SEFS: Security Module for Extensible File System Architectures

Luís Ferreira[†], André Zúquete[‡] and Paulo Ferreira[‡]
[†] ISEL / INESC, [‡] IST / INESC
{luis.ferreira, andre.zuquete, paulo.ferreira}@inesc.pt

Abstract

Data security is a fundamental issue in modern computer systems. In particular, data storage systems are frequently subject to attacks and so need protection. Typical storage systems rely on access control mechanisms, either physical or logical, to prevent unauthorized users from accessing stored data. However, such mechanisms are useless against non-ethical attitudes taken by privileged users, like system administrators. Thus, the ultimate solution for ensuring the privacy of sensitive data is to use cryptographic techniques.

This article describes the design and implementation of a security module, for extensible file system architectures (SEFS), that enforces file security using cryptographic techniques. The security module provides privacy facilities at file granularity. The place where the module is inserted in the file system architecture maximizes the transparency of its operation: existing applications can transparently work with encrypted files, which may be physically stored anywhere (fixed / removable local / remote devices). Furthermore, the proposed solution allows a flexible and discretionary way of marking files as secure, not imposing any modification in their location. A prototype was implemented for the Microsoft Windows 95 architecture, but a similar solution may be adopted for other Windows systems, such as Windows NT or Windows 2000. Preliminary performance evaluations show that the cost of security provided by SEFS is acceptable and mostly due to cryptographic algorithms.

1 Introduction

Data security is a major concern of today's information systems, both for transient and permanent data. Since file systems are the main mechanism provided by operating systems for permanent data storage, the security of long term data implies an effective protection of files.

The traditional policy for protecting files relies on two mechanisms: physical enclosure of the storage media, and logical access control barriers managed by the operating system (e.g. ACL monitors[6, 20, 21]). However, these mechanisms may fail in many different scenarios, for instance: (i) free physical access to a stolen laptop; (ii) access to removable storage devices, e.g. a backup CD-ROM or a floppy disk; (iii) non-ethical access by file server's administrators; or (iv) unauthorized accesses to users' files by an attacker capable of impersonating them. Whence, access control mechanisms, either physical or logical, aren't bulletproof, and that may be a major concern for people dealing with sensitive data.

To make it even worse, current distributed file systems exacerbate some of the previous problems and introduce new ones. On distributed file systems people have to trust on: (i) the physical security of remote file servers, (ii) administration teams (that may blur individual non-ethical attitudes), (iii) the capability of remote authentication mechanisms to prevent personification, and, finally, (iv) they have to trust that attackers cannot understand or modify data flows between the client node and the file server.

All these security issues can be overcome by locally applying user-driven cryptography, the ultimate strong data protection mechanism in multi-user computer systems. For the particular case of file systems, sensitive files should be stored and transferred encrypted, and should only be decrypted just before being used by legitimate users' applications (and encrypted immediately after being used). This solution ensures the protection of files as long as legitimate users are the only ones controlling the encryption / decryption engine (which depends on users' secret keys).

The main goal of our work is to protect sensitive data, stored either in local or remote files, by adding security properties, namely cryptographic privacy, to files and directories. These security properties are specifically requested by users for specific files or directories, and transparently enforced using cryptographic techniques controlled by user's cipher keys. To simplify the manipulation of secure files by applications, they are ciphered and deciphered transparently at file system level by the SEFS module (and not with a user-level tool, e.g. PGP [24]). The names of secure files and directories are also hidden using cryptography. Names are used as well to store per-file security-related metadata used by SEFS.

The transformation of ordinary files and directories into secure ones is handled by specific SEFS tools or plug-ins for popular file system browsers (e.g. Explorer). Secure files are protected by a secret key using symmetric cryptographic algorithms, and, unlike EFS [5], no particular mechanisms are provided for administrative recover of cipher keys (as we want to protect users from non-ethical administrators).

The SEFS module lies within the file system architecture, below applications' API for file manipulation, and uses underlying file systems structures to store secure files. Many existing file system types can be used to store secure files (FAT, FAT32, NTFS, CIFS, etc.). Unlike CFS [1], SEFS allows users to freely sprinkle secure files within those file systems. Extending file system's functionality became possible with more recent file system architectures, that offer the possibility to intercept file system requests. This interception mechanism allows the development of new file system features to improve current systems, instead of developing completely new ones, and we used it to implement SEFS. A first prototype of the SEFS module was developed for the Microsoft Windows 95 architecture, but a similar solution can be adopted for other Windows systems, like Windows NT or Windows 2000.

Preliminary performance evaluations of read / write operations show that the cost of security is acceptable, although significant. The performance degradation is mostly due to cryptographic transformations, which are responsible for 78 to 95% of the total SEFS overhead in read / write operations.

This paper is organized as follows. In §2 we describe the general architecture of SEFS. In §3 we describe the implementation of SEFS. In §4 we evaluate the performance of SEFS. In §5 we present other solutions and we compare them with SEFS. Finally, in §6 we present the conclusions of the paper.

2 SEFS – Security Extension for File Systems

In this section we describe the general architecture of SEFS. First we present the main design goals observed in the design of SEFS and we discuss its architecture; then we describe how SEFS handles cipher keys, and finally we highlight some of the high-level issues in the development of SEFS and the solutions proposed.

The SEFS module was designed in order to achieve four major goals:

1. Provide transparent file system security (i.e. cryptographic transformations of sensitive data) for existing applications with discretionary and easy-to-use file granularity;
2. Use existing file systems, both local or remote, to store secure files and directories mixed with "normal" ones;
3. Maximize users' trust in its security features; and
4. Have the lowest possible impact on the overall performance of the file system.

The first goal is fundamental for simplifying the daily use of secure files, as well as the administration of file systems. Secure files are just ordinary files, but their data and some of their metadata is encrypted. The visibility of secure files and their encrypted data (and metadata, e.g. their name) depends only on the standard access control of the underlying file system. This way, normal administrative procedures, like backups or volume relocations, can still deal with secure files.

Users may specify, with the minimum inconvenience, which files have security properties and which don't. Security properties of directories can be inherited by subdirectories or files, either existing or newly created. Furthermore, secure files may exist anywhere, and not only on special directories or volumes, in order to simplify the management of users' securely stored data.

SEFS provides a "*decrypted view*" over local or remote file systems, allowing authorized users to transparently manipulate the decrypted contents and metadata of secure files (see Figure 1). Each decrypted view is controlled by a set of keys known by a user, the authorized one for that view. Secure and not secure files may coexist in the same volume. Similarly, different users may store secure files in the same volume, but it is assumed that in each machine there is at most one user locally authenticated. Other users can, at the same time, be authenticated on other machines and, throughout their own SEFS, remotely access secure files in the same volume.

