NCryptfs: A Secure and Convenient Cryptographic File System

Charles P. Wright, Michael C. Martino, and Erez Zadok
Stony Brook University

Appears in the General Track of the USENIX 2003 Annual Technical Conference

Abstract

Often, increased security comes at the expense of user convenience, performance, or compatibility with other systems. The right level of security depends on specific site and user needs, which must be carefully balanced. We have designed and built a new cryptographic file system called NCryptfs with the primary goal of allowing users to tailor the level of security vs. convenience to fit their needs. Some of the features NCryptfs supports include multiple concurrent ciphers and authentication methods, separate per-user name spaces, ad-hoc groups, challenge-response authentication, and transparent process suspension and resumption based on key validity. Our Linux prototype works as a stackable file system and can be used to secure any file system. Performance evaluation of NCryptfs shows a minimal user-visible overhead.

1 Introduction

Securing data is more important than ever. As the Internet has become more pervasive, security attacks have grown. Widely-available studies report millions of dollars of lost revenues due to security breaches [19]. Such concerns have prompted regulation efforts for the healthcare (HIPAA [26]) and financial services (GLBA [16]) industries, as well as commitments from software vendors to provide better security facilities.

Yet, software to secure data files is not in wide use today. We believe one of the main reasons for this is that security software is not convenient to use: securing data files cannot be done easily and transparently. For security software to become universal, it has to balance several conflicting concerns: security, performance, and convenience. Whitten reported in 1999 that even experienced computer users could not use PGP 5.0 in less than 90 minutes and that one-quarter of the test subjects accidentally revealed the secret they were supposed to protect [28]. Work on security software in recent years has often focused on increasing the level of security and on performance, as reported in a recent comprehensive survey of storage systems security [21]; not much consideration has been given to user convenience [28]. It is not surprising that in recent years, prominent researchers such as Hennessey and Pike have advocated that the research community begin tackling difficult problems such as software usability [8, 18].

We have designed and developed an encryption file system (NCryptfs) whose primary goal is to ensure data confidentiality, while balancing security, performance and convenience. NCryptfs is a security wrapper that binds to a directory that stores ciphertext data. The ciphertext directory may be on any file system (e.g., EXT2 or NFS). Through NCryptfs, a cleartext view is presented via the standard UNIX file access API. We provide convenience by allowing administrators and users to customize the behavior of NCryptfs, while picking sensible defaults.

The threat models NCryptfs addresses include network sniffers, untrusted servers, and stolen machines. Normally, when exporting file systems over the network, cleartext data is sent over the network and the server must be trusted to keep the data confidential. When NCryptfs is deployed on the clients, only ciphertext file data is sent over the network, and the server does not have access to the cleartext data. Corporations and governments are storing more and more sensitive data on laptops, which are often stolen along with their valuable data. When NCryptfs is used, a stolen laptop will not reveal useful information to the thief. In both of these scenarios, NCryptfs attempts to restrict the information a compromised system reveals to an attacker to just the information that is actively being used.

NCryptfs is a successor to our much simpler encryption file system called Cryptfs [32]. Cryptfs was a proof-of-concept example of what useful features stackable file systems could offer. We used Cryptfs as a starting point and developed a comprehensive stackable cryptographic file system that offers new security and convenience features not available in Cryptfs. Both Cryptfs and NCryptfs were developed from our portable stackable file system toolkit called FiST [30, 33, 34]. Such stackable file systems can use any file system (e.g., EXT3, NFS, or CIFS) as the backing store for encrypted data. Some of the features that NCryptfs includes are:

- Support for multiple users, multiple keys, multiple ciphers, and multiple authentication methods (in-

cluding challenge-response authentication between user processes and the kernel).
- Ad-hoc groups, allowing users to delegate "join" privileges to others, and for others to join or leave groups as needed.
- Per-process and per-session keys, with hooks for processes to be informed of certain activities in NCryptfs (e.g., a request to re-authenticate).
- Key timeouts and revocation, which in addition to per-process keys allows us to suspend and resume processes based on key validity, as well as to add encryption transparently to unmodified programs that have already begun running.

In developing NCryptfs on Linux, we also noticed that a secure file system cannot be easily built as a standalone file system (stackable or native). The reason is that important file system information is accessed by other kernel components without consulting the file system. For example, we enhanced the Linux directory cache (dcache) and inode cache (icache) so that all accesses to cached (possibly cleartext) objects are validated first through NCryptfs. Additionally, we enhanced Linux's process management so that process destruction actions are coordinated with NCryptfs; the latter removes any related security info (e.g., keys and other objects) when a process or session terminates.

Many past secure file systems often made arbitrary decisions along a security-performance axes, with minimal consideration for user convenience [21]. NCryptfs was carefully designed so as to allow many levels of security and still offer ease-of-use and high performance; whenever possible, we allow administrators and users to select among several choices. Our performance benchmarks show NCryptfs's overhead to be just 5% for normal user activities.

The rest of this paper is organized as follows. Section 2 surveys background work. Section 3 describes the design of our system. We discuss interesting implementation aspects in Section 4. Section 5 presents an evaluation of our system. We conclude in Section 6.

2 Background

In this section we briefly describe other cryptographic file systems that provided the motivation for NCryptfs.

SFS SFS is an MSDOS device driver that encrypts an entire partition [6]. Once encrypted, the driver presents a decrypted view of the encrypted data. This provides the convenient abstraction of a file system, but relying on MSDOS is not secure because MSDOS provides none of the protection of a modern OS.

CFS CFS is a cryptographic file system that is implemented as a user-level NFS server [1]. It requires the user to create a directory on the local or remote file system to store encrypted data. The cipher and key are specified when the directory is first created. The CFS daemon is responsible for providing the owner access to the encrypted data via a special *attach* command. The daemon, after verifying the user ID and key, creates a directory in the mount point directory that acts as an unencrypted window to the user's encrypted data. Once attached, the user accesses the attached directory like any other directory. CFS is a carefully designed, portable file system with a wide choice of built-in ciphers. Its main problem, however, is performance. Because it runs in user mode, it must perform many context switches and data copies between kernel and user space.

TCFS TCFS is a cryptographic file system that is implemented as a modified kernel-mode NFS client. Since it is used in conjunction with an NFS server, TCFS works transparently with the remote file system, eliminating the need for specific attach and detach commands. To encrypt data, a user sets an encrypted attribute on directories and files within the NFS mount point [2]. TCFS integrates with the UNIX authentication system in lieu of requiring separate passphrases. It uses a database in `/etc/tcfspwdb` to store encrypted user and group keys. Group access to encrypted resources is limited to a subset of the members of a given UNIX group, while allowing for a mechanism (called *threshold secret sharing*) for reconstructing a group key when a member of a group is no longer available.

TCFS has several weaknesses that make it less than ideal for deployment. First, the reliance on login passwords as user keys is not safe. Also, storing encryption keys on disk in a key database further reduces security. Finally, TCFS is available only on systems with Linux kernel 2.2.17 or earlier, limiting its availability.

