File Sharing using Cryptographically Enforced Access Control

Daniel Randall
Imperial College London

Abstract

File sharing is an everyday activity for some. Currently files can only be shared by one group, or by one individual, at a time before access control requirements have to be specified again. We attempt to achieve greater automation when files are to be shared with many groups of people at a time. We have created a desktop application which utilises cryptography, access control and the Dropbox API to enable users to rank their files and their 'friends.' By doing so they are able to create their own personal hierarchy in which the files they share, automatically flows upwards to friends who are designated a ranking permitting them access to it. We achieve this while creating little overhead and leaving little room for new attacks. Finally we allow potential consumers to test of our work and find a file sharing system such as this may be ideal for the hierarchical structure of the traditional business.

Acknowledgements

I would like to thank my supervisor, Michael Huth, for our many meetings and for his helpful comments.

Contents

1 Introduction

1.1 Motivation

The internet has caused an explosion in the number of ways to share files. People are now, more than ever, relying on the internet to share data. For many years email was the platform of choice to achieve this, however recent years has seen this superseded by many specialised platforms in order to meet the differing needs in users. These have been in the form of file systems in Dropbox, document collaboration in Google Drive and Microsoft SharePoint, as well as social networks in Facebook. In general, the user sharing the file has no interest in sharing it with every user of the internet and so each platform implements some variant of access control. Typically the platforms available today allows for the user to configure the access control for their files by specifying the recipients of the file they are uploading, each time they wish to share data. Some efforts have been made in eliminating the need to specify access control requirements for every file, this can be seen in platforms that offer 'shared folders'. However it is often the case that a user has more than one group they wish to share the file with. Not only does this result in having to upload the file more than once, it also results in multiple instances of the file to be managed. Our solution to the access control problem is a desktop Dropbox App which employs cryptography to allow files to be 'freely' available, with the key which grants access being derivable to all members, in all groups the user permits access, thus automating the entire access control management process. Using the readily available Dropbox API, our App automatically encrypts and stores selected files in a Dropbox folder. When the level of importance of the files has been indicated, it is shared with friends that are permitted to access files at that level.

The basis for this project is a relatively new field, hierarchical cryptography. We harness these techniques to implement access control between users on file resources. We shall describe hierarchical cryptography, the cryptography it relies on as well as general access control in section 2. We will then examine similar products and techniques and show the limitations in their application.

Using the methods and techniques we described, we shall begin to define our solution and evaluate the best way to solve the problems we shall face. These problems shall include how to manage friends, files and the keys to

the files. Once we decide upon the best methods we shall then set about evaluating the best 3rd party software we can use to achieve these tasks and why we should not implement our own solutions.

Once our solution has been implemented we shall proceed to describe what we have done and begin to evaluate it. We will evaluate our implementation in three ways: speed, ease of use and security. We will benchmark our solution against an existing file sharing platform before we allow a number of the users to test drive our work. We will see in which environments our work excels and in which it fails to make an impression. The final part of our evaluation will see us critically assess how our implementation holds up against known attacks.

1.2 Objectives

The objectives for this project are as follows:

- **Automation** We want to design our system such that it automatically shares files with many people at once. However it is important that we only share the files with the friends the user wishes to share the file with, and we share the file with every friend the user wishes to share it with. We want to do this level of accuracy consistently. Users should be able to achieve this by simply labelling each file with a security level and each friend with a security level.

- **Simple to use** - Our solution should not require any knowledge of cryptography we use. Therefore we should deal with the cryptography 'under the hood' without troubling the user with technical requests. Our users should be able to use our system with little explanation.

- **Secure** - By this we mean that it is a requirement for our work not to introduce any security holes and to be relatively robust to the most common of attacks. Our solution should not be any less secure than other file sharing platforms.

- **Fast** - Our system will ideally run at a high speed, similar to alternative file sharing platforms.

2 Background research

2.1 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 used exclusively by the military until the 1970s, when 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] It wasn't until this and new methods of sharing keys before secure communication became possible. These new methods of sharing keys meant that keys could be transferred over insecure channels and thus eliminate the need for physical contact, such as the couriers which were used before this breakthrough. One such technique was a *public key cryptosystem*, in which two distinct asymmetric keys are used. One for encrypting, which is made publicly 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. The 1970s also saw the development of major publicly 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. Its application was not only for privacy but can also be used in many different different ways, such as: "authentication (bank cards, wireless telephone, e-commerce, pay-TV), access control (car lock systems, ski lifts), payment (prepaid telephone, e-cash)."[3] In this report the main focus of cryptograhy will be on access control.

Symmetric cryptography

A symmetric-key algorithm is a algorithm which uses the same key for the encryption of plaintext as it does for decryption of ciphertext. Symmetric-key algorithms can be either block ciphers or stream ciphers. Block ciphers (such as DES) are a type of symmetric key algorithm 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 κ (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 every time the plain-text is encrypted. However this behaviour is generally seen a undesirable. The reason this is undesirable as it releases information to an attacker, who is able to observe the same ciphertext has been encrypted twice. To circumvent this behaviour many modes of operation for symmetric block ciphers use *initialisation vectors* (IVs), which are given as an input to the cipher along with the key and plaintext. Different IVs with the same key and plaintext result in different ciphertexts. To decrypt the ciphertext correctly the IV needs to be issued again along with the key. Generation of IVs needs to be carefully monitored due to the deterministic nature of block ciphers, which means using the same IV twice with the same plaintext results in the same ciphertext.

Block ciphers attempt to obscure information using a process know as *confusion and diffusion*, where confusion refers to the process of creating a complex (non-linear) relationship between the ciphertext and the plaintext and key. Diffusion refers to process of each plaintext and key bit affecting many different ciphertext bits.

Stream ciphers work by using the key to create a *keystream*. The keystream is then used to encrypt the plaintext, often by XORing.

Asymmetric cryptography

Asymmetric cryptography involves the use of a public and private key. Where the public key is publicly available and is used to encrypt file. Because of this, the key used to encrypt the plaintext, the ciphertext and the algorithm used to encrypt the plaintext can be seen. Thus it is necessary for the attacker to be unable to derive the plaintext given this information. In order to achieve this, asymmetric cryptography relies on *Trapdoor functions*. Trapdoor functions are functions which are simple to perform but difficult to invert. For instance, RSA relies on the difficulty of inverting

$$f(m) = m^e (\mathrm{mod} N)$$

where m is the message to be encrypted, e is the value of the public key and N is the modulus.

Asymmetric cryptography is generally slower at performing encryption

and decryption when compared to symmetric. The reason for this is that typically symmetric algorithms use 'CPU-friendly' operations such as XOR (DES, Blowfish, AES) and table-lookups (DES, AES), while asymmetric algorithms often rely on complex Mathematics and employ costly operations such as exponentiation (RSA).

Asymmetric cryptography often results in a much larger ciphertext:plaintext ratio than symmetric cryptography. For instance, as described by PKCS#1, using RSA with a 1024-bit key, you can encrypt a data element only up to 117 bytes which yields a 128 byte ciphertext value.[**?**] Plaintexts larger than 177 bytes are required to be divided into many 177 byte blocks, each resulting in a 128 byte ciphertext. Over large plaintext, such as a file, this would add up quickly.

Due to the speed of encryption and size of the cipher text, asymmetric cryptography is often used to encrypt a symmetric key, which in turn is used to encrypt substantial information.

Hash algorithms

Hash functions are a one-way function, that is they are easy to compute but difficult to invert. Hash functions can accept inputs of any length but always have a fixed size output. This can be written as:

$$h : 2^* \rightarrow 2^k$$

where k is the size of the fixed output.

Most hash functions are *iterated hash functions* (MD5, SHA-1, SHA-2). Iterated hash functions work by using block-sizes, similarly to block ciphers. Hash functions divide input into equal length block-sizes. An internal compression function is then used to process each block. The internal compression function often works in fixed-sized rounds. For instance, MD5 has a block-size of 512 bits and its internal compression function works in 4 128-bit rounds.

The security of hash functions rely on a low amount of *collisions*. A collision occurs when $h(x_1) = h(x_2)$ where $x_1 \neq x_2$. Due to hash functions allowing inputs of any size, while constraining output to a fixed-size, it is impossible for hash functions to be collision free due to the *pigeonhole principle*. However it is desirable for a hash function to have a low collision rate. This is to prevent an attacker from being able to find two inputs with equal

outputs. Often hash functions are used in digital signature schemes making this very important.

2.2 Access control

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 (OS) control access to files by assigning permissions to users (e.g. 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.

2.3 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 $\lambda(u) \leq \lambda(o)$"[7].

2.3.1 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

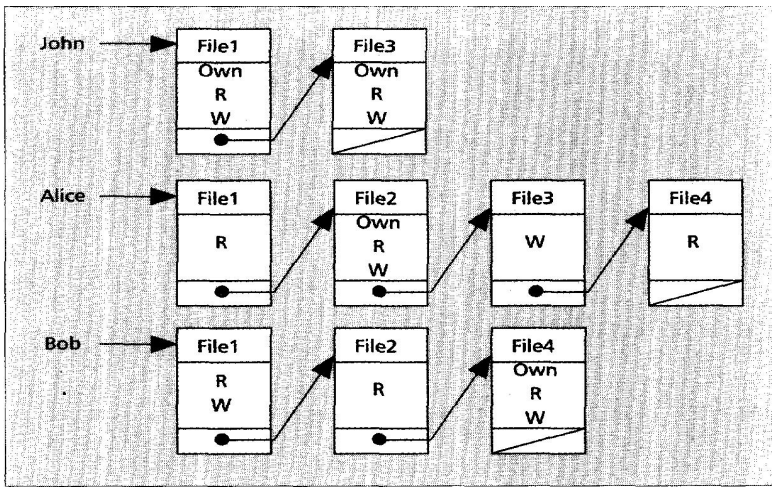


Figure 1: Capabiliy list for four files and three users.[5]

$1 \leq x \leq y \leq n$, 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 $\mathrm{HRec}(\mathcal{A})$ 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 $\prod_{i=1}^{k}[x_i, y_i] \in \mathrm{HRec}(\mathcal{A})$. Then we can 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"[7].

Some common implementations of this scheme are:[7]

- *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

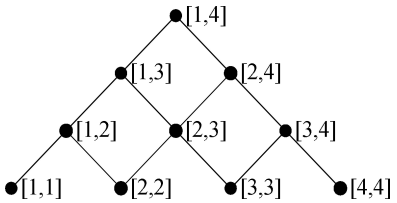


Figure 2: The Hasse diagram of (T4 , $\subseteq$).[7]

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.[8] 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, $T_{m,n} \stackrel{\text{def}}{=} [x_1, y_1] \times [x_2, y_2] : 1 \leq x_1 \leq y_1 \leq m, 1 \leq x_2 \leq y_2 \leq n$".[7] 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."[7]

2.3.2 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."[9] As an example, the simplest possible KAS would be to assign every single key for which $k(y) : y \leq l(u)$ where u is a user and l 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:[10]

- 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 algorithms which meet these criteria are:

- **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.[11] The AFB scheme by Atallah et al. is an example of this KAS, offering:[9]
  - 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:[11]

  - 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 $y \leq x$.
  - 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 publicly.

  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 publicly 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.[12] There has also recently been a number of works on elliptic curve cryptography (ECC)[13][14], which relies on public-key cryptography.

2.3.3 Revocation of permissions and re-encryption

There are occasions when a user previously granted access at some security interval is later revoked of those permissions. If the user was associated with

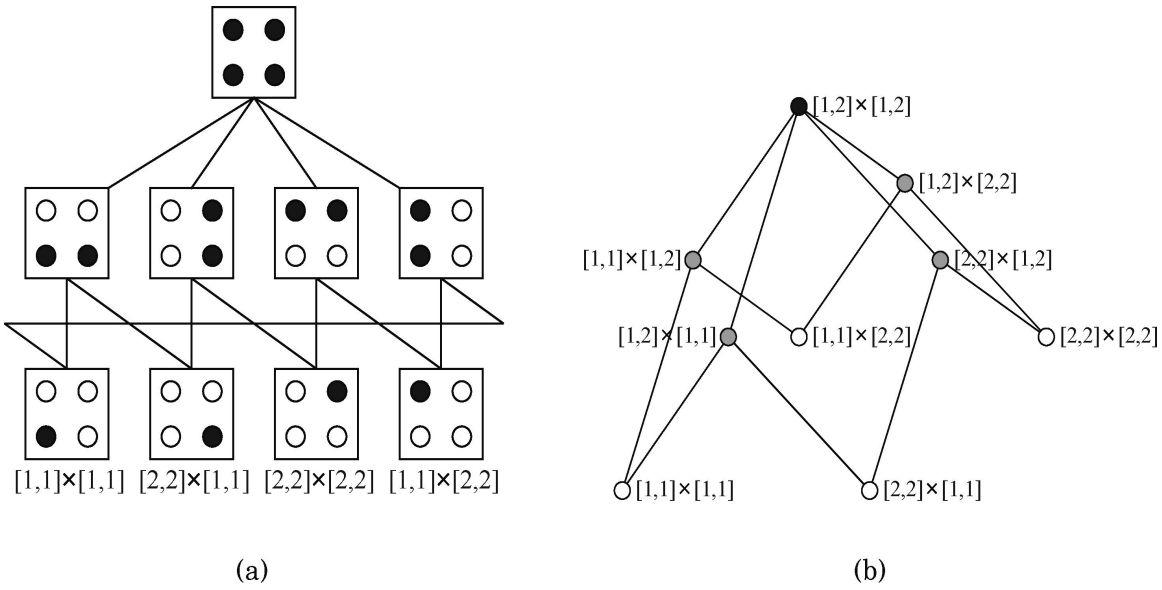


Figure 3: Two representations of $T_2 \times T_2$.[7]

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 no longer has access to the objects she previously had access to, therefore providing robust security. It also ensures 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. This means 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 which takes these points into consideration 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 revocation 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.[11] 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 for them 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.

