Applications of Cryptographically Enforced Access Control
Interim Report

Daniel Randall
Imperial College London

Abstract

Abstract

Acknowledgements

Acknowledgements

Contents

1 Introduction

The problem taken on by this product is that of implementing a concrete application of modern cryptographically enforced access control techniques. This problems stems from changes in the last decade, mainly due to the huge popularity of the internet, which has seen a big increase in the amount of open information available and methods those with malicious intent can use to gain access to information which is protected. This has brought with it a focus from academics and corporations on controlling who has access to what information and how efficient and secure these techniques are. While a lot of recent research has been done on techniques to achieve this there are little freely available implementations of these techniques in a practical environment.

By building a Dropbox app this project aims to implement and analyse a selection of techniques for cryptographically enforced access control. To achieve this a structure will be developed, with care taken not to rely on any features to be tested and instead allow for easy plug-and-play for each feature such that all combinations can be easily analysed. The objectives are to find the extent of the strengths and weaknesses of the chosen proposals and those which work well together and those which are not able to work so well together. With these results the hope is that further insights will be generated as to where future work is needed. Another interesting result will be how well such cryptographically enforced access control measures are able to be used in a 'everyday' environment such as the type which Dropbox offers.

2 Background Research

2.1 Overview

2.2 Technical Research

Cryptography

While protecting sensitive data from adversaries has been a concern for hundreds of years it was not until the First World War that the field of cryptography received serious academic attention however from then on the majority of the work was carried out secretively by states.[1] Cryptography was only employed by the military until the 1970s where cheaper hardware reduced "the design limitations of mechanical computing and brought the cost of high grade cryptographic devices down to where they can be used in such commercial applications as remote cash dispensers and computer terminals."[2] And new methods of sharing keys such that they could be transferred over insecure channels and thus eliminate the need for physical contact (ie. couriers) before secure communication became possible. One such technique was a *public key cryptosystem*, in which two distinct asymmetric keys are used. One for encrypting, which is made publically available, and one for decrypting, which is kept private, thus anyone can encrypt messages with the public key but only the owner of the private key may decrypt the messages. Also, more importantly to this project, the 1970s also saw the development of major publically known symmetric key ciphers such as Data Encryption Standard (DES). Since then improvements on such algorithms have been designed and cryptography has become an integral part of many businesses who use it, not only for privacy but, for many different reasons, such as: “authentication (bank cards, wireless telephone, e-commerce, pay-TV), access control (car lock systems, ski lifts), payment (prepaid telephone, e-cash).”[3]

Block ciphers (such as DES) are a type of symmetric key algorithms used to encrypt data. They map n-bit plain-text blocks to n-bit cipher-text blocks where n is the length of the block. This mapping is one-to-one and thus invertible. The encryption function takes “a k-bit key K, taking values from a subset K (the key space) of the set of all k-bit vectors V_k.” [4] Block ciphers are deterministic and so with the same key and the same plain-text, one should receive the same cipher-text.

This project focuses on access control as the application of cryptography. The purpose of access control is to “limit the actions or operations that a legitimate user of a computer system can perform. Access control constrains what a user can do directly, as well as what programs executing on behalf of the users are allowed to do.”[5] For instance, many operating systems control access to files by assigning permissions to users (eg. read, write, execute.) Typically vanilla access control is based on capabilities or access control lists (ACLs). ACLs works by assigning each object its own ACL detailing for each user which permissions she is authorised to perform on the object. With this setup it is easy to add, edit or remove user permissions and it is easy to see what users have permissions for each object, however all ACLs need to be accessed to determine what objects a user currently has access to. Capabilities are similar to ACLs however each user is associated with a list instead of an object and all capability lists need to examined to determine which users can access a particular object. An example of this, similar to what would be deployed in an OS, can be viewed in Figure 1 where ’R’ refers to read, ’W’ to write and ’Own’ signifies that that user is the owner of the object.

This project is an implementation of the latest theories and ideas on how to securely and efficiently create a solution for hierarchical a for access control. The concrete application domain chosen has been chosen as a public Dropbox. The reasons for this are that the folder structure implemented by filesystems, and incorporated by Dropbox, naturally conform to the strict hierarchy requirements by the chosen encryption method. Dropbox is incredibly popular, with over 100 million users collectively saving 1 billions files per day.[6]

An appropriate alternative application could have been a content distribution network such as BitTorrent[7], however due to a number of different clients being used (Utorrent, Deluge...), many different ways to connect to the tracker (.torrent files, magnetic links...) as well as inconsistent performance when using BitTorrent meant that testing the implementation would have posed a much more difficult task.

