# 6.858 Final Project
# KBox: An Encrypted File System

Giulio Gueltrini (gueltro), Kathleen Laverty (klaverty),
Nicholas Paggi (npaggi), Varun Ramaswamy (vrama)

December 12, 2014

We present the design and implementation of our encrypted file system, KBox. KBox supports the functions expected of a normal file system. Users can create, delete, read, write, rename, and share files and directories. Additionally, users can upload files from their local file system to KBox and download files from KBox from the client. Users can only view files that are either shared with them or they create; this guarantee is cryptographically enforced not enforced with access control lists. The server cannot view the user-given file names, understand the structure of the tree of directories and files, read the contents of a file, or tamper with a file without being detected.

## 1 Design

### 1.1 KNodes

KNodes are abstract constructions that are used to store information in KBox.

On the server, KNode's are represented by text files. A KNode-file's name on the server is a hash of its name, q-bit, and secret: $H(s|n|q)$. A KNode-file can represent a directory, in which case it contains information about the items contained in that directory. This information includes the item's name, information to create the KNode-file's name, lists of users who can read or write to that item, and cryptographic data that can allow users with correct keys to access that item. The whole file is encrypyed under its symmetric key and the cryptographic data is encrypted within that seperately for each user who can access that element under the public key of that user. A KNode-file can represent a file, in which case it contains the ecrypted contents of that file.

To reconstruct a user's file system, the user first finds and decrypts his root KNode. Then, for each item in this KNode, he now has the name of the file and the name of the corresponding KNode. For each item, he uses his public key to attempt to decrypt that item's assymmetric key; if he is successful he knows that he has access to that file. This is done iteratively until the user has touched all the KNodes that he has access to.

On the client-side, the KNode python application represents each KNode as a python object, which will be referred to as the KNode-obj. As the client does the iterative process above to explore KNodes we create each KNode-obj and add its fields as described in figure 1 for a set of all the attributes of this object. A tree of these python objects are created with references between them to reconstruct the file system's structure.

**name:** The user given file or directory name.

**q:** A bit, which is 1 if the KNode represents a directory and 0 otherwise.

**secret:** A unique secret to identify the file.

**ws:** The "write secret", used to prove write access.

**key:** The AES key to encrypt and decrypt the KNode.

**readers:** A list of the public keys of users with read access.

**reader_list:** A list of the key encrypted with the public key of each reader.

**writers:** A list of the public keys of users with write access.

**parent:** An identifier of the parent KNode in the final structure.

**timestamp:** The time corresponding to the most recent modification before the latest pull from the server. This guarantees freshness up to this timestamp.

**contents:** The contents of the file or the directory. If the KNode is representing a file, this is just the contents of the file. If the KNode is representing a directory, the contents is a list of the name, q-bit, secret, and read_list of each file or folder stored in the directory.

## 1.2 Signatures

Our KBox system uses signatures to verify the validity of content of files obtained from the server. Each KNode-file `H(s|n|q).enc` on the server has corresponding signature file `H(s|n|q).sig` contains a signature of the filename and encrypted contents signed with a client's private key. This signature file is updated each time the file is accessed, according to the type of access (see 1.2).

| | |
|---|---|
| `name` | "foo" |
| `q` | 1 |
| `secret` | $s$ |
| `ws` | $ws$ |
| `key` | $k$ |
| `readers` | $[rpk_0, rpk_1, \ldots]$ |
| `readers_list` | $[E_{rpk_0}(k), E_{rpk_1}(k), \ldots]$ |
| `writers` | $[wpk_0, wpk_1, \ldots]$ |
| `parent` | $parent$ |
| `timestamp` | 11:11:11 12/12/2014 |
| `contents` | $[child_1, child_2, \ldots]$ |

Figure 1: A KNode Object

## 1.3 Client-Server Protocols

"Client" here refers to a client-side KBox application. In the KBox file system, it is assumed each user has an RSA public/private key pair. Users are identified by their public key in the KBox application. The following protocols are used in combination to achieve functionality for the KBox API (see KN:API). Users of the client-side KBox application are not able to explicitly call any of the the following functions (unless they create their own KNodes), and instead should interact directly with the API.