BestCrypt BestCrypt is a commercially available loopback device driver supporting many ciphers [10]. Such a loopback device driver creates a raw block device with a single file, called a *container*, as the backing store. This device can then be formatted with any file system or used as swap space. Each container has a single cipher key. The administrator creates, formats, and mounts the container as if it were a regular block device. BestCrypt is ideal for single user environments but unsuitable for multiuser systems. In a single-user workstation, the user controls the details of creating and using a container. In a multi-user environment, however, the user must give the encryption key to a potentially untrustworthy administrator. Moreover, the ability to share containers among groups of users is limited, as BestCrypt gives different users equal rights to the same container. The CryptoGraphic Disk driver is similar to BestCrypt, and other loop-device encryption systems, but it uses a native disk or partition as the backing-store [4].

Cryptfs Cryptfs [32] is the stackable, cryptographic file system that serves as the basis for this work. It was never designed to be a secure file system, but rather a proof-of-concept application of FiST [34]. It supports only one cipher and implements a limited key management scheme.

EFS EFS is the Encryption File System found in Microsoft Windows, based on the NT kernel (Windows 2000 and XP) [14]. It is an extension to NTFS and utilizes Windows authentication methods as well as Windows ACLs [15, 21]. EFS encrypts files using a long-term key. Encryption keys are stored on the disk in a *lockbox* that is encrypted using the user's login password. This means that when users change their password, the lockbox must be re-encrypted. If an administrator changes the user's password, then all encrypted files become unreadable.

StegFS StegFS is a file system that employs steganography as well as encryption [13]. If adversaries inspect the system, then they only know that there is some hidden data. They do not know the contents or extent of what is hidden. This is achieved via a modified EXT2 kernel driver that keeps a separate block-allocation table per *security level*. It is not possible to determine how many security levels exist without the key to each security level. Data is replicated randomly throughout the disk to avoid loss of data when the disk is mounted with an unmodified EXT2 driver and random blocks may be overwritten. Although StegFS achieves plausible deniability of data's existence, the performance degradation is a factor of 6–196, making it impractical for most applications.

3 Design

NCryptfs's primary design goals were to balance the often conflicting concerns of security, convenience, and performance [21]. Our two most important goals were security and convenience:

- **Security** We ensure that the data stored using NCryptfs remains confidential, by using strong encryption to store data. We also modified the kernel to notify NCryptfs upon the death of a process and evict cleartext pages from the cache.
- **Convenience** If a system is not convenient then users will not use it, or will circumvent its functionality [28]. The inconvenience of current cryptographic systems contributes to their lack of widespread adoption. NCryptfs makes encryption transparent to the application: any existing application can make use of strong cryptography with no modifications. We designed NCryptfs to be cipher agnostic, so it is not tied to any one cipher.

We also want the performance of our system be as close as possible to a raw encryption operation as possible. Our final design goal was portability, which we achieve through the use of stackable file systems [33].

In Section 3.1 we describe the players in our system. In Section 3.2 we describe our key management. In Section 3.3 we describe the concept of an attachment, which we use to provide much of the convenience associated with NCryptfs. In Section 3.4 we discuss ad-hoc groups. In Section 3.5 we discuss key revocation and timeouts. We describe system operation in Section 3.6.

3.1 Players

When designing any system, one of the most important questions is who are the players, or more simply who uses it. In NCryptfs we identified three groups of players: (1) the system administrator, (2) owners, and (3) readers and writers. We use the same taxonomy as Riedel, but add the system administrator [21].

System Administrator The system administrator originally mounts NCryptfs and must be able to enforce usage policies. The system administrator is trusted to properly install the NCryptfs kernel and user-space components. However, the system administrator is not trusted with encryption keys.

Owners The owner is the user who controls the encryption key for the data. The owner receives permissions (see Section 3.3) from the system administrator and may delegate them to other users.

Readers and Writers All other authorized users are either readers or writers. The only difference between an owner and a reader or writer is that the owner supplies the encryption key; all other readers and writers do not know the encryption key. An owner is implicitly a reader or writer, depending on the permissions that the system administrator delegates. For the system to be convenient, readers and writers must be able to use encryption transparently. Authorized readers and writers must also be able to delegate permissions received from other authorized readers and writers.

Any user who attempts to exceed their delegated permissions is considered an adversary.

3.2 Key Management

The security of encrypted data is only as strong as the policy that is put in place to protect the keys. NCryptfs makes the assumption that the underlying storage media can be read and tampered with, so to ensure data confidentiality, it must be encrypted. Before the encrypted data is used, the owner must provide the key to NCryptfs (presently by entering a passphrase). Once the key is sent to kernel space, NCryptfs stores it in core memory. NCryptfs will use the encryption key on behalf of readers and writers, without revealing it to them. When cryptographic algorithms are used for authentication, authentication information is distinct from the encryption key. After the initial authentication takes place, the result is bound to a specified user, group, session, or process.

NCryptfs uses a long-lived key to encrypt all data and file names written to disk. If we used a short-lived key, then whenever the key changed, all data would have to be re-encrypted. To avoid this performance penalty, we use long-lived keys. NCryptfs uses the underlying file system to store ciphertext data, but all other data related to the encryption key is stored in pinned core memory that can not be swapped to disk.

NCryptfs is cipher agnostic. It uses cipher modules that are treated as simple data transformations. The only requirement that NCryptfs makes of the cipher is that it must be able to encrypt an arbitrary length buffer into a buffer of the same size. Most widely-used ciphers are able to do this in *Cipher Feedback Mode* (CFB) [25]. CFB mode allows us to keep the size of encrypted files the same. Changing the size of files complicates stackable file systems and decreases performance [31]. Selecting an appropriate cipher allows the user to select where they want to lie on the security-performance-convenience continuum. If the user is more concerned about performance, then a faster but less secure cipher may be chosen (e.g., one with a shorter key length). This also affects convenience: if the cipher is too slow then the user may not use encryption at all.

3.3 Attachments

We associate each encryption key with an *attach*. Attachments, inspired by CFS, allow owners to have personal encrypted directories [1]. An attach is much like an entirely separate instance of a stackable file system. Each attach has a corresponding directory entry within the NCryptfs mount point and stacks on a different lower-level directory. This relationship can be seen in Figure 1. There are three attachments: each attach is a directory entry within `/mnt/ncryptfs` and stacks on a separate lower directory. In this figure the three attaches are `proj`, `mcm`, and `cpw`—which stack on `/proj/src`, `/home/mcm/enc`, and `/home/cwright/mail`, respectively. Encrypted files are stored within the directories `/proj/src`, `/home/mcm/enc`, and `/home/cwright/mail`. The plaintext view of these files is available through `/mnt/ncryptfs/proj`, `/mnt/ncryptfs/mcm` and `/mnt/ncryptfs/cpw`, respectively.

An attach can be thought of as a lightweight user-mode mount. Unlike a regular mount, an NCryptfs attach is not a dangerous operation that only superusers can perform safely. A mount may hide data by mounting on top of a non-empty directory, but an attach can not

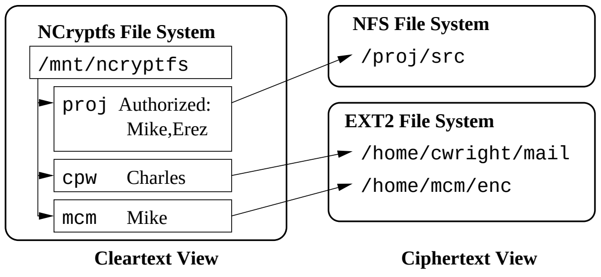

Figure 1: Attach Mode Mounts. This example shows three attachments. Each attach is a directory entry within `/mnt/ncryptfs`.

hide any data because NCryptfs does not allow any files or directories to be created in the root of the NCryptfs file system. A mount may introduce new, possibly dangerous data, such as devices or setuid programs, but NCryptfs only presents an unencrypted view of the existing data in the system, without modifying metadata. Since an attachment does not hide data or introduce new data, unlike a mount, it is a safe operation.

