system's current usage of both physical and virtual memory.

- I `GetComputerName(vol,&dw1);`

  The windows domain name is unique to a domain.

- J `UuidCreate(&uuid);`

  Unique identifier creation. allows each application to come up with a uses include CORBA and COM library interfaces /library name mangling. doesn't actually need

to be random, but just not collide with another. in practice this takes the MAC address of the primary network interface.

- K `QueryPerformanceCounter(&ci);`

  A view into the windows performance monitor subsystem. All well written server windows applications provide data to the system about their running state and provide data on application specific data such as throughput, number of connected users, speeds etc etc. This mixed up data should be tough to determine. Our implementation of the proxy also contributes to this pool via the registration of its global statistical information.

- L `GetCursorPos(&pt);`

  Retrieves the cursor's position, in screen coordinates.

- M `Sleep(1000);`

  Put the thread in to hibernation for a second allowing for some of the counters to be updated. It essentially puts our sample gather at one per second.

Chapter 6

Conclusions

During the course of this work it has been shown that IBE based security protocols are suitable for real world use in applications.

Ongoing research and future work in this area should include

- New key retrieval methods in agents. Currently the fall back of most IBE applications, including my own, is to assume a simplistic private key distribution method, usually manual and without complex real world authentication techniques.
- Complex policy systems. Similarly the key management system usually is very simplistic just asking the user to prove their identity. I believe future systems will involved much more role based identities with complex key construction strategies.
- Systems which replace IBE Key Generation Center's (KGC) Escrow capabilities. The biggest controversy with any existing IBE system is the escrow property. While useful in certain circumstances, it can discourage deployment. Ongoing research hopes to reduce or eliminate the escrow issues with alternatives. This does not appear to be possible for pure IBE but may be achieved for a hybrid IBE/PKI scheme.
- Signatures. Often a parallel applications of security applications, Signatures requirements are weakened by the presence of escrow as signature system are usually restricted to systems where it is provable that only one party can produce a given signature. However the role based models of IBE use propose to open up this field. Future versions of the client software for email clients should integrate signature schemes. Indeed recent research in this area includes the pioneering work of Noel McCullagh [38] and Paulo Barreto in the area of pairing based signature, particularly batch signature verification. [1]. Noel's work has give us a model of verifying signatures in a manner that is much more efficient that those currently employed in

[1] Noel is also a member of the team funded under the same Enterprise Ireland [29]RIF

traditional PKI/RSA based systems where each signature must be checked individually involving relatively large computational expense on an operation that usually resolves in a positive verification. The pairing based batch signature verification can verify any amount of signatures in one operation. This model is more practical when one thinks how email is normally retrieved from a server in batches periodically. In a restricted device such as a phone, this efficient batch signature check may be make a big difference.

- Random Oracle Proof. The Random Oracle security proof has been questioned as to its relevance to IBE. Future work include new IBE methods that do not require random oracle proofs.

- Formal IETF adoption of IBE in XML signature and encryption RFCs, and in VPN protocols like SSL and IPSEC. While my work presents IBE technology in a manner that has made it pragmatic to deploy in current systems, a more idealistic approach is to have IBE embedded in the lower layers of technology and thus truly reap the benefit from its transparent capabilities.

Bibliography

[1] S. A. Vanstone A. Menezes, P. C. van Oorschot. *Handbook of Applied Cryptograpy*, volume 6 of *Discrete Mathematics and Its Applications*. CRC Press, Boca Raton, FL, USA, 2001.

[2] Jeff Alger. *C++ for Real Programmers*. A P Professional. Morgan Kaufmann, New York, 1998.

[3] Tom Austin. *PKI - A Wiley Tech Brief*. Wiley Tech Brief. Wiley Press, Hoboken, New Jersey, 2001.

[4] P.S.L.M. Barreto, H.Y. Kim, B. Lynn, and M. Scott. Efficient algorithms for pairing-based cryptosystems. In *Advances in Cryptology – Crypto'2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 377–87. Springer-Verlag, 2002.

[5] BBC. Increased hack attacks. BBC Internet News Site, 2003. `http://news.bbc.co.uk/2/hi/technology/2231205.stm`.

[6] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. extended abstract. In *Crypto '2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer-Verlag, 2001.

[7] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. *SIAM Journal of Computing*, 32(3):586–615, 2003.