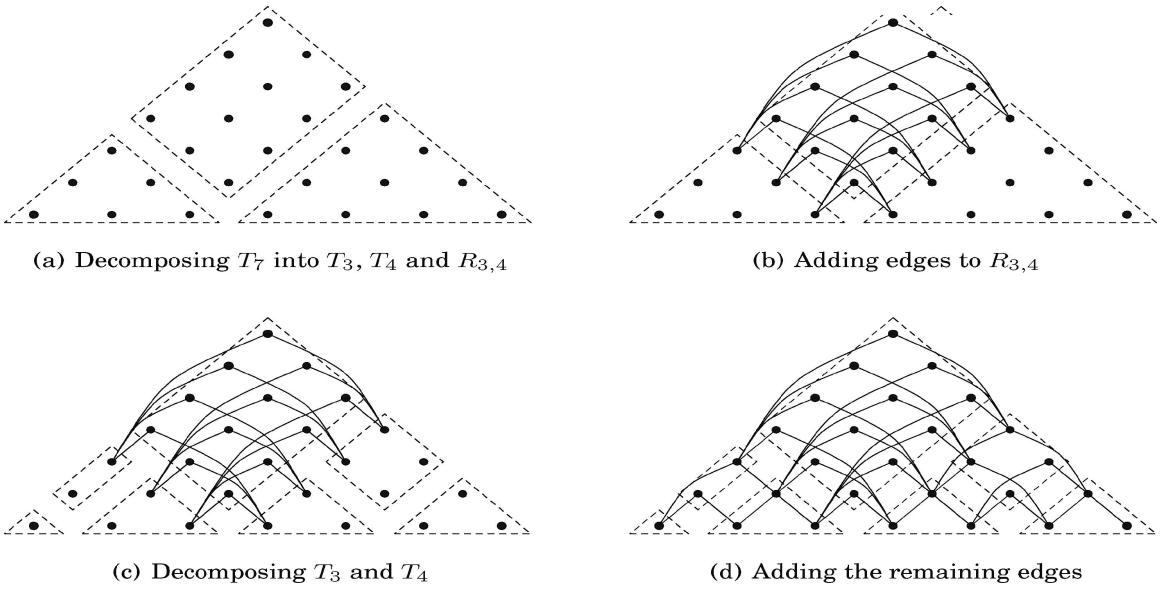
2.3.4 Tree optimisations

Crampton (2011) describes a process of binary decomposition which uses recursive labelling methods to separate the whole tree into nodes.[7] 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 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



(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.[7]

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, i.e. 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.

2.4 Related work

To the best of my knowledge no current commercial software implements hierarchichal cryptography for use in file sharing.

Established file sharing platforms such as Dropbox and Mega offer users a way to share their files with other users in a social environment. This can be in the form of a single file or an entire folder. Our system could be simulated using the folder sharing, where each folder represents a security level and the folders are simply shared with the desired users one-by-one. This method

requires a lot manual work.

For business environments, existing military based models such as the *Bell-LaPadula model* (BLP) could be applied to create a similar environment to which we are attempting. BLP works by labelling each object (file) and user with a security level, such as 'Unclassified' ... 'Top Secret'. BLP prevents subjects from reading objects with a higher level and from writing to objects with a lower level. We could attempt to use this by assigning each user their own BLP and give them full control of labelling. However the military environment is stricter than our target environment. We are also left with the unnecessary and unwanted write restriction and no method of sharing the files outside of the network in which the model is located is specified.

A number of file sharing solutions for businesses are available, such as YouSendIt and Box which offer alternatives to using FTP. Such solutions are vey similar to Dropbox and Mega, whereby each folder has no relation to each other. Access to each folder is controlled independently and options exist to send individual files. Both, YouSendIt and Box, offer the ability to password protect files and folders.

3 Design

3.1 System design

In this section we will discuss the required components of the system, why they are necessary for our system and how we shall go about achieving each one. The required components of the system are:

- **Information storage**
- **Authentication**
- **Encryption of files**
- **Key exchange**
- **Friends**
- **Key revocation**
- **Version control**
- **GUI**

3.1.1 Information storage

We need a method of keeping track of the information associated with the files uploaded, such as the security level, initialisation vector, the user who uploaded it, as well as information about each user, such as authentication details and friends. We need to keep track of this information to be able to display it back to the user and ensure only authorised operations are carried out by authorised users.

The traditional way of achieving this is to allocate a single authority who keeps track of a database. In this model a user is required to communicate with the authority to conduct actions who then verifies that the action is permitted. When an action has been verified for this user the authority may send some information required for the action to process and on completion of the action the user may send the necessary information to the authority to store for latter use. The most popular way of achieving this is using the *client–server model*. Using the client–server model each entity is modelled by either a client or a server. In our environment each user is modelled by

a client and our single authority is modelled by a server. However, there are issues involved in using a centralised authority. Having a single server which may not able to process all of the traffic from the clients introduces a bottle neck. There are also privacy issues involved by housing information in a centralised authority. The user must trust that those who control the information do not have malicious intent, are competent enough to protect the information sufficiently or are simply unlucky enough to fall victim to a 'unpreventable' attack.

It is possible to keep track of the information we need in a manner which does not require a centralised authority. We could attempt to implement a peer-to-peer network similar to that used by Bitcoin today. That is to allow each user to submit updated friends lists and file information, such as a unique file identifier and associated security level, into a public record. However implementing such a system requires a large number of users to donate a large amount of processing power to authorise each submission by users. While Bitcoin can encourage this with financial rewards we can offer no such thing. This model also requires each user's friends list, including each friend and their associated security level, to be public. The number of files each user has submitted at which security level will also be public.

Due to the the client–server model being well established, familiar with many users and the easier option to implement that is what we shall use for this system.

3.1.2 Authentication

As we are using a client–server model we need some way for the client to authenticate themselves with the server. The traditional manner of achieving this is for each user to have a unique, public identifier, such as a username, and private information for the client to verify who they say they are, such as a password. Upon log in the client must provide both pieces of information to the server before being authorised to perform operations unique to that user, and before being provided information unique to that user.

In order for the server to be able to verify that the username and password that the client supplies are indeed correct, both pieces of information need to have been stored previously for the server to retrieve. Due to the username being public there are no privacy issues in storing it as plaintext in the database. The password is private information and so care must be taken before it is stored, due to privacy issues as well as security ones in the case the

database is compromised. This problem can be solved by using a password hasher before the password is stored in the database. Doing so means that the plaintext password can not be read by database administrators and if the database is compromised an attacker can not simply resubmit the hashed password and gain access to a user's account. The hashing must take place on the server side. This is because, while the user's plaintext password can not be read, in the event of the database being compromised the hashed password can be simply re-sent to the server and any security benefits of hashing the password would be lost. Care still needs to be taken in sending the plaintext password to the server. In order to prevent eavesdropping we will send the plaintext password securely, either through a secure channel such as a HTTPS tunnel or encrypted with a public key belonging to the server.

In summary, we shall require each user to register with a username and a password. The passwords will be securely sent to the server, which will then hash the passwords. The hash will be stored in a database to be retrieved and used for authentication given a username and the plaintext password.

3.1.3 Encryption of files

In order for our users to safely upload our files into a 'public' domain they need to encrypt the files. In our scenario the authority on security levels is the only person who we wish to grant access to encrypt and upload files for that level. For this reason we shall use symmetric key cryptography. While we could use public key cryptography to achieve this, symmetric key cryptography is faster and, as we have no prior knowledge on the size of the files the user may wish to upload, this speed difference may well be significant.

3.1.4 Key exchange

Given a key, a user will be able to derive any key with a security level below that for a given user. However assuming that a user wishes to grant another user access to a maximum of some level, the key for that level needs to transferred to the other user without any other user, who is not a part of the transaction, being able to observe it. This is the type of scenario which public key cryptography was designed. In order to achieve this behaviour using public key cryptography, each user can generate their own public and private key pair and send the public key to the server to be stored publicly. When a

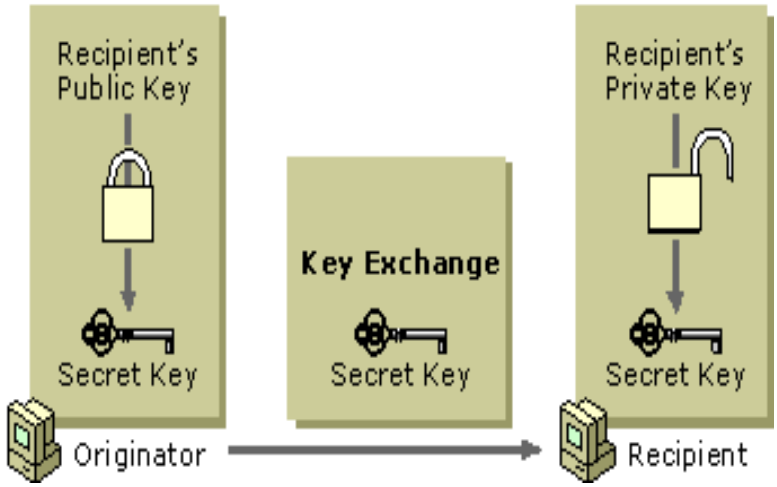


Figure 5: A depiction of two users exchanging a key using RSA. The secret key represents a symmetric key owned by the sender. To send the key the sender encrypts the symmetric key with the receiver's public key and sends the resulting ciphertext. The receiver then decrypts the ciphertext with their private key to complete the proccess and receive the key.

Source: http://technet.microsoft.com/en-us/library/cc962035.aspx

user wishes to share their files with another user they can then request the public key of the user in question and then encrypt the symmetric of the highest level they wish to share at then send that to the server for storage. The receiving user can decrypt the key using their own private key which they have stored.

A visual representation of this can be seen in Figure 5.

3.1.5 Friends

There are a few options available to us in how we handle the adding of friends. We could allow for, either, the owner of files to request to share the files with another user or for users to request access to another user's files. There are no real positives or negatives associated with either method, but it makes more sense for the owner of the files to decide who has access, rather than rely on the correct people to request access.

When the owner of files decides another user is entitled to access some of their files we shall post a request to that user stating the owner wishes

to share files with them. The reason for this is that without requests the user may be subject to un-wanted or potentially malicious files which they would have no way of preventing. It may also be the case that the underlying network charges for the amount of space consumed and the users may wish to be selective on who can share files with them.

As all of the files are encrypted, when a user is granted access to some owner's files at some level, we have the option on which files we share with them. The files they have been granted access to should be be shared with them, however we could simply share all of the files the owner has uploaded with them. This is acceptable as the user is only able to decrypt ones for which they have been able to derive keys for. The benefit of this method is, if in the future the user's access level is increased by the owner no further change is needed in regards to the encrypted files shared with them. The downside is all files uploaded in the future by the owner which they do not have access to has to be shared with them. Also unnecessary space is taken by files the user is not interested in. For this reason we shall only share the owner's files for which the user has access to.

In summary, in our system an owner of files will be able to request to share their files with other users at some level. With the request shall be the symmetric key representing the highest level the user wishes to share with the potential friend. The key shall be encrypted with the potential friend's public key. If the potential friend accepts the request, while all uploaded files are considered to be in the public domain, we will only share files directly to the new friend for which they will be able to decrypt (e.g. if the levels available are 1-5, where 1 is the highest, and this user is permitted for levels $<= 3$ then we shall share files labelled $<= 3$ but not > 3.)

3.1.6 Key revocation

There may come a point when a user no longer wishes for a friend to be able to access their files. We are able to implement this feature in one of two ways, as previously described in the background section - immediate or lazy. The computational requirement for both is the same, however lazy key revocation spreads the computation needed over time. Lazy key revocation requires much more complex logic and so, due to the time constraints on this project only, we shall implement immediate key revocation.

3.1.7 Key storage

As discussed in the previous sections, *Encryption of files* and *Key exchange*, each user is required to store some number of symmetric keys and a single private key. The privacy of these keys is fundamental to the functioning of the system in the way we desire. If the symmetric keys are compromised then all of the user's files can be accessed at will, as well as all of the keys which have been shared with this user. If the private key is compromised then all future keys shared with this user can be intercepted and derived. Due to these reasons, the keys need to be carefully stored. To avoid storing the keys in unprotected files there are software implementations to keep them in called *key stores*. These key stores can be protected by a password which has to be supplied before the keys can be accessed. It would be more secure if we require a user to supply two separate, distinct passwords. One password is sent to the server to be hashed as previously explained and the other is not sent to the server and instead used to create a key store. This means that even in the event of the database at the server being compromised, the attacker is still unable to access the user's key store, even with access to user's particular machine. However this method requires the user to remember two different passwords, which could be considered an unreasonable request for the rare event in which it would provide any extra security. In the case where the likely attackers are known personally to a user, such as in the same company or in the same household, it is likely be a significant improvement in security. As we do not have any real knowledge on our likely userbase's situations we shall simply reuse the password for server authentication as the password for the key store.

3.1.8 File updating

It may be desirable for the users who receive shared files to be able to edit them and update the file owner's version of the file. This would prevent the user who changed the file from having to upload the file themselves and integrate file management into their own security levels. For instance, they may not wish for the owner of that particular file to see any of their own files, which they would be required to do if they uploaded the changed file themselves, at some security level, and shared that level's corresponding key with the owner. Allowing people to update the file at other user's locations circumvents the need for this.

However an issue arises when two users the file owner has shared the file with are editing that file concurrently. If one user updates the version of the file and then the other updates it soon after the first update will be lost in the public domain. Unfortunately it is outside the scope of this work to deal with version control issues in a meaningful way. One measure we can take in an attempt to prevent this is to ripple out the changes to all users who have access to the file. As users are required to download and decrypt the file to a separate location this would not overwrite the changes any user may make but it will alert each user that changes have been made which may wish to incorporate into their own changes before updating the version of the file themselves.

It can not be assumed that allowing their friends to do this is desired for every friend added. We will add an option when a user adds a friend, where they can indicate whether that friend is permitted to update files. This option, in a sense, is similar to granting 'read access' or 'read and write' access. This would work by friend privileges being stored in the database and then when the friend desires to update the file they are required to verify with the server before the action is permitted.

3.1.9 GUI

While the system we are designing is perfectly usable in a command line window, the tables of data, such as a list of the user's own uploaded files, friends and files shared with them, would be far easier to display in visual tables. The multitude of actions they are able to perform with the data would be accessed more easily represented in a graphical form rather than a text based form. It is also possible that retaining the system in a command line window would reduce the potential user base a great deal. For these reasons we shall design a simple interface which can be used to view and manipulate a user's files and friends.

3.2 File sharing platform

The system we are developing needs some way to transfer files from one user to another when they desire. The are many ways in which we can do this. Preferably the method we choose allows for us to easily automate the sending and receiving of the files to minimise the work the user has to conduct to assist the working of our application. This requirement rules out methods

such as email, Facebook and Skype. Ideally the platform also allows us to make the files publicly available. As we intend to encrypt the files, they do not need to be hidden away.