There are compelling reasons to use an attach to achieve this behavior, rather than simply mounting NCryptfs multiple times. In general, only the superuser may mount a new file system. It is possible to allow a user to mount a specific file system with specific parameters, but if we want to allow users to encrypt arbitrary data then this does not suffice, because `/etc/fstab` entries need to be created for each encrypted directory. Finally, most UNIX operating systems have a hard limit on the number of mounts that are allowed; or the OS uses per-mount data structures that do not scale well (e.g., linked lists). Using an attach permits the use of a specific type of stackable enhancement for many lower directories without running into hard limits or degrading system performance for other operations. NCryptfs uses the directory cache (dcache) to store attaches, because in Linux the dcache organizes many entries efficiently.

Attaches were originally designed for convenience. However, separating the name space for each encryption key also provides several other benefits in the context of stackable file systems. Foremost among these benefits is that the dcache can not handle two different views of an encrypted file system. For example, the situation where `cpw` has `/mnt/ncryptfs/foo` encrypted with one key, and user `ezk` has `/mnt/ncryptfs/foo` encrypted with another key is possible without attaches—since `foo` may encrypt to different ciphertexts with different keys. Having two files with the same name causes their data to get intermixed within the dcache and page cache. Even if the data can not be read, knowing the existence of a given file may provide valuable information to an attacker. The better way to allow multiple users to concurrently use a single cryptographic file system is to separate the name space. The only name space mixing using attach mode is the name of the attach (which must be unique). This means that NCryptfs has a completely separate name space for each set of encrypted files.

Each attach has private data that is relevant only to that specific attachment. The per-attach data is made up of an encryption key, authorizations (access control entries), and active sessions. These three data structures separate encryption, authorization, and active sessions. These three data structures model flexible and diverse policies including ad-hoc groups:

Encryption Key This information is specific to the cipher for this attach. This data includes the encryption key and any information (such as initialization vectors) required to perform encryption. NCryptfs passes this data to each encryption or decryption operation, but has no knowledge about the contents of this data. The cipher is wholly responsible for its maintenance and interpretation. This data is opaque to NCryptfs so that a multitude of ciphers can be used without any modifications to NCryptfs.

Authorizations Each attach has one or more *authorizations*. An authorization gives an entity access to NCryptfs after the entity meets a certain authentication criteria. An entity may be a process, session, user, or group. The authentication criteria consists of a method (e.g., password) and data that is specific to this method (e.g., a salted hash of the password).

Active Sessions Each attach also has one or more *active sessions*. An active session contains the description of an entity and the permissions granted to that entity. Note that NCryptfs active sessions are not necessarily the same as UNIX sessions (e.g., an active session can be bound to a user or process). Once an entity has authenticated according to the rules in an authorization, an active session is created. One authorization can map to multiple active sessions (e.g., a user authenticates in two sessions using a single authorization entry). Each active session corresponds to an authorization that exists or existed in the past. If an authorization is removed, the active sessions are allowed to remain (revocation is discussed in Section 3.5).

To ensure maximum flexibility, NCryptfs uses fine-grained permissions. Each authorization and active session contains a bitmask of permissions. NCryptfs permissions are the standard read, write, and execute bits that UNIX already defines plus an additional seven operations:

- **Detach** allows removal of the attachment from NCryptfs. When users have completed their work, the detach operation ensures that all resources (including keys) are freed.
- **Add an Authorization** allows users to delegate a subset of their permissions to new authorizations. By default, only the session that created the attach is authorized. This allows an owner to work with multiple sessions (e.g., using two different xterms) or to give other users permission to use the attach. Using a UNIX session identifier makes it more difficult to hijack an authentication, whereas a UID can be easily changed using `/bin/su`.
- **List Authorizations** allows users to verify and examine which entities (users, sessions, processes, and groups) are authorized to use this attach. Sensitive information (such as the authentication criteria) is not returned.
- **Delete an Authorization** allows users to remove an authorization from an attach.
- **Revoke an Active Session** allows users to prevent a currently-authenticated user from accessing NCryptfs. This can be combined with the authorization-deletion operation to prevent any future use of an attach.
- **List Active Sessions** allows users to verify and examine which users have authenticated to an attach.
- **Bypass VFS Permissions** allows users to take on the identity of the file's owner for files within the attach. This permission is required to implement ad-hoc groups, which allow the convenient sharing of encrypted data (see Section 3.4).

3.3.1 Attach Access Control

By default, any user is allowed to create an attachment with full permissions (except bypass VFS permissions). The system administrator can change this default policy by adding authorizations to the NCryptfs mount point. Each authorization allows a single entity to attach or authenticate to an attach. The main NCryptfs mount point has no active sessions, only authorizations. The mount point can not require authentication, because authentication takes place through an `ioctl`. If the user has not already been granted permission, then the `ioctl` will not be permitted. Once the attach or authentication takes place, the entity receives a subset of permissions in the authorization. Authorizations for the NCryptfs mount point require two additional permissions:

- **Attach** allows a user to create an attach.
- **Authentication** allows a user to authenticate to an attach.

The system administrator can also limit the maximum numbers of attaches and the maximum and minimum key timeouts both on a global and on a per user basis.

3.3.2 Attach Names

There are three methods to generate attach names. First, when a user attaches, by default the user is allowed to choose the name for their attach. This allows a convenient name that the user can easily remember and type, but this name may reveal information about the contents and multiple users may want to use the same name (e.g., `mail` or `src`).

In the second method, NCryptfs generates a name based on the entity doing the attaching to prevent name space collisions between users. We do this by appending a one-letter prefix based on the entity type to the entity's numeric identifier. For example, `u500` represents UID 500, and `u500s200` represents UID 500 with session ID 200. These names are easy to remember, but reveal who is doing the attach. This method is less convenient than allowing users to chose their own names, and may not be unique since each entity can have multiple attaches (e.g., UID 500 may have several encrypted directories, all used in session ID 200).

The third method is to randomly generate unique attach names. This allows a specific user to have multiple generated attaches. This method has the added benefit that it enforces using names for attaches that do not reveal information about the attachments' contents. NCryptfs guarantees randomly generated names that have no name space collisions. These names reveal no information about the contents, but are harder to remember and type. This method is even less convenient to users, but guarantees unique names.

3.4 Groups

NCryptfs supports native UNIX groups like any other entity. A UNIX group has some disadvantages, primarily that a group needs to be setup by the system administrator ahead of time. This means that users must contact the system administrator, and then wait for action to be taken.

NCryptfs supports ad-hoc groups by simply adding authorizations for several individual users (or other entities). The problem with this approach is that each additional user must have permissions to modify the lower level objects, since NCryptfs by default respects the standard lower-level file system checks. If the permissions on the lower level objects are relaxed, then new users can modify the files. However, without a corresponding UNIX group, it is difficult to give permissions to precisely the subset of users that must have them. If the permissions are too relaxed, then rogue users can trivially destroy the data and cryptanalysis become easier—even without the system being compromised.

