A Revocable Backup System
(extended abstract)

Dan Boneh
dabo@cs.princeton.edu

Richard J. Lipton*
rjl@cs.princeton.edu

Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

We present a system which enables a user to remove a file from both the file system and all the backup tapes on which the file is stored. The ability to remove files from all backup tapes is desirable in many cases. Our system erases information from the backup tape without actually writing on the tape. This is achieved by applying cryptography in a new way: a block cipher is used to enable the system to "forget" information rather than protect it. Our system is easy to install and is transparent to the end user. Further, it introduces no slowdown in system performance and little slowdown in the backup procedure.

1 Introduction

On many systems the remove-file command misleads the user into thinking that his file has been permanently removed. Usually the file is still available on a backup tape. This is an important feature used to protect against accidental file erasure or system crashes. However, it has the draw back that the user is unable to completely remove his files. In many scenarios it is desirable on the part of the user to erase all copies of a certain file. This is frequently the case with personal mail messages. Similarly, a user may wish to remove the history and cache files of his web browser. Other examples include a patient removing his medical files from the hospital's system upon checkout and financial data that should be erased after a short period of time. Our goal is to make it so that the sensitive data becomes inaccessible to anyone (including the data's owner).

A naive solution is to erase the data from the file system. Then mount the backup tapes one by one and erase the sensitive data from them. This method is impractical for several reasons. First it inconveniences the user. To implement this scheme the user must call a computer operator whenever such an erasure is to take place. Furthermore, this method is quite complicated considering the backup policy which many institutes employ. Usually a backup of the entire file system is done once every time period, e.g. every month. The backup tape is then stored forever in a "cellar" or sometimes off site. This backup policy enables users to restore their entire directory structure to any point in the past. Clearly the naive approach is doomed to fail whenever this backup policy is used. The computer operator has to remove the data from many backup tapes. This procedure is painstaking, but is also insecure since the operator might "forget" to remove the data from one of the old backup tapes.

We propose a system which avoids the problems mentioned above. Our system will enable the user to remove a file from the file system and all backup tapes without ever mounting a single tape. At first this seems impossible: it is not possible to remove data from a tape without physically erasing the tape. Fortunately, cryptography enables us to do just that. The basic idea can be described as follows: when a file is backed up to tape it is first encrypted using a randomly generated key. The encrypted version of the file is the one written to tape. When the user wishes to remove the file from the backup tape he instructs the system to "forget" the key used to encrypt the file. The act of forgetting the encryption key renders the data on the tape useless. No one, including the file owner, can ever access the file again. In effect, the file has been erased from the tape. Notice that the encrypt-before-backup is completely transparent to the user. It is crucial that no one but the operating system know which key was

*Supported in part by NSF CCR–9304718.

used to encrypt the file during backup. This ensures that when the operating system is instructed to forget the encryption key, the tape data becomes unreadable.

The rest of the paper is devoted to describing an implementation of the above idea. The tricky part of the implementation is managing the encryption keys. Clearly the encryption keys have to be backed up as well. We have devised several methods for backuping the encryption keys while still preserving the fundamental properties described above.

It is interesting to compare our system to the cryptographic file system implemented by Blaze [2]. In a cryptographic file system the files are always stored in an encrypted form. Whenever a file needs to be accessed it is decrypted on the fly using the owner's key. The backup tape is a direct dump of the file system, i.e. the tape contains the encrypted version of every file. Clearly the owner of the file is the only person who has access to the backed up version of the file. This does not guarantee that his backed up files will not be accessed by an unauthorized party. The owner might be forced to reveal his key, e.g. due to a court order. Thus, the cryptographic file system does not conform to our requirements: it does not guarantee that the backup tape data will not be accessed. On the other hand, in our scheme the backed up data becomes inaccessible to everyone once it is "erased" from the tape. In addition it is important to point out that our system is very easy to install. Only the backup system has to be modified. The cryptographic file system requires far more extensive modifications.

2 Key management

In this section we give a detailed description of our backup system. In our system the user can specify a collection of files that are to be encrypted during backup. Each such file has an encryption key associated with it. The encryption key has a life time which is specified by the user. When the key expires the system generates a new key for the file and adds the expired key to a list of old keys. The maximum number of expired keys that the system remembers is a parameter specified by the user. When the list of keys is at its maximal size the oldest expired key in the list is erased to make room for the new key.

The above mechanism is very flexible. We give some examples to illustrate its use. Suppose a user is preparing a document and he wants to make sure that old and invalid copies of his draft are inaccessible to anyone. He could set the key-life for that document to be one month and instruct the system to store 12 expired keys for this document. This means that a new key is generated once a month causing a year old key to be erased. The result is that once a month all backup copies older than a year are revoked. To revoke all copies that are more than 6 month old, the user can manually instruct the system to remove all keys that expired more than 6 month ago. Of course the user can decide to delete the file altogether by instructing the system to remove all keys (including the current key) for the file. In case of disk crash the most recent version of the document can be restored from tape using the current key.