As observed by Fu, Kamara and Kohno, an appropriate file transfer method for hierarchical access control is a content distribution network such as BitTorrent.[6] However, in our case, due to a number of different torrent clients available (uTorrent, Deluge...), many different ways to connect to a tracker (.torrent files, magnetic links...) and inconsistent performance, this would be a challenging task to start from scratch. However there is a a Web API for uTorrent which allow us to interface with a BitTorrent client.[**?**] Unfortunately there are limitations to the uTorrent API. It only provides a way to process a .torrent and so we would still need a centralised BitTorrent tracker. This could be achieved by using a public tracker such as The Pirate Bay, however we would still need some way of uploading the torrents to the tracker website, as well as a way of keeping track of the files uploaded and who they have been uploaded by. The uTorrent API also does not allow us to control where individual files are downloaded, this means that using the API requires us to download the files in the same location as files that have been downloaded using uTorrent outside of our system. Lastly uTorrent as an application is also only available on Windows, thereby limiting our development and deployment platforms.

There are many modern dedicated filesharing tools available today that we can consider. Some of these are: Dropbox, Mega and Wuala. These tools are ideal in the sense that they borrow the folder structure implemented by filesystems which naturally conform to strict hierarchy requirements. This means we have a lot of power in how we can choose to manage our files. Dropbox is incredibly popular, with over 100 million users collectively saving 1 billions files per day.[**?**] It also has an extensive SDK for many different programming languages. However with the discontinuation of its 'public folder'[31], we have no way of making our encrypted files publicly available.

Mega is a relatively new file hosting service (19 January 2013) with over 3 million members.[1] Mega has stricter security requirements than that of Dropbox which closer match what we are trying to achieve. For instance, encryption is performed client side using AES in CBC mode with a 128-bit key.[**?**] However as we are having to perform this actions ourselves, regardless,

[1] As reported by Kim 'Dotcom': `https://twitter.com/KimDotcom/statuses/304003119447158784`

it is irrelevant to us, if not a hindrance due to wasted CPU time. But Mega does allow us to control what degree of access a user has, such as 'read', or 'read/write' which would be useful to us. However at the time of writing an SDK for Mega has been proposed but not released.

As we just described, there is there no existing platform which meets all of needs, which means we will have to make sacrifices. The closest storage which meets the needs for our solution is BitTorrent, as we will be able to make the files public. However the SDK available for it has a number of limitations and the time constraints on this project mean that is not feasible to create solutions for them all. Therefore, due to the extremely large userbase, multi-platform support and an extensive, 'tried-and-tested', SDK we shall use Dropbox to transfer our files.

3.3 Programming language

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 for mobile development there are options for Android and iOS development.
Following is a run down of the different languages and what useful 3rd party software 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.[16] Programming in Python is quick and flexible.

- The main cryptography API available for Java is the Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE) 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[17]. An alternative is The Legion of the Bouncy Castle API which incorporates JCA/JCE[18]. 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."[19].

- Ruby offers a module to interface with the OpenSSL cryptography library written in C. The module offers many modern ciphers such as AES.[20]

- 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++.[21] 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 system.

Due to the author's inexperience with Ruby and wider cryptography support for the other languages available, Ruby is likely not the best choice of language for this project. It also seems unnecessary to take the extra time needed to code the project in C/C++ to risk compatibility issues and possible performance hits by linking the languages. While the Dropbox source code is written in Python and so may well integrate better with the Python Dropbox SDK, Java offers a greater range of cryptographic support.

While mobile development is a possibility, the author's experience is mainly in desktop programming and it feels unnecessary to limit the computational power available to us when compared to a desktop application considering the system would be more no more valuable in the mobile domain.

For these reasons we shall create a desktop application in Java.

3.4 Algorithm choices

In this section we shall decide which algorithms we shall use to perform the tasks described in the previous section and attempt to rationalise each decision.

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 it into a multiple of the block size of that algorithm.

We will not consider inventing our own algorithms to perform cryptography or any security functions. One reason for this is a lack of time, another is 'Schneier's Law.' Bruce Schneier wrote in 1998: "Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break."[**?**] This means that, we can not, in a short space of time, realistically predict and evaluate all threats and prevent them. For this reason we shall not roll out our own cryptography and security designs, and instead use pre-built and well-tested security solutions wherever possible.

3.4.1 Symmetric Algorithm

- **DES** - A block cipher which used to be government standard. DES uses 56-bit keys, however such small key lengths have proven to be insecure (e.g. 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^{112}-bits 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.

AES is by far the most popular block cipher in use today and is currently approved by FIPS for use.[22] 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.

3.4.2 Asymmetric algorithm

As we wish to use asymmetric encryption for key exchange, the options available to us are:

- **RSA** - Relies on the difficulty of factoring large primes
- **ElGamal** - Relies on the difficulty of solving the discrete logarithm problem.
- **DSA** - A variant on ElGamal, typically used for the digital signatures
- **Diffie-Hellman** - A method of key exchange using asymmetric cryptography.

By far the most popular and widely supported asymmetric algorithm used today is RSA. For that reason only, that is what we shall use.

The key sizes supported by RSA are:

- **1024**
- **2048**
- **3072**
- **4028**

Similarly 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.[**?**] 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 unacceptable for use with RSA. A 2048-bit key provides 112-bits of security (pg. 64, Table 2) which is acceptable today through to 2030 which is plenty good enough for our system (pg. 67, Table 4).[25]

Due to how RSA works there is a limit on the size of the plaintexts we can encrypt in relation to the size of the key we choose. As we only wish to use the keys to encrypt AES keys of 128-bit in length, the limit imposed by choosing a 2048-bit RSA key, 2047-bits, is plenty for the task in hand.
We shall use a 2048-bit RSA key.

The PKCS#1 v1.5 padding scheme and the OAEP padding scheme which was standardised in PKCS#1 v2.0 are both endorsed by FIPS for use with RSA. OAEP was designed to overcome two weaknesses that exist with PKCS#1 v1.5, that is to become more resilient in the face of chosen ciphertext attacks and to prove the security that it provides. While OAEP is not infallible[28], it would seem to be an improvement on PKCS# 1 v1.5 and so that is what we shall use.

OAEP needs to be parameterised with a hash algorithm. While RSA labs recommends SHA-1, we know that to be outdated due to the fact it only provides 80-bits of security, less than we required. Instead we shall use a hash algorithm with at least the same number bits of security as provided by our asymmetric algorithm, 112-bits. All of the SHA-2 family of hash algorithms provide at least 112-bits of security. SHA-224 provides 112-bits, SHA-256 provides 192-bits and SHA-512 provides 256-bits. Previously we chose algorithms with the lowest feasible level of security for speed reasons, however this is not the case in this scenario. SHA-512 is faster than SHA-256 on modern 64-bit CPUs. The reason why SHA-512 performs better is because it has "37.5% less rounds per byte (80 rounds operating on 128 byte blocks) compared to SHA-256 (64 rounds operating on 64 byte blocks.)"[29] SHA-224 is a variant on SHA-256 and so will likely perform similarly. For this reason we shall parameterise OAEP with SHA-512.

3.4.3 Password hashing algorithm

Password hashing algorithms are odd in the sense that the trait with which to separate 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, or to allow them to quickly bruteforce the plaintext given a pre-computed hash. Making the hashing algorithm slower makes these tasks much more time consuming.
There are two password hashing algorithms commonly in use. They are:

- **PBKDF2** - Based on the SHA-256 hash function (or can be used with other hash functions.)
- **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.[30] 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[**?**], however it has only existed for a small amount of time (4 years at the time of writing) and so scrypt has not yet 'stood the test of time' like that of bcrypt (nearly 14 years at the time of writing,) which is important for security algorithms.

For these reasons we shall use bcrypt to store passwords in our system.

bcrypt requires us to choose a number of rounds (work factor), where the number of iterations bcrypt has to perform is $2^{workfactor}$. The higher the number of rounds the longer it takes to run and therefore the higher the security. Unfortunately the Java implementation we are to use is slower than the C implementation that the attacker would use, meaning that we can not choose the optimal number of rounds for our hardware. We also do not have the luxury of a powerful server to perform the hashing quickly, as will the

attackers, putting us at a further disadvantage. To help us choose a work factor, benchmarks for each work factor can be seen in Table **??**. It is evident that as the work rate increases our hardware quickly reaches its limit, where at work rate 14 it is unable to hash a single password within a second. If we were to have a substantial user base we would be forced to have a lower work factor in order to be able to compute many hashes in a short amount of time, such as 7, which allows to compute 86 in a second. However we are unlikely to have a large user base as this current time, so for this reason we shall choose a larger one which is still low enough to not pose a significant inconvenience to our users. A work rate of 10 will likely give us significant protection, while only taking 87.7 milliseconds on average. We shall choose a work rate of 10 for bcrypt.

Work factor	**Iterations**	**Time taken (milliseconds)**	**Average (milliseconds)**	**Average passwords per second**
6	10	58.584	5.86	170
7	10	115.982	11.59	86
8	10	221.497	22.15	45
9	10	438.253	43.83	22
10	10	877.052	87.7	11
11	10	1734.642	173.46	5
12	10	3458.536	345.85	2
13	10	6923.244	692.32	1
14	10	13821.333	1282.13	0

Table 1: The time taken to hash passwords using bcrypt at varying work rates on our machine. We calculate the total time to hash 10 passwords which is in the *Time taken* column, the average time to hash a single password in the *Average* column and the amount of passwords our server can hash in a single second in the *Average passwords per second* column.

3.4.4 Random number generator

As we have discussed, in order to complete certain actions with our cryptographic algorithms our users are going to require keys and initialisation vectors. We need a way to generate these when we desire. However it is important that we achieve this in a secure way. By secure, we mean in a manner which can not be feasibly predicted by an attacker, even when given the entire past sequence of numbers generated. This is important because, if the generation of the keys can be predicted in any way (e.g. the 1000^{th} instance will be equal to the 1^{st},) then the attacker would be able to manipulate our solution such that they can find the keys for some user, thereby gaining access to their files. A similar argument can be made for predictable initialisation vectors, if an attacker is able to predict the IVs generated then the symmetric cipher mode we have chosen for our solution, CBC, is not properly implemented and would become equivalent to ECB. A mode which we have already decided is insufficient to meet our requirements.

Due to our requirements, we can not use the most popular software random number generators (RNG) in use today. The majority of the RNGs are not truly random, but are deterministic and so are called psuedo random number generators (PRNG). The majority of PRNG are based on the linear congruential generator (java.util.Random, C rand()) or the linear feedback shift register. PRNG based on linear congruential generators have been proven to be easily predictable.[**?**] Thus we shall avoid all generators based upon such models.

Ideally we do not wish for pseudo random numbers but for 'truly' random numbers. This can be achieved by using hardware RNGs. Such hardware is generally based upon unpredictable physics-based events. For instance hardware RNGs based upon a Geiger–Müller counter detects and counts the radiation particles passing by it every minute.[**?**] Unfortunately there is not enough funds for this project to purchase such equipment and, even if we could, we would then have to transfer the number from the server to the client.

While we can not use completely predictable PSRNGs and we cannot attain truly RNGs, a compromise can be reached. The compromise is in the form of secure PSRNGs. Secure PSRNGs extracts seeds from 'truly' random sources, such as */dev/random* on a Linux machine, and use that to generate pseudo random numbers. To obtain as close to 'truly' random numbers as we can we could simply use the seeds as our random numbers, but this can

be a slow process as the call to /dev/random blocks when enough entropy has not been collected. However as the functioning of our solution relies on the the secure generation of these numbers, there is little we can do about this.

3.5 Algorithm providers

In this section we shall evaluate which implementations of cryptographic and random number generator algorithms are best fit to employ in our system.

There are many different providers of cryptographic 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 unnecessary 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.
- **Ease of use** - How easy it is to incorporate the implementation into our solution.

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.

We shall consider the SunJCE, the suite of cryptography algorithms which come with the JDK. The second we will consider is Bouncy Castle, the popular, extensive set of algorithms. BouncyCastle provide their own API. However we will consider Bouncy Castle through the JCA provider structure due to it being simpler to program and change in the future if necessary (either to other algorithms or to another provider.) Lastly, we shall consider GNU-Crypto, an open source set of algorithms which are claimed to be high quality and provably correct.

3.5.1 AES algorithm provider

We are going to use the AES algorithm to encrypt files before they are uploaded. We can not predict how large or small the file may be. Thus to thoroughly assess how well an algorithm may perform at this task we will need to test them on files of different sizes. That is what we shall do here.

The findings for each implementation of AES can be found in Table **??** for a small file and in Table **??** for a large file.

GNUCrypto is tested with a slightly different algorithm, specifically a different padding 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 been irrelevant 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 be observed as a result.

Looking at the table the most striking result is the disappointing length of time the GNUCrypto implementation takes to complete the tests for encrypting the small, 1.3MB .pdf file. The GNUCrypto implementation takes more than 3 times as long on average, when compared to the SunJCE and Bouncy Castle implementations. However, interestingly, it performs reasonably well when encrypting the large .zip file, better than the Bouncy Castle implementation. Overall, the SunJCE clearly attains the best performance, achieving consistent results over large and small files.

Bouncy Castle offers, by far, the most extensive library of cryptographic algorithms, but this comes at the cost of a very large import, when compared to the alternatives, at 2.3MB. GNUCrypto requires a rather small import of 598.0kB but the SunJCE requires none at all.

Incorporating the GNUCrpyto implementation into the project is far more complicated than the interface provided by the JCE framework (requiring over 15× as much code!)

From these results it is quite clear that SunJCE offers the most attractive implementation for AES, offering the fastest and most consistent performance, with no import necessary and a very simple interface.

Implementation	**Import Size**	**Time taken (seconds)**	**Average (seconds)**
SunJCE	0	11.480270937	0.011
GNUCrypto	598.0kB	34.026706672	0.034
Bouncy Castle (via JCA provider)	2.3MB	11.25560253	0.011

Table 2: Comparisons between different implementations of AES using PKCS5/7 for padding and a 128-bits for the key size. For each implementation a 1.3MB .pdf is encrypted 1000 times. The *Time taken (seconds)* column holds the time taken for the implementation to complete all 1000 iterations. The *Average* column is the average time taken for the implementation to encrypt the file a single time.

Implementation	**Import Size**	**Time taken (seconds)**	**Average (seconds)**
SunJCE	0	185.568239816	1.86
GNUCrypto	598.0kB	187.959853064	1.88
Bouncy Castle (via JCA provider)	2.3MB	193.091386035	1.93

Table 3: Comparisons between different implementations of AES using PKCS5/7 for padding and a 128-bits for the key size. For each implementation a 209.7MB .zip is encrypted 100 times. The *Time taken (seconds)* column holds the time taken for the implementation to complete all 100 iterations. The *Average* column is the average time taken for the implementation to encrypt the file a single time.

3.5.2 RSA algorithm provider

We are only using the RSA algorithm for one purpose, that is to encrypt symmetric keys for transfer between users. For this reason it is sufficient to compare the performances of different implementations solely on how they perform at this task. That is what we shall do here.