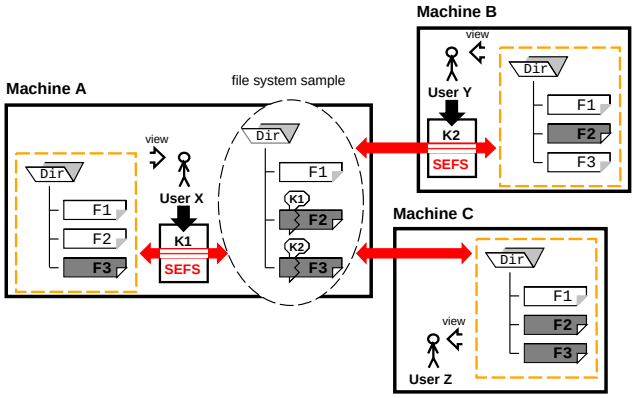

Figure 1: *Example of SEFS decrypted views (dashed boxes) from a file system sample (dashed oval) shared by different machines and users, either using or not SEFS. Those using SEFS have a different view of the original file system contents, and the view depends on the cipher keys provided locally to SEFS. Encrypted files, represented as grey boxes, can only be seen decrypted by users using SEFS and the correct cipher key ($K1$ for* F2 *and $K2$ for* F3*).*

Microsoft Windows allows three ways of extending the system functionality: (i) by extending particular applications' functionalities by means of explicit hooks to plug-ins, (ii) by wrapping legacy code, or (iii) by extending, whenever allowed, the core functionalities provided by the operating system. The first solution is clearly not well suited for providing transparent security features for a broad range of existing applications. Wrapping legacy code is a powerful but delicate approach that should be avoided if one has the possibility of extending the system functional-

ity. Therefore, we chose to implement our secure file system as a module within the Windows file system architecture. This module was complemented by several plug-ins to be integrated in file system browsers (e.g. the Explorer) in order to explicitly manage security attributes of files and directories, and the operational parameters of the SEFS module.

Our three first goals constrained the exact location of the SEFS module within the file system stack. To use file / directory abstractions and existing file system structures, one should insert SEFS above the lowest level of the file system stack where storage details are blurred (like the Vnode level in UNIX file systems[11]). Furthermore, to increase users' trust in the security features one should ensure that sensitive data, once decrypted, is only available to authorized applications (users). This means that all sort of system caching of (decrypted) sensitive data should be avoided to ensure that only locally running applications, on behalf of the authenticated user, can actually access such data. The natural way to meet both goals is to insert SEFS features as close as possible to the file system API used by applications (see Figure 2).

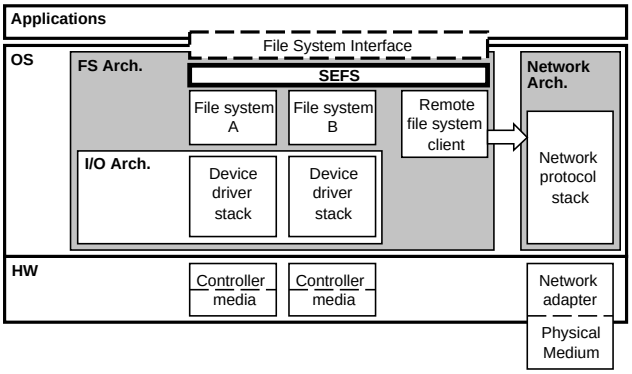

Figure 2: *Generic location of the SEFS module within the Windows file system structure – below the File System Interface used by applications and above specific file system managers, either local or remote.*

The operation principle of the SEFS module is to intercept the file system calls before they are serviced by the file system, perform security related operations depending on the function being called, and let the call go to the original file system that then interacts with device drivers. The driver satisfies the call, possibly accessing the device, and then returns the call to the original file system that does its final actions. On the return of the call, from the file system to the applications, SEFS gains again control of the request and possibly more function-specific security operations are performed.

2.1 Key management

As previously referred, the SEFS module provides a decrypted view for local applications. To build the decrypted view the module uses a set of keys provided by the local user when activating the module.

Cipher keys used by active SEFS modules are not stored anywhere in the file system, therefore some mechanism should be used for safe storage of backup copies (like the one proposed in [2]). SEFS is only a simple key-based data translator, and it does not deal with key management issues. Users are free do use whatever mechanisms they find suitable for maintaining key backups. Similarly, SEFS does not provide any special mechanisms for updating cipher keys being used by secure files, because such task is easier to accomplish with a specific tool.

The same key is used by SEFS, on a user's behalf, to cipher arbitrarily large data items, thus SEFS should use many different cipher keys in order to prevent attacks based on ciphertext pattern matching. Ideally each secure file should be ciphered with one random key, generated by SEFS and stored encrypted somewhere in the file system, which could be recovered using keys provided only by users to the SEFS module. A similar solution is used by EFS [5], but it uses asymmetric cryptography to hide each file-specific key in order to provide administrative recover of keys and file sharing.

As we said before, we do not intend to provide any support for administrative key recovery, and we believe that cryptographic protection is more likely to be useful for protecting personal data, so we do not need to use asymmetric cryptography. Furthermore, asymmetric cryptography raises problems when trying to recover file-specific key ciphered with old or revoked keys.

We used a scheme, based on symmetric cryptography and very similar to the one used in CFS [1], that ensures different cryptographic parameters for each secure file. Each file is bound to a specific random IV (Initialization Vector) value that is used in the cryptographic transformation of the file's data. The main differences between our scheme and the one used by CFS is that (i) we use a random value for each IV, while CFS uses the file i-number, and (ii) we store the IV inside the coded file name (see §2.3), while CFS stores it in the UNIX `GID` field of the file i-node (to prevent i-number modifications after a backup / recovery sequence).

The IV is not a conventional cipher key, as it is not used, directly or indirectly, by a cipher algorithm. The reason for not using a per-file cipher key is that usually cipher algorithms have significant key setup times [22], which are avoided with our cipher scheme: each file IV is used in the data transformation process but using a simple XOR operation (see §3.4). This way we avoid key setup overheads and we still make equal plaintext files, encrypted with the same user key, to produce different ciphertext. Note that a SEFS module is active only for a particular user, so it has to perform key setup operations only after getting user´s cipher keys (or passphrases).

2.2 Encrypted metadata

When encrypting information in a file system one has decide what to encrypt and what is left in clear. Ideally we would like to encrypt all information or meta-information that could help an attacker. Concerning files, for instance, the most important and obvious data to encrypt is their contents. But files have other associated information, such as name, size, and timestamps of previous manipulations. These items are what we call file's metadata, since they provide relevant information about the file but are not the data itself. This information can be very useful in leading attackers to more interesting and sensitive data, thus it should be hidden by cryptographic means. Because one of our goals is to store the secure files in existing file system structures, metadata transformations (encryption) must be carefully tackled, or even prevented, in order to maintain their consistency.