More generally, institutions may wish to adopt backup revocation policies. For instance, to prevent law suits regarding old data, an institution may decide to revoke all backup tapes that are more than 3 years old. This is done by instructing the system to delete all keys that expired more than 3 years ago. We point out once more that the revocation process has the effect of removing files from *all* old backup tapes without ever mounting a single tape.

The most important component of our system is the key management. All encryption keys are stored in one file which we refer to as the *key-file*. This file should be protected meaning that only privileged processes should be allowed to read it. The key-file contains one record per each file that has ever been encrypted during back up. There are two types of entries in the key-file: directory and file entries. The structure of a single entry in the file is described in Figure 1. The fields in a file entry contain the filename, the maximum number of expired keys that the system should store, the life time of a single key and a list of keys. The list of keys includes the current key (as the first entry in the list) followed by the expired keys ordered chronologically. When a new key is generated the keys in the list are shifted and the last (oldest) entry is lost.

The fields in a directory entry specify the directory path and indicate whether all files in the directory and subdirectories should be encrypted during backup. When the cont_flag is turned on the directory is scanned and all new files in it are added to the key-file. The `num_keys` and `key_life` fields for these new file entries are set to the values taken from the corresponding fields in the directory entry. A directory entry must always precede a file entry in order to specify the path to the file.

The key-file is permanently stored on the file system. This file is extremely important since without

it the backup tapes are useless. For this reason the key-file must be backed up as well. However, the file can not be written to tape as is. If it were written to tape the system would not be able to permanently erase keys from the key-file. This would defeat the purpose of our system. Our solution is to generate a new *master-key* during every backup. The key-file will be written to tape after it has been encrypted using this new master-key. The master-key itself is not written to tape. Notice that in case of a disk crash, the master-key is crucial for recovering the key-file. Without the master-key the key-file can not be recovered and as a result *all* backup tapes using our system become useless. For this reason the master-key must be handled with care. We discuss methods for storing the master-key in Section 2.1.

To make sure that the key-file is backed up to tape we treat it as any other file which is to be encrypted during backup. This means that the key-file is stored on the file system for which it used. Further, the key-file contains an entry which corresponds to the key-file itself. The file-name field in this entry contains the key-file name and the key field contains the master-key. Hence, the master-key is actually stored in the key-file and is used to encrypt the key-file during backup. To make sure that a new master-key is generated during every backup the key-life field is set to zero. Similarly, to make sure that only the current master-key is stored, the num-keys field is set to 1. This has the effect of revoking the old copy of the key-file during every backup.

We can now describe the details of the backup and restore operations. The schematics of the backup procedure are described in Figure 2. We briefly sketch these steps here.

Initialize When the backup process is initialized the entire key-file is loaded into memory. Then all directories in the key-file which have the `cont_flag` turned on are scanned and their contents is added to the memory image of the key file. The `touch_key_file()` routine will update the last modification date of the key-file. This guarantees that the key-file is written to tape even during an incremental backup[1].

New keys Next, new keys are generated for all files who need them. A new key is generated for all files for which `date_key_issued + key_life < current_date`. The `date_key_issued` field is updated for each new generated key.

Write all files to tape Loop on all files in the file system. Each file is dumped to tape, after encryption if necessary. The old master-key is erased from the key-file just before the key-file itself is dumped to tape. This ensures that the old master-key is *not* written to tape when the key-file is backed up. Had this not been done, old master-keys could be read from the backup tape. This would be disastrous for our system.

Terminate Finally, write the key-file back to disk and store the new master-key using the methods described in Section 2.1.

The modifications to the restore operation are even simpler. During restore we use the key-file to find the appropriate key for each file. The system first reads the date written on the backup tape. This date is the date on which the backup was done. For each file to be restored the system retrieves the key used to encrypt the file on the backup date. This key is then used to decrypt the file when it is read from the tape.

In case of a disk crash the first thing that has to be recovered is the key-file. This is done by creating a temporary key-file containing a single entry. This entry contains the key-file as its file name and the master-key as the key. The operator can now use restore to recover and decrypt the most recent backed up version of the key-file. Now a full restore of the file system is possible.

2.1 Master key management

As was mentioned above, the master-key is a crucial component of the system. Without it, all the backup tapes are useless. We first describe the properties that a master-key storage system must satisfy. Observe that the master-key can not be written to tape in the clear. If it was, then the key-file on the tape would be accessible and with it all the files on the tape. This will again defeat the purpose of our system. Furthermore, wherever we choose to store the master-keys we must make sure that only the most recent key is accessible. As before, an old master-key will enable access to the contents of old tapes. Hence, any system which stores master keys must "forget" all but the most recent one.