The results from the comparisons can be seen in Table 3. GNUCrypto does not provide an implementation of RSA and so it is omitted from the test. As we can see there is little to separate the two implementations, with both implementations performing extremely quickly.

In order to avoid unnecessary imports and maintain a level of consistency in our algorithm providers we shall simply use the SunJCE implementation of the RSA algorithm.

Implementation	**Import Size**	**Time taken (seconds)**	**Average (seconds)**
SunJCE	0	19.714574141	0.000197
Bouncy Castle (via JCA provider)	2.3MB	19.047213702	0.000190

Table 4: Comparisons between different implementations of RSA using OAEP with SHA-512 for padding and 2048-bits for the key size. Each implementation encrypts a 128-bit AES key 100000 times. The *Time taken (seconds)* column holds the time taken for the implementation to complete all 100000 iterations. The *Average* column is the average time taken for the implementation to encrypt the file a single time.

3.5.3 Random number generator provider

We need to generate random numbers for the following:

- **AES keys**
- **Initialisation vectors for AES**
- **Public/private key pair for RSA**

It will be sufficient for the generation of AES keys and IVs for us to use a seed from the secure PSRNG at the length we desire. The default PSRNG implementations for the *SecureRandom* Java class are NativePRNG for Linux, which uses /dev/random to seed, and SHA1PRNG for Windows, which uses CryptoGenRandom to seed. Both are able to perform at a sufficient level and so we shall use the defaults to create our seeds.

The RSA public/private key pair are related to each in a mathematical way, therefore simply generating two random numbers is not sufficient. It would seem unnecessary, as well as the introduction for possible error, for us to conduct the mathematics necessary to create them ourselves. Therefore we shall use a pre-existing solution. Java offers a *KeyPairGenerator* class which can be supplemented with an algorithm provider and a PSRNG. This what we shall use to create an RSA key pair.

3.6 Databases

Some popular embedded Java database we could use are:

- **Apache Derby** - A simple, popular database with an embedded JDBC driver.
- **H2Database** - An open source, light-weight database promising good performance.

H2, at 1.5MB import, has a smaller footprint than Derby, at 2.6MB. H2 also claims to have superior performance when compared to Derby for simple queries.[**?**] It is unlikely that we will have to execute more complex queries beyond simple SELECT... FROM ... WHEREs meaning that the tests are likely to be highly relevant to our situation.

Hibernate is a database framework designed to be able to make Object/Relational Mapping easier. The only object that would be convenient to store in the database is the 'friends list' we intend to design. Considering the use of Hibernate comes with an overhead and in order to use the latest stable version (v4.2.2) we are required to include 8 .jar files, totalling 6.3MB, it would seem overkill to use solely to store a single object. Instead it may be more sensible to simply use the Java *serialization* process and store that in a 'simple' database.

For these reasons we shall employ the H2Database in our system without the Hibernate framework.

4 Implementation

In this section we shall discuss the implementation created as a result of our discussion in the previous section.

4.1 Structure

An overview of the architecture to our solution can be seen in Figure **??**. This represents the basic communication structure between the major components of our system.

The general structure of our solution for the client can be viewed in Figure 6, while the general structure of the Server can be viewed in Figure 7. In both of these diagrams we have obscured out some, less integral, classes to aid readability.

We will now briefly discuss some of these classes and the roles they play in the functioning of our system. We have many classes in our solution and it would not be sensible to discuss them all. For this reason we shall, at times, discuss many at once.

4.1.1 Client

View - This represents all of the classes used to create the GUI. These classes are used to display information to the user and accept actions which are then forwarded onto the *Controller*.

Controller - This represents the collection of classes used to translate commands from *View* into logic which is held in the *Model* (*Actions, Register, Login*). After actions have been executed it receives updates from the *Model* which it forwards onto the *View*.

Actions - This class contains the majority of the logic in the client side. It communicates with the server to perform the necessary actions (such as 'Add friend', 'Upload file' and 'Download file')

Register - Contains the logic necessary to register the user with the server and Dropbox. If the server accepts the user the keys for each security level are created and stored in the keystore.

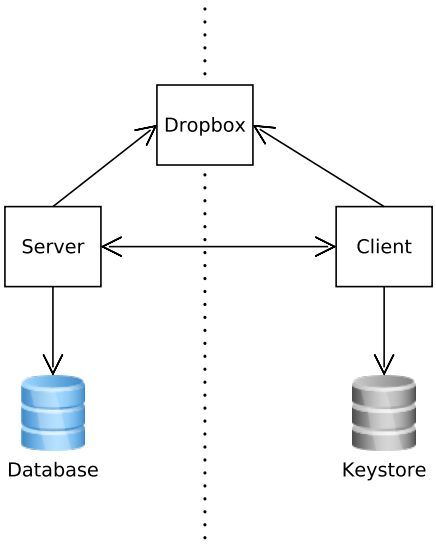


Figure 6: An overview of the architecture of the solution. The dotted line separates the 'client-side' from the 'server-side'. Both the client and server communicate with Dropbox and so that is placed between both sides. The client and the server both communicate with each other, but only the server communicates with the database and only the client communicates with the keystore.

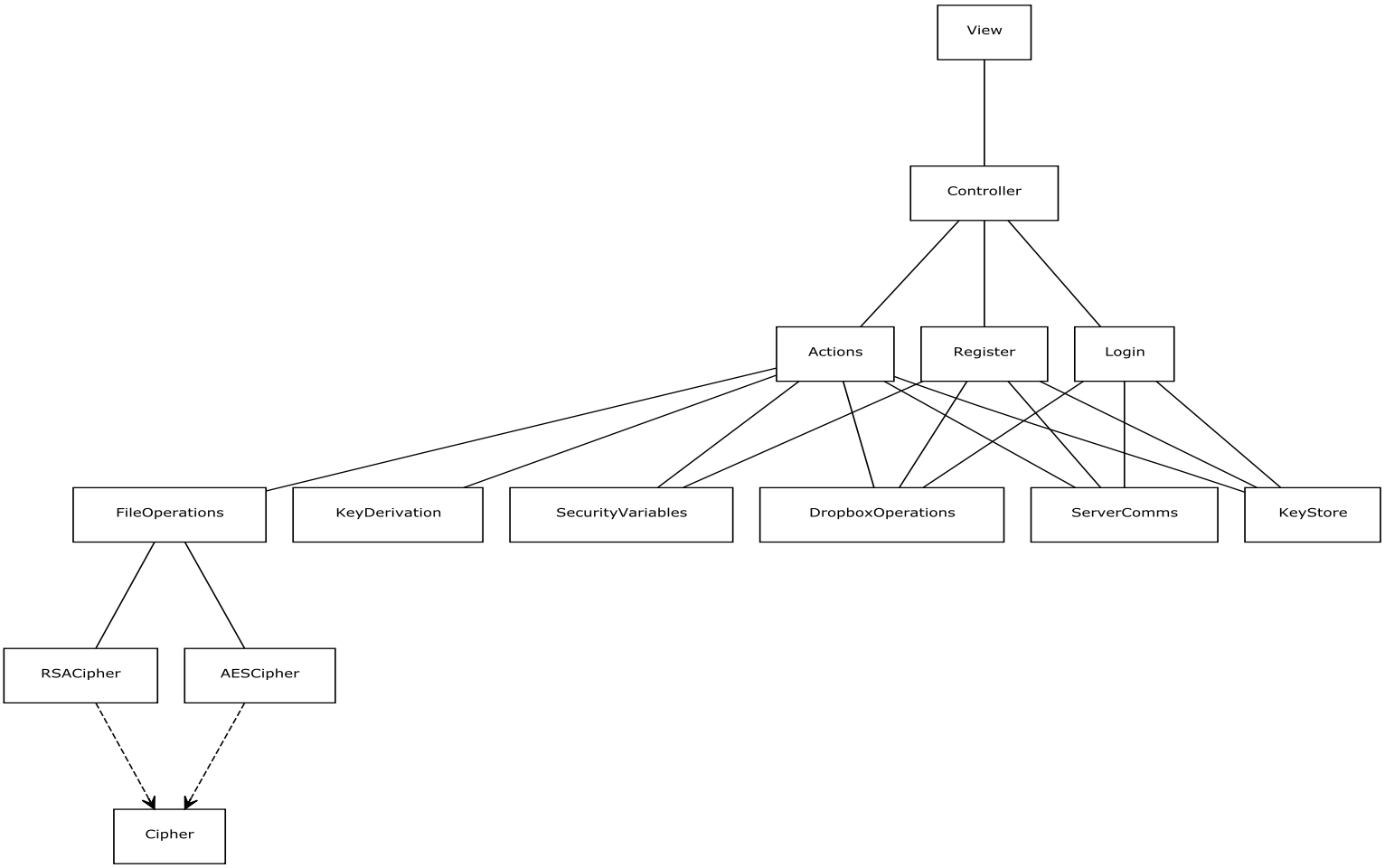


Figure 7: Reduced class diagram for the client

SecurityVariables - This class uses our PRNG to generate our keys, for both symmetric and asymmetric cryptographic algorithms.

AESCipher - Performs the functions of the AES algorithm we decided to include. Performs encryptions and decryptions given a key, iv and a file.

RSACipher - Performs the functions of the RSA algorithm we decided to include. Performs encryptions and decryptions given a key and a file.

KeyStore - This class interfaces with the keystore which holds our keys. Is used to store and retrieve our symmetric and asymmetric keys as well as our friends. Each key is stored with a label which must be referenced when we wish to retrieve it. Our own symmetric keys are stored with the label of their security level. Our asymmetric keys are stored with the label of either "public" or "private". Our friends asymmetric keys are stored with the security level along with a prefix of the friend's username. There is no need to store our friend's public keys. Due to the restrictions on levels and usernames, where we cannot of two levels or two usernames of the same name, we can not have any clashes in the keystore.

KeyDerivation - By interacting with the server, this class performs the necessary logic to derive all of a friend's keys when granted with the key of the highest level for which the user is permitted access. It does this by iteratively decrypting each key with the key before until all keys for all levels this user can access have been acquired. All of the keys are then stored in the KeyStore.

ServerComms - The class which allows us to interact with the server. The class stores the port of the server and carries the streams necessary to communicate with it.

DropboxOperations - When a user logs in the class is activated with a username and the Dropbox specific key and secret pair which are used to create a Dropbox *Session*. This class uses the *Session* to perform Dropbox related functions. These include uploading, downloading and removing files and scanning the folder for files. This class is also queried during login to provide a list of files currently in the related Dropbox folders.

4.1.2 Server

ServerThread - Creates a new each time our application opens. This allows for multiple users to communicate with our server at once. Once a thread has been created it is passed to *Protocol*.

Protocol - The first class which interacts the user. It blocks until a log in or registration attempt is made. *Protocol* uses the *UserOperations* class to retrieve and store password hashes and user data. Once log in or registration has been successful it passes the user on to the *CentralAuthority*.

ServerCentralAuthority - This class mirrors the *Actions* class in client. The class blocks until it receives a decision on which action the client wishes to perform. The class then communicates with the client for information and calls functions necessary to complete the request.

UserOperations - Acts as an interface between the user logic and the table in the database containing user information. It stores and retrieves information relating to the user, such as usernames, password hashes and friends lists.

ServerDropboxOperations - This class performs operations on a user's Dropbox which would be too unwieldy to perform client side. These include transferring files from one user to another and removing files when a user is revoked. These are difficult to perform on the client side as the actions are invoked by users which are not online at the time. Performing them safely on the client side would involve using unnecessary database storage and queries.

4.2 Dropbox tampering

Choosing to use Dropbox as the method to share files does not come free of implications. While these issues have little security concern, they can impact on the functionality of our system. Here we shall discuss what they are and how we addressed them.

One potential issue is the fact that it is entirely possible for a user to manipulate the folder in which the files for this application is stored outside of our application. The manipulation that can occur can be divided into two categories. That is, manipulation of the folder containing their own uploaded

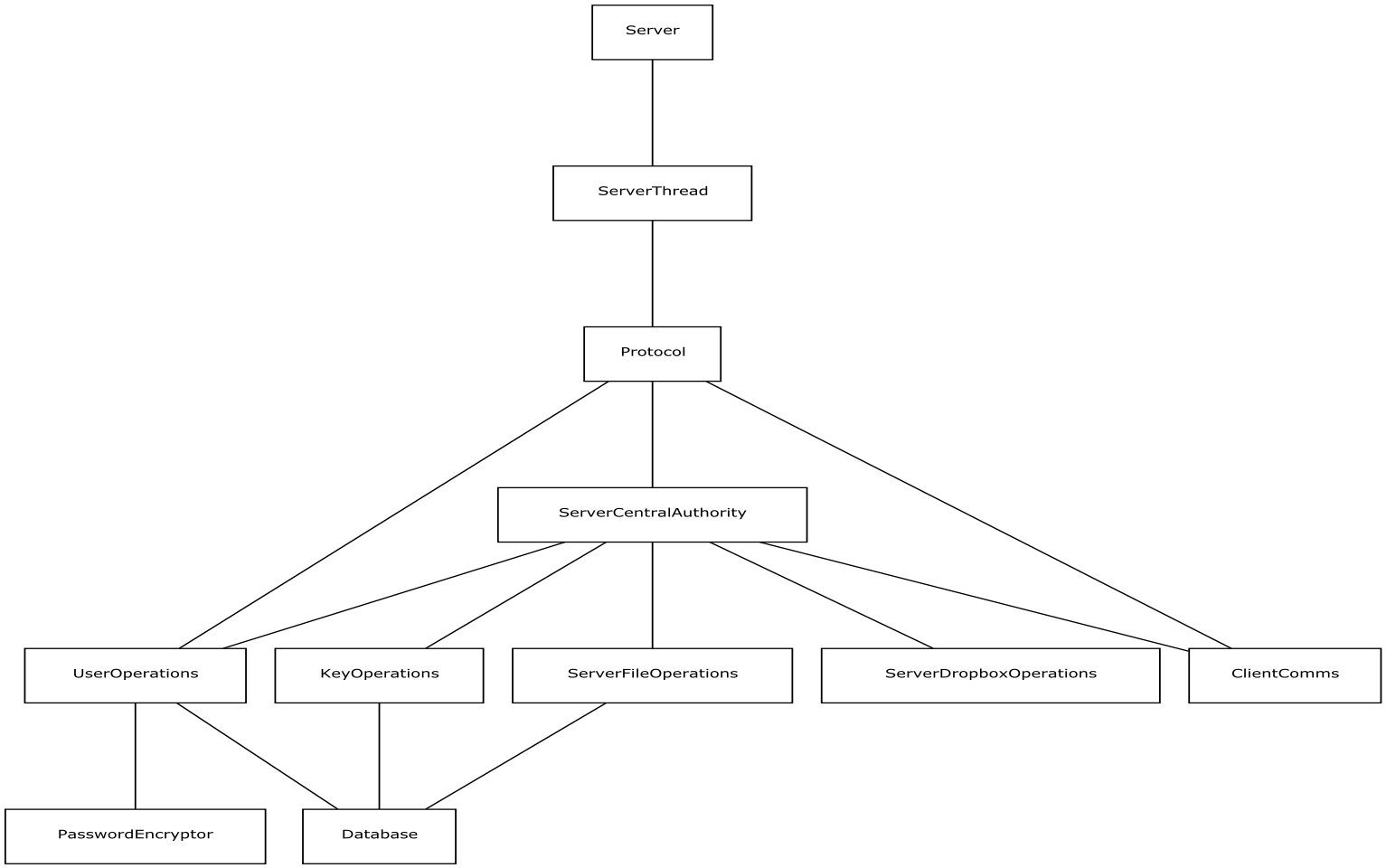


Figure 8: Reduced class diagram for the server

files and manipulation of a folder containing files that their friend has shared with them. We shall address the former first. The potential manipulations that can be performed are:

- Add a file to the folder containing their own uploads
- Remove a file in the folder containing their own uploads
- Edit a file in the folder containing their own uploads

For the manipulation in which a file is added to the folder containing a user's own uploads, the query to the server for file information will result in nothing being returned. We have options in how to deal with this. We could alert the user of the change and request information in order to handle the file, process the file as normal and treat it as regular upload. We could also simply ignore it. For now the file is simply ignored.

