File System Design with Assured Delete

Radia Perlman
Sun Microsystems
radia.perlman@sun.com

Abstract

This paper describes a system that supports high availability of data, until the data should be expunged, at which time it is impossible to recover the data. This design supports three types of assured delete; expiration time known at file creation, on-demand deletion of individual files, and custom keys for classes of data. The obvious approach, of course, is to encrypt the data on nonvolatile storage, and then destroy keys at the appropriate times. However, managing ephemeral keys; robustly keeping them for some amount of time, and then reliably destroying every copy, is difficult. We partition the problem so that the burden of ephemeral key management can be outsourced to a minimally trusted third party we refer to as an "ephemerizer", with negligible performance overhead, resulting in a file system that is easy and inexpensive to manage.

1. Introduction

With traditional systems, making data highly recoverable after a disaster makes it hard to reliably delete it. Making it recoverable requires making a lot of backup copies, and keeping them in diverse locations. The more copies are made, the harder it is to find all copies to delete them.

This paper simultaneously solves two problems:

1. Making all state in a file system (other than files which have been expunged) recoverable by authorized parties from backup media, and
2. making data that has been expunged unrecoverable by *anyone*.

The goal of the design is to make the file system inexpensive and easy to manage, and to provide assured delete without imposing anything more than negligible performance overhead over a simple (without assured delete) encrypted file system.

This is done by storing data encrypted, and then deleting keys to make data unrecoverable. We call keys that must at some point be deleted *ephemeral keys*. Ephemeral keys are much more difficult to manage than permanent keys (keys that do not need to intentionally become irrevocably lost). The reason is that to ensure ephemeral keys are not prematurely lost, copies must be made and stored in many locations, which makes it difficult to ensure that all copies are deleted after their lifetime.

To make the file system easy to manage we design it so that ephemeral key management can be outsourced to a minimally trusted third party which we refer to as an *ephemerizer*.

As we will show, a single ephemerizer, and a single set of time-based keys can serve many mutually distrustful domains, freeing the customers it serves from the burden of managing ephemeral keys. And since our design does not require high reliability of individual ephemerizers (since we use a quorum scheme), ephemerizers become easy enough to manage that many companies, especially those that have many geographic locations to store ephemerizers, can manage some or all of the ephemerizers they use.

Many file systems assume client-side encryption/decryption and, for instance, store files encrypted with the public keys of the authorized readers. Although ephemerizers could be used by the clients (rather than the file system), that would require interacting with ephemerizers on every file open. Instead, the design in this paper handles ephemerization at the file system, which as we will see, gives dramatically better performance while not hindering end-to-end security mechanisms.

We will present designs for three different methods of assured delete:

- **time-based**: files, when created, are declared to have an expiration time. An example application for this form of assured delete might be a medical test which promises that results can be obtained for a period of time, and then afterwards, all information about the test will be expunged.

- **individual file deletion on-demand**: files can be deleted individually at any point. As we will show in section 3.3.1, this form is dangerous, and we do not know of compelling applications for it, although we do show how to implement it in a scalable way.
- **custom classes that can be deleted on-demand**: sets of files can be encrypted with a custom ephemerizer key, and policy can be applied at any time for that key, such as deleting it, or suspending its use until some action is taken. One example application is protecting a spy ship that could be captured. If all data is locked with a remote key, then the remote custom key need not be destroyed in order to make the data unrecoverable by the enemy. Instead, the ephemerizer can be notified to revoke use of the key to anyone captured with the ship. Another application of custom keys is a law firm that might need to expunge all records associated with a particular client once the client severs ties with the firm.

These different types of deletion can be used in combination in the same file system. For example, some files might be created with time-based assured delete, others might be created to never need to be assuredly deleted. These types of assured delete can also be combined for the same file. For example, there might be a custom key for a whole directory, plus files within the directory can have time-based assured delete. If the custom key for the directory is deleted, all files in the directory, regardless of expiration date, become unrecoverable. As long as the custom key remains, files in that directory remain readable until their expiration date.

To support time-based ephemerization, assuming expiration granularity of a day, and lifetimes of up to 30 years, this requires the ephemerizer to have 10,000 ephemeral public keys. This is a scalable solution because the same 10,000 ephemerizer public keys can be uitlized by all organizations. In other words, there is no loss in security for mutually distrustful organizations to use the same ephemerizer, and the same ephemerizer public keys, to provide time-based ephemerization for their file systems.

For custom keys, the ephemerizer needs to keep a key for each class of file, so presumably an organization would pay the ephemerizer to maintain the custom keys for that organization. To support on-demand secure deletion of individual files, as we will see, the ephemerizer only needs to keep two public keys per file system.

Ephemerizers create, advertise, and maintain ephemeral public keys. Ephemerizers should not make backups of keys. Instead of relying on ephemerizers to be reliable, the file system uses multiple ephemerizers, with independent sets of keys. Rather than making an ephemerizer robust by having it make *n* copies of its keys, robustness is achieved instead by the file system, by having the file system use *n* independent ephemerizers, any (quorum (see [25])) of whom can unlock the encrypted data.

Putting the onus of reliability on the file system, by assuming it will be using multiple independent ephemerizers, makes management of an ephemerizer easier, since it is not a disaster for its clients if it loses its keys. It is easier for the ephemerizer to avoid making *any* copies of keys than it would be to attempt controlled key copying.

Our design, as we will see, places minimal trust in the ephemerizers. The protocol for requesting decryption uses blinding [6], so the ephemerizer cannot see what it is decrypting. Decryption requests (for time-based ephemeral keys) can even be done anonymously [10]. The only threats an untrustworthy ephemerizer presents are to not forget the keys when it promises to, or to forget the keys prematurely. These threats are mitigated by having the file system use a quorum of ephemerizers, and to be somewhat careful to choose reputable ephemerizers.

Our system does not, by itself, prevent users, or the file system, from copying and storing the decrypted data, which would violate the assured deletion guarantee. Our system could be coupled with a tamper-proof reader that prevented the client from using the data in a way other than the way our system intends. But it is not part of our system.

2. Previous Work

There are various "disk-scrubbing" systems that concentrate on deleting data on disk [14], [20], by techniques such as overwriting data many times. It might be possible to make data on a disk unrecoverable, but it would be extremely difficult to guarantee that all copies of backup media are destroyed.