NCryptfs's bypass-VFS-permissions option solves this problem. The owner of the attach can delegate this permission (assuming root has given it to the owner). When this is enabled, NCryptfs performs all permission checks independently of the lower-level file system. This allows NCryptfs to be used for a wider range of applications. This feature is described in depth in Section 4. When ACLs become common, they can be used in place of this mechanism [5, 9].

3.5 Timeouts and Revocation

Keys, authorizations, and active sessions all can have a timeout associated with them. When an object times out, NCryptfs executes a user-space program optionally specified at attach time. For example, the user may specify an application that ties into a graphical desktop environment to prompt for the user's passphrase.

NCryptfs takes one of four actions when a timed-out object is referenced:

- All further file system operations fail with "permission denied." This policy is strict and secure, but it is inconvenient.
- Opening a file fails, but already open files continue to function. This is useful because it allows existing work to complete.
- Files that are already open continue to function, but when a user attempts to open a new file, the process is put to sleep until the operation can succeed (e.g., the user re-authenticates). This also allows old work to complete, but no operations will immediately fail, so users do not need to sort out as many partial completions and errors.
- All operations cause the process to be put to sleep until the operation can succeed: open, read, write, etc. block until re-authentication. This prevents even open files from being accessed until the user re-authenticates, and is convenient because no operation will fail until the user has had a chance to re-enter the passphrase. This is similar to the authentication timeout employed in Zero-Interaction Authentication [3].

After the user re-authenticates, blocked processes wake up (or successfully complete operations). If the system is configured to cause all operations to fail or cause all processes to go to sleep, the key timeout deletes the key from memory. If existing files are permitted to continue functioning, then the key must remain in memory, but NCryptfs prevents new files from accessing it.

An authorization timeout prevents new users from authenticating with that authorization, but active sessions may continue to use the attach. This can be used to create login windows, such that all logins must take place between 9:00AM and 10:30AM.

An NCryptfs kernel thread wakes up sleeping processes after a user-specified duration. The function that caused NCryptfs to put the process to sleep returns an error. This prevents processes from waiting indefinitely for an event that may never occur (e.g., a re-authentication).

Active sessions can be revoked. A timeout is a special case of a revocation because it is a scheduled revocation, so an active session revocation has the same behavior as an active session timeout with one key difference. If an active session times out, then it may be indefinitely extended by re-authenticating even if the corresponding authorization was removed. When an active session is revoked, it may not be re-enabled. If manual intervention was taken by the owner to prevent a user from accessing NCryptfs, then we can not safely allow the user to undo that operation.

3.6 System Operation

In this section we describe a typical scenario for NCryptfs usage. The end product of the following example is the structure seen in Figure 1. We show the steps that lead up to the attach structure that is in place.

First, the system administrator mounts NCryptfs on `/mnt/ncryptfs`, then adds authorizations for Mike, Erez, and Charles. User Erez has full permissions including bypass VFS permissions, and other users have full permissions except bypass-VFS-permissions. No data is encrypted just yet. The mount point only contains the "`.`" and "`..`" directory entries.

Next, Charles, Mike, and Erez create attaches. For example, Charles runs the command "`nc_attach -c blowfish /mnt/ncryptfs cpw /home/cwright/mail`". Then, `nc_attach` prompts for a key. The key may be entered as a hexadecimal or ASCII string. An encryption key is then derived from this string using PKCS#5 PBKDF2 [23]. The key is passed to the Blowfish cipher module [25]. After this command is completed, `/mnt/ncryptfs/cpw` presents a decrypted view of the files in `/home/cwright/mail`. In this situation, Charles may access the encrypted files and he has full permissions. The mapping between `mcm` and `/home/mcm/enc` is created in the same manner.

Erez performs the same operation to map `/mnt/ncryptfs/proj` to `/proj/src`. Erez wants to give Mike the ability to read files in this attach as well. Mike and Erez share no UNIX group among themselves, but Erez has the ability to bypass VFS permissions. Erez adds an authorization for Mike with the bypass-VFS-permissions option enabled. To create this authorization, Erez specifies several options aside from the permissions. The first, required setting, is the authorization criteria. In this example password authentication was chosen. A salted MD5 hash of the passphrase is passed to the kernel when creating a password authorization [22]. Using a salted MD5 hash allows Mike to choose his passphrase without revealing it to Erez, and Erez can store it in a configuration file without the original passphrase being revealed. Additionally, Erez may authorize any session that Mike opens, but to use the attach from a session, Mike needs to be authenticated in that particular session. This ties an active session to a specific virtual terminal, to make hijacking the active session more difficult.

Finally, Erez can specify timeouts for the authorization that he is creating. In this example, we chose an authorization timeout of six hours, an active session timeout of one hour, and an inactivity timeout of fifteen minutes. This means that Mike can authenticate to the attach within the next six hours; once authenticated, he can use the attach for one hour without re-authenticating; and if he does not use the encrypted files for more than fifteen minutes, he must re-authenticate. After a timeout, all of Mike's processes go to sleep until Mike re-authenticates. This shrinks the window in which the encryption key may be used.

To use the attach that Erez has created, Mike runs `nc_auth /mnt/ncryptfs proj`, and `nc_auth` prompts him for a passphrase. If Mike is successful, then he may use the files in `/mnt/ncryptfs/proj`. Mike starts a large compile, which lasts for more than an hour. After an hour, the active session timeout is triggered. Mike has configured a user-space hook for his timeout; this hook loads a small X11 program which prompts him for his passphrase, and re-authenticates to NCryptfs. After Mike successfully enters his passphrase, his compile resumes. If Mike is unable to enter the passphrase, then after a configurable amount of time the compile fails with a "permission denied" message.

4 Implementation

We implemented a prototype of NCryptfs on Linux 2.4.18 using FiST, a language for stackable file systems, as a starting point [33]. Although much of NCryptfs was implemented as a standalone stackable file system, we needed to modify some parts of the kernel to increase the security of NCryptfs. Using stackable mechanisms allows us to create new file systems without changes to the rest of the OS. However, Linux is not sufficiently flexible to allow the creation of completely independent secure file systems. Although this reduces portability as compared to a standalone stackable file system, the increased security is worth this trade off. We have designed these features in a way that NCryptfs can be used without them (at the cost of reduced security). In this section, we discuss four interesting aspects of our implementation: on-exit callbacks for tasks, cache cleaning, the imperfect stacking characteristics of permission handling in Linux, and our use of cryptography.

On-exit callbacks Linux does not provide a way for kernel components to register interest in the death of a process. We extended the Linux kernel to allow an arbitrary number of private data fields and corresponding callback functions to be added to the per-process data structure, `struct task`. When a process exits, the kernel first executes any callback functions before destroying the `task` object. NCryptfs uses such callbacks in two ways. First, when processes die, we immediately remove any active sessions that are no longer valid (e.g., if the last process in a session dies, we remove the NCryptfs active session). The alternative to on-exit callbacks would have been to use a separate kernel thread to perform periodic garbage collection, but that leaves security data vulnerable for an unacceptably long window. If an authenticated process terminates and the active session remains valid, then an attacker can quickly create many processes to hijack the active session entry. Using the on-exit call back, the information is invalidated before the process ID can be reused. Also with on-exit callbacks we release memory resources as soon as they are no longer needed.