With respect to file names, these can be encrypted and afterwards coded in a form respecting the valid format and character set for file names. This scheme applies to directory names as well. Timestamps could be easily encrypted, but that would mislead administrative tasks that depend on them, like backups. File size hiding cannot happen without interfering with the actually stored amount data, being difficult to anticipate all the resulting implications in the behavior of management tools.

We believe that name hiding is the most important measure for preventing attackers from finding interesting or sensitive files. Consequently, we decided for keeping all metadata in clear, except names referring to secure files or directories.

2.3 Storage of security-related metadata

Secure files and directories have security-related attributes that should be stored in the file system, just like normal file's attributes (timestamps, etc.). Special user-defined file attributes would be the obvious choice, but unfortunately few file systems support them (e.g. AtFS [13], NTFS [17] or BeOS File System[8]). Using separate directory entries for keeping security-related metadata for all secure files of a directory, or tree of directories below, could be an alternative, but it would create single points of failure to sets of secure files. A separate file for the security-related metadata of each secure file or directory is not a good solution because it would waste a lot storage space.

We chose to store security related metadata within the encrypted name of secure files. In §3.3 we explain exactly how this is done and which security-related metadata is kept this way. For now it is only relevant to say that such metadata contains random fields, like the IV previously referred, that prevent equal and clear file names to generate equal encrypted names (somewhat similar to the protection of encrypted passwords in UNIX systems using *salt* bytes to perturb the `crypt` one-way function[15]).

However, all security-related metadata stored within a secure name should remain constant for the lifetime of the file, in order to avoid the modification of the secure name by any reasons other than the modification of the related original names. This constrain is imposed by our purpose of not disturbing administrative actions, like incremental backups. Consequently, ciphered names cannot be used to store all sort of security-related metadata, but only constant values.

2.4 Distinction of secure files from ordinary ones

Since we want to provide security on a per file basis, we need to clearly distinguish, at the SEFS level, secure from normal files, as both types may exist anywhere. So, when a user / application issues a file system call for a file using its clear name – the only name he knows – SEFS has to check if the name of the file corresponds to a normal or a secure file. In other words, this means that SEFS has to check if the name refers to a file (or directory) containing security-related metadata. Since this metadata is stored within the file name, SEFS has to check the name provided against clear names, and against decrypted secure names. Therefore, SEFS has also to distinguish secure names from normal ones.

We chose to include several hints in secure names in order to provide several kinds of name distinction. The hints, and their purpose, are the following:

identification hints to help in recognizing names as (potential) secure names. Identification hints are constant prefix and suffix strings in secure names. Ordinary names may contain these strings, and thus act as phony secure names, but such occurrences are easy to detect after decrypting them (see §3.3).

lookup hints to minimize the decryption of secure names when doing a clear name lookup. Lookup hints are specific cleartext byte blocks of secure names containing (part of) a value resulting from hashing of the original name together with cipher key. By using the cipher key to compute the lookup hint we reduce the ability of attackers to locate one or more secure files given their original name.

2.5 Correctness test of cipher keys

When one particular user is authenticated against the local SEFS the module knows at least one cipher key that should be used on the user's behalf to encrypt / decrypt secure files. However, the same storage device may contain secure files belonging to different users, usually encrypted with different cipher keys. Thus, at the SEFS level one has to decide if the users' cipher key should be used to decrypt a particular secure file or if, otherwise, the secure file contents should be provided unchanged. Clearly, one should decide for the last case whenever dealing with wrong

encryption keys, i.e. keys not suitable for decrypting particular secure files. This means that SEFS has to detect if a user's cipher key is the correct key for a particular secure file that he intends to manipulate.

This problem is easily solved with the previously referred lookup hints included in the names of secure files. As a lookup hint results from hashing the original file name along with the user's cipher key, only authorized users with the proper key can lookup and find the files they have previously encrypted. Since collisions of lookup hints may happen, the correctness of the key may be further checked when decrypting a presumable valid secure name, as the decrypted result must reveal the original file name.

When listing directories this procedure is reversed: a secure name is decrypted, the recovered original name (if valid) is hashed together with the user's key, and if the resulting value is equal to the lookup hint in the secure name, then the original name is provided to the application, otherwise the secure name is provided unchanged. By chance hint collisions may originate awkward names, though valid ones, which may easily be ignored by users.

2.6 Performance of random accesses to secure files

Files' data is often randomly accessed without any relevant performance penalty when comparing against sequential accesses. Thus, for secure files we would like to keep the usual performance characteristics of random accesses to their data without compromising security. This means that file encryption should avoid cipher techniques requiring feedback for ensuring security. For example, it would be unacceptable, in terms of performance, to use only the CBC block cipher mode [22] to encrypt or decrypt a file, since it would be necessary to fully decrypt it when reading only its last byte. Similarly, it would be unacceptable to use directly most stream ciphers, as they usually need to generate past keystream sequences before generating a particular keystream block.

To solve this problem we decided for a cipher mode, mixing both block and stream ciphers, that imposes a reasonably constant performance penalty without compromising security. This mode will be explained in §3.4.

3 Implementation

In this section we describe the implementation of the SEFS prototype for the Windows 95 operating system. Currently this prototype has a fully functioning cipher engine that encrypts the contents and names of files stored on local file systems. The work for supporting remote file systems is still in progress.

3.1 Windows 95 File System Architecture

The file system architecture of the Windows 95 operating system is based on a component called IFS Manager (see Figure 3). The IFS Manager, on the top layer, centralizes all different ways that applications have to access files, and forwards applications' requests to the proper Instalable File Systems (IFS) controlling the storage space [16]. On the bottom layer lies the Input Output Supervisor (IOS), that manages a stack of drivers for dealing with low-level accesses to devices. Between IFS and IOS is the place for plugging File System Drivers (FSD) of the IFS, like FAT, FAT32, HPFS, etc. Particular cases of FSD are network redirectors, which are clients of distributed file system servers. Finally, within the IFS Manager it is possible to make a chain of interceptors.

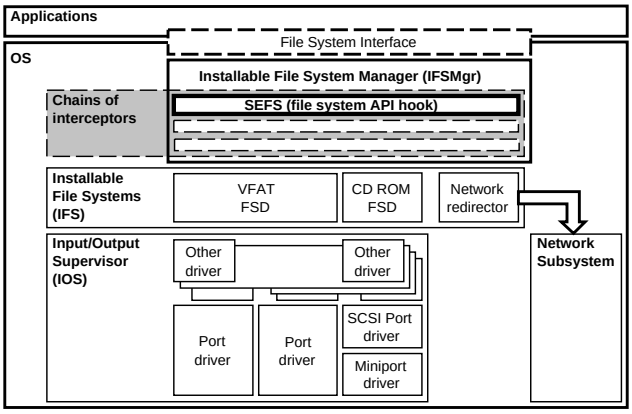

Figure 3: *File system architecture of the Windows 95 operating system and the location of the SEFS module. The SEFS module is plugged-in as an interceptor of the file system API (file system API hook) at the beginning of IFS manager's interception chains.*

As it was not our purpose to deal with file system storage details, but only to perform cryptographic transformations of data stored within existing file system, we decided to implement SEFS as an interceptor within the IFS Manager (a file system API hook, see Figure 3). Note the that the security provided by SEFS can only be ensured if (i) the SEFS module is the first hooker in the hooking chain, or if (ii) the hookers before do not perform any sort of caching of files' data.