There are many designs for encrypting file systems without assured delete, such as CFS [3], SiRius [13], EFS [24], and Plutus [16]. Most of these assume client side encryption/decryption. The assured delete design in this paper is complementary to these schemes. Although individual users could use the ephemerizer, as assumed in [21], this is inefficient, since it requires interaction with (a quorum of) ephemerizers on each file open. The design in this paper instead relies on ephemerization being done by the shared file system, which allows dramatic performance gains and does not interfere with end-to-end security.

Assured deletion was implemented by a company named Disappearing, Inc. [11]. Their system involved a key manager that created and maintained a secret key for every file. A file creator asked the key manager for a key, and it returned a (key ID, key). The file reader requested the key by sending the key ID. This system requires the key manager to create and store a key for every message, and the key manager had to be completely trusted, since it could read all messages.

There are several products being introduced for managing keys for backups [19]. These involve buying one or more boxes that maintain keys, usually a key for each tape. Assuming the database of keys is not backed up (because if it is, then there is no longer assured delete), and assuming the customer has bought sufficiently many copies of the key manager boxes so that the database is never lost, assured deletion can be done by deleting the key from the database. The boxes synchronize with each other so that commanding one of the boxes to delete a key will cause it to tell the others to delete the key from their database as well. As we will explain, the design in our paper is less expensive (because the customer can do replication using tapes rather than expensive boxes), and more robust, as we will explain in section 3.3.1.

In [4], a scheme is presented in which a file system keeps a table of keys for all files in the system. We will call this table the F-table (where each file is encrypted with its own key F). The F-table is backed up, encrypted with a key of a key manager. The key manager maintains several keys (where "several" is a parameter, say "*s*", and creates new keys with some frequency (also a parameter, but let's say one per day), so that at all times the key manager maintains *s* key pairs. When the key database needs to be recovered from backup, the key manager is asked to decrypt it.

This is similar to the on-demand scheme we present in section 3.3, with two problems that we will fix:

- backups of the key database are readable by the key manager if the key manager has ever been asked to decrypt that version of the database
- if a file system using this system were down for several days (say, after a natural disaster that lasts more than *s* days), then all the data would be lost, since the key manager is forgetting keys on a predetermined schedule.

Another difference between the on-demand scheme in [4] and the on-demand variant of assured delete we will present is that in [4], encryption is only done at backup time, so the data on the local disk is unencrypted. Therefore, to really delete data, this scheme would need to also employ disk-scrubbing. The scheme would also need an extra level of key, for having the file system authenticate to the key manager. We add this detail in this paper, but we also argue (see section 3.3.1) that an on-demand scheme is risky, even with the enhancements we provide in this paper, and therefore we would advise relying instead on the other two types of assured delete that we present in this paper.

In [7], a scheme is presented in which a large trusted store for secrets, from which individual secrets can be expunged and made unrecoverable, can be achieved using a small tamper-resistant module with that property, connected to a larger general purpose memory. This scheme could be used to make our ephemerizers out of cheaper components, but is otherwise orthogonal to what we are presenting in this paper.

3. Our Design

We will present three different types of assured delete (time-based, custom keys, and on-demand).

3.1 Concepts Used by all Variants

First we present concepts used by all the variants, and then we discuss, for each type of assured delete, how to structure the file system to use it.

Classes

Two of our designs (time-based expiration, and custom classes) group data into *classes*. A class is a set of files that will be deleted simultaneously. For instance, with time-based expiration, files with the same expiration date will be in the same class.

Overall File System Secret G

We assume there is a secret G associated with the file system. G is accessible to the system administrators, and is necessary to unlock the entire file system.

G might be in the form of a passphrase, kept in the head of multiple system administrators. Or for a more secure approach, G could be a high quality secret. It could be broken into *n* shares for a quorum scheme [25] where any *k* shares can recover G. Each system administrator is given a smart card, each with its own independent high quality secret. Each of the *n* shares of G is encrypted with one of the system administrator's secrets, and the encrypted shares can be stored on backup media and replicated for robustness. Any quorum of *k* system administrators can insert their activated smart cards to recover G, where "activating" the smart card might involve inputting a passphrase or a biometric.

Knowledge of G, together with backups of the state of the file system, enable restoration of the state of the file system (other than files which have been expunged).

Ephemerizers

An ephemerizer is a service that creates, certifies, and publishes ephemeral keys, decrypts using a specified key when requested, and discards keys at the appropriate time. An ephemerizer might be managed by the same organization that manages the file system, or it could be a public service.

An ephemerizer has a permanent public key, which the file system is either securely configured with, or which the file system can find through a PKI. An ephemerizer uses its permanent public key to certify its ephemeral public keys, or to authenticate, in the case of certain operations that require authentication, such as management of custom keys.

An ephemerizer does not need to be highly trusted, because the keys it knows will not allow it to decrypt data. It might, however, fail in one of two ways:

1. forget keys prematurely, or be unavailable when decryption is needed, or
2. fail to forget keys when it should.

Both these failure modes can be handled with a quorum scheme. However, even in a 1 out of *n* scheme, to gain access to data, a malicious ephemerizer would not only need the retained ephemeral key, but also would need access to the encrypted backup media, and G.

Blind Decryption Protocol

The protocol that we will use, for having the file system request a decryption from the key manager, we call "blind decryption". Blind decryption was introduced, and is described more fully in [21]. Blind decryption is conceptually very similar to Chaum's blinded signatures [6]. The idea for blind decryption is to come up with blinding functions (B,U) for "blind" and "unblind" which commute with the (encrypt, decrypt) functions of the ephemerizer's key. If the file system has E_i(M), (a quantity M that is encrypted with the ephemerizer's public key ID *i*), the file system does the following:

To request that the ephemerizer decrypt the encrypted quantity E_i(M):

- The file system creates an ephemeral blinding pair of functions (B, U) that commute with the ephemerizer's E (encrypt) and D (decrypt) functions.
- The file system performs the *blind* function B on the encrypted M to obtain B(E_i(M)).
- The file system sends BE_i(M)) to the ephemerizer, together with *i*, to tell the ephemerizer which decryption key to use.
- The key manager then operates on B(E_i(M)) with its private key *i*, getting D_i(B(E_i(M))), but since D and E are inverses, and B and E commute, the result is B(M), which the key manager returns.
- The file system applies U to read M, and then discards B and U.

Any blind signature function will work, with straightforward modification, as a blind decryption scheme. However one could use more types of functions for blindable decryption, since there is no necessity to have a public key with which signatures could be verified. The ephemerizer's keys could even be blindable secret keys, in which case the file system would have to perform a blinded encryption request in order to encrypt M.