Removing a file from the user's own upload folder can be addressed in one of two ways, if the file has been shared we could replace the file in the user's uploads by copying it from one of their friends and then continue as normal. An alternative is to treat it as regular deletion and remove the file information from the database and remove it where it has been shared. Lastly, we could simply ignore it, however this results in the user losing control of the file and the friends with which they have shared it with can keep it and update it for as long as they like. For simplicity reasons, the file is ignored at this time.

Manipulations in which a file is edited can be considered as a removal (of the file unedited) and an addition (of the newly edited file) and can be treated as such.

Here we shall discuss some potential issues involved in the tampering of folders containing files which have been shared with the user. These issues are:

- Create a folder for a user of which they have access at no level.
- Add a file to a folder containing a friend's uploads
- Remove a file in a folder containing a friend's uploads
- Edit a file in a folder containing a friend's uploads

Creating a folder for a user of which they have access at no level can only have the reaction of being ignored.

When a user adds a file to a folder containing a friend's uploads, our only safe option is to ignore it. To do this we must contact the server to verify the integrity of each file. Whether the file added is an encrypted file they do not have access to is not significant as they do not possess the key to decrypt it and the server can not be tricked into providing it as it does not possess any client keys.

Removing a file from a folder containing a friend's uploads can be ignored, meaning that they will no longer have access to the file, at least until someone updates it. Or we could simply replace the file by copying the owner's current version of the file. We ignore the removal of the file for simplicity reasons.

5 Benchmarking

In this section we shall evaluate the speed at which our solution can perform certain tasks. These tasks include: log in, file upload, file download, sending a friend request, accepting a friend request and revoking a user. In order to fully evaluate our solution we will also perform, and measure, the same actions using Dropbox without any of the cryptography or logic. By doing this we will be able to see how our work stacks up to a 'vanilla' file sharing solution, as well as to help us to investigate where our solution's bottleneck is.

As shown in the section in which we compare the difference in performance between different implementations of the AES algorithms, the time it takes to complete varies greatly on the size of the files we are encrypting. Thus, similar to that section, we shall run tests on files of different sizes. Where possible we shall also use the same files as in that section in order to determine how much of the running time is spent on logic/Dropbox and how much on cryptography.

The benchmarks shall all be measured on the client-side. In the real world it matters how much stress the server is under due to cost and ability to serve all clients. However for the sake of this project we shall concentrate on the user experience. The cryptography, and therefore the vast majority of the computational workload that we have added, is done on the client side and so ignoring the server's workload should not matter so much.

There are occasions in our solution where there is a need to generate parameters for cryptography, such as keys and initialisation vectors. To do this randomly the algorithms need to collect a lot of *entropy*. To achieve this, most secure PSRNG collect values from the computer they are running on, such as the temperature of the CPU, mouse movements and keyboard strokes. Because of this these algorithms are able to run faster when the user is actively using the computer. For this reason, when we are testing an action which requires this, we shall show the time the actions takes at different levels of user activity.

5.1 Uploading a file

These tests for uploading a file are run from the beginning of the *Controller* for this action until it returns. This means we are ignoring the time taken for the GUI listener to detect the request and the time taken for the controller

and GUI to interact and update the display. However this should be minimal and have little effect on the results.

We can see the time taken for our solution to upload different files in Table 4. It takes 4 seconds on average to upload a 1.3MB .pdf. This time includes the encryption of the file before it is uploaded. As we saw when looking for an AES provider, it took an average of 0.011 seconds to encrypt the same file using AES. It is clear that the cryptography has little to no effect on the time this action takes to complete. The other substantial stages of uploading a file using our solution is the communication between the client and server and uploading files to Dropbox.

As we can be seen from Table **??**, uploading the files to Dropbox represents a considerable chunk of time. For instance, on average for the 1.3MB .pdf, uploading the file to Dropbox takes on average 96% of the total time our solution requires. Therefore our solution adds an overage overhead of only 4% for the uploading of this file. However an interesting point is that the average overhead of our solution for the 203.2 kB mp3 is 35%. This is likely due to the uploading of the file being incredibly quick for such a small file, meaning the reasonably small amount of time the cryptography and client/server communication takes has a greater impact than it would otherwise. The tests on other files support this argument. We can see this visually in Figure **??**. For very small files the overhead is fairly large but as the size of the file increases the overhead caused by our solution descends into almost nothing.

The type (extension) of the file we are uploading appears to make little difference in the time it takes to upload. In the pictorial representation of the tests for our solution, Figure **??**, the time taken to upload the 203.2 kB mp3, 1.3MB .pdf, 5.2 MB .jpg and the 10.5 MB .zip, appears to grow linearly with the size of the file, regardless of the type. Comparing the rate at which our solution grows in the time taken to upload files as the size of the file increases, Figure **??**, and the rate which uploading files to Dropbox alone grows, Figure **??**, we can see they are almost exactly the same. From this we can determine that the increase in time as the file increases is mainly caused by Dropbox.

One particularly noticeable result is that the 832.6 MB .iso took longer on average to upload to Dropbox alone than it did using our solution. There is no reason why this would be true and so we shall put it down to the inconsistency of the Dropbox API, and networks in general, for large files.

File	Iterations	Time taken (seconds)	Average
203.2 kB .mp3	10	14.077261255	1.41
1.3MB .pdf	10	38.007998196	3.8
5.2 MB .jpg	10	51.347427923	5.13
10.5 MB .zip	10	79.577470975	7.96
90.9 MB exe	10	1852.5543543	185.32
209.7 MB .zip	10	3483.7958252	348.38
832.6 MB .iso	10	21641.247924	2164.1

Table 5: A table holding the time taken to upload different files using our solution. The *File* column is the files uploaded, *Iterations* is the number of times we upload it, *Time taken* is the length of time taken to upload the file 10 times and *Average* is the average amount of time taken to upload the file once. *Average* is the average amount of time taken to download the file once. Note that the times can vary substantially on the network they were tested on and the load on the Dropbox servers at that time.

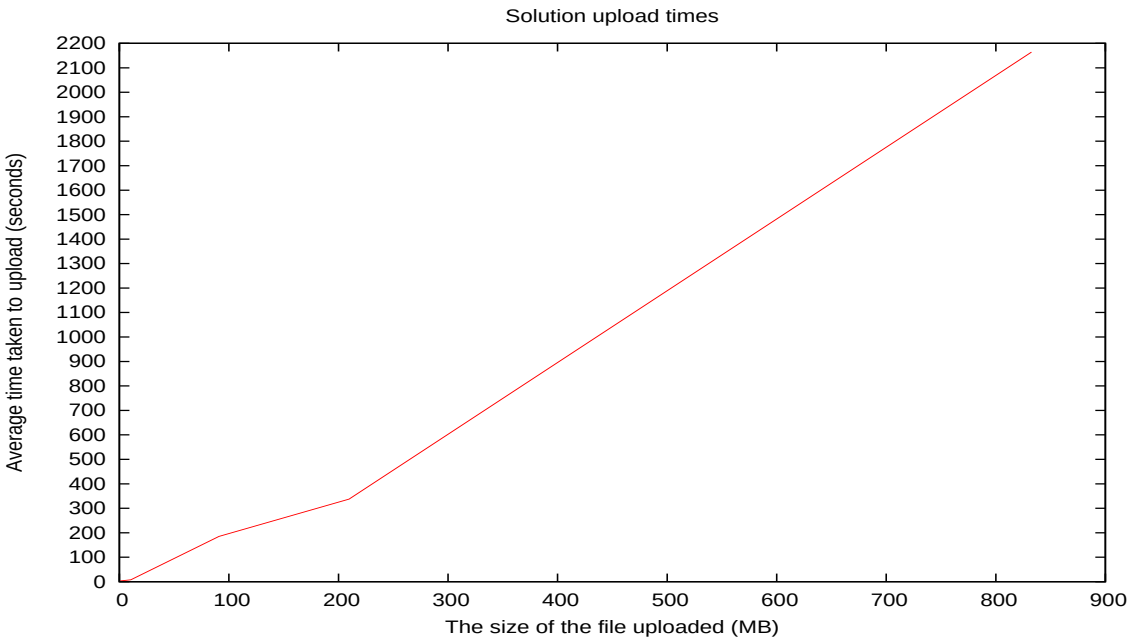


Figure 9: A line graph plotting the time the average time it takes to upload a file to Dropbox via our solution for files of varying size.

File	Iterations	Time taken (seconds)	Average
203.2 kB .mp3	10	9.237756953	0.92
1.3MB .pdf	10	36.34874983	3.63
5.2 MB .jpg	10	42.918766407	4.59
10.5 MB .zip	10	70.938518268	7.09
90.9 MB exe	10	1775.2136753	177.52
209.7 MB .zip	10	3375.2390153	337.52
832.6 MB .iso	10	24259.387692	2425.94

Table 6: A table holding the time taken to read and upload different files to Dropbox directly, without encrypting, communicating with the server or using any logic. The *File* column is the files uploaded, *Iterations* is the number of times we upload it, *Time taken* is the length of time taken to upload the file 10 times and *Average* is the average amount of time taken to upload the file once.
Note that the times can vary substantially on the network they were tested on and the load on the Dropbox servers at that time.

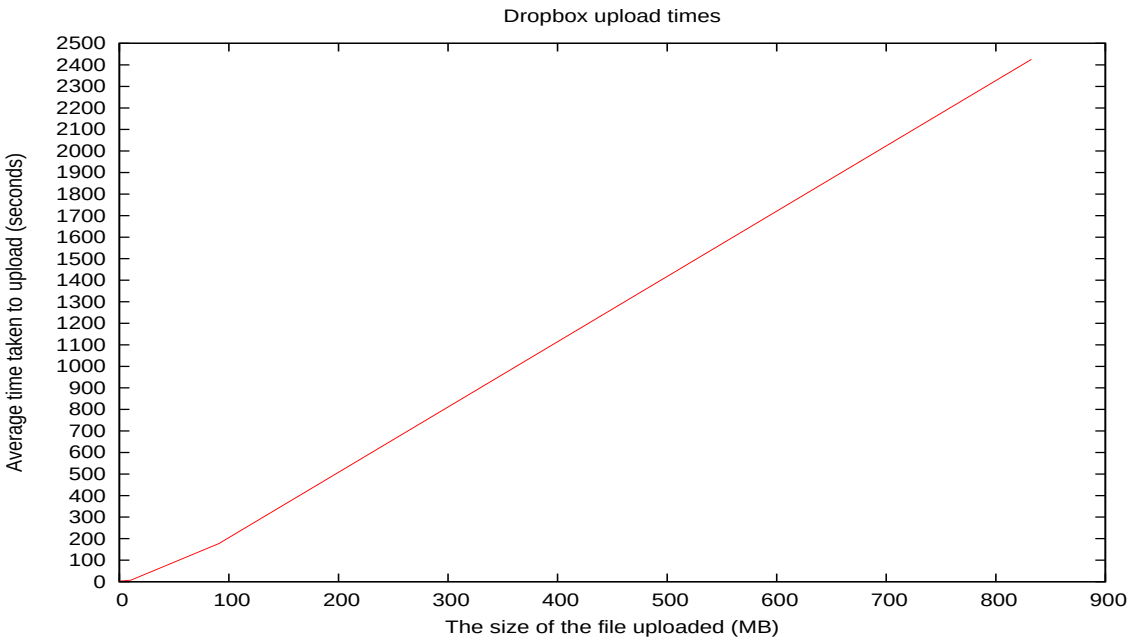


Figure 10: A line graph plotting the time the average time it takes to upload a file to Dropbox directly via the SDK for files of varying size.

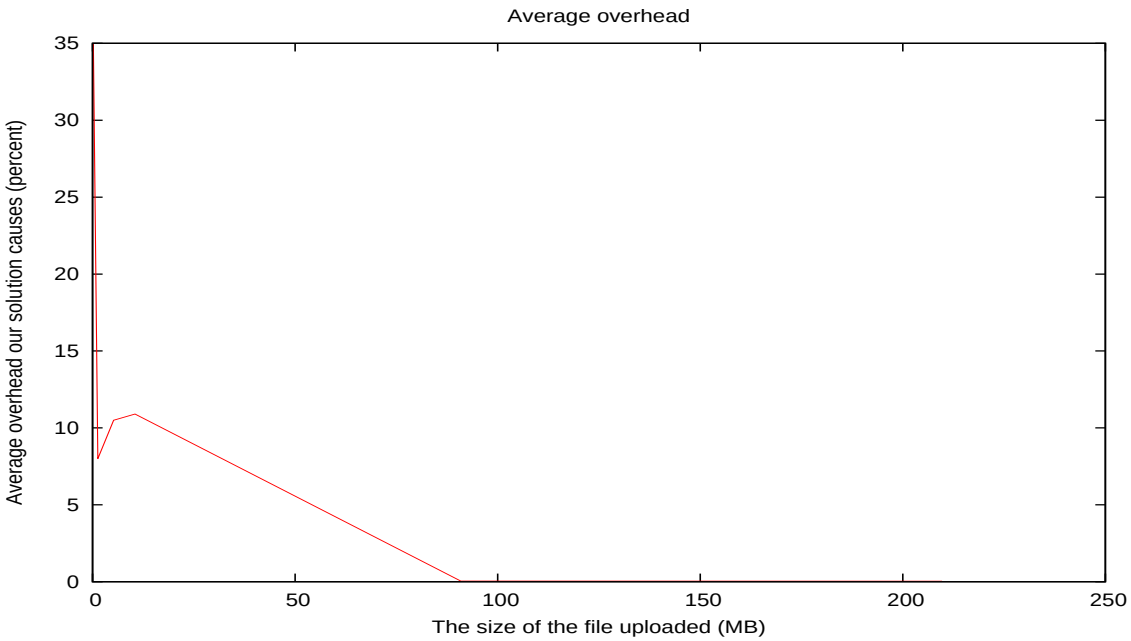


Figure 11: A line graph plotting the time the average overhead caused by our solution when uploading a file to Dropbox using our solution. The graph plots the percentage of overhead when compared to uploading to Dropbox directly for files of varying size.