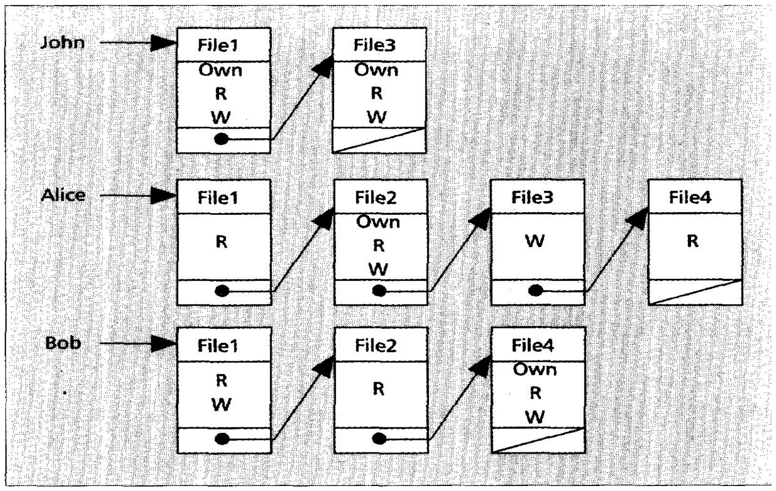


Figure 1: Capabiliy list for four files and three users.[5]

2.2.1 Hierarchical Cryptography for Access Control

Hierarchical access control relies on different levels (or classes) of security which can be represented as labels in a partially ordered set (L, $\leq$). These labels can be applied to users, u, and objects, o, using a many-to-many labelling function $lambda : U \cup O \rightarrow L$ where U is a set of all users u and O is a set of all objects o. "u is authorized to access o if and only if $\lambda(u) \leq \lambda(o)$[8]."

2.2.2 Interval-Based Access Control

Interval-based access control works in a similar way to standard hierarchical access control however, unlike standard hierarchical access control, it is necessary to have a lower bound of access as well an upper bound. If $1, \ldots, n$ is a totally ordered set of all possible security levels and $[x, y]$, where $1 \leq x \leq y \leq n$, is an interval associated with some user then that user can access an object o associated with a security level l where $1 \leq l \leq n$ if and only if $l \in [x, y]$.

This idea can be extended to cover k-dimensional cardinalities:

"Let O be a set of protected objects, let U be a set of users, and let $A_1, \ldots, A_k$ be finite, totally ordered sets of cardinality $n_1, \ldots, n_k$, respectively. We write $\mathcal{A}$ to denote $\prod_{i=1}^{k} A_i = A_1 \times \cdots \times A_k$.

We say $[x_i, y_i] \subseteq A_i$, where $1 \leq x_i \leq y_i \leq n_i$, is an *interval* in A_i. We say that $\prod_{i=1}^{k}[x_i, y_i] = [x_1, y_1] \times \cdots \times [x_k, y_k] \subseteq \mathcal{A}$ is a *hyperrectangle*. We write HRec($\mathcal{A}$) to denote the set of hyperrectangles in $\mathcal{A}$.

We assume that each object $o \in O$ is associated with a unique attribute tuple $(a_1, \ldots, a_k) \in A$, and each user $u \in U$ is authorized for some hyperrectangle $\prod_{i=1}^{k}[x_i, y_i] \in$ HRec($\mathcal{A}$). Then we say that a user u associated with $\prod_{i=1}^{k}[x_i, y_i]$ is *authorized* to read an object o associated with tuple $(a1, ..., ak) \in \mathcal{A}$ if and only if $a_i \in [x_i, y_i]$ for all i[8]."

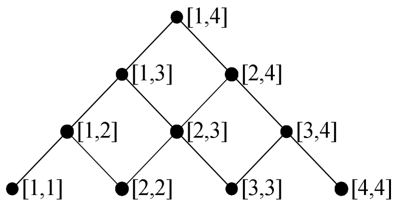


Figure 2: The Hasse diagram of (T4 , $\subseteq$).[8]

Some common implementations of this scheme are[8]:

- *Temporal* ($k = 1$) where $\mathcal{A} = A_1$ and each integer in $1, \ldots, n \in \mathcal{A}$ are in one-to-one correspondence with the time points. A 1-dimensional scheme follows the regular interval-based access control scheme as described above in which a user is associated with an interval $[x, y]$ and each object is associated with an integer. If the integer exists in the interval then the user should possess, or have the means to possess, the key to access the object.
  An example of this can be seen in Figure 2. For instance if a user was associated with the interval [2,4] then she should be be able to derive the keys for the leaf nodes [2,2], [3,3] and [4,4] but should not be able to derive the key to access the leaf node [1,1].