We present three examples of blind decryption protocols in section 7.

Our blind decryption protocol is extremely efficient. Performing a decryption request requires the ephemerizer to perform only a single private key operation, and is just a single message request/response. The key manager does not need to keep any state; just return a single response to a single request. The message only needs to contain a set of bits as big as a public key block (say, 4000 bits for RSA), and a key ID (perhaps 4 bytes). This easily fits into a single IP packet, and therefore there is no need to even create a TCP connection. Perhaps in practice, however, to get through firewalls, the protocol would have to be layered over HTTP.

Because the interaction is blinded, there is no need for authentication in either direction, so there is no need to establish a security association. (Certain operations, like request to create or delete a custom key, would require authentication, but the security of decryption, even with custom keys, does not require authentication of the party requesting the decryption.)

In contrast, the more traditional approach to having something like an external key manager decrypt E(M) would be to establish an SSL connection to the key manager, ask it to decrypt E(M), and have it return M. To contrast that with our protocol:

- this would involve at least two private key operations for the key manager (one to establish the SSL channel, and one to do the decryption),

- would not be as secure (since the key manager, without blind decryption, would directly see M which would enable it to directly decrypt some files on the backups),
- and would involve many packets, for establishing the TCP connection and the SSL handshake.

3.2 Our Time-Based Scheme

The basic idea is that files, when created, will have an expiration date. One or more ephemerizers will be used, which advertise public keys with expiration dates. Data with a particular expiration date will be encrypted with the ephemerizer public key with that expiration date.

Approach 1: Interaction per file

The straightforward approach would be to have each file encrypted with its own key K, and to store K, encrypted with the corresponding ephemerizer's key, in the metadata of the file. However, this would require the file system to interact with the key manager whenever each file was opened. It would also require a lot of storage in each file's metadata, since although K would be a secret key, of perhaps 128 bits, once K is encrypted with a public key, it will be much larger (say 4000 bits if it is an RSA key). And to be decryptable by *k* out of *n* ephemerizers, would require *n* times as much storage.

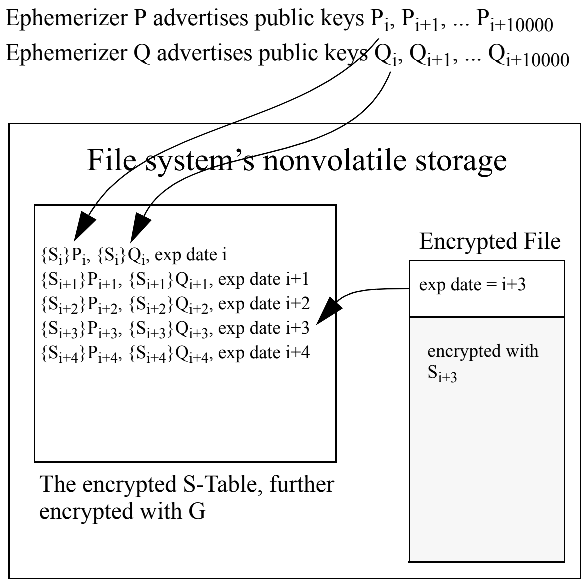
This is the approach that would be taken if this system were used for ephemerizing messages end-to-end, say if Alice creates a file that will expire at some point, that Bob is authorized to read. However, for a file system, we can do much better than this, with the following optimizations.

Optimization 1: Single interaction per expiration date upon boot

With this optimization, as we will see, instead of having the file system interact with the ephemerizer(s) every time a file is opened, the file system will need to interact with the ephemerizers, upon reboot, to build a table of (symmetric) master keys, one for each possible expiration date, which the file system will keep in volatile storage. Once the reboot process completes, the file system can act autonomously from the ephemerizers, and there is no further overhead from the ephemerization.

The file system generates a secret key S_i, for each expiration time *i*, and all files with the same expiration time will be encrypted with the same S_i. (See figure 1).

There is no loss of security in using the same S_i for all files with the expiration time, because at this layer the file system is trusted to read all files (except those that have expired), and to enforce access control. If the file system is not trusted by a user, the user can employ an additional level of encryption layered over the file system, and use the file system encryption only for the assured delete property.



Ephemerizer P advertises public keys P_i, P_{i+1}, ... $P_{i+10000}$
Ephemerizer Q advertises public keys Q_i, Q_{i+1}, ... $Q_{i+10000}$
File system's nonvolatile storage
$\{S_i\}P_i$, $\{S_i\}Q_i$, exp date i
$\{S_{i+1}\}P_{i+1}$, $\{S_{i+1}\}Q_{i+1}$, exp date i+1
$\{S_{i+2}\}P_{i+2}$, $\{S_{i+2}\}Q_{i+2}$, exp date i+2
$\{S_{i+3}\}P_{i+3}$, $\{S_{i+3}\}Q_{i+3}$, exp date i+3
$\{S_{i+4}\}P_{i+4}$, $\{S_{i+4}\}Q_{i+4}$, exp date i+4
The encrypted S-Table, further encrypted with G
Encrypted File
exp date = i+3
encrypted with S_{i+3}

Figure 1. The encrypted S-Table, using 1 out of 2 scheme, with ephemerizers P and Q

There is a one-to-one correspondence between file system secret master keys and ephemeral public keys kept by the ephemerizers. In other words, if there is an expiration time of November 8, 2010, then each ephemerizer used by the file system will have a public key that expires on November 8, 2010, and the file system will have a secret key S that expires on November 8, 2010.

The metadata for a file will contain an indication of which S the file has been encrypted with. For instance, the metadata might contain the expiration time of the file.

The file system contains, in volatile storage, a table we will call the "S-table", which contains the encrypted S's. If the S's are time-based, with granularity of a day, and 30 years' worth, there will be 10,000 S's.

The file system might lose the S-table after a crash, or certainly after a disaster such as the building burning down. So the S-table must be backed up, but in a way that will not interfere with assured delete.

This is accomplished by doing the backup of the S-table as follows. The file system encrypts each S in the S-table with the corresponding public key of the ephemerizer(s), encrypts once more with G, and writes the result onto nonvolatile storage.

In a 1 out of 3 scheme, for example, with ephemerizers P, Q, and R, with public keys for that date represented as P_i, Q_i, and R_i, respectively, the file system's nonvolatile (and backed up storage) will contain 10,000 entries, one for each expiration date, that each look like:

- $\{S_i\}P_i$, $\{S_i\}Q_i$, $\{S_i\}R_i$

The encrypted S's will be further encrypted with G, the overall file system secret.