5.2 Downloading a file

We conduct our tests for downloading files in the same way that we did for uploading files. That is, through the *Controller* and not the GUI.

We can see the results from the test for downloading files using our solution in Table 5 and our results for downloading files straight from Dropbox in Table **??**. The first thing we might notice is that there is slightly faster performance for downloading when compared to uploading the same files in the previous tests. One possible explanation for this is that typically networks have a faster maximum download speed than upload speed. Another possible reason is that Dropbox may have to do more preparation when uploading files, such as generating unique identifiers and creating database entries.

Similarly to the previous test, there is an anomaly in our results for the 209.7 MB .zip, which downloaded quicker on average using our solution than by accessing Dropbox directly. As before, there is no reason for this be true and is further evidence that Dropbox has inconsistent performance when performing actions on large files.

We can see our download tests pictorially in Figure **??** for our solution and in Figure **??** for Dropbox alone. For files under 100 MB the rate of growth in both is almost identical however they differ for files above that size, mainly due to the anomaly described above. From this we can gather, like we did with uploading files, the rate of increase in downloading time is caused largely by Dropbox.

The overage overhead for our solution when downloading files can be seen in Figure **??**. From this we can see our results for downloading have been more inconsistent than when uploading. Because of this we can not accurately measure the overhead. However, ignoring the anomaly, we can still determine a general downward trend in overhead as the file size increases, similar to when uploading files.

File	**Iterations**	**Time taken (seconds)**	**Average**
203.2 kB .mp3	10	13.155384853	1.32
1.3 MB .pdf	10	30.473156395	3.05
5.2 MB .jpg	10	61.894616714	6.19
10.5 MB .zip	10	114.055083186	11.41
90.9 MB exe	10	1446.682795175	144.7
209.7 MB .zip	10	2068.31832507	306.83
832.6 MB .iso	10	8539.4238683	853.94

Table 7: A table holding the time taken to download files of increasing size using our solution. The *File* column is the file downloaded, *Iterations* is the number of times we download it, *Time taken* is the length of time taken to download the file 10 times and *Average* is the average amount of time taken to download the file once. d *Average* is the average amount of time taken to download the file once.
Note that the times can vary substantially on the network they were tested on and the load on the Dropbox servers at that time.

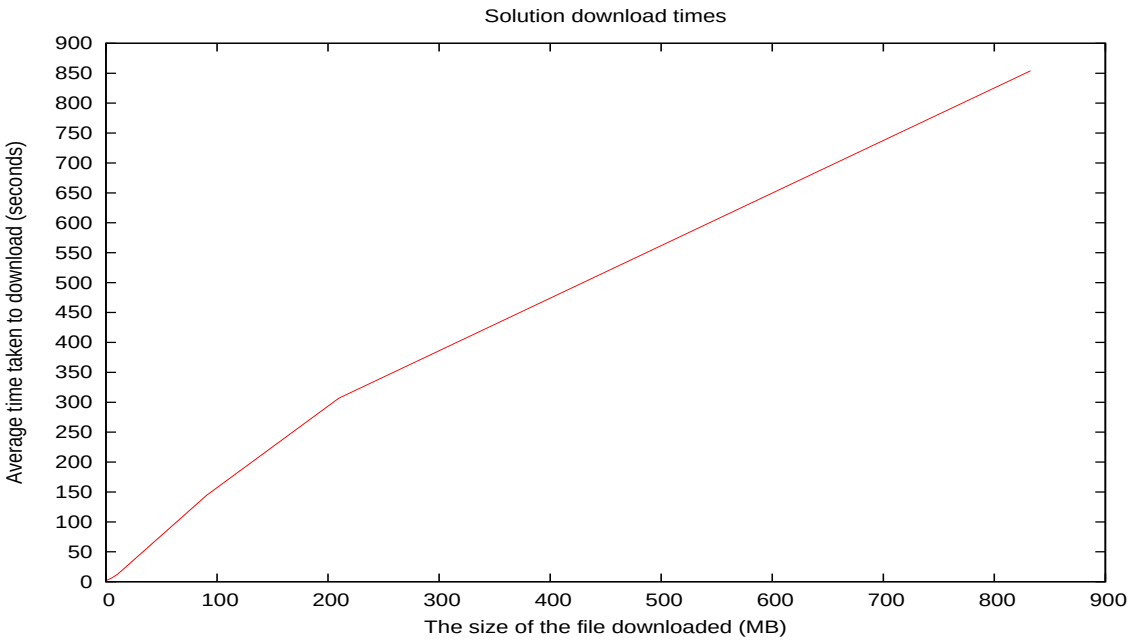


Figure 12: A line graph plotting the time the average time it takes to download a file to Dropbox via our solution for files of varying size.

File	Iterations	Time taken (seconds)	Average
203.2 kB .mp3	10	7.695544583	0.77
1.3MB .pdf	10	24.290205159	2.43
5.2 MB .jpg	10	58.47502913	5.85
10.5 MB .zip	10	94.740813236	9.47
90.9 MB exe	10	1309.980643	131
209.7 MB .zip	10	3375.23901536	377.52
832.6 MB .iso	10	7331.61753447	733.16

Table 8: A table holding the time taken to download files of increasing size from Dropbox directly, without encrypting, communicating with the server or using any logic. The *File* column is the file downloaded, *Iterations* is the number of times we download it, *Time taken* is the length of time taken to download the file 10 times and *Average* is the average amount of time taken to download the file once.
Note that the times can vary substantially on the network they were tested on and the load on the Dropbox servers at that time.

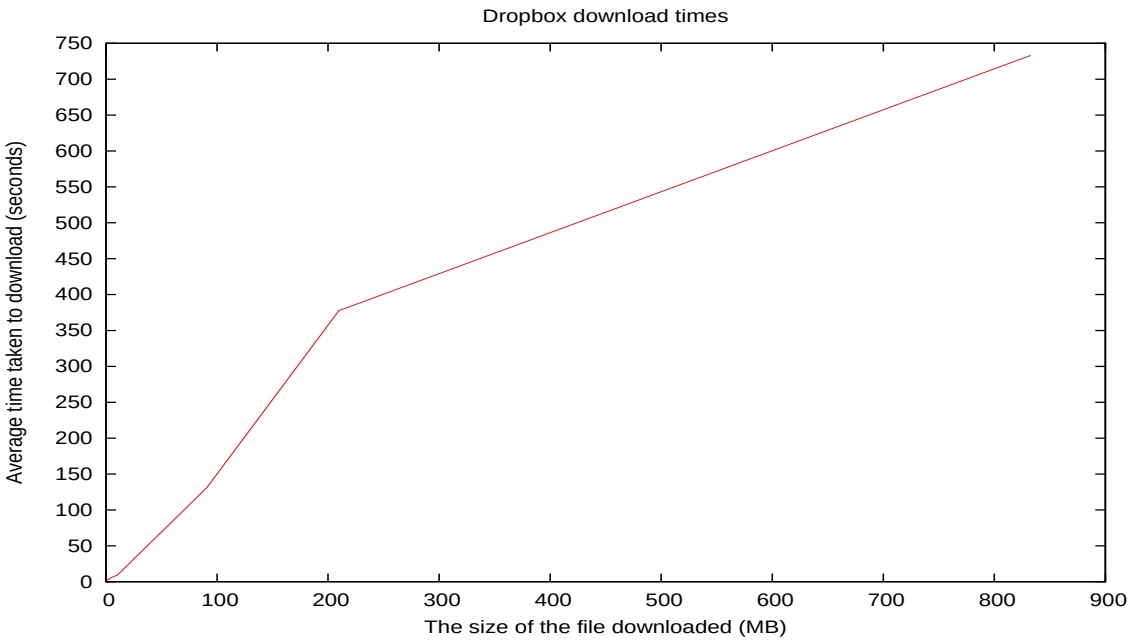


Figure 13: A line graph plotting the time the average time it takes to download a file to Dropbox directly via the SDK for files of varying size.

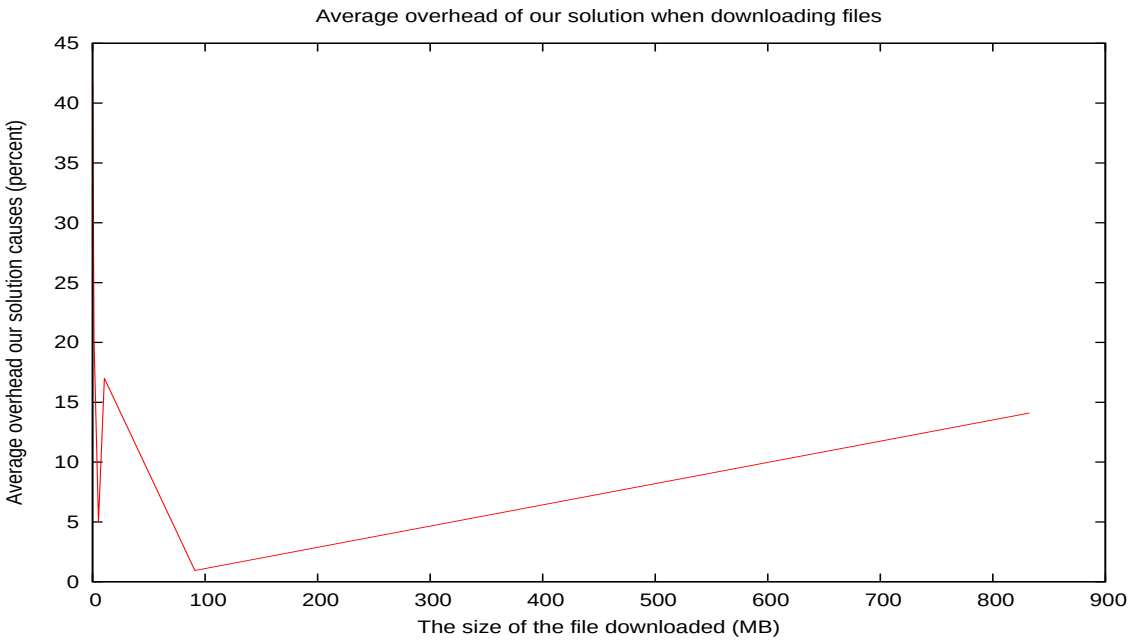


Figure 14

5.3 Log in

When a user logs in the client needs to contact the server to acquire information about its own files and the files friends have shared with them. The information it needs to collect are, the file's security level (for all files) and the file's owner (for files belonging to a friend). Due to this, the time it takes a user to log in will likely correlate with the number of files they currently have access to. Therefore in order to get a good idea of how long this takes we shall measure the times it takes to log in when the user has access to a varying number of files. The results for this can be seen in Table **??**.

There is clearly a significant increase in the time it takes to log in as the number of files the user has access to increases. In Figure **??** we can see a visual representation of the rate of increase. As we can see from the graph, the rate of increase is almost linear. For a small number of files this is not much of a problem, but as the number of files the user has access to grows

large the wait may become rather tedious.

Our results are not unusual; applications which pull information from a server, e.g. Skype, typically have longer log in times than other applications.

Number of files	**Iterations**	**Time taken (seconds)**	**Average**
0	10	28.717884417	2.87
5	10	37.766417567	3.78
10	10	41.748590981	4.17
25	10	62.00080726	6.20
50	10	118.726741348	11.87
100	10	197.461744273	19.46

Table 9: A table holding the time taken to log in using our solution with the user having access to a number of different files. The files used in these tests are duplicates of a 1.3 MB .pdf file. The *Number of files* column is the number of files the user has access to at log in, *Iterations* is the number of times we log in, *Time taken* is the length of time taken to log in 10 times and *Average* is the average amount of time taken to log in once.

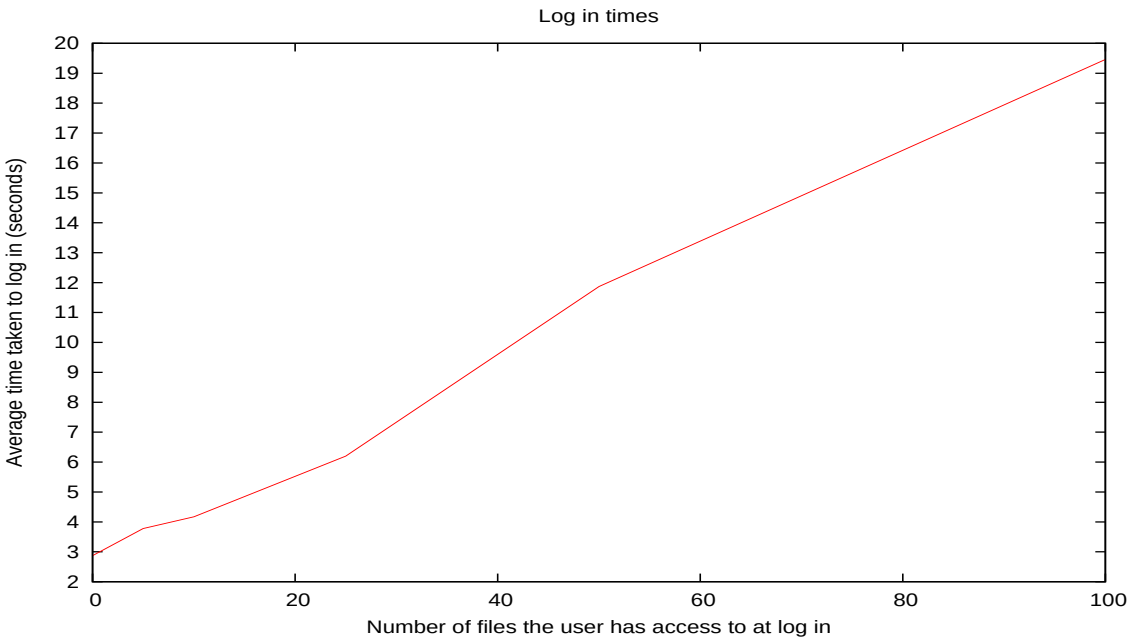


Figure 15: A line graph plotting the time the average time it takes to log in when a user has access to a different number of files.