The second use for private per-process data is in NCryptfs's challenge-response authentication ioctl, which proceeds as follows. First, the user process runs an ioctl to request a challenge. We generate a random challenge, store it as task-private data, and then send the challenge's size back to the user. Second, the user allocates a buffer and calls the ioctl again. This time the kernel actually returns the challenge data. The user performs some function on the data (e.g., HMAC-MD5 [12]), that requires some piece of secret knowledge to transform the data. Finally, the user program calls the ioctl a third time with the response. If the response matches what the kernel expects, then the user is authenticated. Using the task private data sets up a transaction between the kernel and a task. Even though there are several ioctls in this authentication sequence, it is no different than if a process had authenticated itself using a single ioctl. The challenge and its size are not useful to an attacker, since the challenge can only be used for a single authentication attempt. Only the last ioctl call modifies the state of the task. Finally, if an authentication is aborted, then the challenge is discarded on process termination.

Cache Cleaning Cleartext pages normally exist in the page cache. Unused file data and metadata may remain in the dcache and icache, respectively. If a system is compromised, then this data is vulnerable to attack. For example, an attacker can examine memory (through `/dev/kmem` or by loading a module). To limit this exposure, NCryptfs evicts cleartext pages from the page cache, periodically and on detach. Unused dentries and inodes are also evicted from the dcache and icache, respectively. For added security at the expense of performance, NCryptfs can purge cleartext data from caches more often. In this situation, the decryption expense is incurred for each file system operation, but I/O time is not increased if the ciphertext pages (e.g., EXT2 pages) are already in the cache. Zero-Interaction Authentication (ZIA), another encryption file system based on Cryptfs, takes the approach of encrypting all pages when an authentication expires [3]. This is less efficient than the NCryptfs method, because ZIA will also maintain copies of pages in the lower-level file system. ZIA requires the initial encryption of pages, and will use memory for these encrypted pages.

Bypassing VFS Permissions When intercepting permissions checks, it is trivial to implement a policy that is more restrictive than the underlying file system's normal UNIX mode-bit checks. To support ad-hoc groups without changing lower-level file systems, however, NCryptfs needed to completely ignore the lower-level file system's mode bits so that NCryptfs could implement its own authentication checks and yet appear to access the lower-level files as their owner.

The flow of an NCryptfs operation that must bypass VFS permissions (e.g., `unlink`) is as follows:

```
sys_unlink {                              /* system call service routine */
  vfs_unlink {                                          /* VFS method */
    call nc_permission()
    if not permitted: return error
    nc_unlink {                                   /* NCryptfs method */
      call nc_perm_preop()                        /* code we added */
      vfs_unlink {                                      /* VFS method */
        call ext2_permission()
        if not permitted: return error
        call ext2_unlink()                          /* EXT2 method */
      }                                   /* end of inner vfs_unlink */
      call nc_perm_fixup()                        /* code we added */
    }                                            /* end of nc_unlink */
  }                                       /* end of outer vfs_unlink */
}                                               /* end of sys_unlink */
```

The VFS operation (e.g., `vfs_unlink`) checks the permission using the `nc_permission` function. If the permission check succeeds, the corresponding NCryptfs-specific operation is called (e.g., `nc_unlink`). NCryptfs locates the lower-level object and again calls the VFS operation. The VFS operation checks permissions for the lower-level inode before calling the lower-level operation. This control flow means that we can not actually intercept the lower-level permission call. Instead, we change `current->fsuid` to the owner of the lower-level object before the operation is performed and restore it afterward, which is done in `nc_perm_preop` and `nc_perm_fixup`, respectively. We change only the permissions of the current task, and the process can not perform any additional operations until we restore the `current->fsuid` and return from the NCryptfs function. This ensures that only one lower file system operation can be performed between `nc_perm_preop` and `nc_perm_fixup`.

Linux 2.6 will have Linux Security Modules (LSMs) that allow the interception of many security operations [29]. Unfortunately, the LSM framework is not sufficient to bypass lower-level permissions either. Their VFS calls the file-system–specific permission operation first. The LSM permissions operation is called only if the file-system–specific operation succeeds. The LSM operation can allow or deny access only to objects that the file system has already permitted. A better solution is to consult the file-system–specific permission operation. This result should be passed to the LSM module which can make the final decision, possibly based on the file-system–specific result.

Cryptography To ensure data confidentiality, NCryptfs uses strong cryptography algorithms (e.g., Blowfish or AES in CFB mode). File data and file names are handled in two different ways.

Data is encrypted one page at a time, using an initialization vector (IV) specified along with the encryption key XORed with the inode number and page number. For security, ideally the entire file would be encrypted at once, but then random access would be prohibitively expensive; to access the nth byte of data, n bytes would need to be decrypted, and any write would require re-encryption of the entire file. For optimal performance, each byte would be encrypted individually, but without data interdependence, encryption becomes significantly less secure [24].

File names are encrypted with the IV XORed with the inode number of the directory, but the output may contain characters that are not valid UNIX pathnames (i.e., `/` and NULL). To rectify this problem, the result is base-64 encoded before being passed to the lower-level file system. This reduces the maximum path length by 25%. A checksum is stored at the beginning of the encrypted file name for two reasons. First, if a file name is not encrypted with the correct key, then this checksum will prevent it from appearing in NCryptfs. Second, since CFB mode is used, if two files have a common prefix, then they will have a common encrypted prefix. Since it is unlikely that these two files will have the same checksum, prefixing their names with the checksum will prevent them from having the same prefix in the ciphertext. Finally, the directory entries "`.`" and "`..`" are not encrypted to preserve the directory structure on the lower-level file system.

5 Evaluation

We developed a prototype of NCryptfs in Linux 2.4.18. We compare it with CFS, TCFS, and BestCrypt. We chose these three different systems because they represent a cross section of techniques:

- **CFS** is a localhost NFS server, and among the first widely-used cryptographic file systems.
- **TCFS** is an NFS client that implements cryptographic functionality including data integrity assurance.
- **BestCrypt** is an encrypted loopback device driver. BestCrypt is a commercial product.
- **NCryptfs** is our stackable file system.

We begin by describing the various features, security, and convenience aspects of each system in Section 5.1. We compare their performance in Section 5.2.

5.1 Feature Comparison

In this section we present a comparison of the different functionality implemented by CFS, TCFS, BestCrypt, Cryptfs (the predecessor to NCryptfs), and NCryptfs. We identified the following metrics, which we summarized in Table 1:

1. **No keys stored on disk**: If keys are stored persistently, then encryption adds little security. All of the systems we compared, except TCFS, do not store keys on disk. TCFS stores each user's key in a database encrypted using the user's login password. Login passwords are restricted to eight characters, must contain only printable characters, and are often used and thus may be inadvertently exposed in cleartext. If separate passwords were used, then this would not be a weakness in TCFS.
2. **Keys protected from swap devices**: If memory becomes scarce, memory that could contain keys may be swapped. NCryptfs prevents this by pinning keys in physical memory, but cleartext process data can still be written to swap. An encrypted swap partition can prevent all sensitive data (and non-sensitive data) from being written to persistent media in the clear [20]. BestCrypt can encrypt an entire swap device to prevent data from leaking.
3. **Reveals no directory structure information**: CFS, TCFS, and NCryptfs reveal the number of files and their structure as well as inode meta-data information. BestCrypt uses a single file for an entire encrypted file system so it does not reveal this information.
4. **Multiple Concurrent Users**: CFS and NCryptfs allow users to create their own personal attachments. NCryptfs additionally allows multiple users to use a single attachment, with distinct permissions. TCFS allows multiple users to use a common name space with different keys. BestCrypt allows different users to use the same container with different passwords, but with the same permissions.