When the file system is rebooted, a system administrator obtains G. Then G is given to the file system. The file system retrieves the doubly encrypted S-table from stable storage, and decrypts it with G.

Now the file system has (in volatile storage) the encrypted S-table, which is now encrypted with decryption keys known to the ephemerizers.

The file system could, at this point, decrypt all the S's, and keep the entire unencrypted S-table in volatile storage. Or it could decrypt each S the first time a file with that expiration time is accessed. This would require, eventually, 10,000 interactions with the ephemerizer in order to decrypt all the S's after a crash.

Preferably, the S-table would be kept in a tamper-resistant cryptographic accelerator coprocessor. Such devices are available that operate at disk speeds, so that the extra level of encryption will not impact performance, and the tamper-resistance will keep the S-table safe. The coprocessor might retain the S-table across file system crashes, but should erase its state if it is tampered with, or moved. If the coprocessor dies, or has destroyed its state, it is easy to recover the S-table from nonvolatile storage, and with the help of the ephemerizers.

Optimization 2: Single interaction upon boot

We can further optimize the performance by making each S_i be a one-way cryptographic hash of S_{i-1}. An alternative to one-way hashes that would enable the file system to recover all subsequent S's from the first-to-expire S is to encrypt S_{j+1} with S_j.

With this optimization, when the file system is rebooted after a crash, it only needs to interact with the ephemerizer(s) once, to decrypt the earliest-to-expire S, and then locally derive all the remaining S's.

The file system still keeps 10,000 S's (assuming there are files that will not expire for 30 years, and a granularity of one per day). The S-table looks the same as it did for Optimization 1, i.e., that each S_i is encrypted with the corresponding public keys of the ephemerizers.

With optimization 1, each of the 10,000 S's will need to be independently decrypted, with the help of the ephemerizer(s) each time the system reboots. With optimization 2, only the S that will expire first need be decrypted.

Note that if the file system does not crash for some time, it must discard S's from volatile storage when the expiration date occurs.

3.2.1 Changing a file's expiration date

What if the file system wishes to change a file's expiration time? It is relatively easy to extend it. The simplest way would be to re-encrypt the file with an S with a later expiration time.

If the file is very long, however, and if expiration time extension is a common operation, then this can be accomplished more efficiently by having each file be encrypted with its own secret key K, and have K encrypted with the appropriate S_i be associated with the file, for instance, by including it in the metadata for the file.

To extend the life of a file, K need only be re-encrypted using a later S, so the encrypted file data need not be modified.

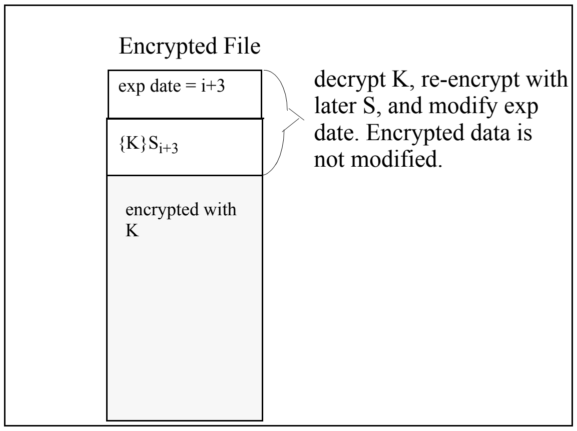

Encrypted File
exp date = i+3
{K}S_{i+3}
encrypted with K
decrypt K, re-encrypt with later S, and modify exp date. Encrypted data is not modified.

Figure 2. Delaying a file's expiration: encrypting data with file key K, encrypting K with S_i

It is not possible, with our time-based scheme, to preserve the assured deletion guarantees if the file's expiration time is made *earlier* than its original expiration time, because the ephemerizer cannot delete a key prematurely, since others might be depending on that key. So even if the file, and its metadata, is deleted from the file system, it would still be recoverable from backup store, as long as the associated S was still recoverable, which it will be, as long as the associated ephemerizer public keys are still available, and a copy of the file encrypted with the later S is still available on some copy of the backup.

3.2.2 Reclaiming space

Garbage collection to reclaim space is easy. In the background, a garbage collector can look for files that have expired, as indicated by the expiration date in the metadata, and free the space.

3.2.3 Adding an ephemerizer

Suppose one or more of the ephemerizers a file system was using was known to permanently fail, for instance, by losing all its ephemeral keys. As long as *n*-*k*+1 of the ephemerizers don't fail simultaneously, or as long as the file system has retrieved the S's (the master class secrets), it is always possible for the file system to add ephemerizers.

It accomplishes this by re-encrypting the table of master class keys, breaking each S_i into the desired number of shares, and encrypting each share with a corresponding ephemerizer key for that class, encrypting the result with G, and storing on backup media.

3.2.4 Security and performance properties of the time-based scheme

Upon reboot of the file system, there is a tiny amount of overhead to interact with ephemerizers and decrypt the S-table. Once the S-table is decrypted, it is accessible locally to the file system, and there is no further performance overhead due to the ephemerization, above what would be needed for any encrypted file system.

The interaction with a ephemerizer divulges no information to the ephemerizer, so there is no need for the file system to authenticate the ephemerizer. Even if the ephemerizer colludes with the storage service, and therefore obtains all information that the file system stores in non-volatile storage, together with the ephemerizer keys, it will be impossible to recover the data without also knowing G, which requires stealing a quorum of system administrator secrets.

There is also no reason for the ephemerizer to authenticate the file system. In fact, the interaction could be done anonymously [10]. There could be a set of ephemerizers available on the Internet, and any file system could choose to use any of them, encrypt with their public keys, and then request decryption blindly and anonymously when needed.

One very convenient aspect of the time-based scheme is that no action need be taken by the file system when a file expires. If the rule is that, say, results of some medical test be maintained for a month, and then destroyed after that, the file will have its expiration time chosen at creation time, and then when it expires, the file will become unreadable, with no further effort by the file system.

What happens if the file system has an incorrect time? If the file system thinks the time is far in the past, say a year old, the file system may, during the time its clock is set incorrectly, create encrypted files that will prematurely expire, possibly even become immediately unrecoverable once the file system loses volatile storage. However, this error is likely to be caught as soon as the file system boots, since the ephemerizers will not be advertising ephemeral keys with past expiration times, and the file system will not be able to decrypt any of the S's that are encrypted with keys that the ephemerizers have discarded. This would make it immediately apparent to the rebooting file system that its clock is very different from the ephemerizer's clock.