5.4 Registration

When a user registers they are required to manually log in to Dropbox and approve our application through a web browser. The time it takes to do this can vary massively between different users and can not be accurately measured. For this reason that part of the registration process is omitted from our measurements.

Registration is the part of our solution which requires the most generation of cryptography values and client/server communication. For this reason we can expect registration to be the most time consuming process in our system. This is not unusual and can be seen in similar applications, such as Mega.[2]

During registration a lot of cryptography values need to be generated.

[2]An image of this happening can be seen at: `http://images.hacktabs.com//2013/01/Mega-generating-key.png`

For this reason registration will likely have the most variation in the time it takes to complete by the amount of entropy our algorithms are able to collect while they are running. Because of this we shall time the registration takes to complete with varying levels of user activity. This can be seen in Table **??**. As we can see registration can vary massively, from 9 seconds with a lot of activity to nearly 2 minutes with no activity. This is as we predicted. Fortunately it is unlikely that our users will not move the mouse at all while they are registering, so the time it takes is likely to be closest to our *casual browsing and typing* result of 15.22 seconds.

Entropy help	**Iterations**	**Time taken (seconds)**	**Average**
No mouse or keyboard movement	10	1145.34437768	114.53
Casual browsing and typing	10	152.188442492	15.22
Wild mouse movement	10	92.557535694	9.26

Table 10: The time taken to register using our solution with varying levels of user activity. The *Entropy help* column refers to the level of activity the user partakes in while registration is being carried out by our solution. *Iterations* is the number of the times the test was conducted. *Time taken* is the total amount of time all iterations take to complete. *Average* is the average time taken for one iteration to complete.
Note that in these tests the entropy pool was unlikely to be full before registration. If the user has a full entropy pool (i.e. they have not recently used any applications which require secure random number generation), before they register, the amount of activity they partake in will likely not have such a big difference in the time registration takes.

5.5 Overhead evaluation

Having measured the times for each action, we shall now evaluate how much overhead our solution results in. We can see the results in Table **??**. The reason for the larger overhead for downloading small files as opposed to uploading is due to Dropbox performing the task quicker for downloading as we have seen previously that, in general, as Dropbox takes longer to perform its task the overhead grows smaller. While the percentages may seem large,

nearly 25% for downloading, the reality is that the overhead is only between half a second and a second for small files.

Having measured all of the actions a user can perform, Dropbox is most definitely the bottle neck in the system. That said, it is clear that our solution does comes with a measurable overhead when compared to Dropbox alone. Uploading and downloading can come with a significant overhead when the files are small, but fortunately the time they take is incredibly small so the overhead is unlikely to be noticeable by the user. However we have seen log in and registration can take a long time under certain circumstances. Generally these are likely to be rare actions for the vast majority of users and should not impact on the experience a great deal.

Action	**Average overhead**
Uploading a file	15.17%
Downloading a file	21.12%

Table 11: The average overhead our solution results in when compared to Dropbox alone. These have only been measured for the file sizes which displayed a consistency, those are: 203.2 kB .mp3, 1.3MB .pdf, 5.2 MB and the 10.5 MB .zip. The larger files have not been measured due to the inconsistency in their performance we observed. It was felt that we would learn very little by including them.
To obtain these results, the overhead for each one has been measured, summed and then divided by the total number of files measured.

6 Evaluation

6.1 User stories

In this section we shall attempt to evaluate how useful our solution is when applied to an 'everyday' environment. There are two possible environments we can test: social and work. To do this we will allow members of the public to use our solution. In order to evaluate our work as accurately as possible we shall attempt to conduct our tests on an audience with a large variety of technical ability.

Each test involves a single user and is divided into two sections: firstly at the beginning of each test we shall conduct a 'cold' run, that is with little help the test subject will be allowed to use the solution as they see fit. Secondly we shall guide the test subject through the solution, telling them exactly what to do where necessary.

Before the beginning of a test the subject will be briefed on what our solution is trying to achieve. There will be little talk of the implementation and the brief will be much like the introduction to this paper. The subject will be given a list of people (with existing accounts) who they are able to, at their pleasure, ignore them, send friend requests to them and accept friend requests from them.
The list of existing users to give to a test subject to use in a social environment is:

- Father
- Mother
- Aunt
- Boss
- Best friend
- Partner
- Friend
- Close work colleague
- Distant work colleague

- Family friend

The list of existing users to give to a test subject to use in a work environment is:

- Supervisor
- Colleague
- Trainee
- Intern
- Chairman
- Project manager
- CEO

Due to the difficulty of describing a generic list for a work environment the test subject will be free to define their own if ours is not suitable.
Once we have described the results of each test we will discuss our findings and what they mean in relation to our work.

6.1.1 Test 1

Technical ability: Advanced
Use: Social

Cold run
The user found login and registration easy and gave little indication that it was taking too long. The user seemed to understand the concept very well and once given the list of people he was able to rank them, but took a while doing so. The user complained that there did not exist a perfect hierarchy, mainly because he felt his friends did not have an overlap of interest in the files he would share. The particular example the user gave was that his mother would not be interested in the files he shares with his father, and vice versa. The user suggested that he would be better able to create a ranking for his friends if he was able to create more than one account and have multiple hierarchies. While ranking his friends the user referred to the security levels by names he designated rather than the numbers which are

assigned by us. The user 'saved' the highest security level for himself, i.e. would permit not any friends to access files at that level. This was surprising to me, and was not a possibility I had considered.

Assisted run This user required little assistance. Direction was needed to locate files he had download onto his computer as he felt the GUI did not make the location clear.

6.1.2 Test 2

Technical ability: Basic
Use: Social

Cold run
Once the application had started the user had no trouble navigating the interface to register her account. Given a list of people she may choose to add she had some trouble assigning them a level of security. After some time she had written some group names and placed them in there, ordering groups of people (such as family/friends) in order, rather than individuals. The user felt that she had to use all security levels by having at least one user in each one. This lead to her coming up with unnatural, or forced, ideas on what each level may represent.

The user found uploading and downloading her own files simple but indicated she was unaware of where the files were downloaded to (a GUI issue.)

Assisted run
After an explanation of how the security levels may be used, the user was better able to formulate an environment which she found more intuitive. Her main improvement was not feeling the need to use all of the security levels. Despite this she still had difficulty organising her friends into a hierarchy she was happy with. With a sub-optimal ranking and faced with the tedious task of revoking and re-adding her friends, she eventually gave up.

6.1.3 Test 3

Technical ability: Average
Use: Work

Cold run
Registration and log in was, again, found to be a simple task. The test subject did not find the allocation of security levels as difficult as the previous two test subject. He felt that the hierarchy requirements conformed to the structure of his business environment. The user often used each security level to represent a branch in his business. The subject suggested that he would like to separate hierarchies to represent branches in which he worked where there was no overlap of files shared or users to share them to (i.e. have more than list of security levels to work with.)
Assisted run
The test subject required little assistance. At times the user required small confirmations about what was expected of him, such as what tasks to complete and how. The user required small explanations on how the GUI functioned (such as where downloaded files were placed.)

6.1.4 What we have learnt

From the tests it is quite clear that the GUI is a limiting factor for user experience. This is to be expected. Our solution was not intended as an out-of-the-box commercial product and so little attention was paid to the user interface. The fact this was the largest complaint is a positive sign.

All test subjects who used our solution struggled to place an intuitive meaning on the security levels. They were not fond of the current labels (1..5) as there was little connection between, for instance, the number 3 and a particular friend. This was a fair complaint. In order to solve this problem we may allow for users to change the name of their security levels. For instance, we could allow a mapping between the security level 3 and the label "family friends" or "interns" This would place a strong intuitive link between the security levels and their friends.

Not all of the test subjects found the premise of our solution to be completely intuitive without explanation but were able to understand what they needed to do after an explanation was given. This is more a failure of the GUI than anything else. A simple solution to this problem would be to have a 'help' or a form of introductory page which displays after registration. The purpose of this page would be to introduce the user to our solution and detail what they need to do to maximise their experience.

In these tests our solution's most promising application was in the work environment. The structure of a business tends to match our design and our

test subject found that to be true in their case. The test subjects who used our solution in a social environment found that their friends do not fit naturally into a strict hierarchy. Instead having distinct groups of friends which have little relationship with each other. From these tests it seems likely that our solution may have limited success in this environment.

We must note that we can not take these tests as a conclusive indicator of the usefulness of our solution. Due to the constraints on time, our sample space is incredibly small and we can not confidently extrapolate that to the general public, or even a specific group of people. There is also a clear limitation on our testing environment, it can not be said that our test environment is representative of any real-life scenario. For instance, in no situation would a user have an observer making notes on what they doing with our solution.

6.2 Security analysis

In this section we shall discuss possible attacks on our solution, how likely they are and how difficult they could be to perform.

In our solution, an attack's end goal is almost always likely to gain access to files they would otherwise not be permitted to access.

An overview of a number of possible attacks to gain access to files can viewed in Figure **??**.

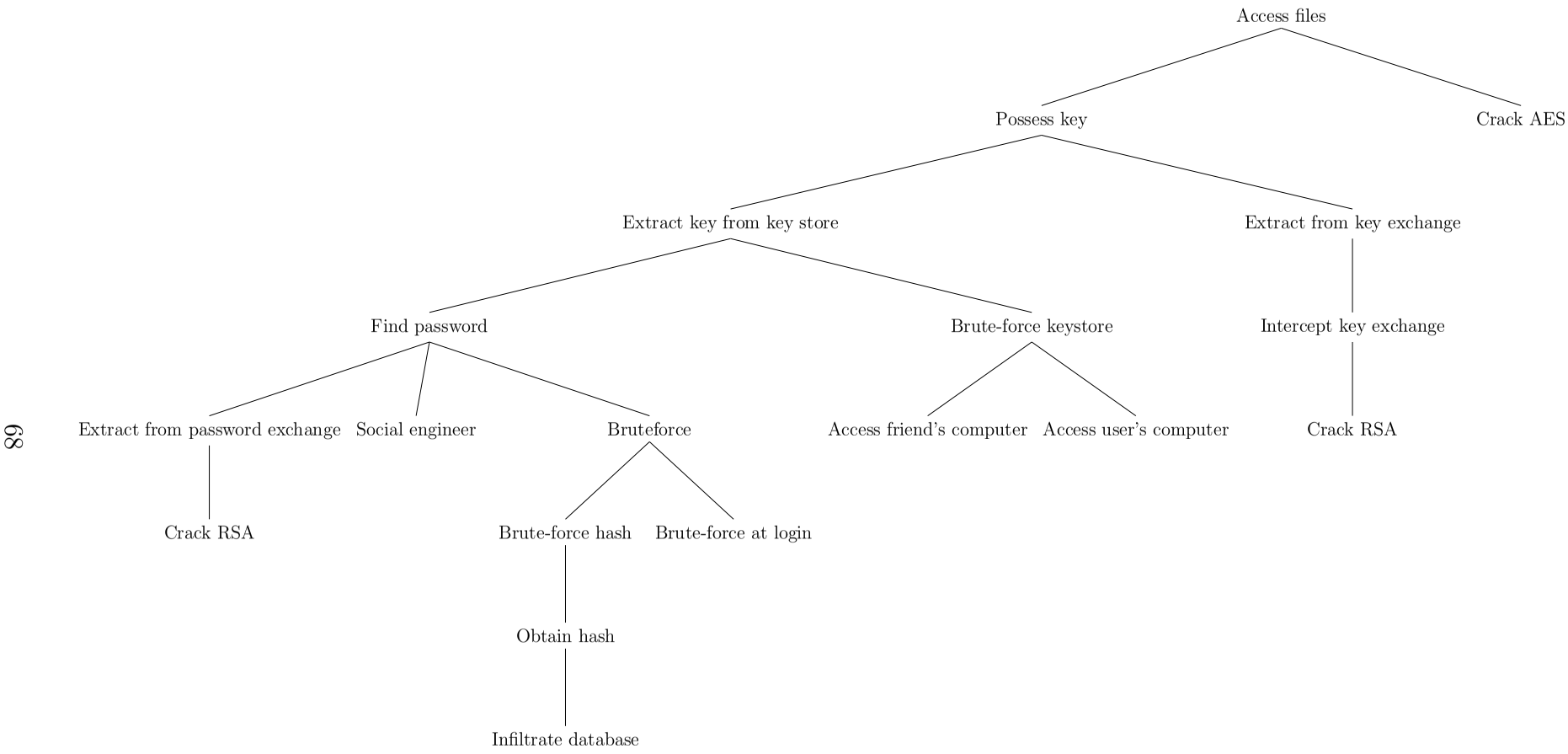


Figure 16: An attack tree representing all possible attacks to gain access to files in our solution. Each branch represents a possible route of attack. A node represents a goal for the attacker and its children are the possible ways in which in they can achieve that goal.

Cracking AES encryption Achieving this would give the attacker access to any file they may want. Assuming the encryption has been applied correctly, this is virtually impossible.

Intercepting key exchange An attacker who does this will possess a key encrypted with another user's public key. To access the key they will need to crack RSA.

Cracking RSA encryption Achieving this would give the attacker access to whatever access the friend has been given. Assuming the encryption has been applied correctly, this is virtually impossible.

Intercepting log in / register passwords By doing this the attacker will possess a password encrypted with the server's public key. Similarly to intercepting key exchanges, accessing the password would require cracking RSA encryption.

Cracking keystore Achieving this gives the attacker access to all of the keys created by the owner of the keystore and all keys shared with them. Unfortunately keystores are quite weak. A number of 'clever' brute-force solutions exists for Java Keystore, such as KeystoreBrute.[**?**] An estimation on the time taken to brute-force a keystore with varying password strengths using KeystoreBrute can be seen in Table **??**. As can be seen, a weak password can be cracked in almost no time at all, but bruteforcing quickly becomes infeasible for any password stronger than an upper/lower alphabet password 8 characters long. However bruteforcing a password of any significant strength requires prolonged access to the user's computer, meaning the user likely has greater problems than the integrity of their keystore.