[8] C. Cocks. An identity based encryption scheme based on quadratic residues. In *VIII IMA International Conference on Cryptography and Coding*, volume 2260 of *Lecture Notes in Computer Science*, pages 360–263. Springer-Verlag, 2001. `http://www.cesg.gov.uk/site/ast/idpkc/media/ciren.pdf`.

[9] CESG. Communications and Electronic Security Group. `http://www.cesg.gov.uk`.

[10] M. Crispin. Rfc 3501 - internet message access protocol. Internet Request for Comments, March 2003. RFC 3501.

[11] J. Schiller D. Eastlake, S. Crocker. *RFC 1750: Randomness Recommendations for Security*. IETF The Internet Engineering Task Force, December 1994.

[12] J. Daemen and V. Rijmen. The block cipher rijndael. *Smart Card Research and Applications, LNCS 1820*, 1820:288–296., 2000. Lecture Notes in Computer Science No. 218.

[13] J. Daemen and V. Rijmen. *The design of Rijndael: AES–The Advanced Encryption Standard.* Springer-Verlag, Berlin, 2002.

[14] J. Daemen and V. Rijmen. Rijndael, the advanced encryption standard. *Dr. Dobb's Journal*, 26(3), March 2001.

[15] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, IT-22:644–654, November 1976.

[16] H. Dobbertin. The status of MD5 after recent attacks. *RSA's CryptoBytes publication*, 2(2):1–6, 1996.

[17] Kitty Niles. Donald E. EastlaKe. Digital cryptography: A subtle art. Sample Chapter, 2002. `http://www.awprofessional.com/articles/article.asp?p=29054&seqNum=4`.

[18] J. Ellis. History of ID-PKI. Web page with links to slides, 2001. `http://www.cesg.gov.uk/site/ast/index.cfm?menuSelected=3&displayPage=31/`.

[19] Dr. S. Henson et al. openssl libary. open source libary via web download, 1988. `http://www.openssl.org`.

[20] J. Galvin et al. Rfc 1847- security multiparts for mime: Multipart/signed and multipart/encrypted. Internet Request for Comments, October 1995. RFC 1847.

[21] Y. Rekhter et al. Rfc 1918 - address allocation for private internets. Internet Request for Comments, February 1996. RFC 1918.

[22] Y. Rekhter et al. Rfc 3369 - cryptographic message syntax (cms). Internet Request for Comments, February 1996. RFC 3369.

[23] American Standard Code for Information Interchange. `:http://www.ansi.org/`.

[24] Apache Software Foundation. Apache http server. Open Source http server, 1996. `http://www.apache.org`.

[25] Peter Gutmann. *Cryptographic Security Architecture Design and Verification.* Springer-Verlag, Berlin, 2004. `http://www.cs.auckland.ac.nz/~pgut001/`.

[26] M. Haddon. *The curious incident of the dog in the night-time.* Jonathan Cape, Great Britain, 2003.

[27] Dr. S. Henson. Renowned security expert and open source developer, 2002. `http://www.drh-consultancy.demon.co.uk`.

[28] IEEE Std 1363-2000. Standard Specifications for Public-Key Cryptography. IEEE P1363 Working Group, 2000.

[29] Enterprise Ireland. `http://www.enterprise-ireland.com`.

[30] C. Neuman J. Kohl. *RFC 1510: The Kerberos Network Authentication Service (V5).* IETF The Internet Engineering Task Force, 1993.

[31] M. Rose J. Myers. Rfc 1939 - post office protocol - version 3. Internet Request for Comments, May 1996. RFC 1939.

[32] A. Joux. A one round protocol for tripartite Diffie-Hellman. In In W. Bosma Ed., editor, *Algorithm Number Theory Symposium – IVth Sympisium*, volume 1838 of *Lecture Notes in Computer Science*, pages 385–394. Springer-Verlag, 2000.

[33] N. Koblitz. Elliptic Curve Cryptosystems, 1987.

[34] Paul Kocher. President Cryptography Research. `http://www.cryptography.com/company/Paul-Kocher.html`.

[35] Steven Levy. The open secret. *Wired*, 07(04), April 1999.

[36] Malpan S. Galperin C. Adams M. Myers, R. Ankney. RFC 2560. X.509 internet public key infrastructure online certificate status protocol - OCSP. Internet Request for Comments, 1999. RFC 2560.

[37] Wenbo Mao. *Modern Cryptography: Theory and Practice.* Prentice-Hall, PTR. HP press, Upper Saddle River, New Jersey, USA, 2004.

[38] Noel McCullagh. god like genius. `http://www.computing.dcu.ie/~nmcculla`.