Setting the time to be too old at the file system will not cause files that have expired to become readable.

If the file system thinks the time is far in the future, then if the user interface specifies expiration times as offsets from "current" (as in "two weeks from present") it will create files that will expire later than wanted, and it will think that files that should be accessible, because they have not expired, are not recoverable. The file system will not attempt to decrypt S's from times in the past.

It would be easy to have the file system specify its notion of the date in the message to the ephemerizer, and/or have the ephemerizer specify the date in the reply. This would be a hint that someone has the time wrong so an alert could be raised.

In practice, setting correct time (approximately) is not that difficult.

What happens if the ephemerizers have the time wrong? Again, time need only be approximate (say within a day). If an ephemerizer thinks the time is in the future, it may prematurely discard keys that should not have expired. If an ephemerizer thinks the current time is in the past, then it may not discard keys on time. With a quorum scheme of *k* out of *n*, it would require more than *k* of the ephemerizers to have clocks set backwards in order for an expired file to be recoverable, and *n*-*k*+1 of them to have clocks set forwards in order for an unexpired file to be unreadable.

Note that booting a ephemerizer with an old time will not cause it to remember keys that were discarded while it was operating before with a correct clock.

What happens if various keys are compromised or lost? If G is stolen, along with the encrypted data, then the thief can read all the data in the file system, just like any system administrator armed with backup data and G. However, like the system administrator, the thief will not be able to read data that has been assuredly deleted.

If an ephemerizer's long term key is lost (but not stolen), then it can not certify any new keys, but the current ephemeral keys (if they are not lost) will still be operational. However, the ephemerizer can get a new long-term key certified through the PKI and continue operating.

If an ephemerizer's long term key is stolen, then the thief might be able to trick a file system into encrypting with bogus keys (which is a denial of service attack but will not disclose data), or with keys that the thief knows the private keys for, and which the thief will not discard. This is exactly the case of a dishonest ephemerizer. It might cause data that should be deleted to be recoverable, but it requires a quorum of colluding dishonest ephemerizers.

If an ephemerizer loses its ephemeral keys, then as long as a quorum of ephemerizers still remain operational, the data that has not expired is still readable. Robustness can be raised at any time by adding new ephemerizers.

If an ephemerizer's ephemeral keys are stolen, then this again is the same as the case of a dishonest ephemerizer. The thief will only be able to read data if it colludes with a quorum of other dishonest ephemerizers (assuming a quorum is more than 1), obtains G, and obtains the encrypted data.

We do require that the public key cryptography used for the blind decryption be robust against a chosen ciphertext attack. It is believed that RSA and elliptic curves (for both of which we present blind decryption schemes in Appendix A) are secure against chosen ciphertext.

3.3 Our Individual File On-Demand Delete Scheme

In this scheme, like in the time-based scheme, there will only need to be a single interaction with the ephemerizer(s) after the file system boots. Once the ephemerizer(s), upon reboot, decrypts a single quantity for the file system, the file system operates autonomously, until it crashes.

In this scheme, each file is encrypted with its own secret key F. The file system maintains a file, which we will call the F-table, of keys for every file in the file system. So, for instance, if the file system has a million files, there will be a million entries in the F-table.

In this scheme, the ephemerizers maintain (at least) two keys for this file system. If the ephemerizer is acting for multiple file systems, it will have a different pair of keys for each file system. In this way it is up to the file system to command the ephemerizer to generate new keys, and when to discard old keys. For simplicity, let's assume just two keys. We will call the pair of keys the "current" key and the "previous" key.

The file system keeps the F-table in volatile storage. However, every time the F-table is modified (due to creating a new file or securely deleting an existing file), the file system chooses a random secret K, encrypts the F-table with K, and stores K encrypted with the ephemerizer's current key. If there are, for instance, three ephemerizers, and we wish to use a 1 out of 3 scheme, then K would be stored 3 times, each time encrypted with a different ephemerizer's "current" key. The encrypted K is further encrypted with G, the overall file system secret.

It is possible to avoid modifying the F-table when files are created by having the file system precompute a batch of F's in advance. In this way, many files can be created without changing the F-table. It is only when there are no unused F's, or when an F is deleted, when the F-table must be modified. Outside of the encrypted F-table, the file system will need to keep track of which F's are available for new files.

When the encrypted F-table (together with the encrypted K) is migrated to replicated non-volatile storage, then the ephemerizers can be informed that they can delete the previous public key, and they will generate a new public key and give it to the file system.

The reason it is important to have key rollover and key deletion under the control of the file system, rather than having it done on a predetermined schedule, with keys shared by many organizations, is that one organization's file system might be down for an extended period of time, perhaps due to a natural disaster. If the ephemerizers delete keys on a schedule, in particular, quickly enough to ensure that a deleted file will become unreadable after a relatively short window, then if a file system were down for longer than that window, all of its data would become forever unrecoverable.

On the other hand, if the ephemerizers rolled over keys on their own schedule (say once per week), then the ephemerizer would not need to use custom keys for

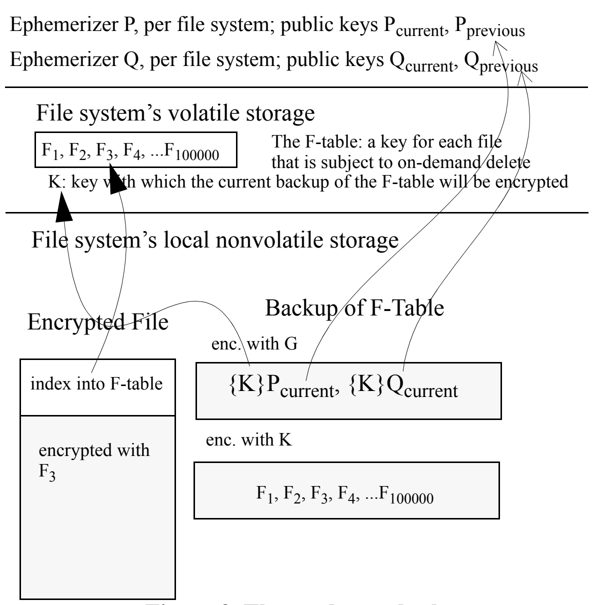

Ephemerizer P, per file system; public keys $P_{current}$, $P_{previous}$
Ephemerizer Q, per file system; public keys $Q_{current}$, $Q_{previous}$
File system's volatile storage
$F_1, F_2, F_3, F_4, \ldots F_{100000}$
The F-table: a key for each file that is subject to on-demand delete
K: key with which the current backup of the F-table will be encrypted
File system's local nonvolatile storage
Encrypted File
Backup of F-Table
enc. with G
index into F-table
$\{K\}P_{current}$, $\{K\}Q_{current}$
enc. with K
encrypted with F_3
$F_1, F_2, F_3, F_4, \ldots F_{100000}$