- *Geo-spatial* ($k = 2$) where $\mathcal{A} = A_1 \times A_2$. $\mathcal{A}$ represents a finite rectangular $m \times n$ grid of cells for which objects and keys are associated with a unique cell. Users are associated with an interval which correspond to a subrectangle of $\mathcal{A}$ where the user is able to derive keys for all cells in the area[9]. More formally, "each object is associated with some point (x, y) and each user is associated with some rectangle $[x_1, y_1] \times [x_2, y_2] = (t_1, t_2) : t_1 \in [x_1, y_1], t_2 \in [x_2, y_2]$. We write $T_{m,n}$ (as an abbreviation of the more accurate $T_m \times T_n$) to denote the set of rectangles defined by a rectangular $m \times n$ grid of points: that is, $T_{m,n} \overset{\text{def}}{=} [x_1, y_1] \times [x_2, y_2] : 1 \leq x_1 \leq y_1 \leq m, 1 \leq x_2 \leq y_2 \leq n$"[8]. Leaf nodes make up the $m \times n$ grid and are of the form $[x, x] \times [y, y]$ or $[x, y]$.
  Two visualisations of this can be seen in Figure 3. Figure 3a displays $T_{2,2}$ as a "partially ordered set of subsets ordered by subset inclusion in which rectangles are represented by filled circles," while the second depicts $T_{2,2}$ where "nodes of the same color have the same area (as rectangles): all rectangles of area 2 are filled in gray, whereas all rectangles of area 1 are filled white."[8]

2.2.3 Key Assignment Schemes (KAS)

A key assignment scheme (KAS) controls the information flow through the tree. The scheme dictates how a user is to derive access to objects she is permitted access to as well as preventing her access to objects she is forbidden from viewing. KAS are "usually evaluated by the number of total keys the system must maintain, the number of keys each user receives, the size of public information, the time required to derive keys for access classes, and the work needed when the hierarchy or the set of users change.[10]" As an example, the simplest possible KAS would be to assign every single key for which $k(y) : y <\leq (u)$ where u is a user and le is a labelling function. This scheme, however, is not ideal and efforts are generally make to reduce the number of keys held by the user. To do this most schemes look to either provide additional public or private information.

Recently a lot of work has been done in key assignment schemes, however not all of the proposed solutions are secure or efficient. As observed by Blanton[2007], the most efficient of the solutions achieve the following properties[11]:

- Each node in the access graph has a single secret key associated with it.
- The amount of public information for the key assignment scheme is asymptotically the same as that needed to represent the graph itself.
- Key derivation involves only the usage of efficient cryptographic primitives such as one-way hash functions.
- Given a key for node v, the key derivation for its descendant node w takes l steps, where l is the length of the path between v and w.

A few potential algorithms to test will be outlined here:

- **Iterative key encrypting (IKE)** - For each edge in the graph the child key is encrypted with the parent key and published as public information. The user is then, using a single private key, able to iteratively derive any child key using that information[12]. The AFB scheme by Atallah et al. is an example of this KAS, offering[10]:
  - A single private key held by the user
  - Permits only a hash functions to derive keys
  - Derivation of a descendant node's key requires $\mathcal{O}(l)$ operations where l is the length of the path between the nodes
  - "Updates [i.e., revocations and additions] are handled locally and do not propagate to descendants or ancestors of the affected part of G"
  - "The space complexity of the public information is the same as that of storing [the] graph."
  - Provably secure against collusion
- **The Akl-Taylor scheme** - Created by Akl and Taylor, this node-based scheme is the first KAS created. Key derivation in a node-based scheme eliminates the need for recursive calculations, instead the algorithm works as follows[12]:
  - The RSA key generator is called to obtain (n, e, d), of which only n is used
  - s is chosen at random from $\mathbb{Z}_n^*$
  - A mapping is chosen $\phi \rightarrow \mathbb{N}^*$ such that $\phi(x)|\phi(y)$ if and only if $y \leq x$.