[39] A. Menezes. *Elliptic Curve Public Key Cryptosystems.* Kluwer Academic Publishers, 1993.

[40] Sun Microsystems. Java 1.1 SDK. Toolkit, 1998. `http://java.sun.com/products/archive/jdk/1.1/index.html`.

[41] V. Miller. Uses of Elliptic Curves in cryptography. In H.C. Williams, editor, *Advances in Crytology - CRYPTO '85*, pages 417–426.

[42] nCipher. A provider of hardware security products and developer toolkits. `http://www.ncipher.com`.

[43] Faranak Nekoogar. Digital Cryptography: Rijndael Encryption and AES Applications. `http://www.techonline.com/community/related_content/14754`.

[44] N.I.S.T. U.s. department of commerce/n.i.s.t., national technical information service. `http://www.itl.nist.gov/`.

[45] Jonathan B. Postel. Rfc . simple mail transfer protocol. Internet Request for Comments, August 1982. RFC 821.

[46] Federal Information Processing Standards Publications. U.s. department of commerce/n.i.s.t., national technical information service. `http://www.itl.nist.gov/fipspubs/`.

[47] Federal Information Processing Standards Publications. Standard specifications for secure hash algorithm -1. FIPS PUB 180-1, 1995. `http://www.itl.nist.gov/fipspubs/fip180-1.htm`.

[48] Federal Information Processing Standards Publications. Standard specifications for the advanced encryption standard. FIPS PUB 197. PDF, 2001. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[49] A. Shamir R. Rivest and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. In *Communications of the ACM*, volume 21(2) of *Communications of the ACM*, pages 120–126. ACM, 1978.

[50] Ron L. Rivest. Founder RSA security and professor electrical engineering MIT. `http://theory.lcs.mit.edu/~rivest/`.

[51] Ronald L. Rivest. Rfc 1321. the MD5 message-digest algorithm. Internet Request for Comments, April 1992. RFC 1321.

[52] M. J. B. Robshaw. MD2, MD4, MD5, SHA and other hash functions. Technical Report TR 101, RSA Laboratories, July 1994.

[53] RSA Data Security, Inc. *PKCS #1: RSA Encryption Standard*, June 1991. Version 1.4.

[54] K. Ohgishi Ryuichi SAKAI and Masao KASAHARA. Cryptosystems based on pairing. In *In Proceedings of Cryptography and Information Security*, Jan 2000.

[55] Ryuichi SAKAI and Masao KASAHARA. ID based Cryptosystems with Pairing on Elliptic Curve. Cryptology ePrint Archive, Report 2003/054, 2003. `http://eprint.iacr.org/`.

[56] Bruce Schneier. Founder and CTO Counterpane. `http://www.schneier.com`.

[57] Micheal Scott. Miracl multiprecision integer and rational arithmetic c/c++ library. Shamus Software Internet Site, 2002. `http://http://indigo.ie/~mscott/`.

[58] A. Shamir. Identity-based cryptosystems and signature schemes. In *Springer-Verlag, Lecture Notes in Computer Science*, volume 30 of *Lecture Notes in Computer Science*, pages 47–53. Springer-Verlag, 1984.

[59] N. Smart. *Cryptography, An Introduction.* McGraw-Hill, Maidenhead, England, 2002.

[60] AEP Systems. A leading provider of security and acceleration hardware products. `http://www.aep-crypto.com`.

[61] C. Allen T. Dierks. *RFC 2246: The TLS Protocol.* IETF The Internet Engineering Task Force, January 1999.

[62] W3C. Xml-signature syntax and processing. W3 Internet Site, 2002. `http://www.w3.org/TR/xmldsig-core/`.

[63] Eric W. Weisstein. Group generators. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/GroupGenerators.html`.

[64] Wikipedia. Differential cryptanalysis. Wikipedia fact index, 2003. `http://www.fact-index.com/d/di/differential_cryptanalysis.html`.

Appendix A

Patent

In September 2002 the team filed a joint patent at both the US and European patent offices with the author as co-inventor.

A.1 Patent Filing

A cryptography system and method

A.1.1 Introduction

The invention relates to Public Key Infrastructure (PKI) cryptography.

Such cryptography involves a sender encrypting a message using the public key of the receiver. The receiver decrypts the message using his/her private key. The public key is made publicly available by the receiver, but the private key is kept private and secure. Such cryptography has been effective from a security viewpoint. However, it has suffered from the problem of being difficult to use. In practice, many people do not go to the trouble of locating the public key of the person they wish to send a message to.