Figure 3. The on-demand scheme

each file system; instead, as with time-based keys, all file systems could use the same "current" and "previous" keys, and again, could communicate with the ephemerizer anonymously. If the scalability of key-sharing across customers were considered an important enough advantage of the on-demand scheme, then the disadvantage above (the danger of losing all data if a file system were down for longer than the ephemerizers' key rollover window) could be lessened by allowing some sort of emergency procedure in very rare cases, for retaining keys longer than the normal rollover period. Customers wanting on-demand with shared keys would need to understand that key rollover might be extended sometimes due to emergencies of other customers.

Note that although it is not strictly necessary, it is a performance advantage to store K (the secret key with which the F-table on nonvolatile storage is encrypted) in volatile storage. That way it is not necessary to decrypt the entire F-table and store it in volatile memory. Instead, when needed, individual F's can be decrypted using K. A new F can be created by encrypting the newly created F with K, and storing it on nonvolatile storage.

However, if an F is deleted from the F-table, a new secret K' must be generated, and the entire F-table must be stored, encrypted with K', on local nonvolatile storage, and then the previous K must be forgotten.

3.3.1 A Downside of the On-Demand scheme

On the surface, the on-demand scheme is attractive. It is very scalable. The ephemerizer only needs to store 2 keys to support a file system that might have thousands of users and millions of files. It can support both pre-determined expiration times (with the file system needing to find and delete expired F's from the F-table), and on-demand delete of individual files.

However, compared with the time-based expiration scheme we present in section 3.2, there is a potential serious reliability issue. With the on-demand scheme, there is a window from the time a file is securely deleted, until the key with which the most recent F-table containing that file's key is discarded by the ephemerizer. Let's say that window is 3 days. So the file is not really securely deleted until that window has passed. Which means we might want that window to be small.

However, suppose the file system were silently compromised, and it took weeks to notice it. Somehow the corrupted file system was corrupting the F-table, and keys for important files that were not recently accessed, were corrupted. Once the ephemeral key with which the last good copy of the F-table is discarded, there is no way to recover those files.

In contrast, the time-based scheme is much safer. As long as the file system is working properly when a file is created, and the file (and its encrypted key) are safely backed up, a subsequent corruption of the file system will not harm that file. It is always possible to restore the state of the world as of some time past, and all files that existed then, and have not yet expired, will be recoverable. (If the files have expired, regardless of what time the file system thinks it is, they cannot be recovered because the ephemerizers would have deleted their expired keys).

Some products that do key management for storage, which employ a key manager station (KMS) that keeps a database of keys, are similar to our on-demand scheme, and have the same disadvantage. In such products, there are often multiple KMS's which synchronize with each other. If the database of keys is backed up outside the KMS's on backup media so that the KMS database can be recovered if all KMS's fail, then there is no assured delete. If the only copies of the key database are in the KMS's, then a single KMS that tells the others to delete keys can cause unintended key deletion, which can result in data becoming unrecoverable, with the problem possibly not being detected for a long time.

3.4 Custom class of file

It might be desirable to have some individual secrets maintained at the ephemerizer. This might be useful, for instance, to protect a set of data that might be captured by an enemy. Rather than destroying the data, by having the ephemerizer destroy the key, if the ephemerizer keeps a custom public key for that set of data, the volatile storage at the file system can be destroyed, and the ephemerizer can be informed that it should lock the key, so that only some extraordinary mechanism could be used to unlock the key; for instance, by having a high-ranking person personally visit the ephemerizer site.

Such a key must not be shared across clients, because it must be lockable, or deletable, on demand. As with the time-based scheme, the file system invents a secret S when the class is first created. Then it requests each ephemerizer it wishes to use to create a corresponding public key for that class.

Why does this scheme not suffer from the problem described in section 3.3.1? The reason is that in this scheme, if the file system is not compromised when the class key for a class of files is created, and that class key is copied, encrypted with the ephemerizers' keys, onto stable replicated storage, then as long as a quorum of ephemerizers still retain their key associated with that file class, the secret file class key, and all data encrypted with that key, can be recovered from backups, regardless of any subsequent compromise of the file system. Deletion of a class key at each ephemerizer requires a very conscious and auditable action. As long as reasonable human mechanisms are in place to ensure that class keys are only deleted when, for instance, a quorum of system administrators agree, classes will not accidentally become unreadable.

With the on-demand scheme, the file system is trusted to maintain and modify the F-table, and if it writes out a corrupted F-table, this may be undetectable until it is too late (once the ephemerizers delete the key with which old backups of the F-table have been encrypted, there is no going back).

3.5 Combined File Types

It is straightforward for the file system to maintain files with the different forms of assured delete, together with files that cannot be assuredly deleted.

- Files of assured delete classes (time-based or custom) each are encrypted with a class master key, and the class master key is encrypted on backup media with a corresponding ephemerizer public key, and then with G.
- Files without assured delete are also encrypted with a class master key S, but this S is only stored encrypted on backup with G (not with an extra wrapping with an ephemerizer public key).
- Files that are capable of being on-demand deleted are encrypted with keys F, stored in an F-table, and the F-table is backed up by encrypting it with a randomly chosen key K, then encrypting K with the "current" public key of the ephemerizer(s), and encrypting the encrypted K with G.

It is also possible to have a file fit into more than one category by encrypting the file key multiple times, once for each of the categories. For instance, if a file has an expiration time, but also should be on-demand deletable, the file should be encrypted with K, and then K should be encrypted with both an F from the F-table and the appropriate class key S. The metadata would indicate which F and which S would need to be used to decrypt K. If any of the keys upon which the file depends become unavailable (i.e., if it expires, or if it is assuredly deleted before the expiration date), the file becomes unrecoverable.

Similarly straightforward to implement is the OR of one of more categories. For instance, if a file should be recoverable if either of two classes is still recoverable, then the file's key K is stored twice in the metadata; once encrypted with the first class's S key, and once encrypted with the second class's S key.