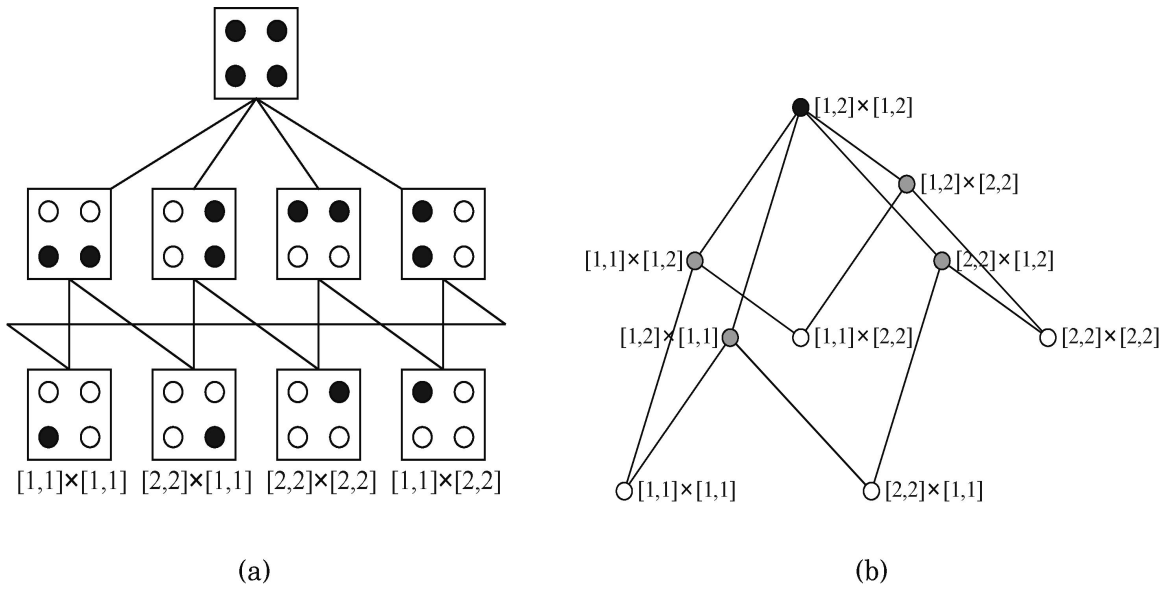


Figure 3: Two representations of $T_2 \times T_2$.[8]

- The key for x, $k(x)$, is defined to be $k(x) = s^{\phi(x)} \bmod n$.
- $n \cup (\phi(x) : x \in L)$ is published publically.

where "(n, e, d) such that: $n = pq$, where p and q are distinct odd primes; $e \in \mathbb{Z}^*_{\phi(n)}$, where $\phi(n) = (p-1)(q-1)$, $e > 1$, and $(e, \phi(n)) = 1$; $d \in \mathbb{Z}^*_{\phi(n)}$, where $ed \equiv 1 \bmod \phi(n)$."
Using private key $k(x)$ to derive $k(y)$ where $y \leq x$ the following formula is used:

$$(k(x))^{\frac{\phi(y)}{\phi(x)}} = (s^{\phi(x)})^{\frac{\phi(y)}{\phi(x)}} = s^{\phi(y)} = k(y).$$

where $\phi(x)$ and $\phi(y)$ is publically available information.
This scheme while secure (when ϕ is chosen appropriately) and fast, requires a lot of public information.

Some KASs for special case scenarios exist, such as for when the number of leaf classes is substantially larger than the number of non-leaf classes.[13] There has also recently been a number of works on elliptic curve cryptography (ECC)[14][15], however this method relies on public-key cryptography and this project will initially focus on symmetric key techniques.

2.2.4 Revokation of Permissions and Re-encryption

There are occasions where a user previously granted access at some security interval is later revoked of those permissions. If the user was associated with a security label l then the revoked user has held a key which grants her access to objects at every node within her security interval, $l' \leq l$. Due to this it is no longer acceptable to continue using the same key for any l' as the revoked user would be able to view or edit future and existing objects that she should no longer have access to. Thus for every l' it is necessary to replace the key associated with the label with a new key with which every object associated with l' is re-encrypted with.

The most obvious way to achieve re-encryption is to immediately re-encrypt every object associated with every l' the moment the user has been revoked and distribute the new keys to the appropriate users to replace the old key. This method ensures the revoked user has no access the objects she previously had access to, therefore providing robust security, and the users who have not had their permissions revoked continue as normal with the new key. However the number of objects associated with l' may well be extremely large meaning that re-encrypting all of them instantly could take a long time, possibly disrupting the service. It also may well be the case that a significant number of the objects may not be viewed or edited for some time, if ever. An alternative to this method is *lazy re-encryption.*