This has led to the "identity based encryption" ("IBE") approach, in which an arbitrary string is used as the public key. For example, if Alice wished to send Bob an encrypted email, there is no need for Alice to obtain Bob's public key certificate. Instead, Alice would be able to encrypt a message to Bob using only his email address as the public key.

While Adi Shamir proposed such a scheme in 1984, IBE remained an unsolved problem until recently. In June 2001 Cocks described a system using the quadratic residuosity problem. In 2001 Boneh et al described an IBE system using the Weil Pairing.

There are four steps in an IBE scheme:

- Setup. This function generates global system parameters and generates a master-key. This is essentially picking the parameters of the elliptic curve set out above.

- Extract. This function uses the master-key to generate a private key that corresponds to an arbitrary public key string ID.
- Encrypt. Takes a message and encrypts it using the public key ID.
- Decrypt. Decrypts a message using the corresponding private key.

To encrypt a message the encrypting party needs to have access to the public global system parameters of the Key Generator and (optionally) the policy of the Key Generator for constructing a public key string from an identity.

At present, there exist relatively few practical implementations of the IBE scheme. These schemes have yet to solve the problem of scalability of the system, focusing on the direct crypto aspects and assume that the parameters and/or policy we speak of are fixed in a global solution or distributed manually with each instance.

We believe this approach to be limiting.

An object of the.invention is to enhance the IBE scheme with methods to allow for more scalable, distributed, dynamic model. Another object is to allow for multiple closed groups with their own security parameters, to offer a model with multiple realms of security policy.

A.1.2 Statements of Invention

According to the invention, there is provided a cryptography method comprising the steps of a sender encrypting a message using a recipient's public key and the recipient decrypting the message using a private key, wherein the public key is generated using parameters and a policy retrieved from a domain.

In one embodiment, the domain is associated with the recipient.

In another embodiment, the domain provides the parameters and policy either before or after the recipient has registered with it.

In a further embodiment, the domain also provides a private key to the recipient.

In one embodiment, the domain comprises a key distributor which distributes private keys to registered users, and parameters and policies for potential senders to construct public keys.

In another embodiment, the domain address is made available via search engines in a manner whereby it is easy to locate because of the business or other link with the recipient.

In a further embodiment, the domain enables only read-only access by potential senders.

A.1.3 Detailed Description of the Invention

The invention will be more clearly understood from the following description of some embodiments thereof, given by way of example only with reference to the accompanying drawings in which:-