**Pull** The pull operation describes how to go from the file representation to the object representation of a KNode. Suppose a client has access to a KNode-obj representing a directory, and the client wants to decrypt one of the files contained in the directory. Then, the client has access to the content of the directory, so the client has access to name, q-bit, secret, and reader_list of the file. So the client can compute $H(s|n|q)$, the name of the file on the server and send an `scp` request to fetch the KNode-file. The client also uses their private key to try to decrypt each item in the reader_list until the AES key for the file is found. This enforces that a client can get the contents of a file only if the client has read access.

Additionally, whenever a client pulls a file from the server, he or she pulls the signature file as well and verifies that the encrypted file was properly signed by a user with write access. If this verification fails, the client is alerted that the file may be corrupted.

Note that there is the issue of bootstrapping - a client must have access to some directory in order to get access to its subdirectories. To this end, we assume that users store locally the information necessary to construct a root directory. This information is provided when starting the KBox application.

**Populate** Populate is the explicit function that iteratively builds the structure of the file system. We first create the KNode-Obj for the root directory based off local information, this allows us to set all the fields on that obj except for its children. When we call populate on that obj it requests the KNode-file for correponding to the root KNode which gives us information to construct KNode-Obj's for all the children off the root that are complete except for their children. We then call populate on each of the children, which repeats the process and iteratively creates the children of each child.

Note that populate only has an effect when called on a directory, we do not automatically pull the contents of files from the server to save space. When the user wants to access a file a different function called fillup() is used.

**Write Request** Unlike read permissions, enforcing write permissions cannot be done client-side. This is the purpose of the "write-secret" field in a KNode. The server stores a table mapping filenames on the server to their respective write-secrets encrypted with the server's public key. Sending the write secret for a particular KNode prooves to the server that the client has write access, at which point the server will allow the client to perform writes.

**Push** The push operation describes how to go from a KNode-obj to a KNode-file. The KNode-obj is flattened and then encrypted with the AES key of the keynode. An `scp` request is sent to the server to add or overwrite that file. Since this is a request that involves writes, the client must also send a write request. When a KNode is pushed, the corresponding signature file is update

**Delete** When a client requests that a file be deleted, the corresponding KNode-file on the server is actually deleted via `ssh` request. The contents of the KNode's parent is updated to show the deletion, and the parent is pushed. Since both of these actions involve file modification, both must be performed with a write request. The corresponding signature file remains. The signature is updated to be a signature of a delete token concatenated with the filename signed with the private key of the client.

**Change Permissions** When a client wishes to grant read permissions of a KNode to another client, the client simply adds the other client's public key to the list of readers, and adds the encryption of the AES key of the KNode to the reader_list. If the KNode represents a directory, this permission change is recursively done to all items in the directory.

In order to revoke a client's permission of a KNode and guarantee that the client will not be able to view or modify the KNode once their permission has been revoked, it is necessary to change the secret, key, and write secret of that KNode. This is equivalent to deleting the KNode and creating a new KNode with the same contents and the appropriate permission lists, so no method for removing is explicitly specified.

## 1.4 API and Usage

In order for a user to interact with the application, `KBox` provides its own shell. The `KBox` shell supports a number of unix operations to allow the user to efficiently navigate, view, and modify the files represented by the existing kNodes. The `kBox` shell also supports commands to load files from the local filesystem to the server, and to store local files from the server.

The user can initialize the `KBox` shell by calling `KBox` in the command line. The user must specify the host and port of the file server, as well as the location of a file corresponding to the user's root KNode-file. The application pulls the root node, and then recursively populates it, giving the user access to the full filesystem for which they have permission.

Once initialized, the user can navigate the filesystem using `pwd` and `cd`. Both of these work identically to the identically name unix commands. The KBox shell also supports a number of other unix commands. The user can view the contents of a directory with `ls` and the contents of a file with `cat`, which both take a valid path in the virtual filesystem as an arguent. The user can also make modifications to the files existing on the server using `mkdir`, `rmdir`, `rm`, `cp`, and `mv`.