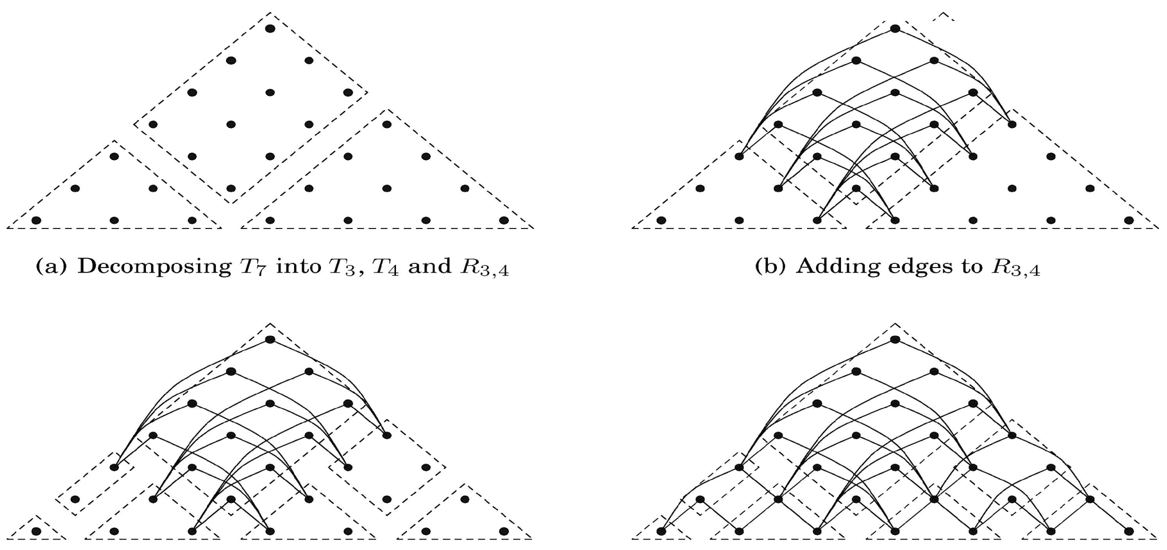
Lazy re-encryption does not instantly re-encrypt all objects, instead an object is re-encrypted with the newly assigned key for the security label l' only when it is edited for the first time after the revokation by any user. The result is that the workload of re-encryption is spread out over time and is only performed when absolutely necessary. Employing lazy re-encryption requires the user to possess more than one key - one key for objects yet to be encrypted and one for objects encrypted with the new key.[12] This means that the revoked user has the capacity to view objects while they remain unchanged and not yet re-encrypted. While logic suggests that the user who has been revoked has already been able to see the object during the time their permissions were valid meaning that it is unlikely to pose a real threat, this may not be acceptable in every scenario. To avoid this problem another solution may be to, instead of waiting for the object to be edited, wait for the object to be requested for a read. This way the user will need to possess the newly assigned key to read the object however the burden of re-encryption is still spread out over time. It can also be noted that if there are a number of users revoked over time and there are sporadic reads/writes of different files there will be a large number of keys being used for files associated with the same security level that a single user is required to hold. To keep the number of keys down it may be desirable at times to re-encrypt objects even when they are yet to be accessed.

2.2.5 Building the Tree

Crampton[2011][8] describes a process of binary decomposition which uses recursive labelling methods to separate the whole tree into nodes. In general the result is a smaller tree with some unnecessary edges being removed meaning that key derivation takes less time and the required storage space for the tree is smaller. The following describes how the process works:

If T_m represents the set of intervals where m is the cardinality then "let $l = \lfloor m/2 \rfloor$ and $r = \lceil m/2 \rceil$. Now T_m comprises:

- A copy of T_l, containing the minimal elements $[1, 1], \ldots, [l, l]$.
- A copy of T_r, containing the minimal elements $[l + 1, l + 1], \ldots, [m, m]$.



(a) Decomposing T_7 into T_3, T_4 and $R_{3,4}$

(b) Adding edges to $R_{3,4}$

(c) Decomposing T_3 and T_4

(d) Adding the remaining edges

Figure 4: The binary decomposition of T_7.[8]

- A copy of rectangle $R_{l,r}$, containing the remaining nodes in T_m."

The result of this procedure can be seen in Figure 4a for T_7.
The next step is to begin adding edges. An edge is added from every node in $R_{l,r}$ to two other nodes, a single node in T_l and a single node in T_r. "In particular, for node $[x, y]$ such that $x < l \leq y$, we add edges from $[x, y]$ to $[x, l]$ and from $[x, y]$ to $[l + 1, y]$."
This algorithm is then recursively applied to T_l and T_r until $l, r \leq 1$. The final result of this process can be seen in Figure 4d.

It is clear to see that the information flow policy is still upheld, ie. each node still has access to all other nodes it previously had access to before the binary decomposition however it has not gained access to any new nodes.

Crampton details how this algorithm can be extended to temporal, geo-spatial and general interval access control.

2.2.6 Programming Languages and Usable Software

Dropbox has a choice of SDKs which can be used for this project: the options for desktop development are written in Python, Java and Ruby, while there are options for Android and iOS development.
Following is a run down of the different languages and what they offer:

- Python has a few cryptographic tools available, such as PyCrypto which has a number of hash functions and encryption algorithms ready to use. The low level functions in PyCrypto are written in C for speed.[17] Programming in Python is quick and flexible.

- The main cryptography API available for Java is the Java Cryptography Architecture (JCA) which incorporates two packages, *javax.crypto*, which contains an interface for low level cryptography such as encryption, decryption, and hashing, and *java.security* which contains an interface for key management, certificate management, and signatures. The algorithms to implement the cryptography are included with a Service Provider Interface (SPI), with these there are many symmetric algorithms available such as AES, DES, DESede, Blowfish and IDEA[18]. An alternative is The Legion of the Bouncy Castle API which incorporates JCA/JCE[19]. Another alternative is The GNU Crypto project which is native on Linux machines and is coded in Java and offers many algorithms including AES, DES, Blowfish. An important point is that "GNU Crypto does not implement any algorithms that are encumbered by patents, and does not rely on any non-free code or documentation."[20].