3.2 VxD Hook Module

SEFS was implemented as a Windows 95 Virtual Device Driver (VxD) that hooks itself to the File System API in the IFS Manager. The hooking step consists in registering a VxD function that will be called by the IFS Manager when dispatching any file system request. The registration function returns the address of the next hook in the chain, which should be called by SEFS whenever the file system request requires the contribution of lower file system layers. When the SEFS VxD is installed it stores in memory a cipher key provided by the installing user. This key remains active and associated to that user until the removal of the SEFS VxD from the chain of hooks (see Figure 4).

This hooking mechanism allows the VxD to intercept all file system requests with the same interface

```
void CtrlMsgDispatch()
{
  ...
  case Sys_Dynamic_Device_Init:
    /* when the VxD is installed */

    nextHooker = IFSMgr_InstallFileSystemApiHook(SEFShook);

    SEFSuser = /* Get the user using SEFS   */;
    SEFSkey =  /* Get the user's cipher key */;
    break;

  case Sys_Dynamic_Device_Exit:
    /* when the VxD is removed */

    IFSMgr_RemoveFileSystemApiHook(SEFShook);

    /* clean SEFSuser */
    /* clean SEFSkey  */
    break;
  ...
}
```

```
int SEFShook(..., ioreq * request)
{
  if (SEFSuser == request->ir_user) {
    return SEFSswitch(..., request);
  }
  else {
    return nextHooker(..., request);
  }
}

static
int SEFSswitch(..., ioreq * req)
{
  /* SEFS pre-processing
     according to req->ifs_func */

  /* call nextHooker if necessary */

  /* SEFS post-processing
     according to req->ifs_func */
}
```

Figure 4: *Pseudo-code of the main entry points of the SEFS VxD, showing the macroscopic handling of some system requests (like the installation of the driver) and file system requests through the hook chain.*

used for file system drivers. However, the SEFS module is much simpler than a file system driver because it doesn't need to deal with the effective data storage on devices, leaving all those tasks to the underlying file system drivers. The exact activities performed by the SEFS VxD depend on to three aspects: (i) the function being called, (ii) the user performing the call, and (iii) the keys currently known by SEFS. If SEFS possesses a user key and the request comes from the same user, then SEFS does some security-related pre-processing of data (given by the application) or post-processing of data (retrieved by the next hooker). Otherwise, SEFS simply chains the call to the next hooker in the hooking chain (see Figure 4). Some of the actions performed by SEFS will be described further below, in §3.6, after describing how SEFS encrypts the name and contents of secure files.

3.3 Encryption of file names

As discussed in §2.2 and §2.3, SEFS encrypts the names of secure files and adds security-related metadata to the resulting ciphertext. The resulting bytes are encoded in ASCII characters before being stored in the underlying files systems, in order to avoid the occurrence of reserved characters; the encoding step uses a BASE64 code map[1]. Our map uses characters that are usually supported by all file systems, namely all alphanumeric characters, the hyphen and the underscore.

The layout of a secure name, and the algorithm to generate each of its components, is shown in Figure 5. A secure name has three main components:

- plaintext bytes — the #S prefix and the .S# extension. These strings ensure a proper presentation of secure names (when not decrypted by SEFS) in directory listings ordered by name or by type. The prefix and the suffix are also used by SEFS for fast pattern-based lookup of secure names, and as first-stage identification hints of secure names.

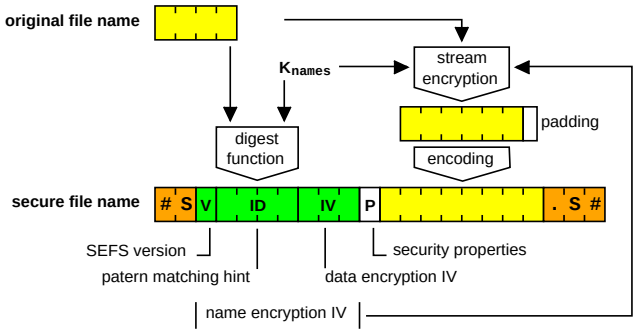

Figure 5: *Format of a secure name, used to refer a secure file or directory, and the steps followed to generate some of its components.*

- encoded ciphertext — resulting from encrypting and encoding the original file name (padded to a 3-byte boundary).

- encoded bytes — the security-related metadata, formed by 5 different values:

  V – the version of the SEFS module that created the name;
  ID – a digest of the name and the cipher key, used as a name lookup hint;
  IV – a random IV used in the encryption of the file's data; and
  P – the security properties of the file.

  The fifth value, formed by the three first ones – V, ID and IV – is used as an IV for the encryption of the original name.

This structure was chosen for improving the performance of SEFS without compromising security and capacity to operate. The extra information, both plain-

[1] Mapping groups of 6 bits into distinct 8-bit characters.

text prefix, plaintext suffix and security-related metadata, occupy a fixed number of bytes (14) that are enough for ensuring security, and not excessive in order to keep the secure name as short as possible. The exact number of bytes of a secure name is given by:

$$L_{secure} = 14 + (L_{original} \times 8 + 5)\,/6$$

where $L_{original}$ is the length of the original filename, $L_{padding}$ is the length of a null-byte padding applied to the original name before its encryption, and L_{secure} the resulting full length of the secure name. Since Win32 long file names cannot exceed 255 bytes, then $L_{original}$ should not exceed 180 bytes, which we believe is not a dramatic limitation.

This format for secure names is not very effective in hiding the length of original names, since for each length of a secure name there are only one possible value for the length of the original name. However, we believe this is not a significant source of useful information for attackers. Besides, it was unacceptable, in terms of performance, to always use maximum-length secure names for hiding original name lengths.

3.3.1 Security-related metadata

The security-related metadata includes the 5 fields of a secure name previously referred: V, ID, IV, P, and the block formed by three first ones. The V field indicates the version of SEFS that produced the secure name and needs no further clarification; the other fields are explained next.

The ID field is a pattern-matching hint used by SEFS to locate secure names corresponding to original ones. This hint is a 24-bit value, stored encoded, resulting from applying a digest function (MD5 [19]) to two values: the original name and the name cipher key (K_{names}). Thus, when SEFS wants to find a secure name corresponding to a given original name, it generates a hint with the original name and the current SEFS key, and looks for names starting by 7 known bytes — `#S<SEFS version><hint>`. False hints are detected going one step further, decoding and deciphering the secure name.