We propose two solutions which can be used in conjunction with one another. The first one is the simplest; at the end of each backup the computer operator is asked to write down (on paper or on a floppy disk) the current master-key. He then destroys his copy of the previous master-key.

The second method involves an internet server

[1] An incremental backup of a certain level only backups files that were modified since the last backup of that level.

```
union keyfile_entry {

  struct file_entry {              /* Structure of file entry         */
    char *filename;
    char *keys[];                  /* Keys used to encrypt file       */
    int  num_keys;                 /* Number of keys to be saved      */
    time_t key_life;               /* Life time of a single key       */
    time_t date_key_issued;        /* Date current encryption key     */
                                   /*    for file was issued          */
    time_t last_backup_date;       /* Date file was last backed up    */
  } file

  struct dir_entry {               /* Structure of directory entry    */
    char *dirpath;                 /* Directory path                  */
    char cont_flag;                /* Indicate if files in dir        */
                                   /*   should be added to key-file   */
    int  num_keys;                 /* Default num keys to be saved    */
    time_t key_life;               /* Default life time of a key      */
  } dir;
}
```

Figure 1: key-file structure

```
load_key_file();
scan_sub_dir();
generate_new_keys();
touch_key_file();                              /* key-file will be stored    */
                                               /* during incremental backup */
Loop on all files in file-system {

  file_entry = get_key_file_entry(current_file);
  if (file_entry != NULL) {                    /* If file has an entry in    */
    key = get_key(file_entry);                 /* key-file then get          */
  else                                         /* encryption key. Otherwise, */
    key = NULL;                                /* don't encrypt file.        */

  if (is_key_file(current_file))               /* Erase current master-key   */
    erase_master_key_from_disk();              /* before writing key-file.   */

  Loop on blocks in file {                     /* Write file blocks to tape  */
                                               /* encrypting them if         */
    if (key != NULL)                           /* necessary.                 */
      encrypt(current_block, key);

    dump(current_block);                       /* Write block to tape.       */
  }
  if (file_entry != NULL)                      /* Update backup date field   */
    file_entry->last_backup_date = current_date;
}

write_key_file();      /* Write key-file back to disk.                       */
store_master_key();    /* Take extra care to store the new master key.       */
```

Figure 2: Backup operation

which is used for master-key storage. Suppose a site performs daily incremental backups. This is common in many sites. The master-key-storage-server generates a public/private key pair on a daily basis. Every day all sites performing backups encrypt their daily master-key using the server's current public-key and write the resulting string on their backup tape. In case of a disk crash a site can recover its daily master-key by performing the following steps: the local computer operator first reads the encryption of the master-key written on his tape. He then sends the encrypted master-key to the storage-server. The storage-server, after authenticating the identity of the sender, decrypts the master-key and sends the result back to the computer operator. The operator can now restore his file system. The same public/private key pair can be used by all sites in the world which use daily (incremental) backups. Hence, the master-key-storage-server simply provides the service of generating a public/private key pair on a daily basis.

The above description is not much different than a standard key escrow system. The new twist is that to make sure that old master-keys are inaccessible the master-key-storage-server must erase all but its most recent private-key. Otherwise old master-keys can be recovered from old tapes. These master-keys then enable anyone to read the contents of old tapes. Since the storage-server is providing a commercial service it is in its best interest to be trustworthy and indeed "forget" all old private keys. To increase the security and reliability of the scheme one can use k out of l secret sharing techniques [4]. This means that a given site will employ l storage-servers and exactly k of them are required to recover the daily master-key. Now an old master-key can not be recovered even if $k-1$ of the storage-severs are untrustworthy. Similarly, even if $l-k$ storage-servers crash and lose their current private-key a site can still recover its current master-key. The parameters l and k can be fine tuned to the site's needs.

For increased reliability some sites may wish to be able to access a small number (e.g. three) of old master-keys. For instance, if the most recent backup tape is lost, the previous one can be used if the corresponding master-key is still accessible. As a result some sites may want the storage-server to save a small number of its most recent private-keys. To accommodate this need the storage server can offer k (where $k < 10$) types of public/private key pairs. For type 1 only the most recent private key is available. For type 2 the two most recent private keys are available, etc. This arrangement requires the storage-server to generate k new public/private key pairs daily. A site who wishes to have access to its three most recent master-keys may use a type 3 public key published by the storage server.