- Ruby offers a module to interface with the OpenSSL cryptography library written in C. The module offers many modern ciphers such as AES.[21]
- It is also possible to create this project in C/C++ using OpenSSL. This can be done by linking the C code to the Dropbox API using Java JNI or creating built-in modules for Python.

Keyczar is an API that exists for Java, Python and C++.[22] However it does not appear flexible enough to be used in this case. It is intended for those who have little understanding of cryptography and so it hides too much information to be useful in our case.
As a large part of the evaluation for this project is based around performance I have ignored the possibility of creating the system on a mobile platform where it may not perform to its full potential. As for desktop languages, due to my inexperience with Ruby and wider cryptography support for the other languages available I decided it was not the best language for this project. It also seems unnecessary to take the extra time needed to code the project in C/C++ to risk compatability issues and possible performance hits on linking the languages. While the Dropbox source code is written Python and so may well integrate better with the Python Dropbox API, Java offers a greater range of cryptographic support. The most appealing of the APIs is GNU Crypto as the system will always run on a Linux machine and the lack of unpatented code will prevent the system from becoming tied down if there is a desire to freely distribute it after production.

2.3 Related work

Dropbox Bitbox? Mega

3 Initial Assessments

3.1 Algorithm choices

There are 4 key decisions to be made when choosing to employ a Crytographic algorithm. These are:

- **Algorithm** - The algorithm itself
- **Mode** - The mode of operation used
- **Key size** - The size of the key used for encryption and decryption
- **Padding scheme** - The method to increase the size of the file to be encrypted in order to force into a multiple of the block size of that algorithm.

Here we will discuss which algorithms to use for this project and why.

Symmetric Algorithm

- **DES** - A block cipher which used to be government standard. DES uses 56bit keys, however such small key lengths have proven to be insecure (eg. by the 'EFF DES cracker') and can no longer be used safely.
- **3DES** - A variant on DES in which plaintext is fed through 3 seperate instances of DES using up to 3 different keys. 3DES, with 3 unique keys, is secure up to $2^1$12bits of security, which is very secure. However DES is designed for hardware implementations and so it is slow when used in software which, unfortunately, is what we wish to use it for.
- **Blowfish** - 64 bit block size with up to 448 bit keys.
- **Twofish** - Successor to Blowfish. 128 bit block size with up to 256 bit keys
- **AES** - Uses 128 bit blocks with up to 256 bit keys. The current government standard.

There is no reason not to use AES and so that is what we shall use.

AES supports the following modes of operation:[23]

- **ECB** - Has been shown to be vulnerable if it used to encrypt more than one block of data with the same key. In our case, it can not be used to encrypt a file larger than 128 bits. ECB also has the undesirable trait where each plaintext block encrypted with the same key results in the same ciphertext block. This is undesirable as attackers are able to detect trends in an individuals encryptions. Requires padding.
- **CBC** - Each plaintext block is XORed with the previous block before it is encrypted by block cipher. An initialisation vector (IV) is used to XOR the first block. The ciphertext generated from the plaintext will different each time iff the IV is different. Requires padding.
- **CFB** - Requires padding
- **OFB** - Turns the block cipher into a stream. Does not require padding.

- **CTR** - Similarly to OFB, CTR turns the block into a stream cipher. Does not require padding.
- **EAX** - A variation on CTR which includes integrity checks
- **GCM** - A variation on CTR which includes integrity checks

While it is true that the modes that require padding, such as ECB, CBC and CFB can be subject to padding oracle attacks, in our implementation we will have no such oracle and so we need not worry about such attacks.[24]
We will use CBC for no reason other than it is widely supported and relatively simple to implement while not sacrificing any security if implemented correctly.

AES supports the following key sizes:

- **128 bit**
- **192 bit**
- **256 bit**

The only weakness possible resulting from the size of the key chosen is that of brute force attacks. A 128 bit AES key provides 128 bits of security (pg. 64, Table 2) and is speculated to be safe beyond the year 2031 (pg. 67, Table 4).[25] While we may choose to use a 256 bit key 'just to be safe' it is unlikely that this system will be around in 30 or 40 years and so this sort of precaution is unnecessary. Using a 192 bit key instead would cause us to take a performance hit of around 20% while a 256 bit key would cause a 40% hit, due to the fact that 128 bit, 192 bit and 256 bit keys require 10, 12, 14 rounds respectively (pg. 14, Figure 4)[26]. Due to this, using a smaller key is desirable for performance.
For these reasons we shall use a 128 bit key for AES in our system.

Asymmetric algorithm

Some asymmetric ciphers that exist today are:

- **RSA**
- **DSA**
- **ElGamal**

By far the most popular asymmetric algorithm used today is RSA. That is what we shall use.