The IV field is a random IV used both in the encryption of names and data of secure files. It is a 18-bit value, stored encoded, and is currently generated by getting the 18 least significant bits of the timestamp counter fetched with the Pentium RDTSC instruction. Longer 64-bits IV values, used when processing names or data of secure files, are generated by simple concatenation of the 18-bit IV. This repetition is not dangerous for the security of the system, because the primary purpose of the random IV is to complicate ciphertext-matching attacks on names and contents of secure files.

The P field is a bit field indicating which security properties are enable for the secure file (or directory) referred by the secure name. It is a 6-bit value, stored encoded, and each bit reveals a specific security property. Currently we support only two properties:

- propagation of privacy properties to files (belonging to the directory the name refers to); and
- propagation of privacy properties to subdirectories (of the directory the name refers to).

In the future we intend to had other properties, like (i) the cipher algorithms to be used, (ii) the integrity control of the secure file, and (iii) the propagation of integrity control properties to files or directories belonging to a secure directory.

We said in §2.3 that security-related metadata should be keep constant for the lifetime of a secure file. In fact, this is what happens, except for the properties field, that can be updated. However, we believe this field would not be updated frequently, so it does not raise significant problems to our goal of keeping secure names as constant as possible (i.e. changing only when that happens to the original name).

3.3.2 Encryption of original names

Original names are encrypted using a stream cipher built from a block cipher (IDEA [12]) operating in 64-bit CFB mode. Original names are padded with null bytes until 3-byte boundaries to prepare the resulting ciphertext to the BASE64 coding phase. The initial state of the stream cipher is the field referred as *name encryption IV* in igure 5, i.e. the first 8 bytes of the security-related metadata.

With this algorithm we prevent comparison attacks on secure names without extra costs in cipher operations. Equal original names, on different directories, produce different secure names because a random field (the data encryption IV) is also used as part of the IV of the stream cipher. The stream cipher also ensures that similar original names produce very different secure names. Key-setup overheads are avoided by using a constant key (K_{names}) for ciphering all secure names for a particular user.

3.4 File encryption

SEFS encrypts the contents of secure files, but not the contents of secure directories. Directories have a system-defined structured that cannot be modified in order to keep the coherence of the file system. Therefore secure directories have only a ciphered name and security properties that can be propagated for files and directories below, but not encrypted contents.

The contents of secure files are encrypted using the method shown in Figure 6. This method is very similar to the one used in CFS, named EBC+OFB [1], and operates as follows. Each plaintext block of the file, properly aligned, is XORed with two values, one picked from a global mask, and the other from the file name (the IV field). The result is encrypted using a block cipher operating in ECB mode. The global mask

is created using a stream generator, and only when the SEFS VxD is installed for a particular user.

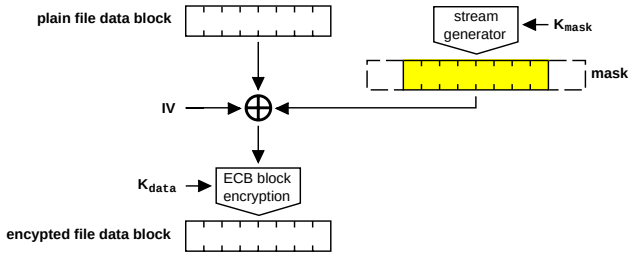

Figure 6: Cipher method used for generating the contents of secure files.

The overall performance of this method, when doing random updates on a secure file, is close to a pure ECB. Similarly, this method is probably as secure as a pure ECB [22], but, like for CFS, the mask reduces the probability of producing equal ciphertext blocks from two equal plaintext blocks along the same file, and the IV ensures that equal files ciphered with the same keys (K_{mask} and K_{data}) produce different ciphertext. Therefore, the mask complicates structural analysis of secure files, and the IV complicates pattern-matching attacks against pairs of files.

The differences between our method and the one used by CFS are only of operational nature. Currently SEFS uses the IDEA algorithm and 128-bit keys for both for the stream generation (operating in 64-bit OFB mode) and EBC block encryption; CFS uses DES [22] and 56-bit keys. The IV is a 64-bit value built from the 16-bit IV field of the file name; CFS builds an equal-length value from a 32-bit $i-number$ or GID. The length of the mask was decided as a trade-off between security, efficiency and resource allocation. While CFS uses a $\frac{1}{2}$ Mbyte mask that can be paged-out, SEFS uses a 1 Kbyte mask in non-pageable memory. However, the short mask kept by SEFS is effectively used as a long 1 Mbyte mask, by means of a vector multiplication of the short mask with itself (using XOR and considering each 64-bit block as a vector element). To avoid the structured occurrence of null values in the long mask, resulting from XORing equal blocks, the short mask is multiplied with a 32-bit shifted copy of itself. With this algorithm we guaranty, with a minimum increase of XOR operations, that SEFS uses a long and reasonably random mask without occupying an excessive amount of non-pageable memory.

Incomplete blocks at the end of files must be handled differently, but not padded. The reason for not padding them is the following. We cannot store any frequently-updated metadata in secure names (see § 2.3), so we cannot use them to store real sizes or even contents of incomplete blocks. Thus, padding would have to be fully quantified using file contents, and that would modify the size of all secure files. As there are numerous system requests that return the size of non-opened files (e.g. directory listing requests, queries of file's metadata), it would be unacceptable, in terms of performance, to fetch padding information from secure files, to correct their actual size, in all those requests. Consequently, the size of secure files must remain equal to the original one for keeping performance in reasonable levels.

Consequently, we had to choose a different cryptographic method for handling incomplete blocks different from padding them. Ciphertext stealing was considered, but it does not apply to files shorter than a block. Therefore, we decided for adding the incomplete block to the result of encrypting a null-byte block. Unfortunately, this method cannot prevent deterministic modifications of incomplete blocks, thus allowing limited known-plaintext attacks. The solution for this problem is still an open issue, but, in any case, it does not compromise the security provided by SEFS in terms of privacy.

3.5 Encryption keys

As we saw along this section, SEFS uses three different keys, named K_{names}, K_{mask} and K_{data}. All these keys are generated from a variable-length password provided by the user installing the SEFS VxD. The three keys are 128-bit values generated by successively applying a digest function (MD5 [19]) to the password and all the previously generated keys.