Brute-force passwords Achieving this would give the attacker access to the user's account, therefore their keystore and all of the files they have uploaded as well as all of the files that have been shared with them. Ignoring social engineering of the passwords, this can be done in one of two ways. Compromising the database and acquiring the password hashes and proceeding to brute force the passwords, or bruteforcing the passwords via

Password format	Number of characters	Number of possible combinations	Time to brute-force
Lower case alphabet	4	26^4	0.3 seconds
Upper/lower alphabet	4	52^4	4.9 seconds
Upper/lower alphanumeric	4	62^4	9.8 seconds
Upper/lower alphanumeric and symbols	4	95^4	54 seconds
Lower case alphabet	8	26^8	1.61 days
Upper/lower alphabet	8	52^8	1.13 years
Upper/lower alphanumeric	8	62^8	4.6 years
Upper/lower alphanumeric and symbols	8	95^8	140 years

Table 12: The time it will take an attacker to brute-force a keystore using KeystoreBrute. We assume the attacker can attempt 1500000 passwords per second (10× the number recorded by the creator who used a Core2Quad CPU.)
Note: Any real attacker will likely have more processing power than we have described here.

the GUI log in form. For both of these cases the time it is likely to take the attacker to bruteforce the hashes varies on the length of a user's password and whether the user's password consists of dictionary words. We are able to prevent brute forcing at the log in form by simply blocking log in attempts for a username after n consecutive failed log in attempts. Brute-forcing the password once the hash has been acquired can not be prevented. Whether the attacker can find a matching password for the hash in a feasible amount of time largely depends on the user's password. An estimation on the time it will take for different non-dictionary passwords can be found in Table **??**. While this is a crude estimation, it can be seen that, for a longer password the attacker will find it very difficult to find a match, while a shorter password can be brute-forced in a matter of hours or days. For dictionary based passwords, or the hardware accelerated attacks as described in the bcrypt analysis, the time taken to brute-force the passwords will be much lower.

Password format	Number of characters	Number of possible combinations	Time to brute-force
Lower case alphabet	4	26^4	1 hour 6 minutes
Upper/lower alphabet	4	52^4	17 hours 49 minutes
Upper/lower alphanumeric	4	62^4	36 hours
Upper/lower alphanumeric and symbols	4	95^4	8 days 6 hours
Lower case alphabet	8	26^8	58 years
Upper/lower alphabet	8	52^8	14867 years
Upper/lower alphanumeric	8	62^8	60719 years
Upper/lower alphanumeric and symbols	8	95^8	1.8 million years

Table 13: The time it will take an attacker to brute-force a matching plaintext password for a hash. We measure this for passwords of varying strength. We assume that an attacker has 10× the amount of processing power than we do, meaning they can compute a hash at an average of 8.77 milliseconds. Note: Any real attacker will likely have more processing power than we have described here.

Database security

Database compromising Almost all information in the database is considered public. In our solution the only sensitive information, aside from the password hashes, is the key/secret pair used for the server to access a user's Dropbox. As all files in a user's Dropbox is encrypted there is little chance of using this information to access encrypted files, but it would be more of an annoyance, e.g. a malicious user adding or removing files.

SQL injections We allow for user input for the username and password during log in and registration which makes us vulnerable to SQL injection attacks. An SQL injection attack is when a user inserts malicious SQL commands in a field which will then be executed in our database. However we use the Java PreparedStatement[**?**], which automatically escapes SQL characters[**?**]. Meaning that injections will fail.

Unauthorised login We do not wish for attackers to be able to access our database even if they are able to gain access to the server. For this reason we have set a username and password to access the database. This username and password is required to be entered before any actions can be performed. It is a concern that they may be able to brute-force the password, similar to the way in which the keystore can be brute-forced. H2Database prevents against such attacks by setting a 250 millisecond delay after a failed log in attempt before log in can be attempted again. This delay proceeds to double with each log in attempt thereafter, reaching a maximum 4 second delay.[**?**]. Providing we use a password of significant length, the attacker should be unable to gain access using brute-force in any reasonable length of time.

Security evaluation

Bruce Schneier said in *Secrets & Lies*, "Security is a chain; it's only as secure as the weakest link."[**?**] For this reason we shall ignore the parts of our system that are clearly robust, such as the cryptography, and base the overall security of our system on the weakest parts.

From our analysis for the possible routes of attack, it appears that the user's password is the weakest link in our solution. While we can not control how safely a user keeps their password, we are able to exert control on the strength of password they choose. Enforcing upper/lower alphanumeric passwords of at least 8 characters in length would give them significant security from brute-force attacks should their hashes become compromised.

The security of the keystore is weaker than the security than the password hash provides, however the security gained by the keystore's location on the user's machine means that it is likely a more difficult route of attack than acquiring hashes and brute-forcing them.

7 Conclusion

Throughout this project we have looked at the background of cryptography, access control and how they can be brought together to create our hierarchy, we looked at how we can use these to create our system and how we can solve potential issues in the *Design* section. From there we described the structure of our work and went on to evaluate it in a number of ways. We benchmarked our system against Dropbox, looked at how potential users found using our system and what markets our work may perform well in, before finally looking at how secure our system is.

This project aims to increase the automation in file sharing with different groups of friends who require different access control constraints. Our solution achieves this by using cryptography. This is done without troubling the users with technicalities. A user need not know what a 'key' is, or how to derive and use them to access files. Most actions are performed in the GUI by selecting desired files and clicking the button for the desired action.

We wanted our solution to be quick enough to be usable. From the *Benchmarking* section we found that our solution comes with an overhead, as to be expected. Fortunately the most time consuming parts of our solution, registration and log in, only need be performed rarely. The most common actions a user is likely to perform, uploading and downloading files, bear little overhead, meaning our solution is a realistic one when it comes to performance. A performance which was in-line with Dropbox's performance.

We wanted our solution to be secure, that is not to introduce any unnecessary threats to user. From the *Security analysis* section we found no clear security holes in our work. With any security threats likely coming from 'traditional' attacks.

However our solution does not come without constraints. It requires the user to order the people they wish to share their files with in a fairly strict hierarchy. The degree to which most people can adopt this into their lives is yet to be seen and thorough case studies, or a release, would have to be done before we can know for sure. From the *User stories* section we can see that there may be some people this suits well, while others may find the lack of flexibility frustrating. The division of these two sets of people is likely to be in the environment in which they intend to use our solution. It appears that our solution may be very useful in a business environment but may struggle to make an impact on a user who intends to use it in a social environment. Due to this we can see that our work may have a place alongside other file

sharing software, like general file sharers, such as Dropbox, and collaboration software, such as Google Drive. Our potential users are organisations which have an inherently hierarchical structure, who may well find our solution a more efficient option than applications such as YouSendIt and Box.

7.1 What I would have done differently

I would not have chosen to use Dropbox as the storage for files. While it is not stated explicitly, after using the Dropbox API for some time it is clear that the API is not designed to be used in such an intensive way. The performance is very inconsistent and most actions, such as searching folders for files, are quite slow.

The alternatives are slim, but given more time and the chance to do it again, I would have chosen to use the uTorrent API. This would have been more difficult to implement and test, and it would have come with its own limitation, but the rewards may well have been greater.

7.2 Future work

Storage

Public storage Our solution uses private Dropbox folders which are accessed through an SDK. However using the techniques described and applied in this project we do not need to do this. As they are encrypted, and the encryption is not feasibly broken, they do not need to be hidden and only shared with people with the key to decrypt. Thus our solution is more applicable, and may achieve superior results, with a more public domain. In this domain the files can be uploaded along with its IV and labelled with its owner and no centralised server is then needed.

User experience

Client-side database In order to reduce the number the number of calls to the server, we may introduce a client-side database. The database would mainly store the IVs and security levels for files that they have comes across before. This would enable the client to independently decrypt files and log in slightly quicker. This would be useful for users who frequently download the same file, or for files which are unlikely to change in the near future.

Whether to implement this would come down to a trade-off between time and space.

Client-side threading We could mitigate the overhead required by our solution by executing commands on the client side in a new thread. By doing this we could allow the GUI to continue functioning. This would give the illusion that the commands are executed instantly. Implementing this feature would introduce concurrency issues if another command was executed before the previous one had finished. This could be prevented if we ignore, or queue, new commands until the previous command had finished executing.

Customisable security level bounds It may be the case that the user requires more or less than 5 security levels. Increasing the number of levels after they have already been created would not be difficult, it would be little different to creating the keys at registration as we do now. However the extra keys created would have to be either the lowest or highest security levels to be done without fuss. If the level required to be added was in the middle, the key one below would have to be re-encrypted with the new key and re-sent to the server. Reducing the number of levels could be done by simply discarding the files associated with that level, removing that key from the server and re-encrypting the key one level down from the level to be removed by the key one level up from the level to be removed.

Adjustable friend security levels At this point in time, if a user wishes to change the security level of a friend they would have to revoke the friend entirely and add him again. This involves unnecessary work for what may be a computationally simple task. If the friend's permissions are to be increased this could be done in the same way a friend is added now, that is send the friend the key for the new highest level they can access encrypted with their public key and allow them to derive any keys in-between their new level and their old level. If the friend's level is to be lowered, all keys between their new level and old level have to be revoked, new keys generated and distributed and files associated with those levels have to be re-encrypted.

Multiple trees In our solution a user is only able to create a single hierarchy. This means if all of the people they want to share files with do not do fit into a perfect hierarchy (i.e. they really are in the need for two or

more hierarchies) they are forced to leave people out or have a sub-optimal solution. A potential improvement to this is to allow the user to create more than one hierarchy. That is, they will have more than one hierarchy with distinct sets of keys, distinct sets of files and distinct sets of friends. The hierarchies would not be intended to be used with an overlap of friends. The introduction of this feature would be intended to cover many different possible hierarchies in a user's life (e.g. a freelance worker who wishes to share files with many different companies who each have their own hierarchy.)

Adjustable lower bounds for friends Currently a user can only state an upper bound for a friend. However it may be the case that the user only wishes to share a certain interval or that the friend is not interested in their low-level files. This could be achieved by using Jason Crampton's method as explained in the background section. Achieving this may also allow us to efficiently implement the feature described in *Multiple trees* without actually having to store many different individual trees. For instance, we could allow a 'inner-tree' to consume levels between an x and a y where any users with a lower bound $< x$ can not have a higher bound $\geq x$ and any users with a higher bound $> y$ can not have a lower bound $\leq y$. Therefore each inner-tree would be self-contained within the big tree. An advantage this brings would be that we would not have to search through many different trees in order to find the tree we would like. This would not be possible if we had completely individual trees.

Rather than presenting the user with level names which begin with a number 1 greater than the last level in their previous inner-tree. The levels within each inner-tree can be presented to the user in a meaningful way, such as a name of their choosing. This way the user need not know the inner-trees are connected in any way.

Lazy computation Currently our solution attempts to do as much work as possible when the work arrives. This often causes delays which the user must wait out before they can continue using our application. This is most evident when registering a new account and revoking a user. For registration we could create no security levels until after the account is created and the user is then prompted to state how many security levels they would like. This would avoid having to create any unnecessary levels.

Lazy revocation can be implemented, as described in the background

research, to avoid the large delay our system causes when a user is revoked.

User interface The user interface at this point in time is lacking. It is not particularly aesthetically pleasing and has little in the way of instructions. An introductory page which explains the functionality at a high level and gives guidance on how the user may best use the features on offer would be necessary before our solution could be publicly released.

Security

Automatic key replacement Every time a key is used to encrypt a file it produces ciphertext. An attacker can store these ciphertexts and perform cryptanalysis. The larger the number of ciphertexts the attacker has the easier it is for them to do this. For this reason the same key should not be used for long periods of time. In our current solution the public/private key pair are never replaced and the symmetric keys are only replaced when a friend is revoked access. Future work to improve security would be to automatically replace the keys after they have been active for a set period of time or after they have produced a certain number of ciphertexts. Having a time limit on keys would likely be the superior option as it would be simpler for us to measure and we would not have to store any invasive information.

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] RSA Laboratories, *PKCS #1: RSA Cryptography Standard*, Available at: `http://www.rsa.com/rsalabs/node.asp?id=2125`

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

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

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

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

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

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

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

[13] 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.

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

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

[16] uTorrent, *"uTorrent Web API"*
Available at: `http://www.utorrent.com/community/developers/webapi`

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

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

[19] Mega, *"Developers - 1.4 Cryptography"*,
Available at: https://mega.co.nz/#developers

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

[21] Oracle, *"Java™ Cryptography Architecture (JCA) Reference Guide"*,
Available at: http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html

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

[23] The GNU Crypto project,
Available at: https://www.gnu.org/software/gnu-crypto/\#introduction

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

[25] Keyczar, Available at: http://www.keyczar.org/.

[26] Bruce Schneier, *"Crypto-Gram Newsletter"*, October 15th 1998.
Available at: http://www.schneier.com/crypto-gram-9810.html\#cipherdesign

[27] Randall J. Easter and Carolyn French, *"Approved Security Functions for FIPS PUB 140-2"*, May 30 2012,
Available at: http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexa.pdf

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

[29] 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`

[30] Elaine Barker, William Barker, William Burr, William 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`

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

[32] 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`

[33] Javamex, *"RSA key lengths"*,
Available at: `http://www.javamex.com/tutorials/cryptography/rsa_key_length.shtml`

[34] James Manger, *"A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0"*, 2001.

[35] Shay Gueron, Simon Johnson and Jesse Walker, *"SHA-512/256"*, October 1, 2010.

[36] 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`

[37] Scrypt,
Available at `http://www.tarsnap.com/scrypt.html`

[38] Hugo Krawczyk, *"How to Predict Congruential Generators"*, Computer Science Department, Technion, Haifa, Israel, May 1991.

[39] Black Cat Systems, *"GM-10 Geiger Counter Radiation Detector"*, Available at: `http://www.blackcatsystems.com/GM/products/GM10GeigerCounter.html`

[40] H2Database, *"Performance"*, Available at: `www.h2database.com/html/performance.html`

[41] Erik Zivkovic, *"KeystoreBrute"*, Available at: `https://github.com/bes/KeystoreBrute`

[42] Java SE 6 Documentation, *"PreparedStatement"*, Available at: `http://docs.oracle.com/javase/6/docs/api/java/sql/PreparedStatement.html`

[43] The Open Web Application Security Project, *"Preventing SQL Injection in Java"*, Available at: `https://www.owasp.org/index.php/Preventing\_SQL\_Injection\_in\_Java\#Prepared\_Statements`

[44] H2Database, *"Security Protocols - Wrong Password / User Name Delay"*, Available at: `http://www.h2database.com/html/advanced.html\#security\_protocols`

[45] Bruce Schneier, *"Secrets and Lies: Digital Security in a Networked World"*, John Wiley & Sons, March 2011.