4. Layering with an end-to-end encryption scheme

Our scheme is transparent to storage (which is unaware of the encryption). Our scheme does protect data on backup media, since all the file system data will be encrypted. It is also transparent to clients, other than the ability for clients to specify, when creating a file, that the file should be assuredly deletable. One reasonable interface is to allow a directory (and descendents) to be associated with an ephemeral custom key, and/or to allow a particular file to be created with an expiration date, and/or to specify that a particular file should be on-demand deletable.

Our scheme alone does not provide end-to-end encryption. In other words, if Alice creates a file that she wants only Bob and Carol to read, unless there is a layer of encryption above what our system provides, the file system will be able to read the file, and Alice will have to trust the file system to enforce the ACL (that specifies only Bob and Carol should be able to read the file).

If instead Alice would like to do end-to-end encryption, she would do it exactly as she would in a normal file system, by encrypting the file with her own chosen key J, and storing J encrypted with each authorized recipient's public key along with the encrypted file, in a way transparent to our layer of the file system. Our layer would see everything Alice stores (including the encrypted J) as data. If Alice wants to store a file for Bob and Carol that will expire, Alice stores the file as she would ordinarily, for Bob and Carol, and specifies to our level the file system, the expiration date. Our level of the file system chooses its own file key K, encrypts everything Alice would store with K, and stores K encrypted with the corresponding class key, as what our level of the file system interprets as metadata for the file.

When Bob asks to retrieve the file, the file system enforces the ACL, notes Bob is authorized, decrypts K with the class key, decrypts the "data", and sends it all to Bob. Bob interprets what he receives as metadata (J encrypted with his public key), followed by the file encrypted with J.

The data will be doubly encrypted, but secret key encryption can be done without loss of performance today, especially with hardware accelerators.

5. Building an Ephemerizer

Ideally an ephemerizer would contain a tamper-resistant component that generates the ephemeral keys and does decryptions, and never divulges the private keys. The remainder of the ephemerizer functions, e.g., responding to decryption commands, or commands for creating custom keys, can be implemented on a general purpose machine. For time-based keys, once a day the ephemerizer can overwrite (in the tamper-resistant portion) yesterday's key with a new key.

6. Protecting the master class keys

If the file system is built on a general purpose computer, there is the danger that it could become compromised, and divulge the master class secrets. Therefore, it might be desirable to structure the file system so that the keys are kept in a tamper-resistant coprocessor. To recover from a disaster, G and the backup of the keys is input into the secure coprocessor, which recovers the class keys (with the help of ephemerizers). Let's call this trusted portion the KM (key manager).

The KM needs no permanent state, so if the KM fails, a new one can be substituted, and its database of keys can be recovered from G, backup tapes, and interaction with ephemerizers.

The file system communicates securely with the KM, either because of physical proximity, or through an established security association, and requests decryption, either of the file key when the file is opened, or even the data (assuming a high bandwidth path between the KM and the file system.)

The KM could be a cryptographic accelerator card attached to the file system, or it could be a free-standing machine in a data center, and it could act on behalf of multiple file systems. It could also contain a hardware random number generator, with which it might be able to generate better file encryption keys than the file system built on a general purpose machine.

7. Blind Decryption Functions

In this section we present three blind decryption functions. The first two use public keys for encryption, so only decryption needs to be blinded. The third one uses secret functions for both encryption and decryption, so encryption (as well as decryption) requires blinded interaction with the ephemerizer.

7.1 RSA Keys

This form is almost identical to blind signatures with an RSA key [6]. The ephemerizer's public RSA key is (e,n). We assume all arithmetic is done mod n, so for readability, we leave out "mod n".

The file system encrypts M using the ephemerizer's public key by computing M^e.

The file system, FS, gets the ephemerizer, KM, to blindly decrypt M^e, (i.e., retrieve M) by doing the following:

1. FS chooses random R
2. FS computes R^e.
3. FS computes $M^e * R^e$ and sends that to KM
4. KM raises $M^e * R$ to d to obtain $M^{ed} * R^{ed} = M * R$.
5. FS divides by R to obtain M.

The file system never writes R onto nonvolatile storage.

7.2 Blind encryption/decryption with a Diffie-Hellman public key

This form of blind decryption does not have a similar blind signature scheme, and it works with any Diffie-Hellman group, including ECC. Assume that the ephemerizer has a public Diffie-Hellman key, g^x, where the group **G**, including g and the order of **G**, is known. The private key is x.

To encrypt message M with the ephemerizer's public key, the file system performs the following:

1. FS chooses random y, and computes g^y and g^{xy}. This is done by raising the publicly known base g

to y, and the ephemerizer's public Diffie-Hellman key g^x to y.

2. FS uses g^{xy} as a secret key to encrypt M, obtaining $\{M\}g^{xy}$. FS saves $\{M\}g^{xy}$ and g^y, and discards y and g^{xy}.

To retrieve M, the FS asks the ephemerizer to blindly decrypt $\{M\}g^{xy}$ by doing the following:

1. FS knows $\{M\}g^{xy}$ and g^y.
2. FS chooses random z, and compute's z's exponentiative inverse z^{-1}. (What we refer to as exponentiative inverse is more conventionally known as z's multiplicative inverse modulo the order of **G**.)
3. FS computes $(g^y)^z$, sends g^{yz} to ephemerizer.
4. Ephemerizer applies its private key (x) and sends to FS: g^{xyz}
5. FS raises g^{xyz} to z^{-1} to obtain g^{xy}, with which it can decrypt $\{M\}g^{xy}$.

7.3 Blind encryption/decryption with a secret encryption function

This form of blind decryption does not have a similar blind signature scheme (and couldn't, because there is no public key with which to validate a signature). We use exponentiation mod p, a blinded version of Hellman-Pohlig [15]. Instead of a public key, the ephemerizer has two secret numbers, x and x^{-1}, which are exponentiative inverses mod p. "Encrypt" will be done by exponentiating with x, "decrypt" with x^{-1}. Blind encryption in this scheme, as with blind decryption, requires the involvement of the ephemerizer.

Blind encryption requires authentication of the ephemerizer. Unlike the other schemes, where the public encryption key is certified by the ephemerizer's long-term key (so there is implicit authentication of the ephemerizer), in this scheme there is no certified public key, so authentication must be done explicitly during the encryption request. The risks of asking for encryption from the wrong party are:

- denial of service; there is no way of knowing whether the encryption worked
- using something that does not throw away keys

Authentication of the ephemerizer can be done through any number of conventional ways, for instance, using SSL.