	Feature	CFS	TCFS	BestCrypt	Cryptfs	NCryptfs
1	No keys stored on disk	✔	[a]	✔	✔	✔
2	Keys protected from swap devices			✔ [b]		✔
3	Reveals no directory structure			✔		
4	Multiple concurrent users	✔[c]	✔	✔		✔
5	Users do not need root intervention	✔				✔
6	Multiple ciphers	✔	✔	✔		✔
7	Automatic cipher loading		✔			✔
8	Separate permissions per user					✔
9	Group support – UNIX GID		✔			✔
10	Group support – ad-hoc					✔
11	Challenge-response authentication					✔
12	Data integrity assurance		✔			
13	Per-file encryption flag		✔			
14	Threshold secret sharing		✔			
15	Key timeouts	✔				✔
16	User-space timeout callback					✔
17	Process sleep/wakeup on key timeout					✔
18	Implementation technique	NFS server	NFS client	loop device	stackable	stackable
19	No. of systems available	any UNIX	3[d]	2	3	1[e]
20	Additional Blowfish LOC (Lines Of Code, excludes cipher implementation)	33	109	99	0	76
21	Total core LOC	5258	14731	3526	4943	6537

Table 1: Feature comparison. A check mark indicates that the feature is supported, otherwise it is not.
[a] TCFS stores on disk keys encrypted with login passwords.
[b] BestCrypt can encrypt the entire swap device.
[c] CFS supports multiple users, but is single threaded.
[d] TCFS provides cipher modules and data integrity assurance only on Linux.
[e] NCryptfs is based on the FiST templates, which are available on three systems.

5. **Users do not need root intervention**: After an initial setup, users do not need root intervention to create new sets of encrypted data in CFS or NCryptfs. In TCFS, the system administrator must run `tcfsadduser` for each user who needs TCFS access. In BestCrypt and Cryptfs, root must allow each encrypted directory to be mounted.
6. **Multiple Ciphers**: CFS, TCFS, BestCrypt, and NCryptfs support this.
7. **Automatic cipher loading**: TCFS and NCryptfs support this feature. In CFS, all ciphers are statically compiled into `cfsd`. BestCrypt loads all available ciphers, whether they are used or not.
8. **Separate permissions per user**: When using groups, NCryptfs allows each member to have individually-defined permissions (e.g., read, write, or detach). Other systems treat all users the same as each other.
9. **Group support – UNIX GID**: TCFS and NCryptfs support UNIX groups.
10. **Group support – ad-hoc**: Only supported in NCryptfs.
11. **Challenge-response authentication**: Only supported in NCryptfs.
12. **Data integrity assurance**: TCFS detects modifications to the ciphertext. If data is modified on the underlying file system, then CFS, NCryptfs, and BestCrypt do not detect this.
13. **Per-file encryption flag**: TCFS allows users to specify whether data is encrypted on a per-file basis. This may confuse users, since not all files within TCFS are be encrypted.
14. **Threshold secret sharing**: TCFS allows a group key to be split into n pieces. If m of these n members of the group insert their key into TCFS, then the full key can be reconstructed.
15. **Key timeouts**: CFS can automatically detach an attach after a certain period of time. NCryptfs can time out keys, active sessions, and authorizations.
16. **User-space timeout callback**: NCryptfs can optionally execute a user-space program on timeouts.
17. **Process sleep/wakeup on key timeout**: NCryptfs has four types of possible behavior on timeouts: all operations fail, new files operations fail, all operations put the calling process to sleep, or operations on new files put the calling process to sleep.
18. **Implementation technique**: CFS is a user-space localhost NFS server that works with standard NFS

clients. Running in user-space decreases performance, but increases portability. TCFS is a kernel-space NFS client that works with any NFS server. BestCrypt is a kernel loopback device driver. This means that it has lower overhead than other systems. Cryptfs and NCryptfs are stackable file systems that run in kernel space. Since stackable file systems run in kernel space, they have better performance than user-space file systems, and are easier to develop than disk or network-based file systems.

19. **Number of systems available**: CFS can run on any UNIX system. TCFS runs on Linux, OpenBSD, and NetBSD, but is only feature complete on Linux. BestCrypt runs on Linux and Windows. Cryptfs runs on Linux, FreeBSD, and Solaris. The NCryptfs prototype runs on Linux, but is based on the FiST templates, which run on Linux, FreeBSD, and Solaris.
20. **Additional Blowfish *Lines Of Code* (LOC)**: The total number of lines of code needed to interface with an existing cipher is a good metric for how difficult it is to add additional ciphers. To interface with Blowfish, CFS, TCFS, BestCrypt, and NCryptfs use small wrappers. Cryptfs hard-codes the calls to Blowfish.
21. **Total core LOC**: The number of LOC in the file system is a good measure of maintainability, complexity, and the amount of initial effort to write the system. CFS, Cryptfs, and NCryptfs have roughly the same number of LOC. TCFS re-implements an NFS client and is more than twice the size of any other system. BestCrypt has the smallest implementation, which is to be expected because it is a loopback device driver, not a file system.

NCryptfs supports a rich feature set that allows system administrators and users to tailor it to their site-specific security, performance, and convenience needs.

5.2 Performance Comparison

We compared the performance of CFS, TCFS, BestCrypt, and NCryptfs. We ran all benchmarks on a 1.7GHz Pentium 4 machine with 128MB of RAM. All experiments were located on a 30GB 7200 RPM Western Digital Caviar IDE disk formatted with EXT2. The machine was installed with Red Hat Linux 7.3. For CFS, BestCrypt, and NCryptfs, we ran a vanilla 2.4.18 kernel. For TCFS, we used the most recent supported kernel, 2.2.17, with the TCFS patch applied. To ensure cold cache, we unmounted the file systems where the experiments took place between each test. All other executables and libraries (e.g., compilers) were located on the root file system. We ran all tests several times, and our computed standard deviations were less than 5%. We chose Blowfish with a 128 bit key for our cipher, since it is widely available and performs well in software. We recorded elapsed, system, and user times for all tests.

5.2.1 Configurations

We used the following ten configurations:

- **EXT2-24** A vanilla EXT2 running on the 2.4.18 kernel. It serves as a baseline for performance of other configurations.
- **EXT2-22** A vanilla EXT2 running on the 2.2.17-tcfs kernel. It serves as a baseline for TCFS performance.
- **CFS-NULL** CFS using the identity cipher (this copies data with no modification). This demonstrates the overhead of the file system without cryptographic operations.
- **TCFS-NULL** TCFS using the identity cipher.
- **BC-NULL** BestCrypt using the identity cipher.
- **NC-NULL** NCryptfs using the identity cipher
- **CFS-BF** CFS using the Blowfish cipher. This demonstrates the overhead of CFS including cryptography.
- **TCFS-BF** TCFS using the Blowfish cipher. TCFS is designed to generate several keys per file. However, due to a memory leak we discovered in TCFS, we were forced to use a single key for all encryption operations. This means that the overhead introduced by TCFS will be underrepresented because initializing a Blowfish key is an expensive operation, which uses 4168 bytes of memory and requires 521 iterations of Blowfish encryption [25].
- **BC-BF** BestCrypt using the Blowfish cipher.
- **NC-BF** NCryptfs using the Blowfish cipher.