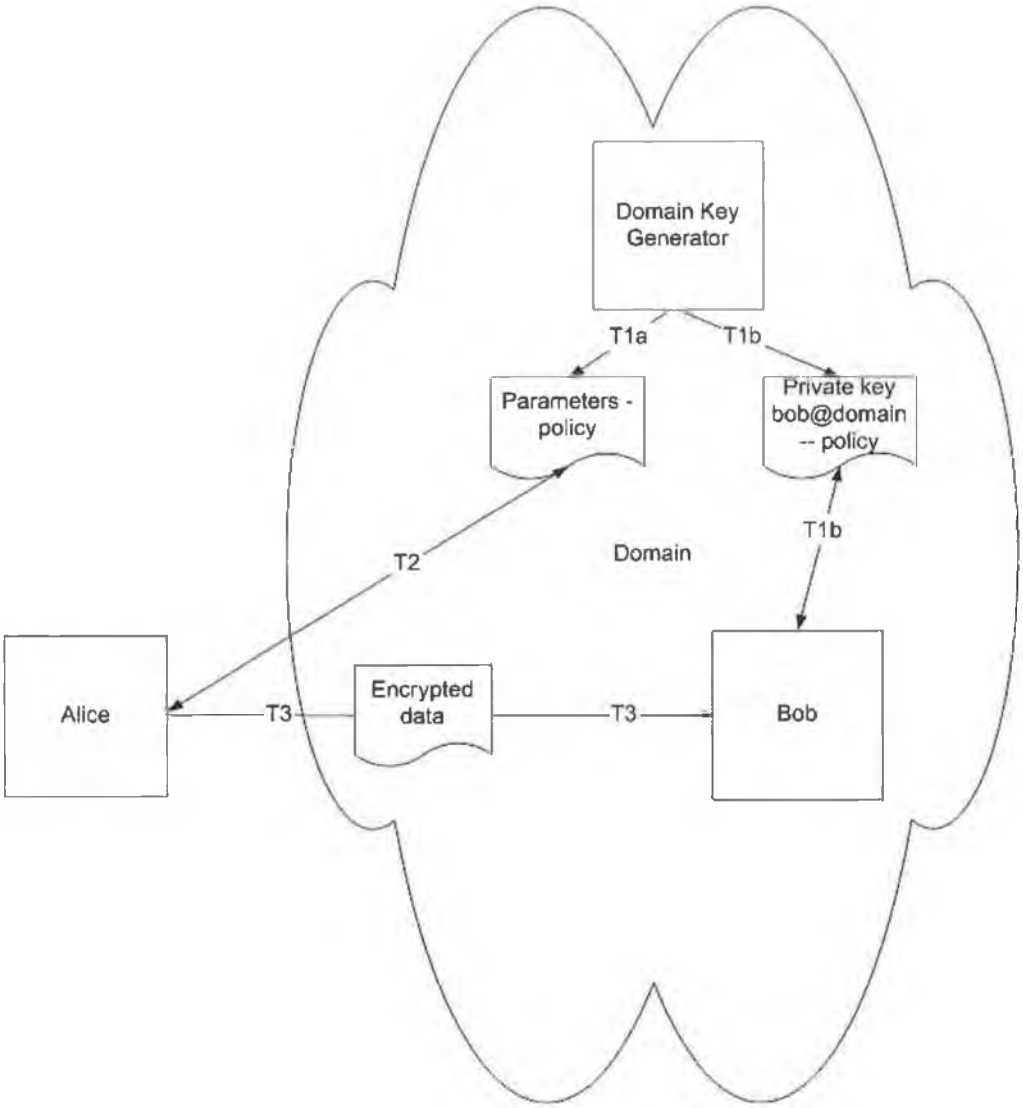

Fig. 1 is a message transfer diagram illustrating a cryptography method of the invention.

In the invention, a cryptography method includes locating the global system parameters and/or policy for an identity based on a domain that identity is a member of. This domain is possible to derive from the identity. For example, to encrypt a message to *neil@costigan.com* one does a 'network lookup' to the domain costigan.com to locate the global public parameters (and/or policy) to initialise an Encrypt operation.

Examples of methods of lookup could be:-

- a DNS lookup request to an IBE record in a DNS host table similar to how a mail

relay finds an MX record to locate a mailhost.

- a HTTP request to the URL ibe.costigan.com.

A would-be infringer would have to include a lookup function based on the concept that an identity is described by a string and that identity can be placed in a domain. This domain is used to dynamically locate the correct parameters to secure data being transmitted to the identity.

In more detail, a Key Generator (KG) publishes parameters and various policies relevant to construction of public keys for entities it will distribute private keys to. To encrypt data to the identity Bob the encrypting entity Alice needs to locate suitable parameters and policy to construct a public key for 'Bob'.

To do so they will determine a likely Key Generator for the domain or group 'Bob' is likely to be able to get a suitable private key from.

This relationship between Bob and the Key Generator does not need to be in place at the time of encrypting.

Alice derives the source of the parameters and policy for 'Bob' based on a domain 'Bob' is part of.

For example:

To encrypt and send a message to *bob@un.org*, Alice drives via a mapping function (f in Fig. 1) to the domain for Bob (un.org), and then does a lookup of the parameters from a key generating source for un.org. The parameters could be generated based on the request.

At some time T_{1a} the domain key generator DKG publishes parameters and a policy.

At some time T_{1b} the domain key generator DKG gives a private key on request to Bob who should authenticate himself to obtain this key.

At a time T_2 prior to encrypting data to *Bob@un.org*. Alice drives via a mapping function $F()$ a suitable domain for Bob based on his identity, i.e. for Bob (un.org) F(bob@un.org) =un.org. Alice either determines she already has, or needs to request public parameters and policy from this domain . If she needs to request Alice then does a lookup of the parameters and/or policy from a key generating source (DKG) for un.org. The parameters could be generated based on the request type or contents.

At some time T_3 after T_2 Alice uses these parameters to form the public-key for the identity Bob.

It will be appreciated that the invention allows for much simpler location of the set of parameters that an identity is most likely to be associated with. This is achieved by use of a domain associated with the recipient.

The invention is not limited to the embodiments described but may be varied in construction and detail.

A.1.4 Claims

1. A cryptography method comprising the steps of a sender encrypting a message using a recipient's public key and the recipient decrypting the message using a private key, wherein the public key is generated using parameters and a policy retrieved from a domain.