The difficulty of guessing the keys depends directly on the length of user's passwords. Short passwords, though probably providing very different keys, make the SEFS security weak against exhaustive guessing attacks. Nevertheless, to increase the cost of such attacks, SEFS generates K_{mask} first, directly from the user's password, because this key is not directly used in any exposed ciphertext. The key K_{names} is the last to be produced because it is solely and directly used to produce the ciphertext of secure names.

$$\begin{aligned} K_{mask} &= MD5(password) \\ K_{data} &= MD5(password|K_{mask}) \\ K_{names} &= MD5(password|K_{mask}|K_{data}) \end{aligned}$$

Therefore, to guess passwords using known original file names, an attacker as to execute the digest function three times on each trial. Similarly, the attacker as to execute the function twice when using original file contents.

3.6 Behavior of intercepted file system calls

The functions intercepted by SEFS perform some extra processing whenever dealing with names or contents of secure files. In this section we will briefly highlight some of actions taken by SEFS in both cases.

3.6.1 Secure names

There are two major groups of system calls that deal with file names: (i) those providing the complete

name of a file, either existing or not, to do something with it (create, open, rename, delete, query / set attributes, etc.), and (ii) those getting file names from the file system, either using or not pattern-matching constrains (directory listings).

For first group of functions SEFS usually tries to use first the normal name; if it fails, then a secure name is generated and used. One exception to this rule is when creating a file on a directory with the privacy propagation property, that obligates SEFS to create the file as a secure one (thus with a secure name). In both cases SEFS does not know if the secure name exists, which implies a pattern-constrained search among secure names to find it out. This search uses the 7-byte pattern described in §3.3.1.

The second group of functions, to perform directory listings, allow a name-by-name gathering of directory entries using a listing context and three functions (FindFirst, FindNext, and FindEnd). When these functions are not used with pattern-matching constrains, SEFS simply calls the next hooker to get a directory entry and translates it, whenever referring to a secure name ciphered with the user's key, before returning to the caller. When they are used with pattern-matching constrains, SEFS first calls the next hooker to get all normal names matching the pattern. After a failure SEFS calls the next hooker to get all secure names, and checks itself the pattern-matching after translating the returned secure name. This two-phase search implies that SEFS must keep itself search contexts referring current search phases.

3.6.2 Secure contents

The functions that manipulate file contents use a handle provided by a create or open system call, both referring a file name. When SEFS detects that the name provided in one of these calls is (or will be) a secure one, then it stores the handle returned by the next hooker in a table of secure handles. Read or write system calls not using secure handles are ignored by SEFS and immediately forwarded to the next hooker; otherwise some data processing must be done by SEFS. Seek calls are always ignored by SEFS, because the offset of data in an encrypted file is the same as in the corresponding plaintext.

Read calls are simpler to handle than write calls. On a read call SEFS reads a block-aligned buffer containing the file contents required, decrypts it, and returns the required data to the caller. Only when reading the last and incomplete block of a file it is necessary to use a different decryption algorithm.

On a write call SEFS encrypts the block-aligned portion of the buffer provided and writes it. If the block is not aligned, then one or two data blocks are read, decrypted, and combined with the incomplete blocks in the begining or end of the buffer, encrypts and writes them. Finally, some different or extra actions must be done when writing an incomplete block at the end of the file, or when writing a buffer after the end of a file terminating with an incomplete block. In the first case it is necessary to use a different encryption algorithm. In the second case the incomplete block must be read, decrypted, padded with garbage or combined with the first bytes of the buffer, and written again.

4 Performance evaluation

To evaluate the performance of SEFS we measured our prototype in a variety of scenarios and with several types of specific benchmark programs. For running the benchmarks we used a PC Pentium-MMX 233 Mhz with 64 Mbytes of RAM and an EIDE disk of 2.6Gb formatted with FAT32 and a 32 Kbyte file system blocks. The benchmarks used only local files, as the current SEFS prototype does not intercept all the system calls used for remote files (through redirectors). We expect to fully support remote file systems in a near future.

The first benchmark consists of reading data from a file and writing it into another file. The read and write operations used 5 different block – of 1 K, 10 K, 100 K, 1 M and 10 Mbytes – starting from the begining of 10 Mbyte files. The benchmark was executed in three different scenarios:

1. without SEFS;
2. with SEFS and normal files; and
3. with SEFS and secure files.

The time results presented in Table 1 and Figure 7 are the arithmetic mean of elapsed times evaluated in 10 executions of the benchmark after an initial READ / WRITE phase to warm-up the file system cache. The table also presents, in percentage, the overheads introduced by SEFS when processing normal and secure files. Finally, the table shows the time expended by SEFS in the data cipher mode (subdivided in two operations, the XOR of IV and mask, and the ECB encryption with IDEA), and the percentage of each of these operations to the total overhead introduced by SEFS.

The overhead introduced by SEFS for normal files is negligible (below 4%, except when reading 1 Kbyte buffers), and mainly due to a search of the application's file handle in the table of handles to secure files managed by SEFS. Regarding secure files, the overhead introduced by SEFS is significant, but mostly due to the cipher method: the XOR step is responsible for about 6% of the total SEFS overhead, while the cipher step with IDEA is responsible for 72 to 88% of the total SEFS overhead. Altogether, the cipher method is responsible for 78 to 95% of the total SEFS overhead.

5 Related work

In this section we discuss other solutions that use also cryptography for improving security in storage systems. There are three typical solutions for managing encryption operations: (i) encryption tools

Operation	Buffer lenght	Without SEFS time (μs)	with SEFS: normal file time (μs)	normal file overhead (%)	secure file time (μs) Total	time (μs) XOR	time (μs) IDEA	secure file overhead (%) Total	overhead (%) XOR	overhead (%) IDEA
READ	1K	66	79	19.7	568	32	465	761	5.6	81.8
	10 K	295	302	2.3	5273	277	4566	1687	5.2	86.6
	100 K	1947	1964	0.9	52971	3077	45590	2620	5.8	86.1
	1 M	27345	27464	0.4	532894	34772	469687	1849	6.5	88.1
	10 M	239956	241056	0.5	5314088	352393	4696778	22115	6.6	88.4
WRITE	1K	130	135	3.8	621	36	448	378	5.8	72.2
	10 K	297	302	1.5	5106	312	4350	1617	6.1	85.2
	100 K	2035	2045	0.5	50610	3155	43529	2387	6.2	86.0
	1 M	27106	27172	0.2	513458	34437	449795	1794	6.7	87.6
	10 M	245298	246613	0.6	5096911	347196	4501715	1979	6.8	88.3

Table 1: *Evaluation of elapsed time during READ and WRITE operations in multiple scenarios and with different buffer sizes. The total overheads introduced by SEFS are calculated dividing total elapsed times by the values obtained without SEFS. The XOR and IDEA overheads are computed dividing XOR and IDEA elapsed times by the total time spent with SEFS.*