5.2.2 Workloads

We tested our configurations using two workloads: one CPU-intensive and another that is I/O intensive.

The first workload was a build of Am-utils [17]. We used Am-utils 6.0.7: it contains over 50,000 lines of C code in 425 files. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation: a total of 265 additional files. The Am-utils compile contains a fair mix of file system operations; it is CPU intensive and performs many meta-data operations during the `configure` process. This workload demonstrates what type of performance impact a user may see while using NCryptfs.

The second workload we chose was Postmark [11]. We configured Postmark to create 20,000 files and perform 100,000 transactions in ten directories. This benchmark uses little CPU, but is I/O intensive. Postmark focuses on stressing the file system by performing a series of file system operations such as directory lookups, creations, and deletions on small files. A large number of small files is common in electronic mail and news servers where multiple users are randomly modifying small files. We chose the above parameters for the number of files and transactions as they are typically used and recommended for file system benchmarks [11, 27].

5.2.3 Am-utils Results

The elapsed time overhead for Am-utils is shown in Figure 2 and the first two rows of Table 2. This shows that CFS and TCFS performance suffer from using the network stack. TCFS performance additionally suffers from data integrity checks. BestCrypt uses a kernel thread to perform all encryptions, so cryptographic operations may continue after the termination of the instrumented process. This helps to improve perceived performance, which we further explore later in Section 5.2.4. NCryptfs only minimally impacts performance.

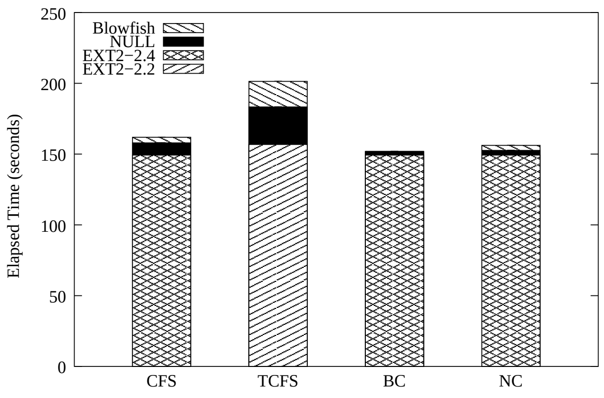

Figure 2: Am-utils build elapsed time. Each bar represents an EXT2 *configuration, a NULL configuration, and an encrypting configuration.*

Configuration	CFS	TCFS	BC	NC
Elapsed Time – NULL	5.7	16.9	1.5	2.2
Elapsed Time – BF	8.4	28.4	1.7	4.5
System Time – NULL	25.5	50.3	0.7	4.6
System Time – BF	39.5	93.7	1.8	17.0

Table 2: Am-utils percentage overheads over EXT2

The system time used by a process demonstrates how much CPU was used by the additional file system overhead and encryption. These results can be seen in Figure 3 and the last two rows of Table 2. CFS has a user-space process, `cfsd`, which performs all encryption. BestCrypt has a kernel-space thread that performs encryption. We added the time used by these processes into the system time to represent the total time used on behalf of the process. TCFS makes use of `knfsd`, but as this can be on a remote server we do not include its overhead. These results show that the overhead for CFS and TCFS is quite large compared to BestCrypt and NCryptfs. This was expected because of the added network overhead. BestCrypt has a simpler interface and hence a smaller overhead than NCryptfs.

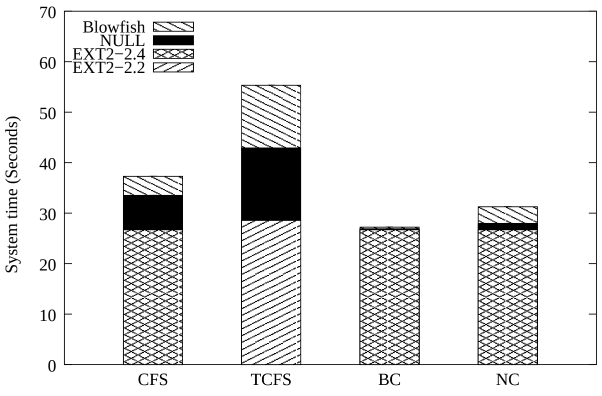

Figure 3: Am-utils build system time. Each bar represents an EXT2 *configuration, a NULL configuration, and an encrypting configuration.*

The user times for all tests were within 2% of the EXT2 configuration. This is expected as the encryption happens outside of the process in all of these systems.

5.2.4 Postmark Results

The elapsed time overheads for Postmark are shown in Figure 4 and the first two rows of Table 3. This shows that for I/O-intensive operations, all of the cryptographic file systems we tested have a non-negligible impact on system performance. Increasing security often affects performance, so these results were consistent with our expectations. The results for CFS and TCFS show a larger performance overhead than for Am-utils, but the overhead of user-space data copies and the NFS protocol explain this. BestCrypt performed significantly better in the Am-utils test than in the Postmark test; this is because BestCrypt uses a single kernel thread for encryption that often disables interrupts. The I/O intensive nature of the Postmark test exposes this behavior. Other elapsed times did not significantly increase when encryption was used.

Configuration	CFS	TCFS	BC	NC
Elapsed Time – NULL	119	106	101	56
Elapsed Time – BF	123	106	127	59
System Time – NULL	553	50	95	51
System Time – BF	821	118	280	156

Table 3: Postmark percentage overheads over EXT2

Figure 5 and the last two rows of Table 3 shows system time in the same manner as described in Section 5.2.3. CFS has the worst performance degradation over EXT2, caused by the excessive number of data copies from user-space to kernel-space and within the network

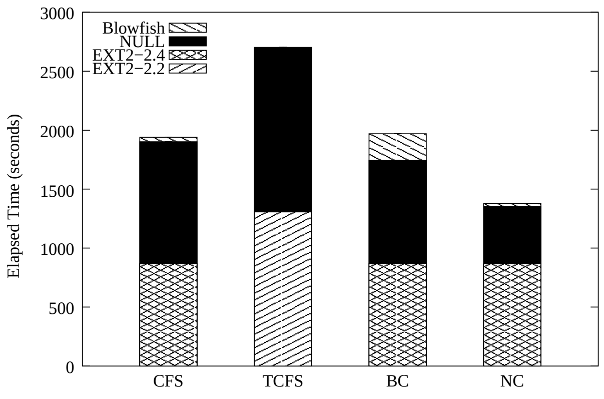

Figure 4: Postmark elapsed time. Each bar represents an EXT2 *configuration, a NULL configuration, and an encrypting configuration.*

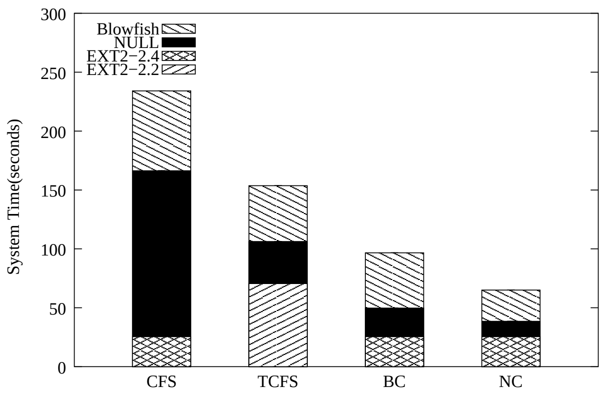

Figure 5: Postmark system time. Each bar represents an EXT2 *configuration, a NULL configuration, and an encrypting configuration.*

stack. TCFS also suffers from data-copy overheads in the network stack. BestCrypt and NCryptfs both have smaller overheads than CFS and TCFS.