The key sizes supported by RSA are:

- **1024**
- **2048**

- **3072**
- **4028**

Similarily to the choice of the symmetric key size we could go for an exceptionally large key 'just in case'. However each time we double the size of an RSA key decryption is 6-7 times slower.[1] Therefore we wish to use the smallest reasonable key size. A 1024 bit RSA key provides 80 bits of security (pg. 64, Table 2) which is currently deprecated today (pg. 67, Table 4), meaning that this key size is currently unnaceptable for use with RSA. A 2048 bit key provides 112 bits of security (pg. 64, Table 2) which is acceptable today through 2030 which is plenty good enough for our system (pg. 67, Table 4).[25]
We shall use a 2048 bit RSA key.

The padding recommended by FIPS is PKCS#1 v1.5 and, as we have no reason to not to, we shall use that.

Password hashing algorithm

Password hashing algorithms are odd in the sense that the trait with which to seperate them by is which is the slowest. It is desirable for a password hashing algorithm to be slow to prevent an attacker bruteforcing a password dictionary in the hope of entering the correct one into a log in form. Making log in slower makes this task much more time consuming.
There are two password hashing algorithms commonly in use. They are:

- **PBKDF2** - Based on SHA-256(VERIFY)
- **bcrypt** - Based on Blowfish.

NIST recommends the use of PBKDF2 in the use of the password storage.[27] However bcrypt has a desirable trait which PBKDF2 lacks - the computational cost of the password scheme increases as the hardware it runs on improves. It does this by requiring fast RAM which GPUs do not possess but CPUs do.[28] What this means in a practical sense is that the algorithm is more efficient on a CPU rather than a GPU - quick for servers and slow for the attackers.
There currently exists GPUs which contain modern field-programmable gate arrays with RAM which can be used to optimise attacks on bcrypt. A new password scheme has been designed to combat this problem called scrypt[2], however it has only existed for a small amount of time (4 years at the time of writing) and so has not yet 'stood the test of time' like that of bcrypt (nearly 14 years at the time of writing.)
For these reasons we shall use bcrypt to store passwords in our system.

[1]`http://www.javamex.com/tutorials/cryptography/rsa_key_length.shtml`
[2]http://www.tarsnap.com/scrypt.html

Algorithm providers

There are many different providers of AES algorithms for Java, however there is little literature on the differences between them. This is problematic due to the fact our system is very likely to execute these algorithms very frequently and if the implementation is inefficient we may end up with unecessary delays. The possible ways we can differentiate between the different implementations are as follows:

- **Correctness** - The algorithm is implemented correctly.
- **Speed** - The time the implementation takes to execute.
- **Size** - The size of the .jar import needed to use the implementation.

Determining the correctness of the implementations is outside of the scope of this report and so it will be assumed all implementations covered here are implemented correctly. The speed and size of the import needed are easy enough to measure and so they are the features we shall be looking at.

AES algorithm provider

The findings for each implementation of AES can be found in Table 1. As can be seen GNUCrypto is tested with a slightly different algorithm, specifically a different spadding scheme. This is because it does not implement PKCS5 padding (or any in common with the other implementations being tested), only PKCS7. Because of this a fairer comparison would have been to compare each using no padding, which all of them implement, however we have no control over the size of the files the user may encrypt and so the results would have irrelevent to the situation we face. For this reason each implementation used the padding which we would use should it be included in our system. However, it is true that PCKS5 and PCKS7 are practically interchangeable and so no difference should observed as a result.
Looking at the table the most striking result is the length of time the GNUCrypto implementation takes. Incorporating the GNUCrpyto implementation into the project is far more complicated than the interface provided by the JCE framework (requiring 15x as much code!) and so that may be an explanation but, in any case, henceforth we shall disregard the GNUCrypto implementation. As for the fastest implementation, there is little between the SunJCE and Bouncy Castle implementations. The real difference is not in speed but in the much more extensive library of ciphers BC provides against not requiring an external jar in the case of SunJCE.

RSA algorithm provider

GNUCrypto does not provide an implementation of RSA and so it is omitted from the tests.

Implementation	Algorithm	Import Size	Time taken
SunJCE	AES/CFB/PKCS5 padding/256bit key	0	14.67506586
GNUCrypto	AES/CFB/PKCS7 padding/256bit key	598.0kB	37.15129076
Bouncy Castle (via JCA provider)	AES/CFB/PKCS5 padding/256bit key	2.3MB	14.70026627

Table 1: Comparisons between different implementations of AES when encrypting a 1.3MB pdf file 1000 times

Implementation	Algorithm	Import Size	Time taker
SunJCE	RSA/ECB/PKCS1 Padding/2048bit key	0	2.18826906
Bouncy Castle (via JCA provider)	RSA/ECB/PKCS1 Padding/2048bit key	2.3MB	2.35624939

Table 2: Comparisons between different implementations of RSA when encrypting a 256bit AES key 10000 times

File sharing platform

Database

- **Hibernate**
- **H2Database**

Programming language

4 Implementation

4.1 Design

- Class diagram - Purpose of each class

5 Testing

6 Evalutation

6.1 What I would have done differently

Replacing keys after a set period of time. Implement lower bounds as per Jason Crampton's algorithm. Employ a more lazy key revocation scheme.