3 The user interface

Two utilities enable a user to interact with our system. The first utility enables a user to declare that a file or directory has the revocable-backup attribute. The second enables a user to revoke backup copies of his files. These two utilities comprise the most basic interface. Naturally utilities for obtaining the status of a given file are also provided.

mkrvcbl The make-revocable command will add a file (or directory) to the key-file. The user can specify the key-life and num-keys parameters on the command line. The default values for these parameters is infinity for the key-life and one for num-keys. This means that there is a single key associated with the file throughout the life of the file. This enables the user to revoke all backup copies of the file when he wishes to completely remove the file from the system. Executing `mkrvcbl dir-name` will add the directory to the key-file with the `cont_flag` turned on. During backup all files in the directory and its subdirectories will be added to the key-file. Only the owner of the file (or directory) is permitted to apply `mkrvcbl` to the file (or directory).

rvkbck The revoke-backup command is activated as `rvkbck file-name [date]`. Only the file owner is permitted to apply `rvkbck` to a file. The command removes all keys that expired before `date` from the file's entry in the key-file. If the `date` parameter is left blank the entire file's record is removed from the key-file. In doing so, all keys used to decrypt the file from the backup tapes are lost. As a result, the file can no longer be restored from the backup tapes. Unfortunately this is not quite true. Recall that the key-file itself is also backed up on tape. Hence, the removed entry can still be found in the backup version of the key-file. However, at the next backup the master-key will change making the old backed up version of the key-file useless. Therefore, the entry is permanently removed from the key-file after the next backup operation. For example, in a system where an incremental backup takes place daily the `rvkbck file-name` command will permanently remove the file within 24 hours of the time the command is issued.

The two utilities described above enable a user to

manipulate the key-file according to his needs. By definition, the key-file contains an entry for every file that has ever been encrypted during backup (and has not been revoked). This could make the key-file large. Once every time period, e.g. once a year, the computer operator may wish to prune the key-file by removing all entries which correspond to files which are no longer present on the file system. This is done using the *prune-key-file* utility.

prunekeyfile The command `prunekeyfile date` will remove all entries in the key-file which correspond to files which no longer exist on the file system and whose last backup date is prior to 'date'. The utility will write the removed entries to a file called `key-file.'date'`. This file will be backed up to tape as a regular file (without encryption) and then removed from the file system. Notice that by doing so the user is no longer able to revoke the backed up copy of the files corresponding to the removed entries. This is fine since these files are old files which are no longer present on the file system.

4 Summary and future work

We described a system which enables a user to permanently remove a file from the file system and all backup tapes. The ability to revoke backup copies of files is important and may be of interest to many institutions. Our system is very easy to install and provides a simple user interface.

Our scheme applies cryptography in a new way. Here cryptography is used to erase information rather than protect it. Since the backed up files are stored for extended periods of time it is desirable that the block cipher used to encrypt the files be extremely secure. Hence we suggest using block ciphers with longer keys than are usually used. For instance one could apply triple DES twice to obtain a 224 bit key.

To simplify the installation of our system we chose not to modify the existing file system. Our implementation has the drawback that the backup key information does not become a part of the file. Thus, when the file is moved or copied the resulting file will not be securely backed up. A full-scale implementation of our scheme could embed a new file attribute in the file header. This attribute would indicate that the file is to be securely backed up. This way when the file is moved or copied, the new file will have the same attributes.

The size of the key-file can be reduced by incorporating the ideas used in Lamport's one-time password scheme [3, pp. 230–232]. We thank Steven Bellovin [1] for pointing this out. Let f be a one way permutation. The idea is to only store the oldest accessible key in the key-file. The more recent keys can be obtained by repeatedly applying f to this key. When the oldest key k expires it is simply replaced by $f(k)$. Since given $f(k)$ it is hard to compute k we may say that k has been "forgotten". For efficiency it is desirable to store both the oldest and most recent keys in the key-file. Using this approach we can make do with storing only two keys per key-file entry.

As a final note we point out that standard UNIX backup utilities, e.g. dump and tar, do not enable a user to specify a collection of files that should not be backed up. There are several types of files for which this option is important due to privacy considerations. The typical example is the history and cache files of the user's web browser. On the one hand, the information is usually not important to the user. On the other hand the information might be private and the user may not want extra copies of it floating around. We suggest that this feature be incorporated into commercial backup systems.

Acknowledgments

We thank Jim Roberts and Matt Norcross for a very helpful explanation of our local backup system.

References

[1] S. Bellovin, private communications.

[2] M. Blaze, "A Cryptographic File System for Unix",
available at `http://www.cert-kr.or.kr/doc/Crypto-File-System.ps.asc.html`

[3] C. Kaufman, R. Perlman, M. Speciner, "Network Security", Prentice Hall, 1995.

[4] A. Shamir, "How to share a secret", CACM, Vol. 22, Nov. 1979, pp. 612–613.