To get the ephemerizer to blindly encrypt M:

1. Alice chooses random z, and its exponentiative inverse z^{-1}.
2. She computes M^z, sends it to the ephemerizer, with the request to "encrypt".
3. The ephemerizer applies x and returns M^{xz}
4. Alice applies z^{-1} to obtain M^x.

To get the ephemerizer to blindly decrypt M^x:

1. Alice chooses random y, and its exponentiative inverse y^{-1}.
2. She computes M^{xy}, sends it to the ephemerizer, with the request to "decrypt".
3. The ephemerizer applies x^{-1} and returns M^y
4. Alice applies y^{-1} to obtain M.

8. Conclusions

We presented three schemes for supporting assured delete; time-based, custom classes, and individual file on-demand. These schemes can be combined. In the same file system, some files can be stored with expiration times, others in a class with a custom keys, others that can be deleted individually, on-demand, in an assured manner, and still others that have no assured deletion properties, so that even if deleted, they would be recoverable as long as they still exist on backup media.

The schemes can also be nested, e.g., by having a directory encrypted with a custom key, and individual files in that directory also having expiration dates.

For data with the securely deletable property, the file system employs the services of remote ephemerizers, whose sole purpose is to create, certify, advertise, and manage ephemeral keys, and to perform decryptions with ephemeral keys upon request. A file is recoverable if and only if someone has the encrypted file, the overall file system secret G, and a quorum of ephemerizers still retain the key associated with that class of file.

We achieve robustness without any copying of ephemeral keys by using multiple independent ephemerizers with independent keys. This makes an ephemerizer sufficiently inexpensive, and easy to manage, that organizations might choose to run all or some of the ephemerizers they depend on themselves.

The communication with the ephemerizer is done through "blind decryption", and the ephemerizer gains no information from this exchange. Therefore the ephemerizer can be a relatively untrusted third party, as can the organization that manages the non-volatile storage. Even if the ephemerizers and the storage organization collude, they will not be able to read the data. The cryptographic algorithm used for the public keys of the ephemerizer needs to be resilient to chosen ciphertext attack.

In all our variants, the only time it is necessary for a file system to communicate with an ephemerizer is after the file system reboots. The ephemerizers will need to do one decryption to unlock all the time-based-expiration files on the file system, one decryption to unlock all the on-demand-deletable files on the file system, and one decryption for each file class with a custom key.

Thus there is minimal performance overhead of our scheme beyond simple encrypted storage. There are cryptographic accelerators that can be used in the file system that will work at disk speeds. And other than an initial network-based interaction with a ephemerizer to recover the set of file system master keys for file system classes, there is no further interaction with the ephemerizers after the file system boots.

This system assumes the file system is trusted to read the data. If this is not a desired property, then an additional level of encryption can be done at the user level, while the assured delete is still done at the file system level.

Even if end-to-end encryption is layered over the file system, doing ephemerization at the file system level, as we do in this paper, achieves great performance advantages.

9. Bibliography

1. Anderson, R., “Two remarks on public-key cryptology”, Invited lecture, Fourth ACM Conference on Computer and Communications Security, April, 1997.
2. Ateniese, M., Fu, K., “Improved Proxy re-encryption schemes with applications to secure distributed storage”, NDSS 2005.
3. Blaze, M., “A cryptographic file system for Unix”, CCS, 1993.
4. Boneh, D., and Lipton, R., “A Revocable Backup System”, Usenix Security Symposium, 1996.
5. Camp, L. J. (1997, February). Web security & privacy: An American perspective. ACM SIGCAS CEPC ’97 (Computer Ethics: Philosophical Inquiry).
6. Chaum, D., “Blind signatures for Untraceable payments”, Advances in Cryptology - proceedings of Crypto 82, 1983.
7. G. Di Crescenzo, N. Ferguson, R. Impagliazzo, and M. Jakobsson, "How To Forget a Secret.", STACS '99. LNCS 1563. Springer-Verlag, 1999. pp. 500--509.
8. Damiano, E., De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi S., Samarati, P., “Key Management for Multi-user Encrypted Databases”, StorageSS’05, 2005.
9. Diffie, W., and Hellman, M., “New directions in cryptography”, IEEE Transcations on Information Theory”, 1976.
10. Dingledine, R., Mathewson, N., Syverson, P. “Tor: The Second-Generation Onion Router”. Usenix Security Symposium, 2004.
11. Disappearing, Inc., web site: http://www.specimen-box.com/di/ab/hwdi.html
12. Fu, K., “Group sharing and random access in cryptographic storage file systems.” Master’s thesis, MIT, 1999.
13. Goh, E., Shacham, H., Modadugu, N., and Boneh, D., “SiRiUS: Security Remote Untrusted Storage”, NDSS, 2003.
14. Gutman, P, “Secure Deletion of Data from Magnetic and Solid-State Memory”, Usenix Security Symposium, 1996.
15. Hellman, M, and Pohlig, S., U.S. Patent 4,424,414, 1984.
16. Kallahala, M., Swaminathan, R., Fu, K., “Plutus: Scalable Secure File Sharing on Untrusted Storage”, FAST03, Usenix Conference on File and Storage Technologies, 2003.
17. Kher, V., Kim, Y., “Securing Distributed Storage: Challenges, Techniques, and Systems”, StorageSS’05, 2005.
18. Naor, M., and Yung, M., “Public-Key Cryptosystems Provably Secure Against Chosen Ciphertext Attacks”, 22nd Annual ACM Symposium on Theory of Computing, 1990.
19. Network Computing Reports, “Tape Encryption Devices: Host-based vs. Appliance”, Nov 24, 2005.
20. Peterson, Z., Burns, R., Herring, J., Stubblefield, A., Rubin, A., “Secure Deletion for a Versioning File System”, FAST ‘05: 4th Usenix Conference on File and Storage Technology, 2005.
21. Perlman, R., “The Ephemerizer: Making Data Disappear”, Journal of Information System Security, 2005.
22. Perlman, R., “Secure Deletion of Data”, 3rd International IEEE Security in Storage Workshop, 1995.
23. Rivest, R., Shamir, A., and Adleman, L., “A method for obtaining digital signatures and public-key cryptosystems”, Communications of the ACM, 1978.
24. Russinovich, M., Solomon, D., “Microsoft Windows Internals 4th Edition”, 2005.
25. A. Shamir, ``How to share a secret'', CACM, Vol. 22, Nov. 1979.