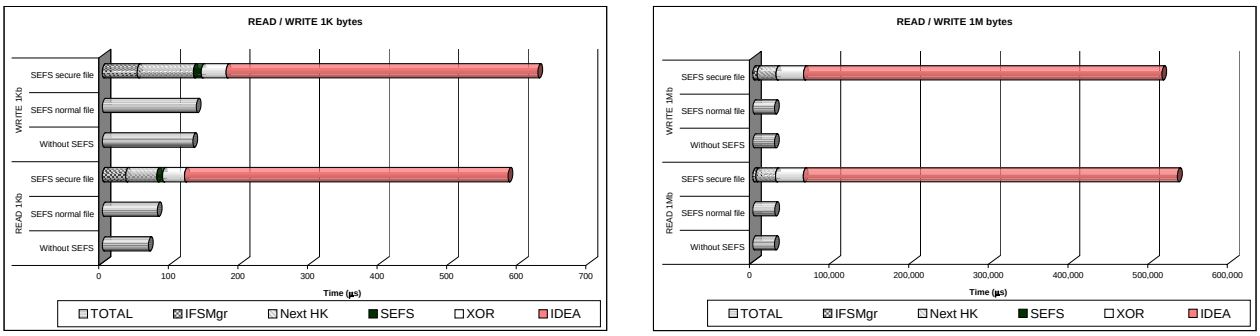

Figure 7: *Time diagrams of the values presented in Table 1 for buffers with 1 K and 1 Mbytes. The bars relative to non-secure read/write operations show only total elapsed times. The bars relative to secure read/write operations show a breakdown of total elapsed times in 5 components: time expended by the IFS manager above SEFS (***IFSMgr***), time expended by the next hooker below SEFS (***Next HK***), time expended inside SEFS excluding the cipher method (***SEFS***), and the time expended by the cipher method (***XOR*** and ***IDEA***).*

(e.g. PGP [24]), (ii) disk encryption systems (e.g. PGPdisk [18]), and (iii) file or directory encryption systems (e.g. CFS [1]). We are going to focus mainly on this last one, and then we give a brief summary on some disk encryption systems; encryption tools are far out of our objective and will not be addressed.

The first example of file or directory encryption systems is the Cryptographic File System (CFS) [1], developed by M. Blaze at AT&T around 1993. This system was developed for UNIX platforms and allows users to encrypt files on a per-directory basis, both on local and remote file systems. The system intercepts the file system calls via the NFS interface. Each client machine runs, in user space, an altered NFS server named CFS daemon (cfsd) implementing the cipher engine and exporting encrypted mount points through the localhost interface. The CFS daemon then accesses local file or remote file systems through the standard UNIX interface to store secure data. Thus, accessing secure remote files through CFS requires a double mount technique; this is one of its major drawbacks since it implies large performance costs.

To use CFS, a user attaches an exported encrypted directory to a special mount point that is the home of all secure directories, and the only place where files can be seen and accessed in clear. This scheme forces changes in file location when adding security properties to them, and so lacks flexibility.

The CFS offers data privacy by encrypting file contents and names using DES [22]. For encrypting file contents CFS uses a combination of the ECB and OFB cipher modes in order to improve security without compromising performance of usual random block accesses. This cipher mode also uses a IV for each file to prevent comparison attacks, but the system cannot fully guaranty the recovery of this value, putting in risk the decryption of file data in some circumstances.

Following the work made by M. Blaze, a team from University of Salerno came with Transparent Cryptographic File System (TCFS [4]). This system provides deeper integration between the encryption service and the Linux file system architecture. TCFS operates like a NFS client, using the same system calls, and solves some of the problems of CFS. The main difference is that the user does not has to have his secure directories attached to a special mount point where all

encrypted files can be seen in clear form. Instead, in TCFS encrypted and normal files can have the same location and be accessed in the same way. Another difference is that CFS works in user space while TCFS works in the kernel space, thus resulting in improved performance. However, the problem of distinguishing secure from non-secure files is solved by adding a special attribute to the file that is interpreted by a special attribute server running on the server machine of a remote file system. The introduction of this attribute server prevents transparent use of existing distributed file servers without changes.

The Secure File System (SFS [10]) is a component of the UFO architecture developed in University of Minnesota, based on UNIX architectures. The idea is to put the cipher engine layered between the applications and the original file systems. On the client machine, SFS intercepts the file system calls using the native debugging interface, to process file name and data, encrypting before sending it and decrypting after receiving it from the remote storage system.

The control information associated with the secure files, like user-key pairs, is saved in a header in the file content, implying that a file as to be opened and partly read to obtain his security properties. Because SFS uses the UFO architecture, where file processing is based on the open call, and read / write system calls are not intercepted, it requires an entire file being encrypted or decrypted on open with the performance cost associated. Another weakness of SFS is that the names of the files in a directory are all encrypted with the same key creating problems in having files belonging to different users in the same directory.

The Encrypted File System (EFS [5]) is a component of the Microsoft Windows 2000 operating system. EFS is a module that adds encryption to data stored in local NTFS volumes, allowing to encrypt or decrypt data on a per-file and per-directory basis. EFS uses a mix of public key and symmetric cryptography: it uses DES and randomly generated keys to encrypt file data, and file owner's public key to encrypt the DES key.

EFS uses the extendable characteristics of NT file system architecture to add the encryption module on top of NTFS, intercepting all file system calls. One problem with EFS, however, is that it saves security control information in special file attributes of NTFS, and so becoming dependent of it. This, besides disallowing the use of the encryption module on top of other common local file systems, like FAT32 or HPFS, also raises problems on using distributed file systems like CIFS [14] (LanMan, SMB). Since the protocol used by network connections doesn't have knowledge of this special attributes supported by NTFS, a secure file can not be stored on a remote file server. The exception to this is when the server runs Windows 2000 with an EFS / NTFS installation, but in this case the client is forced to trust the server with the clear data [5]. Another security weakness of EFS is that it doesn't encrypt file names, which are useful hints to attackers. Finally, in the case of remote servers, files and directories may be encrypted on server but the data decrypted and transmited in clear to remote clients.

Another approach to security in storage architectures is putting security at device driver level, working by encrypting and decrypting entire disk blocks . Also, they have to define the volume as the minimum granularity for security, since it's the only concept in common between drivers and users. So, these systems lack the flexibility to define security properties at file and directory level and have very weak control on the amount of data that really need to be processed. As cipher operations are very time consuming, this can lead to great inefficiency. This approach was common on the MS DOS operating system, since there was no way to easily expand its file system and it was relatively easy to build and install a block device driver for the storage system. Nevertheless, there are still many systems that use this approach, as it is easy to implement and manage. Examples of such systems are Secure File System (SFS) for MS DOS / Windows [9] and PGP Disk [18] for MS Windows and UNIX systems.

5.1 Comparison with SEFS

Concerning the implementation, the SEFS approach is similar to the ones followed by TCFS and EFS: they all work at kernel level and provide security for particular files and directories. However, SEFS is more flexible, because it allows users to store secure files over several existing file systems, either local or remote. SEFS does not rely on special file attributes (as EFS does) to store the metadata of secure files, or particular mount points for triggering its action (as TCFS does), thus being more capable of spanning many different local and remote file systems.