6 Conclusions

Our main contribution is in designing and building a cryptographic file system that for the first time, to our best knowledge, was developed with the express goal of balancing all of these four conflicting aspects: security, performance, convenience, and portability.

We achieved high security by including support for many ciphers and authentication methods, addressing vulnerabilities in OS caches and the task manager, separate OS name spaces, separate encryption from authentication keys, session-based and process-based encryption, reduced user need for superuser privileges, and key timeouts that are transparent to processes.

We achieved high performance by designing NCryptfs to run in the kernel. Our performance benchmarks show a small 5% overhead for normal system operation.

We achieved ease of use by allowing administrators and users alike to tailor the levels of security, performance, and convenience to their needs: by providing encryption and authentication that is transparent to users and processes; by allowing users to quickly attach and detach from NCryptfs; by supporting ad-hoc encryption groups for shared yet secure collaboration; and by automatic loading of ciphers and authentication modules.

Lastly, we achieved high portability by building NCryptfs as a stackable file system. This allows users to use any file system as the backing store for encrypted data, and helps to reduce the development and porting effort of NCryptfs to other systems. Although our first prototype works in Linux alone, we developed NCryptfs using the FiST stackable templates system, which also supports Solaris and FreeBSD [34].

6.1 Future Work

In the immediate future, we plan to integrate a lockbox mode and cryptographic checksumming into NCryptfs. One possible method to achieve integrity assurance is to store block-by-block checksums of the file in a separate checksum file. This technique is similar to using index files in size-changing file systems [31]. We will also investigate the best way to store a unique IV along with each file, so that NCryptfs does not rely on the inode number of files.

Currently, a stackable file system can change lower-level file system data independently from upper-level data. Data on multiple levels must be associated within OS caches to present a coherent system view. We plan to modify the VFS caches to associate each upper level cache object (dentry, inode, and page) with its lower level object [7].

Presently, NCryptfs exposes the owner and other inode meta-data of a file. We cannot remove all of this structure without making incompatible changes to the lower-level file system. We can perturb some data, such as adding padding to the file's name or storing ownership information outside of the lower-level inode. These operations may complicate backup and restore. We also expect this to decrease system performance.

7 Acknowledgments

We wish to thank Jeffrey Osborn, Amit Purohit, and Kiran Reddy for reviewing early drafts of the paper and assisting with benchmarking. We thank Tzi-cker Chiueh, Fred B. Schneider, Jukka T. Virtanen, our shepherd Bennet Yee, and the anonymous USENIX reviewers for the valuable feedback they provided. This work was partially made possible by an NSF CAREER award EIA-0133589, and HP/Intel gifts numbers 87128 and 88415.1.

References

[1] M. Blaze. A cryptographic fi le system for Unix. In *Proceedings of the first ACM Conference on Computer and Communications Security*, 1993.

[2] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 245–252, June 2001.

[3] M. Corner and B. D. Noble. Zero-interaction authentication. In *The Eigth ACM Conference on Mobile Computing and Networking*, September 2002.

[4] R. Dowdeswell and J. Ioannidis. The CryptoGraphic Disk Driver. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, June 2003.

[5] A. Grüenbacher. Extended attributes and access control lists. http://acl.bestbits.at, 2002.

[6] P. C. Gutmann. Secure fi lesystem (sfs) for dos/windows. www.cs.auckland.ac.nz/˜pgut001/sfs/index.html, 1994.

[7] J. Heidemann and G. Popek. Performance of cache coherence in stackable fi ling. In *Proceedings of Fifteenth ACM Symposium on Operating Systems Principles*. ACM SIGOPS, 1995.

[8] J. Hennessey. The Future of Systems Research. *IEEE Computer*, 32(8):27–32, 1999.

[9] IEEE/ANSI. Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API)—Amendment: Protection, Audit, and Control Interfaces [C Language]. Technical Report STD-1003.1e draft standard 17, ISO/IEC, October 1997. *Draft was withdrawn in 1997.*

[10] Jetico, Inc. BestCrypt software home page. www.jetico.com, 2002.

[11] J. Katcher. PostMark: a New Filesystem Benchmark. Technical Report TR3022, Network Appliance. www.netapp.com/tech˙library/3022.html.

[12] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. Technical Report RFC 2104, Internet Activities Board, February 1997.

[13] A. D. McDonald and M. G. Kuhn. StegFS: A Steganographic File System for Linux. In *Information Hiding*, pages 462–477, 1999.

[14] Microsoft Corporation. Encrypting File System for Windows 2000. Technical report, July 1999. www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp.

[15] R. Nagar. *Windows NT File System Internals: A developer's Guide*, pages 615–67. O'Reilly, September 1997. Section: Filter Drivers.

[16] National Association of Insurance Commissioners. Graham-Leach-Bliley Act, 1999. www.naic.org/GLBA.

[17] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.0.4 edition, February 2000. www.am-utils.org.

[18] R. Pike. Systems Software Research is Irrelevant. Bell Labs, February 2000. www.cs.bell-labs.com/who/rob/utah2000.pdf.

[19] R. Power. Computer Crime and Security Survey. *Computer Security Institute*, VIII(1):1–24, 2002. www.gocsi.com/press/20020407.html.

[20] N. Provos. Encrypting virtual memory. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[21] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 15–30, Monterey, CA, January 2002.

[22] R. L. Rivest. The MD5 Message-Digest Algorithm. Technical Report RFC 1321, Internet Activities Board, April 1992.

[23] RSA Laboratories. Password-Based Cryptography Standard. Technical Report PKCS #5, RSA Data Security, March 1999.

[24] J. H. Saltzer. Hazards of fi le encryption. Technical report, 1981. http://web.mit.edu/afs/athena.mit.edu/user/other/a/Saltzer/www/publications/csrrfc208.html.

[25] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 2 edition, October 1995.

[26] U.S. Dept. of Health & Human Services. The Health Insurance Portability and Accountability Act (HIPAA), 1996. www.cms.gov/hipaa.

[27] VERITAS Software. Veritas fi le server edition performance brief: A postmark 1.11 benchmark comparison. Technical report. http://eval.veritas.com/webfi les/docs/fsedition-postmark.pdf.

[28] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the Eigth Usenix Security Symposium*, August 1999.

[29] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[30] E. Zadok. Stackable fi le systems as a security tool. Technical Report CUCS-036-99, Computer Science Department, Columbia University, December 1999.

[31] E. Zadok, J. M. Anderson, I. Bădulescu, and J. Nieh. Fast Indexing: Support for size-changing algorithms in stackable fi le systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 289–304, June 2001.

[32] E. Zadok, I. Bădulescu, and A. Shender. Cryptfs: A stackable vnode level encryption fi le system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998.

[33] E. Zadok, I. Bădulescu, and A. Shender. Extending fi le systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, June 1999.

[34] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.