The kBox shell also provides several commands not found in unix. The command `changeperms` allows the user to remotely change write permissions on a file, provided the user has write permissions. `changeperms` takes as arguments a path to the file to be modified, whether the permissions are to be added or removed, whether the read or write permissions are to be changed, and the list of new permissions. Since the permissions themselves are public keys, the list of new permissions is specified by the path to a local file containing the public keys.

The command `down` allows the user to move the contents of a specified remote file to a local file. In the case where file is a directory, this is just the representation of the corresponding kNode.

Finally, the command `mkfile` allows a user to add a local file or directory to the server. `mkfile` takes as arguments a path in the tree of kNodes and the path to a local file and an optional "remote" flag. If the flag is not set, then `mkfile` will just encrypt the local file and push it to the server. If the flag is set, then `mkfile` will attempt to parse the local file as a kNode-file into a new kNode. Much like in initialization, the server then attempts to pull and populate the kNode and add it to the local filetree.

## 2 Security Guarantees

In analyzing the security of this design, we consider a number of different vectors: a malicious KBox user, an attacker with server access, and an attacker who can intercept packets from the server. For each of these threats, we must consider both the case where the server is trusted and the case where the server is untrusted. Finally, we must consider an attack by a malicious server.

### 2.1 Kbox User

First consider the simple case in which a malicious user wants to read from a file which should be inaccessible. Recall that read access is granted by putting a user's public key in the list of readers for a KNode and putting the encryption of the AES key under the user's public key in the reader_list. In order to get the AES key necessary to decrypt the file, then, a user without write permission would have to invert another user's public key encryption scheme, a cryptographically difficult task. So KBox guarantees that users without read access will not be able to view the plaintext contents of the file. Furthermore, for a user without access to the parent directory of a file, it will difficult to know that a file with a particular name even *exists* on the server.

In the case of a trustworthy server, the guarantee for write protection is equivalent, since a user without write permissions would have to invert another user's public key encryption scheme in order to recover the write secret. So in the case of a trusted server, KBox guarantees that a users cannot modify or remove any files without write permissions. However, an untrusted server may leak write permissions, or just ignore the write-request protocol altogether. In this situation, a malicious user with read access to a directory may modify the contents of that directory arbitrarily. Like before, it would be difficult for a malicious user without read access to ascertain whether or not a particular file exists on the server. Finally, a modification or removal of a file by a malicious user will break the verification of a signature file, providing valid users with evidence of tampering.

### 2.2 Server Access

Suppose an attacker is able to gain access to the server hosting the KNode and signature files. Such an attacker would know the names of all files existing on the server. Since all the decryption is done client side, such an attacker would only have access to the encrypted contents of each file, and gaining the plaintext contents would be cryptographically hard. Furthermore, since storage is flat on the server this attacker would not be able to determine any information even about the structure of the filesystem.

In the case of a trustworthy server, modification require proof of write access so the guarantee is the same as that for a malicious user. In the case of an untrusted server, the attacker may be able to modify or delete *any* files, including the signature files. However, modification of either a KNode file or a signature file will give valid users evidence of corruption. In order for this attacker to overcome this, the attacker would have to spoof the signature file, which requires the true name of the file. Hence, the attacker would have to determine secret for the file and invert the hash function which computes the file's name on the server - this is highly unlikely.

### 2.3 Network Access

Suppose an attacker is able to gain access not only to the server, but is also able to sniff packets being sent from and received by the server. Since a user's root KNode is populated whenever that user connects, one might worry that the attacker would be able to glean the file structure from the pattern of requests sent. Such an attack would give the attacker information about what files on the server were associated with a particular user's filetree. However, since the attacker does not know how populate actually fills the file tree, the attacker could only determine possible topological ordering on these files rather than their true structure. This knowledge would not allow an attacker to get any information regarding the contents of the files. However, an attacker seeking to do maximal damage may be able to target root nodes more easily. Although it was not implemented, a possible solution is to create dummy roots.