Concerning the location of secure files, SEFS is more flexible than CFS and SFS, because it allows users to sprinkle secure files everywhere within existing file systems. CFS and SFS provide a decrypted view over completely encrypted file systems or directories, while SEFS provides an ordinary or decrypted view over partially encrypted file systems. This allows secure files from different users to co-exist in the same file system.

Both SEFS and CFS encrypt secure file's names, but CFS uses only the resulting ciphertext for naming secure files, while SEFS includes also security-related metadata in the names of secure files. CFS always decrypts all directory names when doing name lookups, while SEFS minimizes decryptions using hints on name lookups.

Finally, SEFS, CFS and SFS ensure network security to remotely stored secure access, as they are only decrypted by the clients. On the contrary, EFS decrypts the files before sending them to clients, and relies on secure network protocols, such as Secure Sockets Layer / Private Communication Technology (SSL / PCT [7, 23]), to encrypt data accesses over the network.

6 Conclusions

SEFS provides seamless privacy at file and directory level by means of a module inserted in the file system stack, using any underlying file system to store secure files and directories. The file system API is not changed by SEFS, allowing existing applications to transparently work with secure files. Administrative procedures, like backups, are not affected by the presence of secure files, and deal with them just like with other ordinary files.

Secure files may be stored anywhere, mixed with normal files, both in local or remote file systems. Network security is ensured because decryption happens locally to applications. The names of secure files and directories are also encrypted to complicate the task of finding interesting sensitive files. The names of secure files are also used to store per-file security-related metadata, and is one of the innovative aspects of SEFS.

The privacy provided by SEFS is based on symmetric cryptography, both for encrypting names and contents of secure files. Each installed SEFS module operates for a given user, and uses a set of keys computed from a password obtained when installed. For that user SEFS provides a decrypted view of file systems, i.e. a view where files and names ciphered with the user's password are decrypted before being provided to applications. The current implementation of SEFS supports only one password per user, but its easy to allow more than one. SEFS does not provide any means for administrative recover of secure files' data.

To improve performance SEFS uses several techniques to reduce the impact of cryptography. To keep the usual performance characteristics of random accesses to secure file's data, without compromising security, it uses the technique designed for CFS [1]. For reducing the number of decryption in name lookup operations, the name of secure files contain identification and lookup hints.

A prototype of SEFS was developed, for the Windows 95 operating system, as VxD that hooks itself in the hooking chain provided by the IFS Manager. The privacy provided by SEFS is ensured for as long as the SEFS VxD remains the first hooker of the chain. In a near future we expect to develop an SEFS prototype for the Windows Driver Model (WDM) architecture, supported by the Windows 98 and windows 2000 operating systems [3].

In terms of performance SEFS introduces a significant overhead when reading / writing secure files, but that is mostly due to cryptographic transformations. Since such transformations are necessary to enforce privacy, either with application-level tools or with a file system module like SEFS, we can conclude that the performance of the current SEFS prototype is good for the functionality it provides.

References

[1] Matt Blaze. A Cryptographic File System for Unix. In *1st ACM Conf. on Comm. and Computing Security*, pages 9–16, Fairfax, VA, USA, November 1993.

[2] Matt Blaze. Key Management in an Encrypting File System. Technical report, AT&T Bell Laboratories, 1993.

[3] Chris Cant. *Writing Windows WDM Device Drivers*. R&D Books, 1999.

[4] A. Celentano, A. Cozzolino, A. del Sorbo, E. Mauriello, and R. Pisapia. Transparent Cryptographic File System. In *Pluto Meeting 1998 – LiMe'98*, Rome, Italy, October 1998. `http://impchim2.ing.uniroma1.it/-LUG/lime98/atti/index.html`.

[5] Microsoft Corporation. Step-by-Step Guide to Encrypting File System (EFS). `http://www.microsoft.com/-TechNet/win2000/efsguide.asp`.

[6] Gary Fernandez and Larry Allen. Extending the UNIX Protection Model with Access Control Lists. In *Proc. of the USENIX Summer Conf.*, pages 119–132, San Francisco, California, USA, June 1988.

[7] Alan O. Freier, Philip Karlton, and Paul C. Kocher. SSL Protocol Version 3.0. Internet-Draft (expired), March 1996. currently available in `http://developer.netscape.com/-docs/manuals/security/sslin/-contents.htm`.

[8] Dominique Giampaolo. *Inside the BeOS; Modern File System Design*. Morgan Kaufmann Publishers, October 1998.

[9] Peter Gutmann. Secure FileSystem (SFS) for DOS / Windows, 1996. `http://www.cs.aukuni.ac.nz/-~pgut001/sfs/index.html`.

[10] James Hughes, Chris Feist, Steve Hawkinson, Jeff Perrault, Matthew O'Keefe, and David Corcoran. A Universal Access, Smart-Card-Based, Secure File System. Atlanta Linux Showcase, October 1999. `http://www.securefs.com`.

[11] S. R. Kleiman. Vnodes: An architecture for Multiple File System Types in Sun UNIX. In *Proc. of the USENIX Summer Conf.*, Atlanta, Georgia, USA, June 1986.

[12] X. Lai and J. Massey. A Proposal for a New Block Encryption Standard. In *Advances in Cryptology – EUROCRYPT '90 Proceedings*, pages 389–404. Springer-Verlag, Berlin, 1990.

[13] Andreas Lampen. Advancing Files to Attributed Software Objects. In *Proc. of the USENIX Winter Conf.*, pages 219–229, Dallas, Texas, USA, January 1991.

[14] Paul Leach and Dan Perry. CIFS: A Common Internet File System. *Microsoft Interactive Developer*, November 1996. `http://www.microsoft.com/-mind/1196/cifs.htm`.

[15] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[16] Stan Mitchell. *Inside the Windows 95 File System*. O'Reilly & Associates, 1997.

[17] Rajjev Nagar. *Windows NT File System Internals*. O'Reilly & Associates, 1997.

[18] PGP Disk. `http://www.pgpinternational.com/-products/pgpdisk.shtml`.

[19] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, April 1992. available via DDN Network Center.

[20] J. H. Saltzer. Protection and control of information sharing in Multics. *Comm. of the ACM*, 17(7):388–402, July 1974.

[21] M. Satyanarayanan. Integrating Security in a Large Distributed System. *ACM Trans. on Computer Systems*, 7(3):247–280, August 1989.

[22] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley & Sons, Inc., second edition, 1996.

[23] Daniel Simon. The Private Communication Technology Protocol. Internet-Draft (expired), April 1996. `draft-benaloh-pct-01.txt`.

[24] Philip Zimmermann. *The Official PGP User's Guide*. MIT Press, 1995.