2. A cryptography method as claimed in claim 1, wherein the domain is associated with the recipient.

3. A cryptography method as claimed in claim 1 or 2, wherein the domain provides the parameters and policy either before or after the recipient has registered with it.

4. A cryptography method as claimed in claim 3, wherein the domain also provides a private key to the recipient.

5. A cryptography method as claimed in any preceding claim, wherein the domain comprises a key distributor which distributes private keys to registered users, and parameters and policies for potential senders to construct public keys.

6. A cryptography method as claimed in any preceding claim, wherein the domain address is made available via search engines in a manner whereby it is easy to locate because of the business or other link with the recipient.

7. A cryptography method as claimed in any preceding claim, wherein the domain enables only read-only access by potential senders.

8. A cryptography method substantially as described with reference to the drawings.

9. A server for performing the operations of a domain as claimed in any preceding claim.

10. A computer program product comprising software code for performing a method of any of claims 1 to 8 when executing on a digital computer

Appendix B

Annotated Configuration

```
Windows Registry Editor Version 5.00

[HKEY_CURRENT_USER\Software\DCU]
```

location defined by Microsoft guidelines.

```
[HKEY_CURRENT_USER\Software\DCU\Socket Relay]

"pos.x"=dword:000000b0

"pos.y"=dword:000000e8

"pos.cx"=dword:00000310

"pos.cy"=dword:000001a2
```

To save windowing position every time the application is closed we save the cordinates of the top right and bottom left corners and reopen the applicaion using these points.

```
"verbose"=dword:00000001

"debug"=dword:00000000
```

Level of detail in output log window.

```
"thread.pool.enable"=dword:00000001
```

Tells the application to build a pool of worker threads ready to take action on connections. Improves performance by requesting resources.

```
"perf.mon.enable"=dword:00000000
```

Enables forwarding of some statistical information to the NT performance monitor subsystem.

```
"tcp.linger"=dword:00000001
```

```
"tcp.nodelay"=dword:00000001
```

Enables small TCP optimisations.

```
"log.nolog"=dword:00000000
```

```
"log.err"=dword:00000001
```

```
"log.warn"=dword:00000001
```

```
"log.info"=dword:00000001
```

```
"log.debug"=dword:00000000
```

```
"log.detailed"=dword:00000001
```

```
"log.max"=dword:00000000
```

Allows the application user to filer the GUI display to relevant logging detail.

```
"toolbar.size"=dword:00000000
```

Enables a GUI option to have small or large tool-bar icons.

```
[HKEY_CURRENT_USER\Software\DCU\Socket Relay\Rule1]
```

Each rule is a TCP port to listen on and an action to take on connections to the port and incoming, connecting, IP address pair.

```
"disable"=dword:00000000
```

Enable the rule or not.

```
"peer.addr.str"="localhost"
```

```
"peer.port.str"="25"
```

```
"peer.mask.str"="255.255.255.255"
```

The listening IP Port in number or name. A connecting IP address, fully resolved or by hostname. Mask applied to this address.

```
"dest.addr.str"="mail.ibe.dcu.ie"
```

```
"dest.port.str"="25"
```

Port and server address to connect to.

```
"secure.in"=dword:00000000
```

```
"secure.out"=dword:00000001
```

Determines if there is encryption on incoming or outgoing connections. Ignored if SMTP or POP3 settings as enabled.

```
"smtp"=dword:00000001
```

SMTP setting.

```
"ibe.private.parameters"="c:\\ibe\\neil_ibe.dcu.ie.ibe"
```

Location of ID key if using direct VPN mode.

```
"ibe.public.parameters"="c:\\ibe\\server_curve0.ibe"
```

Location of Server domain parameters if explicitly set.

```
[HKEY_CURRENT_USER\Software\DCU\Socket Relay\Rule2]
```

```
"disable"=dword:00000000
```

```
"peer.addr.str"="localhost"

"peer.port.str"="110"

"peer.mask.str"="255.255.255.255"

"dest.addr.str"="mail.ibe.dcu.ie"

"dest.port.str"="110"

"secure.in"=dword:00000001

"secure.out"=dword:00000000

"pop3"=dword:00000001
```

Set for POP3

```
"ibe.private.parameters"="c:\\ibe\\neil_ibe.dcu.ie.ibe"

"ibe.public.parameters"="c:\\ibe\\server_curve0.ibe"
```