### 2.4 Malicious Server

At first glance, it might seem like a malicious server could do a fair amount of damage. However, a malicious server would only be able to access the stored encrypted files and be able to listen to requests from users. In other words, a malicious server is no more powerful than an attacker with server and network access, and hence all guarantees from the previous section carry over: a malicious server could not read plaintext from the stored files, modify the files without alerting the users to corruption, and can only gain minimal information about the file structure.

## 3 Use Cases and Performance

### 3.1 Performance Tradeoffs

While loading, the client loads all directories that the user can access immediately and lazily loads files when the user requests to view or download them. The first decision results in more time spent initializing but allows instantaneous movement through directories after it is loaded, the second requires more time to load files but the alternative, loading all files at start up, would result in prohibitively long initialization time and would use too much memory.

While adding or modifying a file we must encrypt and push only files to the server: the contents of the file itself and the knode corresponding to the directory that the file is contained in. This means that the only delays are encrypting and scping time. The first is unavoidable in any encrypted system and ours is fast due to the use of assymmetric key encryption for files. The second is minimial because knodes are generally small, containing only the names of the contents along with metadata and encryption information.

### 3.2 Use Case Timing

Our first test case involved uploading a large file to the server and then downloading it. To isolate the performance of KBox, we compared it to using SCP to transfer the files and ran both the server and client on the same machine to remove network effects. The upload time for our system includes reading from disk, generating keys and sending to the server. The download time includes making the request, waiting for the download, and unencrypting, but does not include initializing the system. Results are presented in in Figure 2, averages are across 50 trials. We can see that our systems performance is slower than SCP, but not by much. Additionally, it is capable of handling large files.

| System | File Size | Upload Avg | Upload Ratio | Download Avg | Download Ratio |
|---|---|---|---|---|---|
| `scp` | 179 MB | 2.36 s | 1 | 2.65 s | 1 |
| KBox | 179 MB | 6.53 s | 2.76 | 5.75 s | 2.16 |
| `scp` | 360 MB | 4.47 s | 1 | 4.99 s | 1 |
| KBox | 360 MB | 11.37 s | 2.54 | 11.07 s | 2.22 |
| `scp` | 502 MB | 6.58 s | 1 | 7.23 s | 1 |
| KBox | 502 MB | 17.25 s | 2.62 | 16.44 s | 2.27 |

Figure 2: Timing data for upload and download of single large file. SCP vs KBox.

Our second test demonstrated a particularly interesting aspect of our design. It involved a simulation of someone storing a set of documents. In this case, we uploaded a folder containing 131 mp3 files totaling 504 MB. `scp` uploads this folder in 7.49 s while kbox is slower, taking 102.1 s. However, once we uploaded, we could initialize our system in only .93 s which then allows us to navigate through the folder and then select a song to download, which for a 6 MB file takes only 0.50 s. This means that while there is a reasonable delay to upload large folders, once they are uploaded the system can quickly open, move through, and download items from them.

## 4 Further Work

We designed a challenge protocol for the clients to prove that they had write access in order to prevent unauthorized users from writing to a file. However, in the way in which we attempted to implement this protocol, we experienced race condtions which we not able to resolve without introducing a large amount of latency. In the future, KBox should be modified to guarantee both write protection and efficiency.

Additionally, `scp` is considerably more efficient when transferring batches of files, rather than transferring the files individually. This improvement could be integrate into the KBox system by storing multiple KNode files in a temporary batch folder and using a single call to `scp` to transfer them all.

Another possible improvement to our system could include an expansion of the idea of lazily fetching the KNode-files. In our current implementation, directories are populated recursively. Alternatively, we could optimize the depth of the tree to which we pull initially depending on the experienced latency.

## 5 Conclusion

KBox allows users to securely store their file on an untrusted remote server with confidence that their files may not be modified without detection. A notable property is that a malicious server has no more power to corrupt or steal data than a malicious user.