7 Conclusion

7.1 Future work

References

[1] Bruce Schneier, *"Applied Cryptography: Protocols, Algorithms, and Source Code in C"*, 1995.

[2] Diffie and Hellman, *"New Directions in Cryptography"*, IEEE Transactions on Information Theory, Vol. IT-22, No.6, 1976.

[3] S Vaudenay, *"A Classical Introduction to Cryptography: Applications for Communications Security"*, 2005.

[4] A. Menezes, P. van Oorschot and S. Vanstone, *"Handbook of Applied Cryptography"*, Chapter 7, CRC Press, 1996.

[5] Sandhu and Samarati, *"Access Control: Princples and Practice"*, IEEE Communications Magazine, September 1994.

[6] *"Dropbox company info"*, Available at: `https://www.dropbox.com/news/company-info`.

[7] Fu, Kamara and Kohno, *"Key Regression: Enabling Efficient Key Distribution for Secure Distributed Storage"*, 2005.

[8] Jason Crampton, *"Practical and Efficient Cryptographic Enforcement of Interval-Based Access Control Policies"*, Royal Holloway, University of London, 2011.

[9] Atallah et al, *"Efficient Techniques for Realizing Geo-spatial Access Control"*, Purdue University, 2007.

[10] Atallah et al, *"Dynamic and Efficient Key Management for Access Hierarchies"*, Purdue University, 2005 (revised 2009).

[11] Blanton, *"Key Management in Hierarchical Access Control"*, Ph.D. Thesis, Purdue University, 2007.

[12] Jason Crampton, *"Cryptographically-Enforced Hierarchical Access Control with Multiple Keys"*, Information Security Group, Royal Holloway, University of London, 2009.

[13] JW Lo, MS Hwang and CH Liu, *"An Efficient Key Assignment Scheme for Access Control in a Large Leaf Class Hierarchy*, Information Sciences, 2011.

[14] AK Das, NR Paul and L Tripathy, *"Cryptanalysis and Improvement of an Access Control in User Hierarchy Based on Elliptic Curve Cryptosystem"*, Information Sciences, 2012.

[15] YL Lin, CL Hsu, *"Secure key management scheme for dynamic hierarchical access control based on ECC"*, Journal of Systems and Software, 2011.

[16] Radia Perlman, *"File System Design with Assured Delete"*, Sun Microsystems, 2005.

[17] Dwayne C. Litzenberger, *"PyCrypto - The Python Cryptography Toolkit"*,
Available at: `https://www.dlitz.net/software/pycrypto/`.

[18] *"Java^{TM}Cryptography Architecture (JCA) Reference Guide"*,
Available at: `http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/Cry`

[19] *"The Legion of the Bouncy Castle"*,
Available at: `http://www.bouncycastle.org/java.html`.

[20] *"The GNU Crypto project"*,
Available at: `https://www.gnu.org/software/gnu-crypto/#introduction`.

[21] *"openssl: Ruby Standard Library Documentation"*,
Available at: `http://www.ruby-doc.org/stdlib-1.9.3/libdoc/openssl/rdoc/index.html`.

[22] *"Keyczar"*, Available at: `http://www.keyczar.org/`.

[23] NIST, *Recommendation for Block Cipher Modes of Operation*, 2001 Edition
Available at: `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`.

[24] Juliano Rizzo and Thai Duong, *Practical Padding Oracle Attacks*, May 25th 2010,
Available at: `http://static.usenix.org/events/woot10/tech/full_papers/Rizzo.pdf`.

[25] Elaine Barker, William Barker, William Burr, Wi lliam Polk, and Miles Smid, *NIST Special Publication 800-57*, Recommendation for Key Management – Part 1: General (Revision 3), July 2012,
Available at: `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf`

[26] FIPS, *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*, November 26, 2001,
Available at: `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[27] Meltem Sönmez Turan, Elaine Barker, William Burr, and Lily Chen, *NIST Special Publication 800-132*, Recommendation for Password-Based Key Derivation, Part 1: Storage Applications, December 2010,
Available at: `http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf`.

[28] Niels Provos and David Mazières, *A Future-Adaptable Password Scheme*, 1999 USENIX Annual Technical Conference, June 6–11 1999,
Available at: `http://static.usenix.org/event/usenix99/provos/provos.pdf`.

[29] Dropbox API Team, *"New sharing model will replace Public folder"*, 15th June 2012, Available at: `https://www.dropbox.com/developers/blog/19`.